CENG443
Heterogeneous Parallel Programming

Introduction to CUDA

3 Ways to Accelerate Applications

Libraries	Compiler Directives	Programming Languages
Easy to use Most Performance	Easy to use Portable code	Most Performance Most Flexibility

Libraries

Ease of use: Using libraries enables GPU acceleration without in-depth knowledge of GPU programming

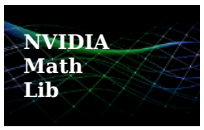
“Drop-in”: Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes

Quality: Libraries offer high-quality implementations of functions encountered in a broad range of applications

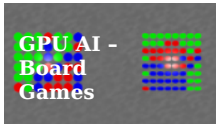
GPU Accelerated Libraries

Linear Algebra
FFT, BLAS,
SPARSE, Matrix

Numerical & Math
RAND, Statistics



Data Struct. & AI
Sort, Scan, Zero Sum



Visual Processing
Image & Video

NVIDIA
Video
Encode

Vector Addition in Thrust

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void)
{
  thrust::device_vector<float> X(3);
  thrust::device_vector<float> Y(3);
  thrust::device_vector<float> Z(3);

  X[0] = 10; X[1] = 20; X[2] = 30;
  Y[0] = 15; Y[1] = 35; Y[2] = 10;

  thrust::transform(X.begin(), X.end(),
                    Y.begin(),
                    Z.begin(),
                    thrust::plus<float>());

  for (size_t i = 0; i < Z.size(); i++)
    std::cout << "Z[" << i << "] = " << Z[i] << "\n";

  return 0;
}
```

Compiler Directives

Ease of use: Compiler takes care of details of parallelism management and data movement

Portable: The code is generic, not specific to any type of hardware and can be deployed into multiple languages

Uncertain: Performance of code can vary across compiler versions

OpenACC

Compiler directives for C, C++, and FORTRAN

