COSC 407

Intro to Parallel Computing

Topic 13: CUDA Threads

1

Outline

Previous pre-recorded lecture (Students' led Q/As):

- CUDA basics: program structure
- Useful Built-in CUDA functions
- Function Declarations (global, device, host)

Today:

- Error Handling, cudaDeviceSynchronize
- Hardware architecture: sp → SM → GPU
- Thread Organization: threads → blocks → grids
  - Dimension variables (blockDim, gridDim)
- Thread Life Cycle From the HW Perspective
- Kernel Launch Configuration: 1D grids/blocks

Next Lecture:

- Kernel Launch Configuration: nD grids/blocks
- CUDA limits
- Thread Cooperation
- Running Example: Matrix Multiplication

2

Memory

CPU and GPU have separate memory spaces

- Need to move data to device (GPU) if it is processed there
- Need to move results back to CPU memory
- Functions: `cudaMalloc, cudaFree, cudaMemcpy`

Pointers

- Hold memory addresses in either CPU or GPU memory
- Can't differentiate CPU pointers from GPU pointers by just checking their values.
  - There, you must use pointers in their appropriate locations.
    - Dereferencing CPU pointer in kernel will likely crash
    - Dereferencing GPU pointer host code will likely crash

3

Error Handling

CUDA has two sources of errors :

(1) Errors from CUDA API

E.g. cannot allocate memory space on the device

(2) Errors from CUDA Kernel

i.e. errors that happen inside your kernel code.

4

Handling CUDA API Errors

- CUDA API functions return an error code of type **cudaError_t**
- For example:
  - **cudaSuccess** (=0, if no problems)
  - **cudaErrorMemoryAllocation** (=2, if cannot allocate memory)
  - Other error codes (positive values) are possible
    - see here for the full list.

Such errors should be handled using some extra code. For example:

```
cudaError_t err = cudaMalloc(&d_a, num_bytes);
if (err != cudaSuccess) {
        printf("Can't allocate CUDA Memory");
        ...//more code to handle error
} else {...}
```


Handling CUDA API Errors

A better IDEA: to avoid repeatedly writing if statements after each CUDA call, you can define and use a macro as following:

```
#define CHK(call) {                                         \
    cudaError_t err = call;                                 \
    if (err != cudaSuccess) {                               \
      printf("Error%d: %s:%d\n",err,__FILE__,__LINE__);     \
      printf(cudaGetErrorString(err));                      \
      cudaDeviceReset();                                    \
      exit(1);                                              \
    }
}
```

destroys and clean up all resources associated with the current device in the current process immediately

Then use this macro whenever you call a CUDA API function

```
CHK( cudaMalloc(&d_a, num_bytes) );
```

Handling CUDA Kernel Errors

- The other type of errors is the one that happens during the execution of YOUR kernel function
- You can check for this error as follows

```
Kernel<<<..,..>>>();        //call kernel
CHK(cudaGetLastError());          //1
CHK(cudaDeviceSynchronize());     //2
```

- Statement #1 will **check for kernel launch errors**
    - e.g. too many threads per block
  - CUDA runtime maintains an error variable that is overwritten each time an error occurs. **`cudaGetLastError()`** returns the value of this variable and resets the variable to **`cudaSuccess`**.
- Statement #2 will block the host until GPU is done
  - Any asynchronous error is returned by (`cudaDeviceSynchronize`)

More details: https://devblogs.nvidia.com/how-query-device-properties-and-handle-errors-cuda-cc/

7

cudaDeviceSynchronize()

- CUDA functions **and** host code are ***asynchronous***
  - i.e. they return control to the calling CPU thread before they finish their work
- `cudaDeviceSynchronize()` can be used to block the calling CPU thread until all CUDA calls made by this thread are finished
- Example use: time your kernel
  - (must include time.h and cuda lib)

```
//on host
double t = clock();
Kernel<<<..,..>>>();
cudaDeviceSynchronize()
t = (clock()-t)/CLOCKS_PER_SEC;
```

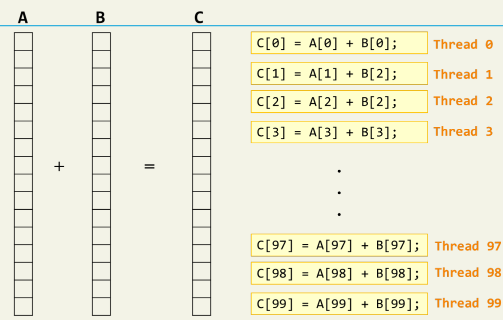
Synchronization is expensive, so don't overuse it!

