CUDA

- CUDA is the most popular programming model for writing parallel programs to run on GPU
- developed by NVIDIA

CUDA keywords and kernel

- A CUDA program has two parts of code
  - Host code: the part of code run on CPU
  - Device code: the part of code run on GPU
- The functions that will be run on the GPU device are marked with CUDA keywords
- A function that is run on GPU is called a kernel function
-

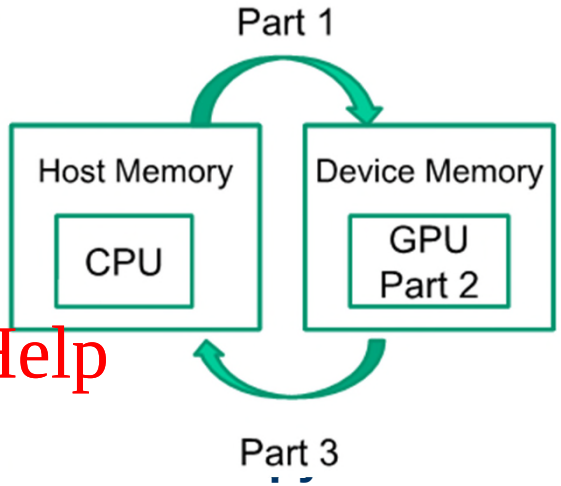
Keywords in function declaration

- __global__
  - A global function is a kernel function to be executed in GPU
  - can only be called from the host code
- __host__
  - A host function (e.g., traditional C function) executes on the host
  - can only be called from another host function
  - By default, all functions are host functions if they do not have any CUDA keywords

An Example of CPU Code

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n
{
    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 each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

Outline of a vecAdd() function for GPU




```
#include <cuda.h>
…
void vecAdd(float* A, float* B, float* C, int n)
{
…
```

Part 1: Allocate device memory for A, B, a
from host memory to device memory

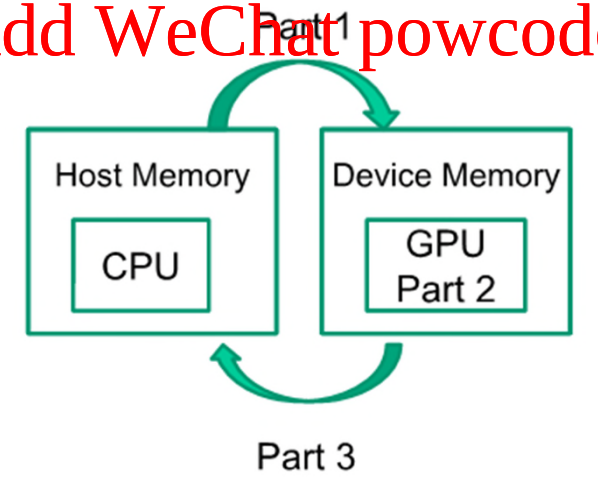
Part 2: Launch kernel code to perform the actual operation on GPU

Part 3: Copy the result C from device memory to host memory; Free device vectors

```
}
```

Part 1 and Part 3: dealing with GPU memory

- Allocate GPU memory
- Copy the data from CPU memory to GPU memory
- Copy the result in GPU memory back to CPU memory



Memory Management in GPU

- **cudaMalloc(void ** *devPtr*, size_t *size*)**
  - **Allocate the device global memory**
  - **Two parameters**
    - **devPtr: a pointer to the address of the allocate memory**
    - **size: Size of allocated memory**