```
#pragma acc parallel loop copyin(input1[0:inputLength],input2[0:inputLength]),
copyout(output[0:inputLength])
    for(i = 0; i < inputLength; ++i) {
        output[i] = input1[i] + input2[i];
    }
```

Programming Languages

Performance: Programmer has best control of parallelism and data movement

Flexible: The computation does not need to fit into a limited set of library patterns or directive types

Verbose: The programmer often needs to express more details

GPU Programming Languages

Numerical analytics	MATLAB Mathematica, LabVIEW
Fortran	CUDA Fortran
C	CUDA C
C++	CUDA C++
Python	PyCUDA, Copperhead, Numba
F#	Alea.cuBase

CUDA C

Easy to use
Most Performance

Easy to use
Portable code

Most Performance
Most Flexibility

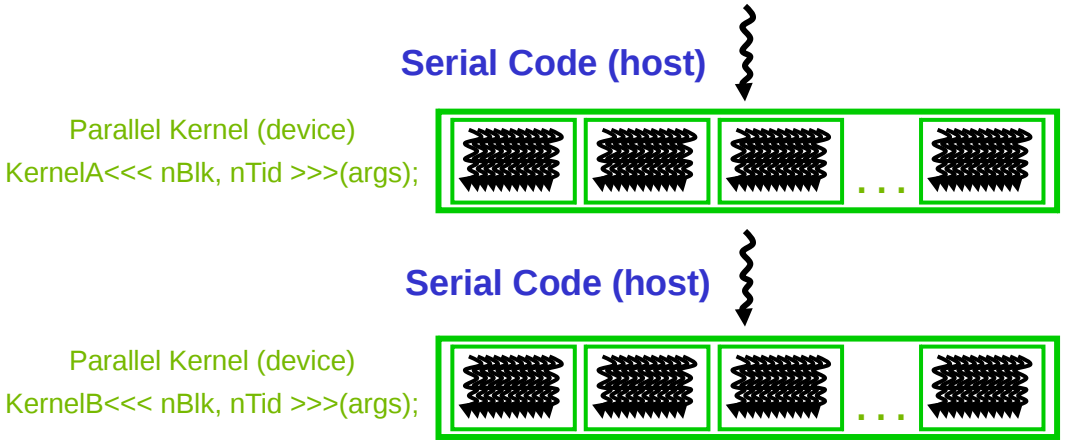
CUDA Programming

CUDA (Compute Unified Device Architecture)

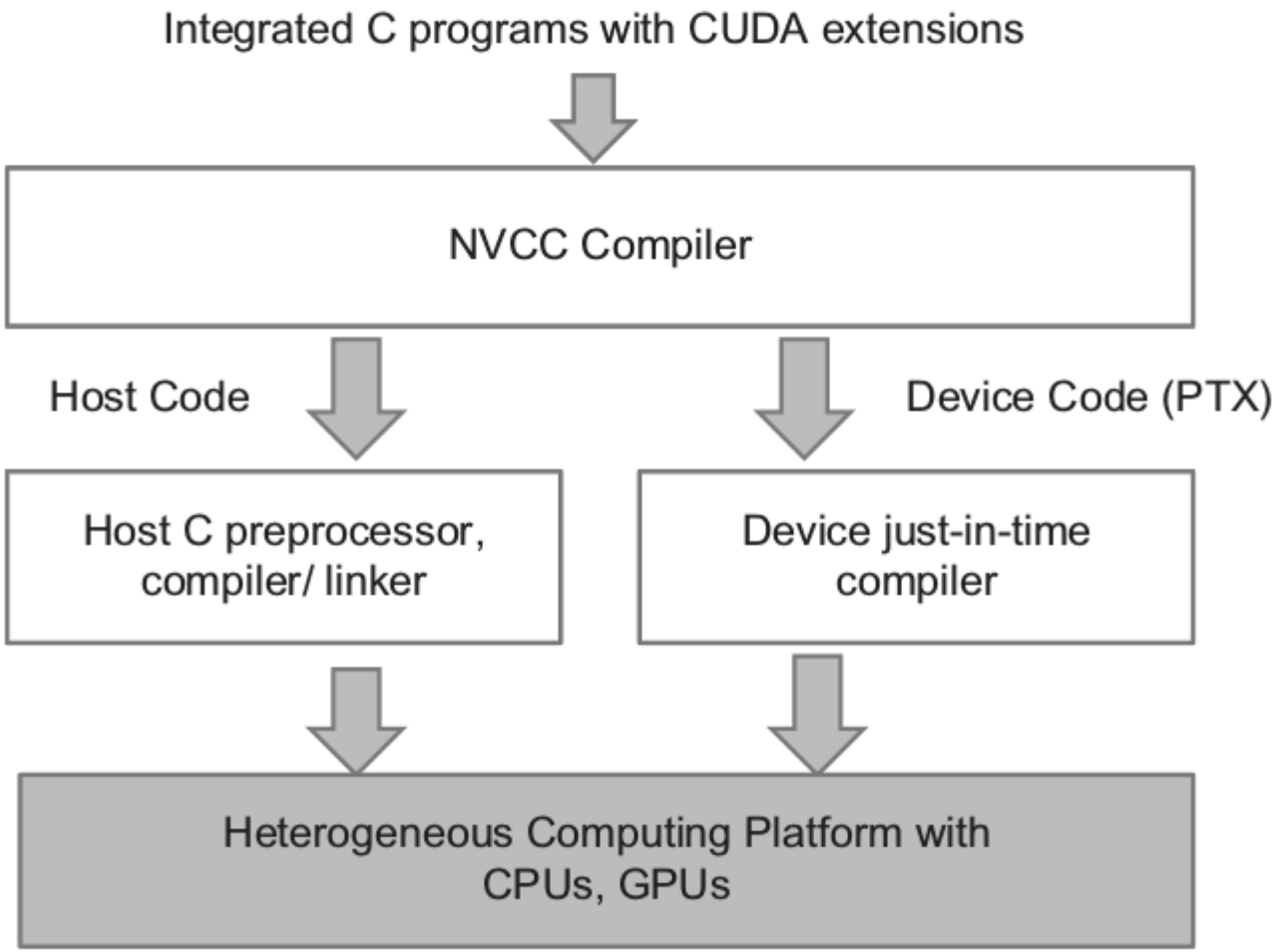
Integrated host+device app C program

Serial or modestly parallel parts in host C code

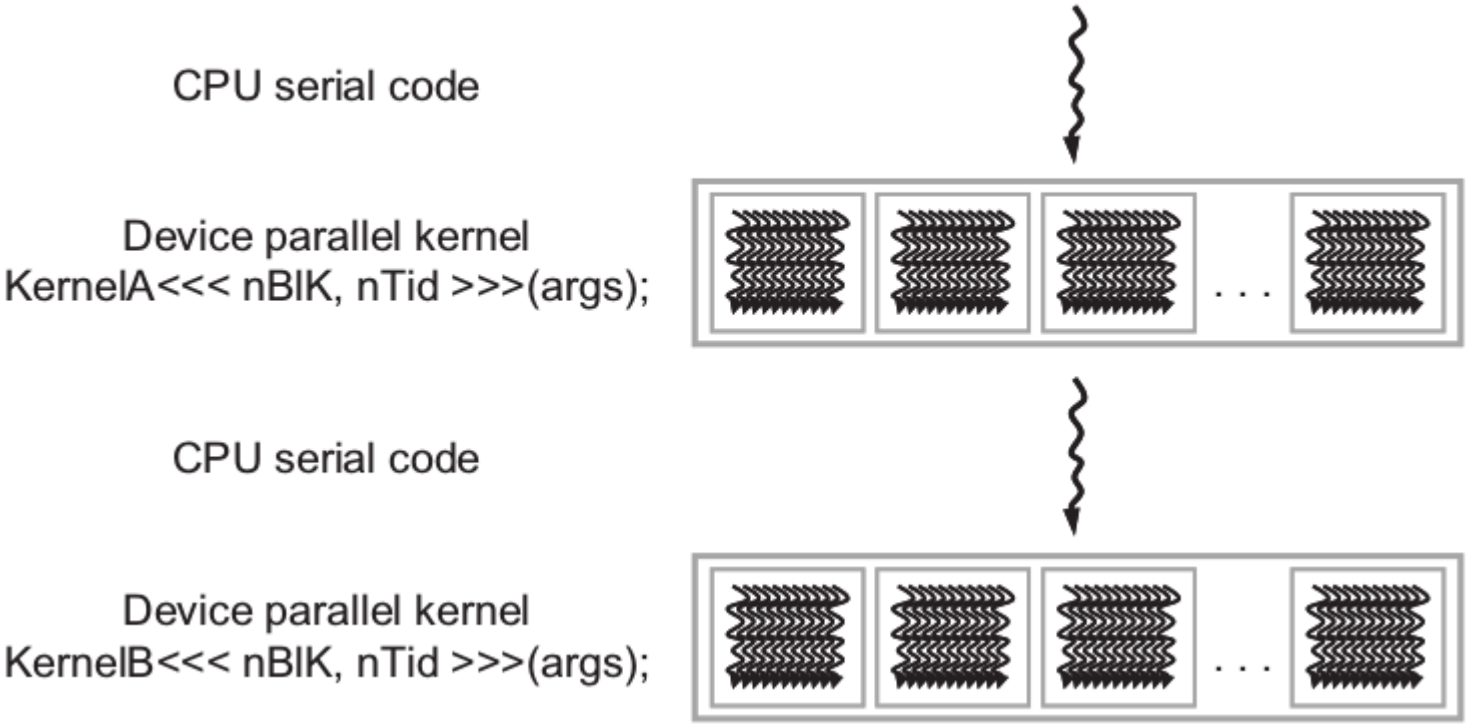
Highly parallel parts in device SPMD kernel C code



CUDA C Program Compilation



CUDA C Program Execution



Vector Addition – Traditional C Code

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
  // I/O to read h_A and h_B, N elements
  …
    vecAdd(h_A, h_B, h_C, N);
}
```