8

Adding Vectors: Revisited

- You saw before that we usually assign a thread to process each element in an array.
- Assigning threads to *vector* elements was easy

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



9

Typical GPU Program

Host code (running on CPU)

1. Allocate space on GPU
2. Copy CPU data to GPU
3. Launch kernel function(s) on GPU

   define launch-configuration before that.

4. Copy results from GPU to CPU
5. Free GPU memory

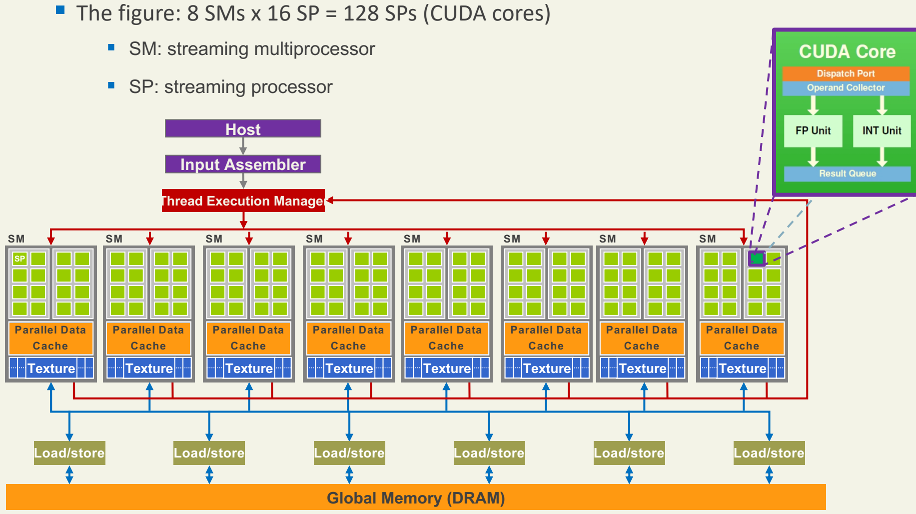
Kernel code (running on GPU)

- Write kernel function as **if it will run on a single thread**
  - Use IDs to identify which **piece of data** is processed by this thread
    - Remember that this SAME kernel function is executed by many threads
- Parallelism of threads is expressed in the host code

10

GPU Design

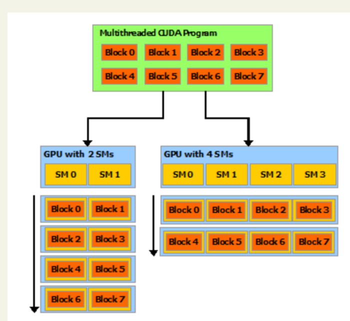
- Massively threaded, sustains 1000s of threads per app
- The figure: 8 SMs x 16 SP = 128 SPs (CUDA cores)
  - SM: streaming multiprocessor
  - SP: streaming processor



11

GPU Design

- A scalable array of multithreaded *Streaming Multiprocessors* (*SMs*)
- A multithreaded program is partitioned into blocks of threads that execute independently from each other
  - GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.



https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

12

GPU Design

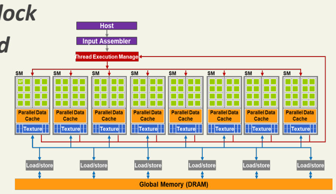
- When host invokes a kernel grid
  - Blocks of the grid are enumerated
  - Distributed to multiprocessors with available execution capacity
- The threads of a thread block execute concurrently on one multiprocessor
  - Multiple thread blocks can execute concurrently on one multiprocessor
- As thread blocks terminate
  - New blocks are launched on the vacated multiprocessors
- Designed to execute hundreds of threads concurrently. To manage such a large amount of threads (*SIMT*)