Memory Management in GPU

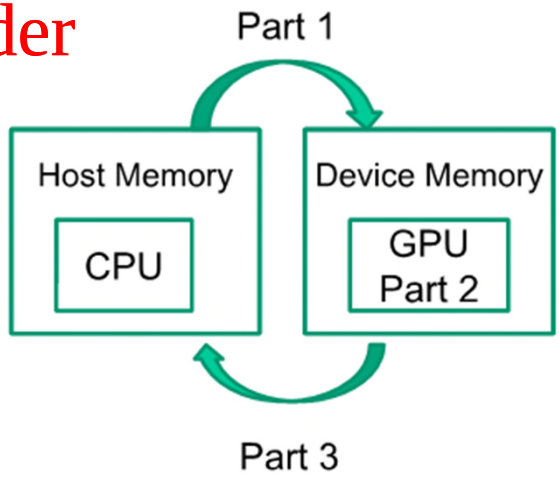
- cudaMemcpy(dst, src, count, kind)
  - Memory data transfer
  - Four parameters
    - 1. destination location of the data to be copied
    - 2. source location of the data
    - 3. size of the data
    - 4. The types of memory copying: host to host, host to device, device to device, device to host

vecAdd Function

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size=n*sizeof(float);
    float *dA, *dB, *dC;

    cudaMalloc(&dA, size);

    cudaMemcpy(dA, A, size, cudaMemcpyHostToDevice);

    cudaMalloc(&dB, size);

    cudaMemcpy(dB, B, size, cudaMemcpyHostToDevice);

    cudaMalloc(&dC, size);