Heterogeneous Computing

Setup inputs on the host (CPU-accessible memory)

Allocate memory for outputs on the host

Allocate memory for inputs on the GPU

Allocate memory for outputs on the GPU

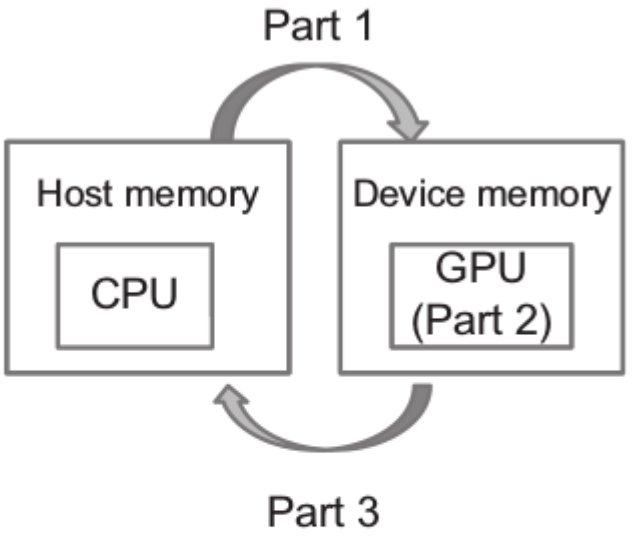
Copy inputs from host to GPU

Start GPU kernel (function that executed on gpu)

Copy output from GPU to host

vecAdd CUDA Host Code

```
#include <cuda.h>
…
void vecAdd(float* A, float* B, float* C, int n)
{
   int  size = n* sizeof(float);
   float  *d_A *d_B, *d_C;
   …
1. // Allocate device memory for A, B, and C
    // copy A and B to device memory

2. // Kernel launch code – to have the device
    // to perform the actual vector addition

3. // copy C from the device memory
    // Free device vectors
}
```

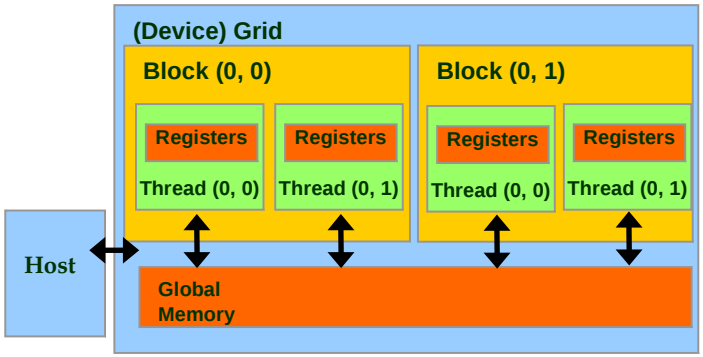


Partial Overview of CUDA Memories

Device code can:

R/W per-thread registers

R/W all-shared global memory



Host code can:

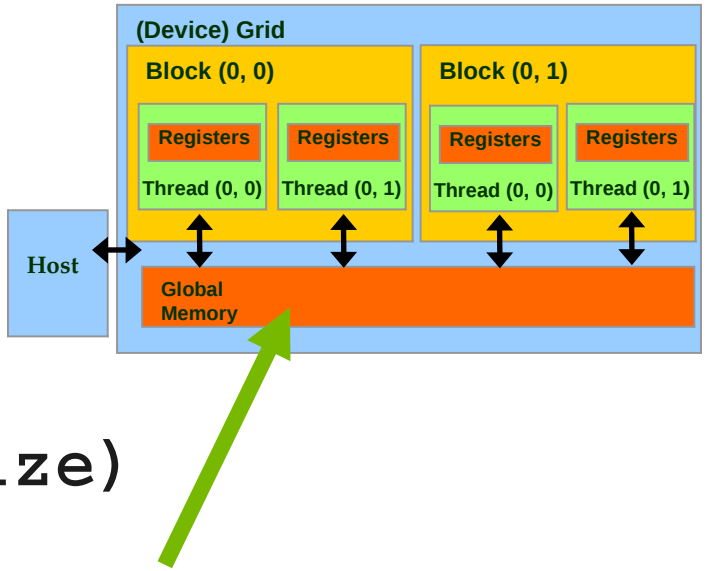
Transfer data to/from per grid global memory