13

Threads Organization: Basics

On SOFTWARE (code) side:

- Threads are grouped into Blocks
  - All threads in a block execute the same kernel program (*SPMD*)
- Blocks are grouped into a Grid
- **IDs**:
  - Each ***thread*** has a unique ID *within a* ***block***
  - Each ***block*** has a unique ID *within a* ***grid***



On HARDWARE side:

- Each **block** runs on one **SM.**
  - An **SM** might run **more than one block**
- Each thread runs on an SP (within an SM)
  - An SP can only run one thread at any time
  - Might run many successive threads.
- *More about this later*

14

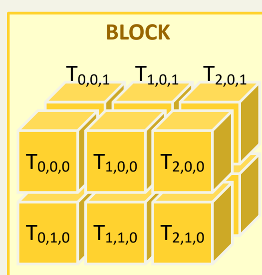
Device Properties

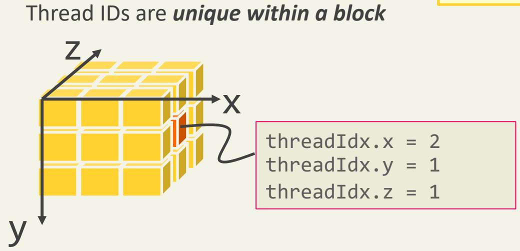
```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

int main()
{
//just to check
cudaDeviceProp prop;
int count;
cudaGetDeviceCount(&count);
for (int i = 0; i < count; i++)
{
    cudaGetDeviceProperties(&prop, i);
    ...
    //examine members of the strcut
```

```
----- General Information for device 0 ---
Name:   Tesla K80
Compute capability:     3.7
Clock rate:     823500
Device copy overlap:    Enabled
Kernel execution timeout: Disabled
----- Memory Information for device 0 ---
Total global mem:       11996954624
Total constant Mem:     65536
Max mem pitch:  2147483647
Texture Alignment:      512
----- MP Information for device 0 ---
Multiprocessor count:   13
Shared mem per mp:      49152
Registers per mp:       65536
Threads in warp:        32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:    (2147483647, 65535, 65535)
```


Threads in a Block

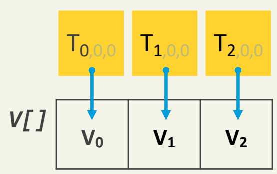
- Threads in a block are organized in **1D**, **2D**, or **3D** array of threads.
- Built-in **ID variables**
  - `threadIdx.x`
  - `threadIdx.y`
  - `threadIdx.z`
- Thread IDs are ***unique within a block***



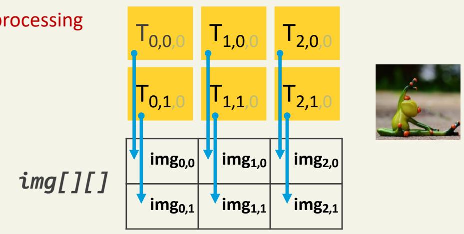


Why 1/2/3-D Organization?

Simplifies memory addressing when processing **multidimensional data**

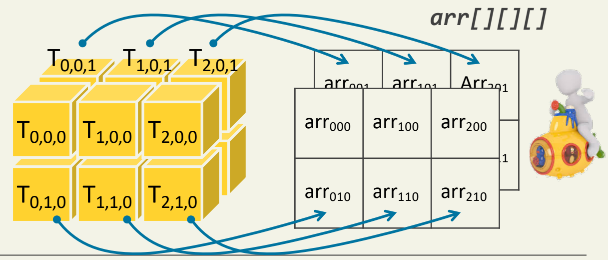


1D threads are most suitable for processing **vectors**



2D threads are most suitable for **2D arrays** (e.g. images)

3D threads are most suitable for **3D arrays** (e.g. 3D environments)