    Part 2: Launch Kernel code to perform the actual operation on
            GPU


    cudaMemcpy(C, dC, size, cudaMemcpyDeviceToHost);
    cudaFree(dA); cudaFree(dB); cudaFree(dC);
}
```

Part 2: Launch and Run the Kernel Code

- Launch and execute the Kernel function
- Various related issues in Part 2
  - Execution model of the kernel function
  - Thread structure
  - Execution configuration
  - Kernel execution



Part 2: Launch and Run the Kernel Function

- **Various related issues in Part 2**
  - **Execution model of the kernel function**
  - **Thread structure**
  - **Execution configuration**
  - **Kernel execution**

Execution Model of GPU

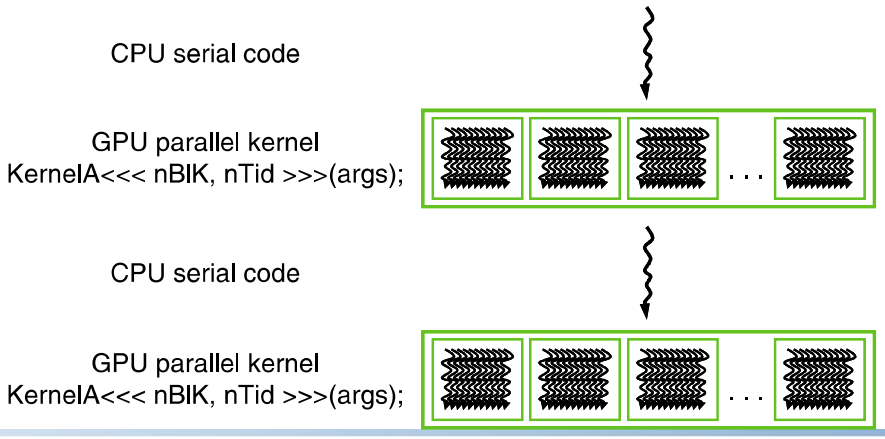
- The execution starts with host (CPU) execution

- When a kernel function is called, it is executed by a large number of threads on the GPU

  - All the threads to run a kernel are collectively called a *grid*

- When all threads of a kernel complete their execution, the corresponding grid terminates

- The execution continues on the host until another kernel is called

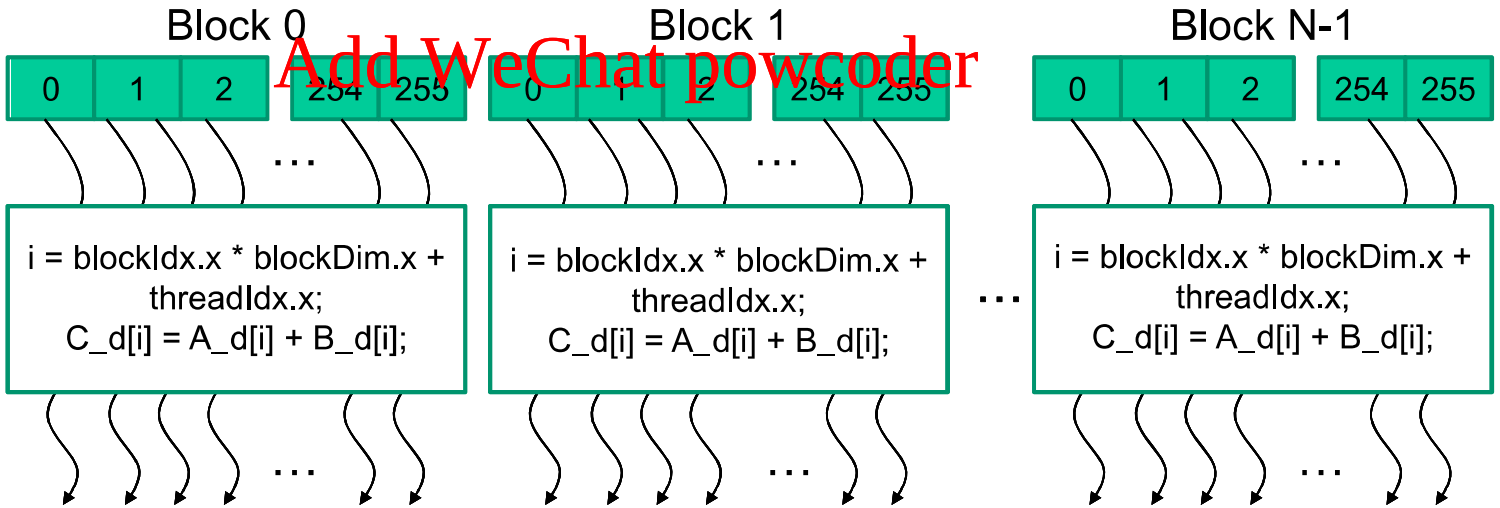


GPU code – Part 2

- Launch and execute the Kernel function
- Various related issues in Part 2
  - Execution model of the kernel function
  - Thread structure
  - Execution configuration
  - Kernel execution

Thread Structure for Running a Kernel

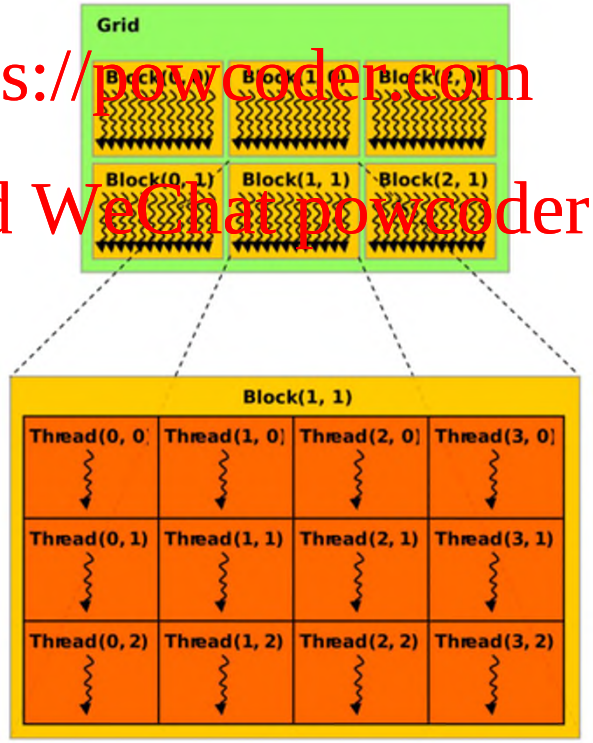
- When a host code launches a kernel, CUDA generates a grid of thread blocks
  - Each block contains the same number of threads (up to 1024)
  - Each thread runs the same kernel function