CUDA Device Memory API functions

cudaMalloc()

- Allocates an object in the device global memory
- Two parameters
  - Address of a pointer to the allocated object
  - Size of allocated object in terms of bytes

```
cudaError_t cudaMalloc (void ** devPtr,
                        size_t  size)
```



CUDA Device Memory API functions

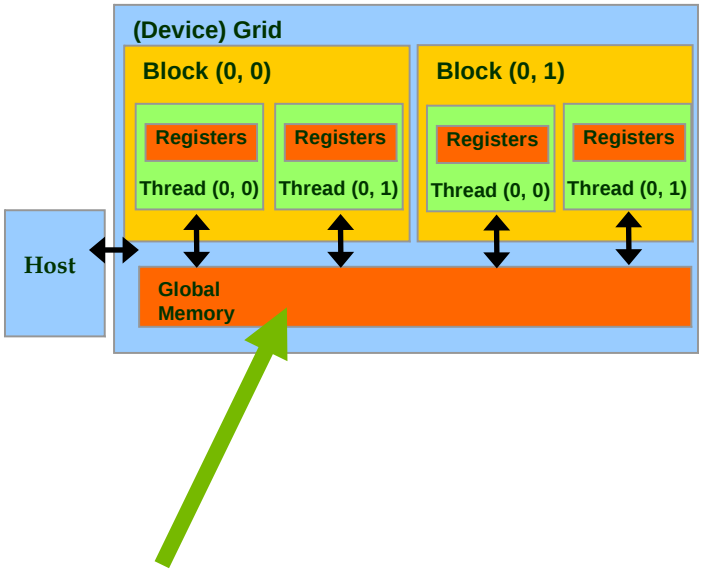
cudaFree()

Frees object from device global memory

One parameter

Pointer to freed object

```
cudaError_t cudaFree(void* devPtr)
```



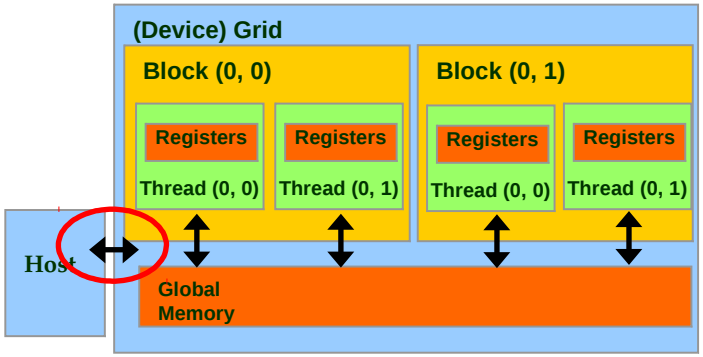
Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    //transfer data to device
     cudaMalloc((void **) &d_B, size);
    //transfer data to device
     cudaMalloc((void **) &d_C, size);

     // Kernel invocation code – to be shown later

    //get data from device
     cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

Host-Device Data Transfer API functions

cudaMemcpy()

- memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer




```
cudaError_t cudaMemcpy (void *   dst,
                        const void * src,
                        size_t    count,
                  enum cudaMemcpyKind  kind    )
```

Transfer to device is asynchronous

Host-Device Data Transfer API functions

cudaMemcpy()

kind is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice

Calling cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behavior

Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_B, size);
     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_C, size);

     // Kernel invocation code – to be shown later

     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
     cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