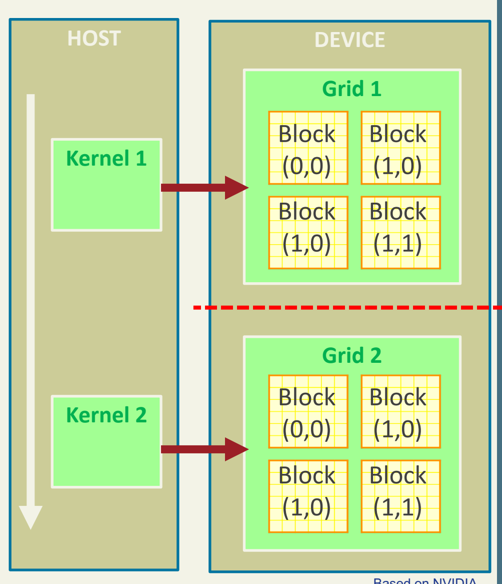
17

Blocks in a Grid

- Kernel code (on host) may initiate one or more blocks, each with many threads.

```
__global__ kernel1(..) {..}
...
kernel1<<<gridSize,blockSize>>> (..);
...
```

- **All blocks** for a given kernel belong to a **grid**
- All blocks in a grid ***must finish before the next kernel runs***.
  - *Synchronization point!*
- Remember that each block runs on one SM

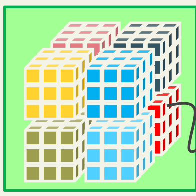


Based on NVIDIA

18

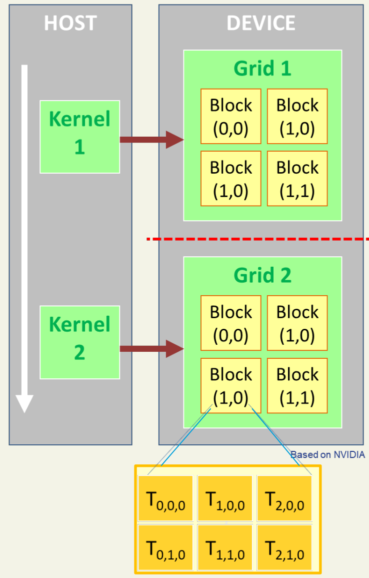
Blocks in a Grid

- Blocks in a grid are organized in 1D, 2D, or 3D array of blocks.
- Built-in ID variables
  `blockIdx.x`
  `blockIdx.y`
  `blockIdx.z`



```
blockIdx.x = 1
blockIdx.y = 1
blockIdx.z = 1
```

- Block IDs are ***unique within a grid***



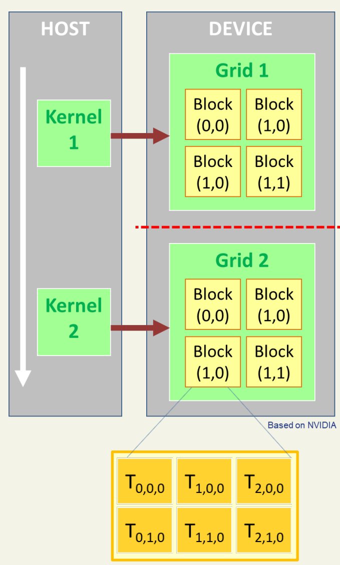
19

Dimension Variables

A dimension variable holds the number of elements over this dimension

- Dimensions may be unique for each grid and are set at launch time
  – cannot change a kernel's dimensions once it is launched.
- Built-in ***dimension*** variables
  `blockDim.x, blockDim.y`
  `blockDim.z`
  `gridDim.x, gridDim.y,`
  `gridDim.z`