Thread Organization

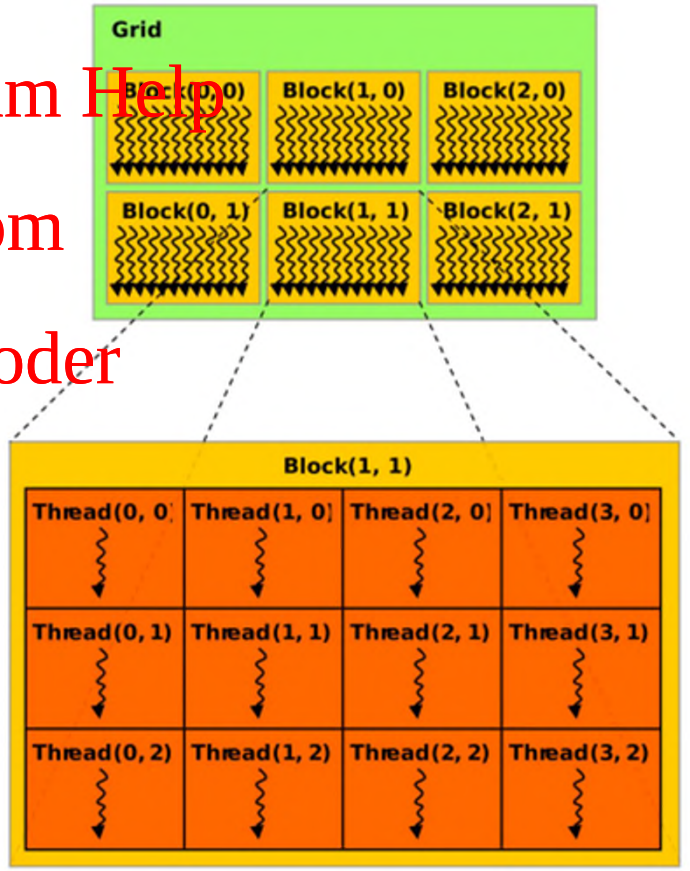
Threads are organized into a grid of blocks (Two-level architecture)

The grid and blocks can be multidimensional



Thread Organization

- gridDim(x, y, z): the dimensions of the grid,
- blockDim(x, y, z): the dimensions of the block,
- blockIdx(x, y, z):
  - the coordinate (ID) of the block in the grid,
  - it can be accessed by the calling thread to obtain which block it is in
- threadIdx(x, y, z):
  - the local coordinate (ID) of a thread in a block,
  - It can be accessed by the calling thread to obtain its local position in the block



Build-in variables

- gridDim: the dimensions of the grid
- blockDim: the dimensions of the block
- blockIdx: the block index within the grid
  - All the threads in a block share the same *blockIdx* value
- threadIdx: the thread index within the block
- Their values are preinitialized by the CUDA runtime library when invoking the kernel function
- Can be accessed in the kernel function

GPU code – Part 2

- Launch and execute the Kernel function
- Various related issues in Part 2
  - Execution model of the kernel function
  - Thread structure
  - Execution configuration
  - Kernel execution

Execution configuration of kernel launch

- ***Execution configuration*** **sets the grid and block size**
  - **Set between the <<< and >>> before the C function parameters**
  - **First parameter defines grid size: the number of thread blocks in the grid**
  - **The second specifies the block size: the number of threads in each block**


