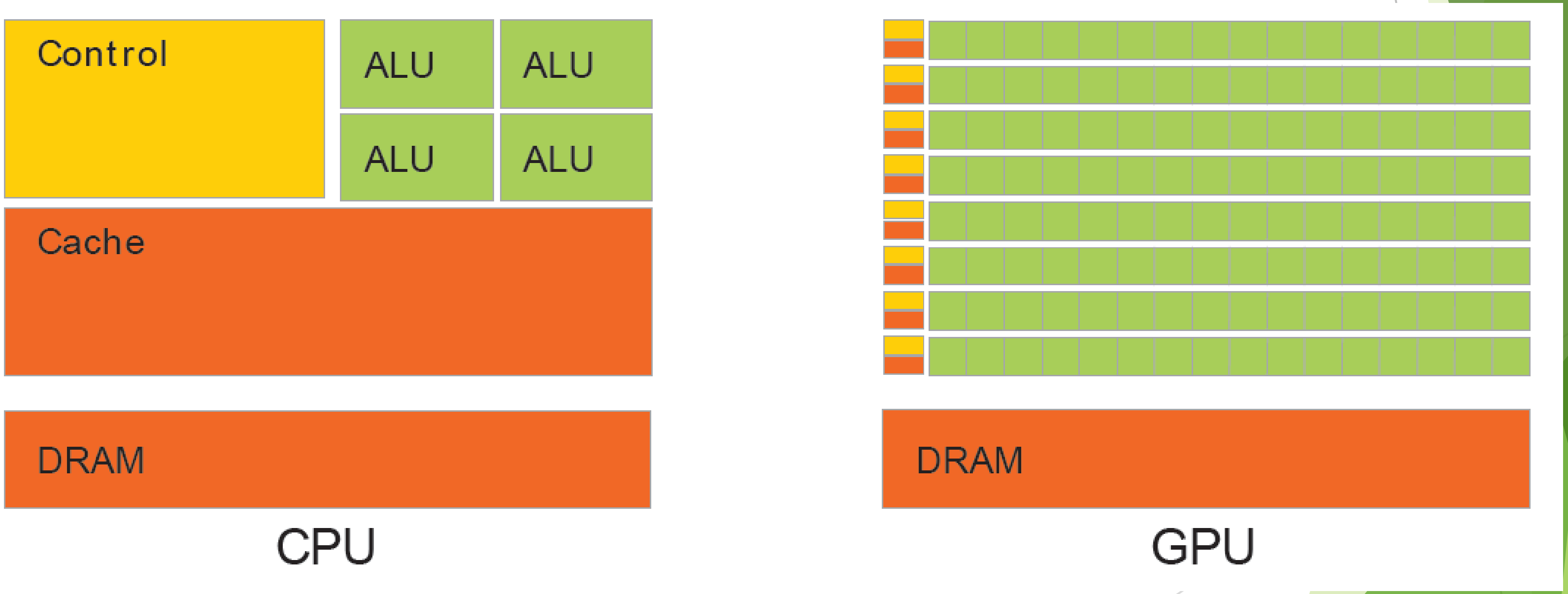
Basic CUDA Programming

Computer Architecture 2020 (Prof. Chih-Wei Liu)