  – E.g. Grid 2 in the figure:
    gridDim.x = 2 blockDim.x = 3

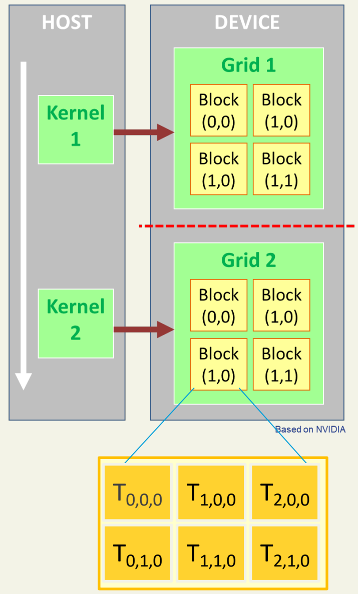


20

Thread Life Cycle in HW

Incomplete version

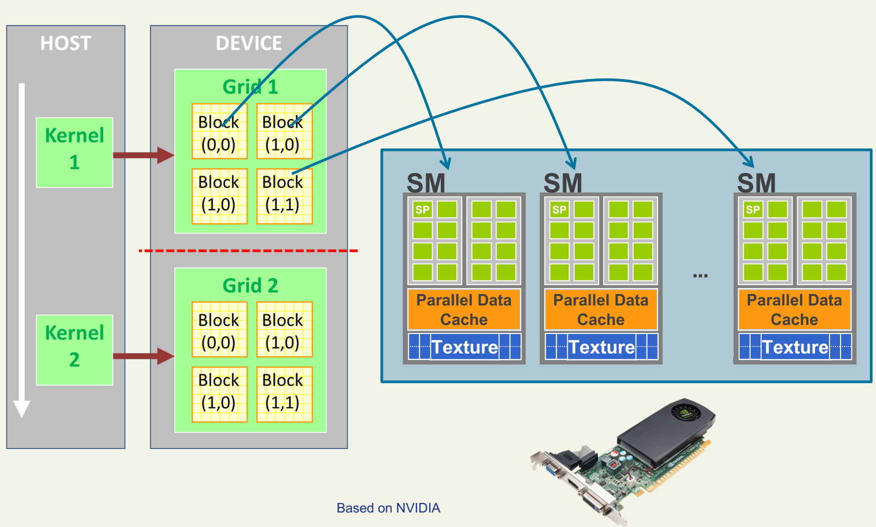
1. Grid is launched
   kernelFoo<<<gridSize,blockSize>>> (..);
2. *Blocks* are distributed to *SM*
   – Potentially more than one *Block* per *SM*
3. Each SM launches the threads in its block.
   – One thread per core (SP)
4. As Blocks complete, resources are freed.



Note: this is not the complete lifecycle. We are still missing the "warps" which are discussed later.

Thread Life Cycle in HW

Incomplete version



Based on NVIDIA

Kernel Launch Configuration

- From the Vector Addition Example

```
vectorAdd<<<1, N>>>(...);
```

  - Statement tells the GPU to launch **N** threads on **1** block

- The general format:

```
kernelFunc<<<gridSize, BlockSize>>>
```

- You can:
  - Run **as many blocks** at once (all belong to the same grid)
  - **Each block can have a maximum** of:
    - **1024** threads on newer GPUs.
    - **512** threads on older GPUs
- We **cannot** specify which block runs before which.

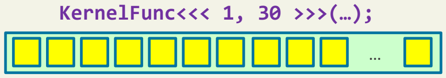
23

Kernel Launch Configuration

- You should choose the breakdown of threads and blocks that **make sense to your problem**.

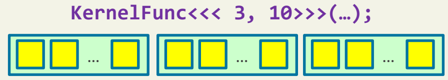
Example: For a vector, choose 1D setup with options

```
KernelFunc<<< 1, 30 >>>(…);
```



30 threads: 1 Block with 30 threads

```
KernelFunc<<< 3, 10>>>(…);
```



30 threads: 3 Blocks, each with 10 threads

- **Dimensionality of above example**: 1D blocks and 1D grid
- **x-dimension** is used by default for 1D items
  - Can define ***higher dimensionality*** using `dim3`. (more about this later)

Remember, **each block is assigned to one SM. If you want to fully use the GPU, then #blocks should be ≥ # of SMs**

24

Vector Addition: Revisited

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

int main() {
 int *a, *b, *c, *d_A, *d_B, *d_C;
 //...allocate space on CPU and GPU
 //...initialize a,b
 //...copy a,b to GPU at d_A, d_B
 //launch the kernel
 vectorAdd <<<1,N>>> (d_A, d_B, d_C, N);
 //...results back from d_C to c
 //...free up memory
 return 0;
}
```

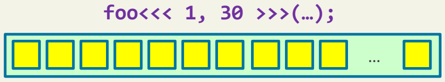
Only x-dim is used

This means 1 grid with 1 block running on 1 SM, and N threads organized in 1D array (over the x-dimension only)

25

Computing Array Index for Each Thread

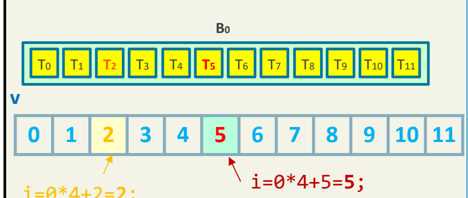
`foo<<< 1, 30 >>>(…);`



1 Block with 30 threads