```
int vectAdd(float* A, float* B, float* C, int n)
{
//  d_A, d_B, d_C allocations and copies omitted
// Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

The same kernel can be launched with different execution configurations

Execution Configuration

- Execution configuration sets
  - Grid and block are multidimensional
  - Execution configuration sets the grid and block dimensions
  - Dimensions values are stored in the built-in variables gridDim and blockDim

Example: dim3 a(3, 2, 4); dim3 b(128, 1, 1);

vecAdd <<<a, b>>>(...); Then,

- gridDim.x=3, gridDim.y=2, gridDim.z=4
- blockDim.x=128, blockDim.y=1, blockDim.z=1

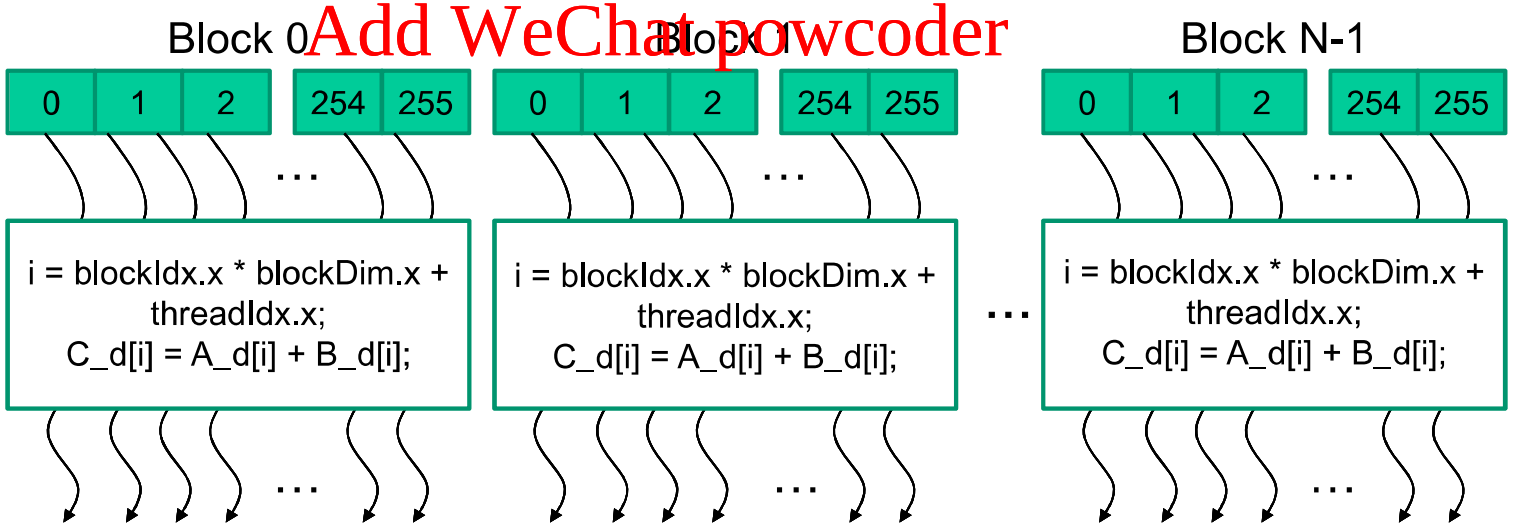
- Question: how many threads will be generated?
- Answer: 3*2*4*128

GPU code – Part 2

- Launch and execute the Kernel function
- Various related issues in Part 2
  - Execution model of the kernel function
  - Thread structure
  - Execution configuration
  - **Kernel execution**

Kernel execution

- Different threads process different parts of data in the kernel code
- We need to match different threads to different parts of the data