CUDA Tutorial

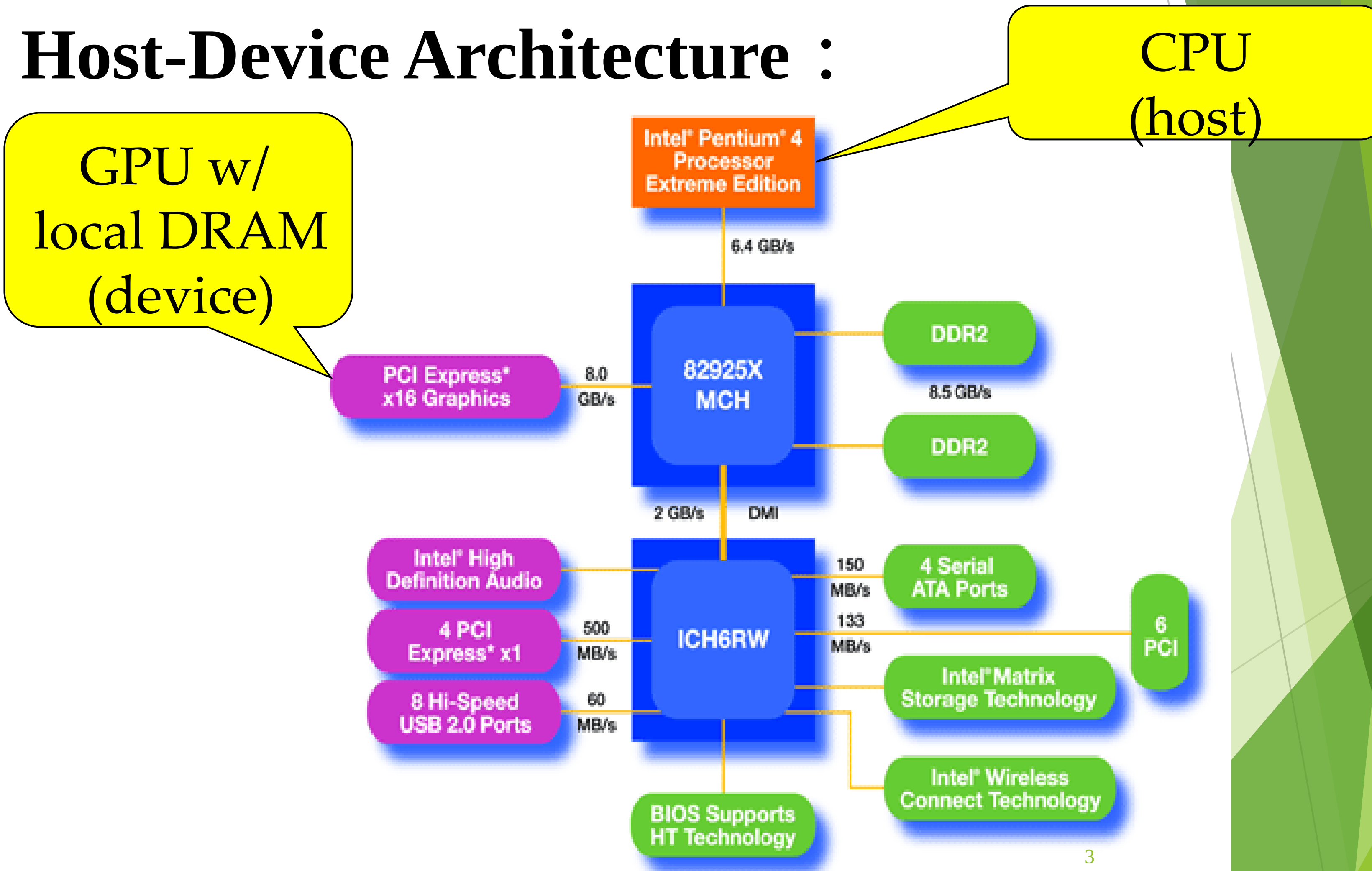
From Graphics to General Purpose Processing – CPU vs GPU

- CPU：general purpose computation (SISD)
- GPU：data-parallel computation (SIMD)