Error Check

```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf("%s in %s at line %d\n",   cudaGetErrorString(err), __FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```

-For brevity, we will omit this!

Kernel Function

Our “parallel” function

Given to each thread

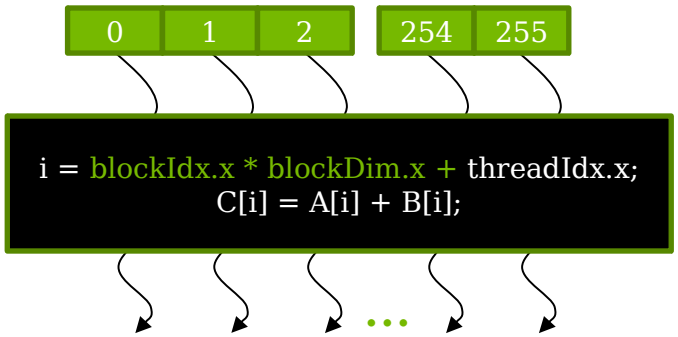
Simple implementation:

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    //Decide an index somehow
    c[index] = a[index] + b[index];
}
```

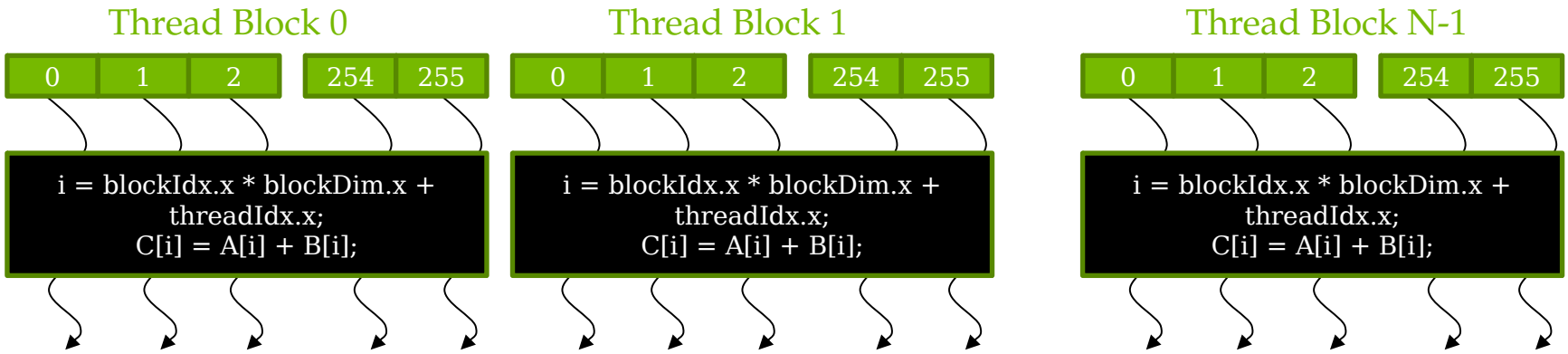
Arrays of Parallel Threads

A CUDA kernel is executed by a grid (array) of threads

- All threads in a grid run the same kernel code (Single Program Multiple Data)
- Each thread has indexes that it uses to compute memory addresses and make control decisions