```
void foo(...){
    int i = blockIdx.x * blockDim.x
    ...         + threadIdx.x
}
```

Example: computing *i* for v[2], v[5]

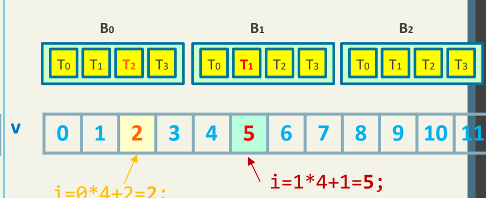


`foo<<< 3, 10>>>(…);`

3 Blocks, each with 10 threads

```
void foo(...){
    int i = blockIdx.x * blockDim.x
    ...         + threadIdx.x
}
```

Example: computing *i* for v[2], v[5]



29

Computing Array Index for Each Thread

- The general formula to compute the thread index:

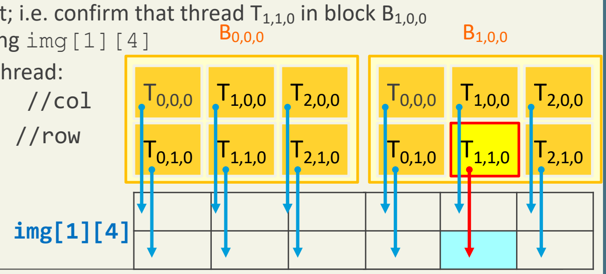
```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
int z = blockIdx.z * blockDim.z + threadIdx.z;
```

Use above formulas *in the kernel function* to identify which data element is accessed by each thread.

- **Example**: using the formulas above, compute the (x,y) of the highlighted element; i.e. confirm that thread $T_{1,1,0}$ in block $B_{1,0,0}$ will be only accessing `img[1][4]`

Answer: For this thread:

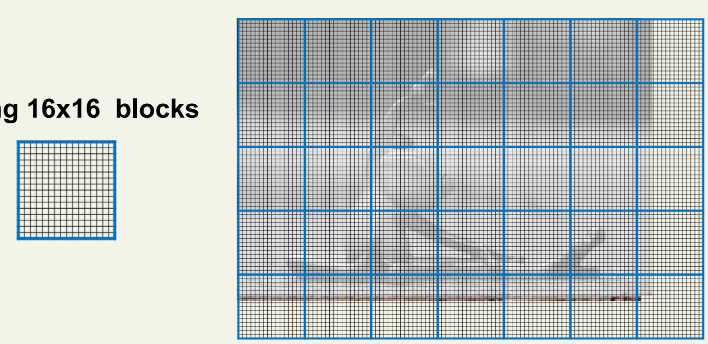
```
x = 1*3+1 = 4  //col
y = 0*2+1 = 1 //row
```



Processing 100x70 Picture

```
__global__ void PicKrnl(float* d_Pin,float* d_Pout,int w,int h){
// Calculate row # of the d_Pin and d_Pout element to process
int y = blockIdx.y * blockDim.y + threadIdx.y;
// Calculate column # of the d_Pin and d_Pout element to process
int x = blockIdx.x * blockDim.x + threadIdx.x;
// each thread computes one element of d_Pout if in range
if((y<h)&&(x<w)) d_Pout[y * w + x] = f(d_Pin[y * w + x]);
```

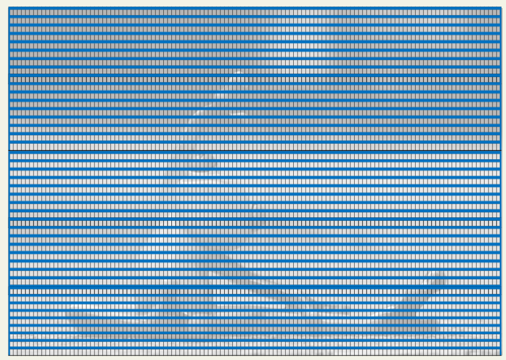
Using 16x16 blocks



Processing 100x70 Picture