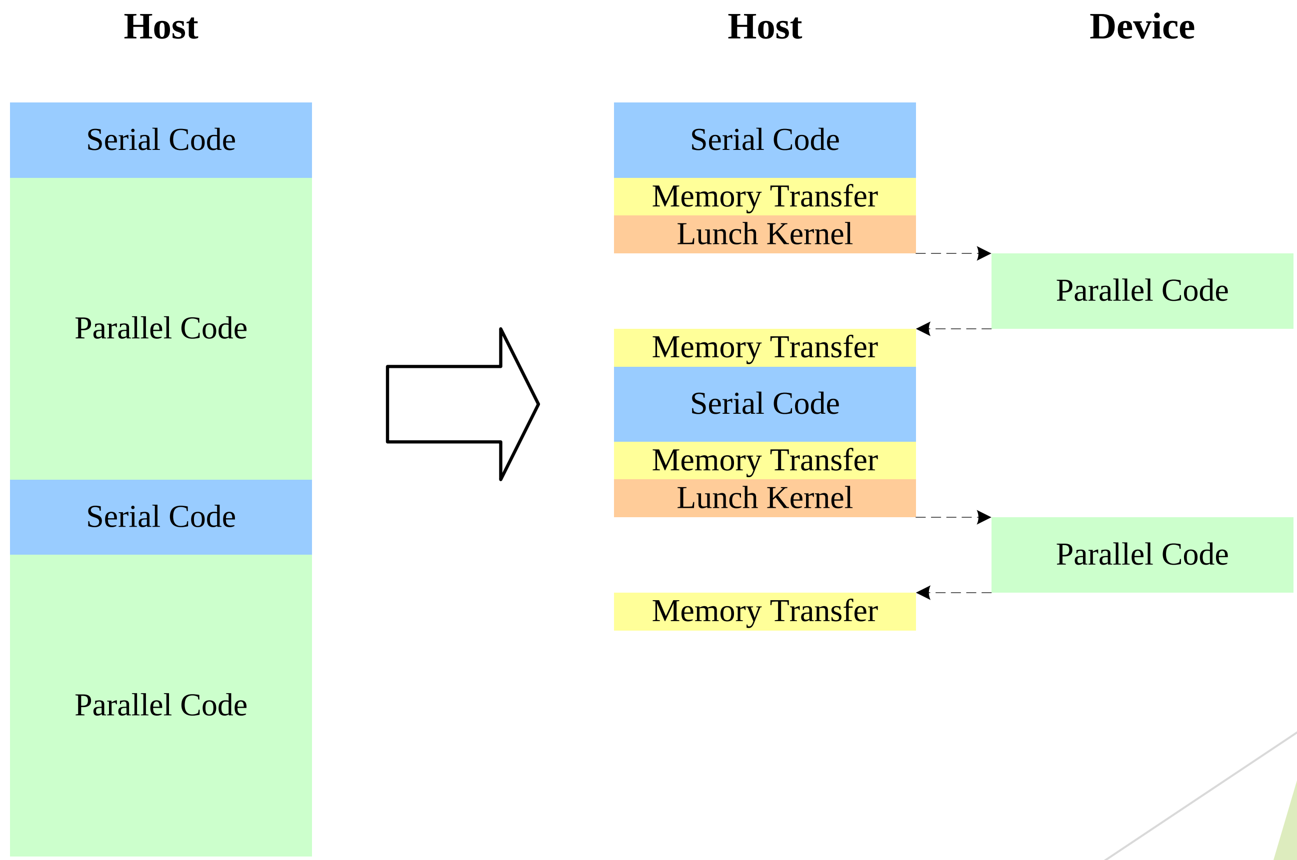


Heterogeneous computing: CPU+GPU Cooperation

Host-Device Architecture：

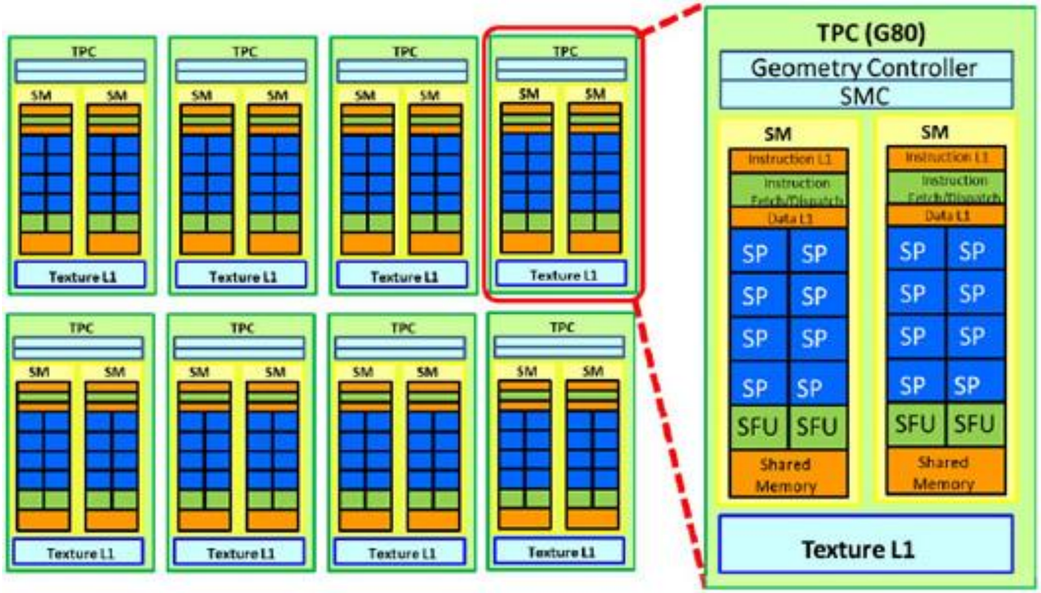


Heterogeneous computing: CUDA Code Execution



Heterogeneous computing: NIVIDA G80 series

- Streaming Multiprocessor (SM)
  - Streaming Processor (SP) / **[CUDA core]**
  - Shared memory
- For NV GTX 1080, the number of SPs is 2560.