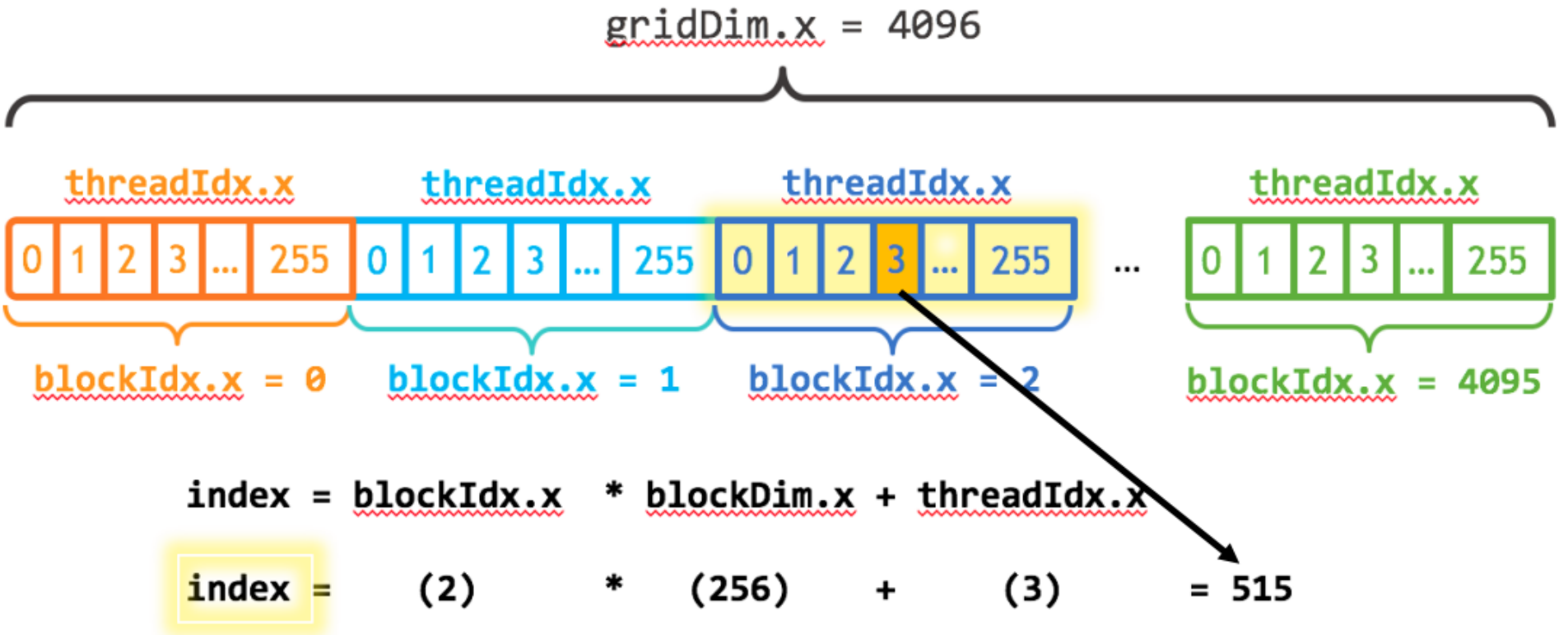
Thread Blocks: Scalable Cooperation



Divide thread array into multiple blocks

- Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
- Threads in different blocks do not interact

Kernel, Block, Thread



Vector Addition Kernel Function – Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = threadIdx.x+blockDim.x*blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

Vector Addition Kernel Function – Device Code

```
// Compute vector sum C = A + B
// each thread calculates two (adjacent) output elements of a vector addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = (threadIdx.x+blockDim.x*blockIdx.x)*2;
    if(i<n) C[i] = A[i] + B[i];
}
```

CUDA C Keywords for Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

__global__ defines a kernel function

- Each "__" consists of two underscore characters
- A kernel function must return void