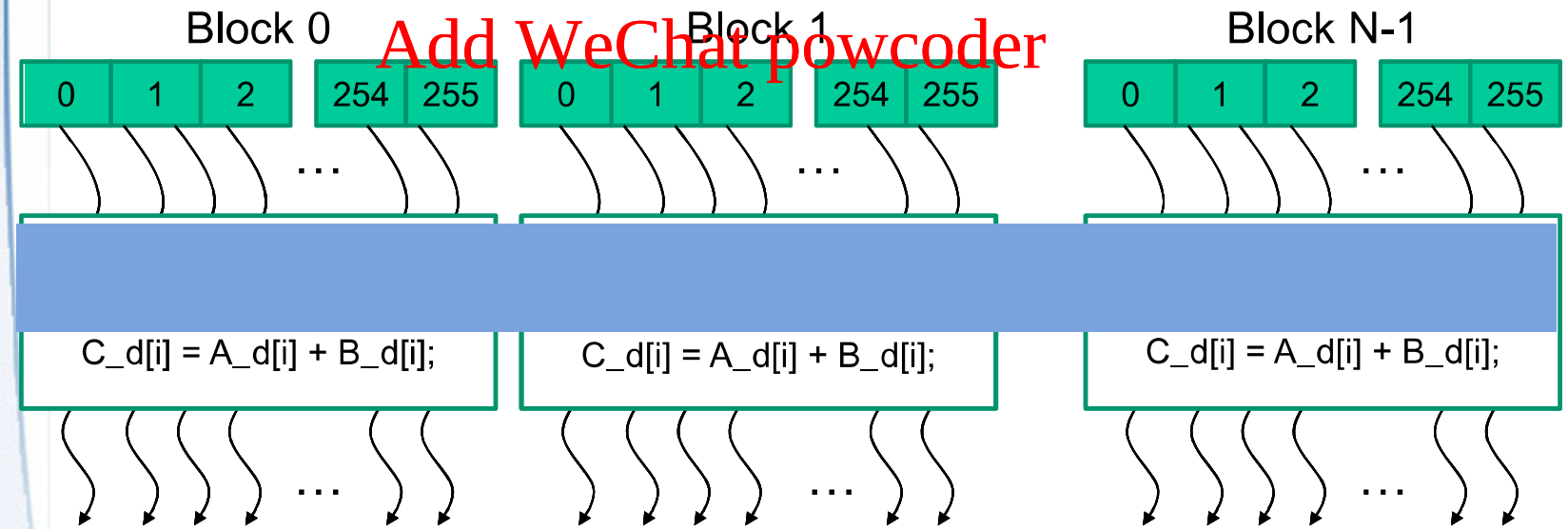


Match threads to data items

- Assume the following grid of blocks are generated to compute C_d=A_d+B_d

Griddim(x, y, z)=(N, 1, 1) ,blockdim(x, y, z)=(256, 1, 1), blockidx(x, 0, 0), threadidx(x, 0, 0)

- Question: how to match a thread (threadidx) to compute A_d[i]+B_d[i]?