```
__global__ void PicKrnl(float* d_Pin,float* d_Pout,int w,int h){
  // Calculate row # of the d_Pin and d_Pout element to process
  int y = blockIdx.y;
  // Calculate col # of the d_Pin and d_Pout element to process
  int x = threadIdx.x;
  // each thread computes one element of d_Pout if in range
  if((y<h)&&(x<w)) d_Pout[y * w + x] = ƒ(d_Pin[y * w + x]);
```

Using 100x1x1 blocks



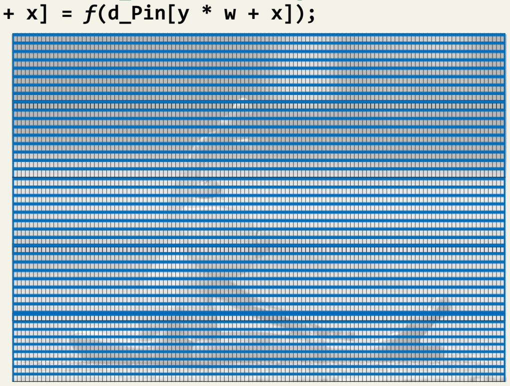
Only use red code if you are sure you have a 1D block.

32

Processing 100x70 Picture

```
__global__ void PicKrnl(float* d_Pin,float* d_Pout,int w,int h){
  // Calculate row # of the d_Pin and d_Pout element to process
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  // Calculate col # of the d_Pin and d_Pout element to process
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  // each thread computes one element of d_Pout if in range
  if((y<h)&&(x<w)) d_Pout[y * w + x] = ƒ(d_Pin[y * w + x]);
```

Using 100x1x1 blocks



Using the General Formula gives same output (and it's preferred)

Try it: since is a 1-row block, put *threadIdx.y=0 , blockDim.y = 1, blockIdx.x = 0* and the code will be *the same as the previous slide*, **except that** it also works if you decide to change dimensionality

33

Choosing Launch Config

(A) 1D Grids / Blocks

- Assume you know total number of **threads (N)** needed
  - Should be equal to the number of data elements
- How do you determine the launch configuration?

```
KernelFunc<<< ???, ??? >>>
```

Steps:

1. Choose the number **threads per block (nthreads)**.
2. Compute the **number of blocks** as follows:

```
nblocks = (N-1)/nthreads + 1
```

Note (again): if you want to fully use the GPU, then

- #threads per block should be large (≥ #SPs per SM)
- #blocks should be ≥ # of SMs

Launch Configuration Examples

Assume we choose `nthreads` = 256 (i.e. #threads per block)

- # of array elements ***N*** = 200
  - `nblocks` = 199/256 + 1 = 1 → total # threads = 256
- # of array elements ***N*** = 256
  - `nblocks` = 255 / 256 + 1 = 1 → total # threads = 256
- # of array elements ***N*** = 400
  - `nblocks` = 399 / 256 + 1 = 2 → total # threads = 512

Note: *use `if(i<n)` to discard the extra threads*

Example: Vector Addition

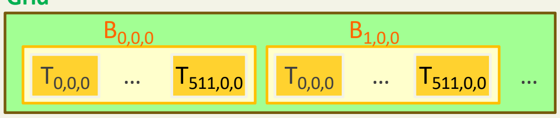
```
__global__ void vectorAdd(int* a, int* b, int* c, int n) {
     int i = blockIdx.x * blockDim.x + threadIdx.x;
     if(i<n)  c[i] = a[i] + b[i];
}
```

Now you know why