__device__ and __host__ can be used together

__host__ is optional if used alone

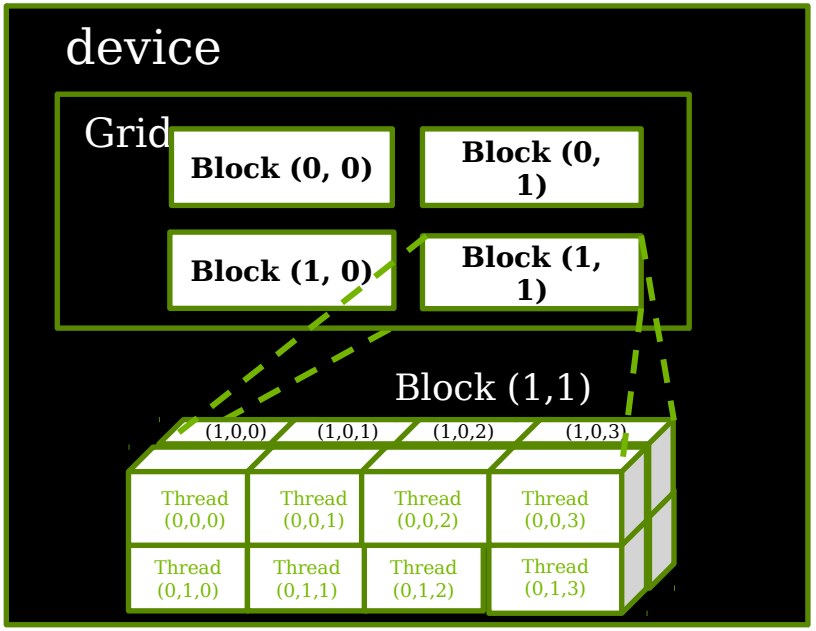
Indexing

Each thread uses indices to decide what data to work on

- blockIdx: 1D, 2D, or 3D (CUDA 4.0)
- threadIdx: 1D, 2D, or 3D

Simplifies memory addressing when processing multidimensional data

- Image processing
- Solving PDEs on volumes
- …



Vector Addition Kernel Launch - Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and copies omitted
 // Run ceil(n/256.0) blocks of 256 threads each
  vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
}
```

More on Kernel Launch – Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  dim3 DimGrid(ceil(n/256.0), 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

Vector Addition with 8000 Elements

Each thread calculates one output element

Thread block size is 1024 threads

The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements

How many threads will be in the grid?

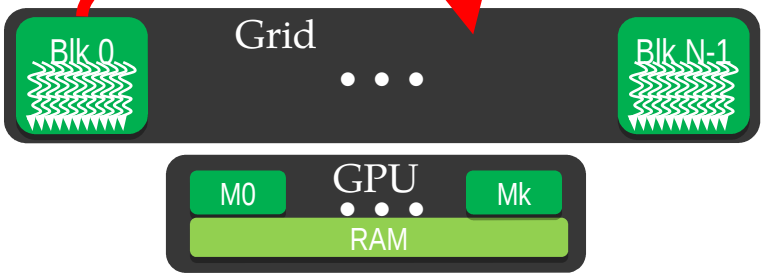
ceil(8000/1024) * 1024 = 8 * 1024 = 8192

(the minimal multiple of 1024 to cover 8000 is 1024*8 = 8192)

Kernel Execution

```
__host__
void vecAdd(…)
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);
vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B,d_C,n);
}
```

```
__global__
void vecAddKernel(float *A,
     float *B, float *C, int n)
{
   int i = blockIdx.x * blockDim.x
     + threadIdx.x;

   if( i<n ) C[i] = A[i]+B[i];
}
```



Important CUDA Syntax Extensions

Declaration specifiers

__global__ void foo(...); // kernel entry point (runs on GPU)

Syntax for kernel launch

foo<<<500, 128>>>(...); // 500 thread blocks, 128 threads each

Built in variables for thread identification

dim3 threadIdx; dim3 blockIdx; dim3 blockDim;

Example: Original C Code

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
int main()
{
    // omitted: allocate and initialize memory
     saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY kernel
   // omitted: using result
}
```

CUDA Code

```
__global__ void saxpy(int n, float a, float *x, float *y) {
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        if(i<n) y[i]=a*x[i]+y[i];
}
int main() {
        // omitted: allocate and initialize memory
        int nblocks = (n + 255) / 256;
        cudaMalloc((void**) &d_x, n);
        cudaMalloc((void**) &d_y, n);
        cudaMemcpy(d_x,h_x,n*sizeof(float),cudaMemcpyHostToDevice);
        cudaMemcpy(d_y,h_y,n*sizeof(float),cudaMemcpyHostToDevice);
        saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
        cudaMemcpy(h_y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
        // omitted: using result
}
```