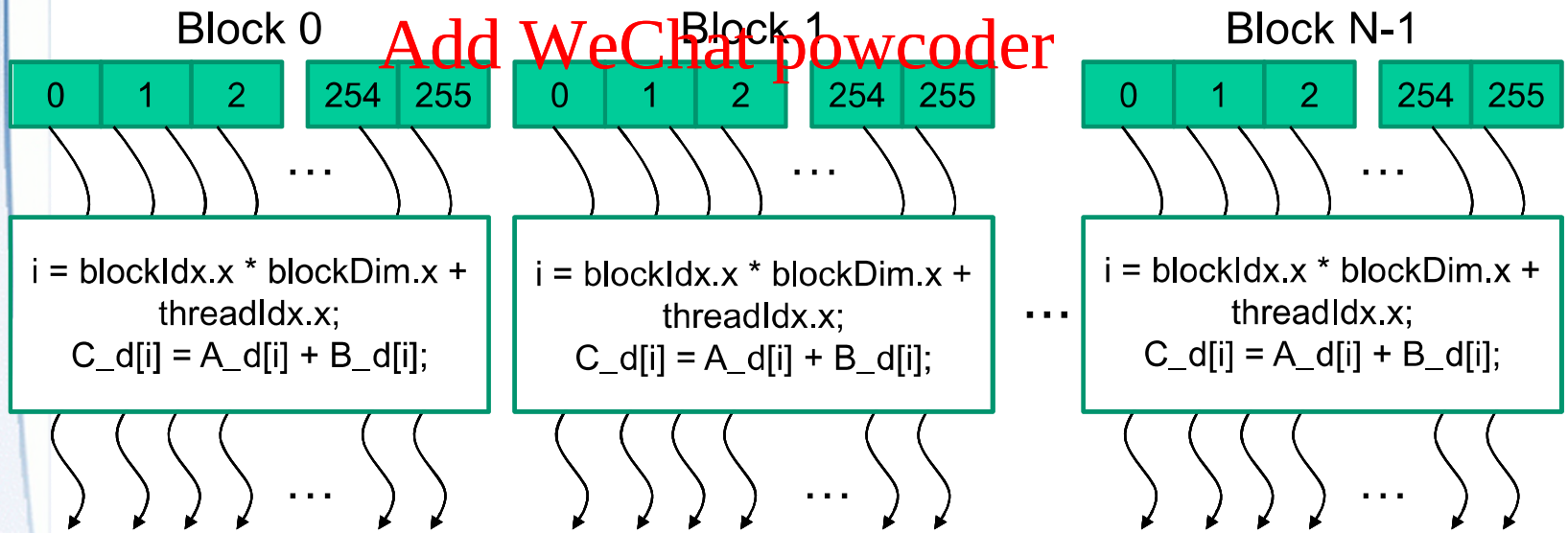


Match threads to data items

- Assume the following grid of blocks are generated to compute C_d=A_d+B_d

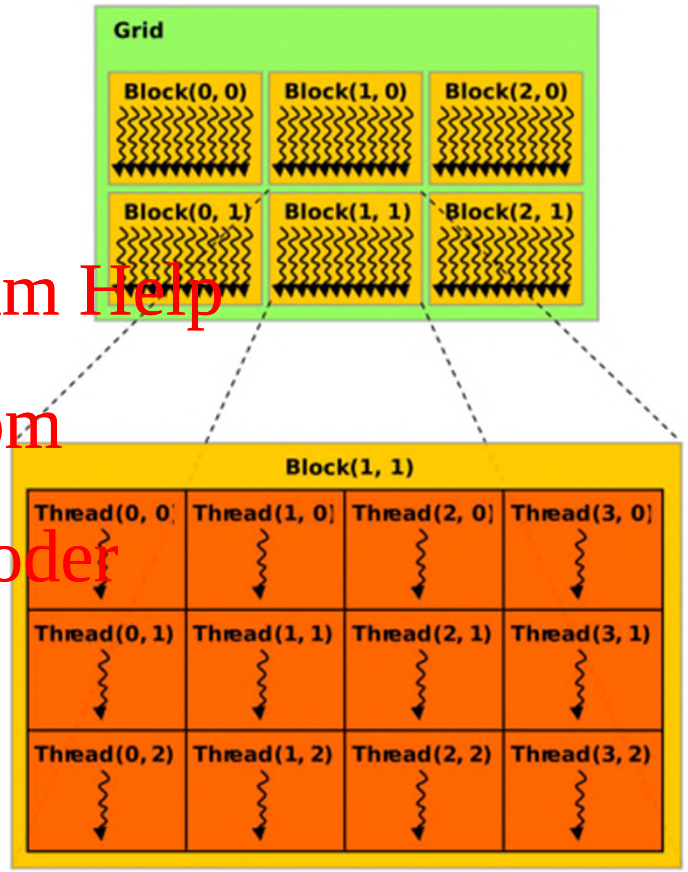
Griddim(x, y, z)=(N, 1, 1) ,blockdim(x, y, z)=(256, 1, 1), blockidx(x, 0, 0), threadidx(x, 0, 0)

- Question: how to match a thread (threadidx) to compute A_d[i]+B_d[i]?



Exercise

- Calculate C=A+B; A, B, C are 6*12 matrices
- Assume a grid of blocks on the right are generated:
  griddim(2, 3), blockdim(3, 4)
- Question: How to match a thread to calculate C[i]=A[i]+B[i]?
- Answer: calculate the global index of a thread in the grid
- X=blockidx.x*blockdim.x+threadidx.x
- Y=blockidx.y*blockdim.y+threadidx.y



Kernel Execution for vecAdd

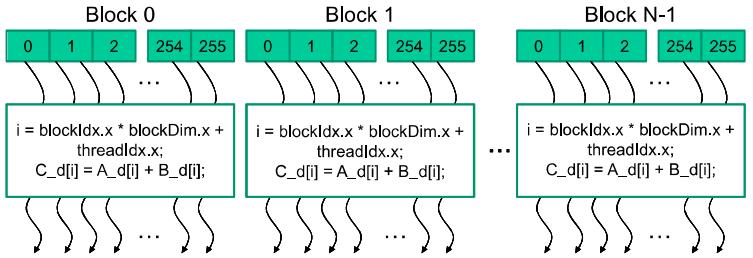
- All threads in a grid execute the same kernel function