```
int main() {
 int *a, *b, *c, *d_A, *d_B, *d_C;
 //...allocate space on CPU and GPU
 //...initialize a,b
 //...copy a,b to GPU at d_A, d_B
 //launch the kernel – Assume array has N elements
 int nblocks = (N-1)/512+1;      //512 threads per block
 vectorAdd <<<nblock,N>>> (d_A, d_B, d_C, N);
 ...
```

1D array of blocks, each having 1D array of threads

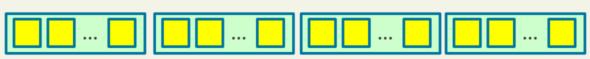
Grid



36

Vector Addition: Full Code

Rewrite the serial program below so that vectorAdd runs on the GPU with 4 blocks each having 256 threads



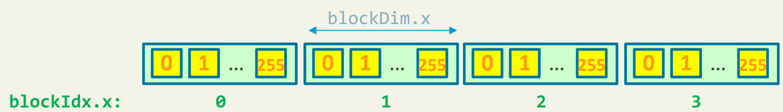
SERIAL CODE

```
#define N 1024
void vectorAdd(int* a, int* b, int* c, int n) {
   int i;
   for (i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}

int main() {
   int *a, *b, *c, i;
   a = (int*) malloc(N * sizeof(int));      // create three arrays
   b = (int*) malloc(N * sizeof(int));
   c = (int*) malloc(N * sizeof(int));

   for(i=0;i<N;i++) a[i] = b[i] = i;        // initialize a and b for testing
   vectorAdd(a, b, c, N);                   // serial vector addition
   for(i=0;i<10;i++) printf("c[%d] = %d\n", i, c[i]);
   free(a);free(b);free(c);                 // free up memory taken by a,b,c
   return 0;
}
```


40

Vector Addition : Parallel Code

Step1: parallelizing the function




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

Vectors are 1D, so we assume we launch a 1D grid of threads

Vector Addition: Parallel Code

Step2: modify the main method

```
int main() {
    int *a, *b, *c, i;      //pointers to host memory + loop counter
    int *d_A, *d_B, *d_C;  //pointers to device memory

    a = (int*) malloc(N * sizeof(int)); //allocate space on host
    b = (int*) malloc(N * sizeof(int));
    c = (int*) malloc(N * sizeof(int));

    //allocate space on device
    cudaMalloc(&d_A, N * sizeof(int));
    cudaMalloc(&d_B, N * sizeof(int));
    cudaMalloc(&d_C, N * sizeof(int));

    for(i=0;i<N;i++) a[i]=b[i]=i; //intialize a,b (for testing)
    //copy data (i.e. a and b) from host to device
    cudaMemcpy(d_A, a, N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_A, a, N * sizeof(int), cudaMemcpyHostToDevice);
```

Vector Addition: Parallel Code

```
//specify threads configuration & pass pointers to device memory
int nThreads = 256;
int nBlocks = N/nThreads
if (N%256) nBlocks++;

vectorAdd<<<nBlocks, nThreads>>>(d_A, d_B, d_C, N);

//copy results from device to host
cudaMemcpy(c, d_C, N * sizeof(int), cudaMemcpyDeviceToHost);

for(i = 0; i < 10; i++)        // print first 10 elements(for testing)
     printf("c[%d] = %d\n", i, c[i]);

free(a);free(b);free(c);       // free the host memory taken by a,b,c

//free device memory
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);

return 0;
}
```

Another way to specify the # of blocks given # of threads and # of threads/block

Summary: How Many Blocks?

Given that **nthreads** is # threads **per block**

Method1:

```
nblocks = (N-1)/nthreads + 1
dim3 gridSize(nblocks, 1, 1);
dim3 blockSize(nthreads, 1, 1);
```

Method2:

```
nblocks = N/nthreads;
if(N%256) nblocks++;  //if there is remainder, add one more block
dim3 gridSize(nblocks, 1, 1);
dim3 blockSize(nthreads, 1, 1);
```

Summary

Previous pre-recorded lecture (Students' led Q/As):

- CUDA basics: program structure
- Useful Built-in CUDA functions
- Function Declarations (global, device, host)

Today:

- Error Handling, cudaDeviceSynchronize
- Hardware architecture: sp → SM → GPU
- Thread Organization: threads → blocks → grids
  - Dimension variables (blockDim, gridDim)
- Thread Life Cycle From the HW Perspective
- Kernel Launch Configuration: 1D grids/blocks

Next Lecture:

- Kernel Launch Configuration: nD grids/blocks
- CUDA limits
- Thread Cooperation
- Running Example: Matrix Multiplication