- The threads use their coordinates (i.e., blockidx and threadidx) to

  - distinguish themselves from each other

  - identify the appropriate part of the data to process


```
__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];
}
```



Local Variables in a Kernel Function

- Local (automatic) variable in the kernel function are private to each thread
  - Each thread has a local copy of the variable


```
__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];
}
```

If statement

```
__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];
}
```


- Only first *n* threads perform the addition
  - Because not all vector lengths can be expressed as multiples of the block size
- Allows the kernel to process vectors of any lengths

Comparison between CPU and GPU version

- There is a “for” loop in the CPU version
- In the GPU version, the grid of threads is equivalent to the loop
- GPU version


```
__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];
}
```

- CPU version

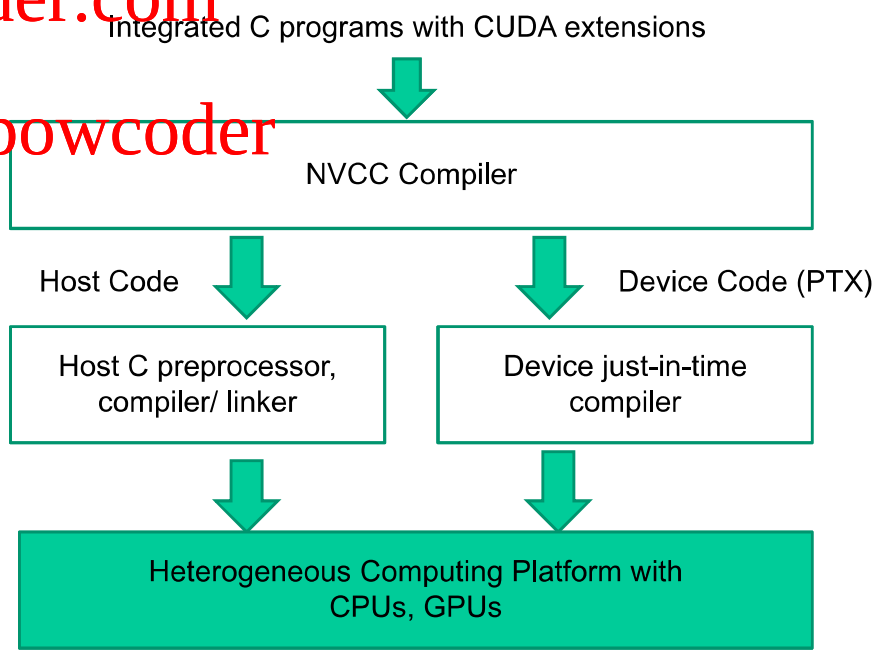
```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}
```

The complete program of vecAdd

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size=n*sizeof(float);
    float *dA, *dB, *dC;

    cudaMalloc(&dA, size); //Part 1

    cudaMemcpy(dA, A, size, cudaMemcpyHostToDevice);

    cudaMalloc(&dB, size);

    cudaMemcpy(dB, B, size, cudaMemcpyHostToDevice);

    cudaMalloc(&dC, size);


    vecAddKernel<<<ceil(n/256.0), 256>>>(dA, dB, dC, n); //Part 2


    cudaMemcpy(C, dC, size, cudaMemcpyDeviceToHost); //Part 3
    cudaFree(dA); cudaFree(dB); cudaFree(dC);
}
```

Compilation Process of a CUDA Program

- NVCC compiler uses the CUDA keywords to separate the host code and device code
- The host code is further compiled with standard C compiler and run as a CPU process
- A device code is first compiled by NVCC to PTX code
- The PTX code is further compiled by NVCC to executable



Timing the GPU code

- Using Events for timing on GPU
- Events are special kernels that can be invoked for timing on GPU

Timing the GPU code

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
vecAddKernel<<<ceil(n/256.0), 256>>>(dA,
dB, dC, n);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```


- cudaEventRecord() is used to place the start and stop events into the execution of kernel
- The GPU will record a timestamp for the event when the Kernel function reaches the event