CUDA Programming Model (1/7)

- Define
  - **<u>Programming model</u>**
  - **<u>Memory model</u>**
- Help developers map the applications or algorithms onto CUDA devices more easily and clearly.
- It is important to follow CUDA's programming model to obtain higher performance of program execution.

CUDA Programming Model (2/7)

- **C/C++ for CUDA**
  - Subset of C with extensions
  - C++ templates for GPU code
  - CUDA goals:
    - Scale code to 100s of cores and 1000s of parallel threads.
    - Facilitate heterogeneous computing: CPU + GPU

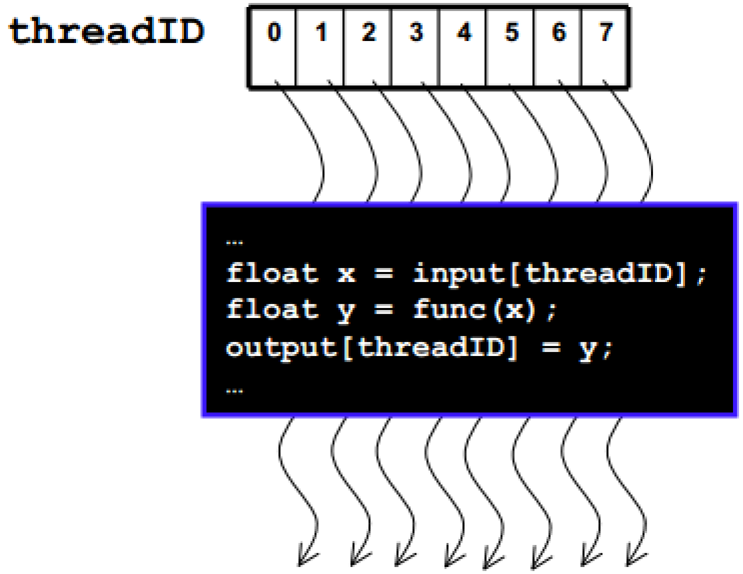
CUDA Programming Model (3/7)

- **CUDA Kernels and Threads**：
  - Parallel portions of an application executed on the device are **kernels**. And only one **kernel** is executed at a time.
  - All the **threads** execute the same kernel at a time.

CUDA Programming Model (4/7)

Arrays of Parallel Threads：

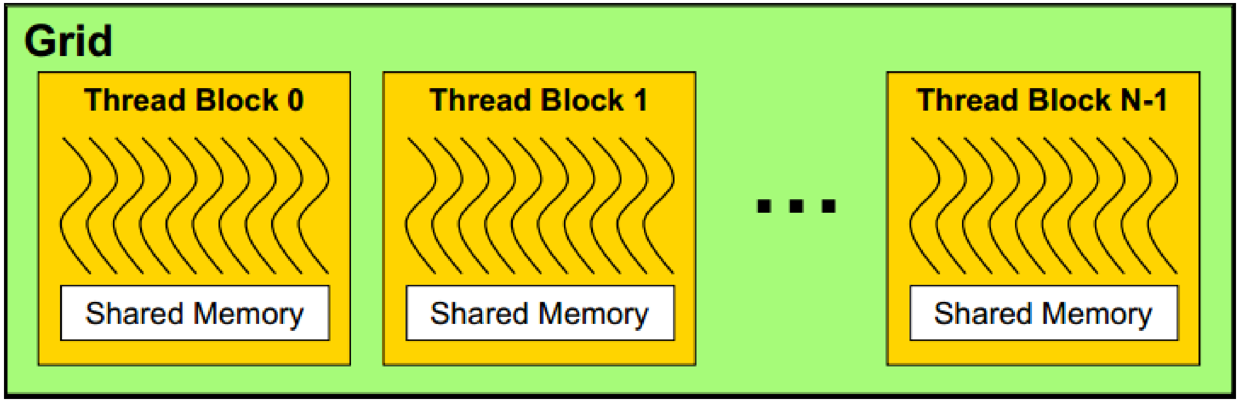
- A CUDA kernel is executed by an array of threads
  - <u>Each thread has an ID that it uses to compute memory addresses and make control decisions</u>



CUDA Programming Model (5/7)

Thread Batching：

- Kernel launches a **grid** of **thread blocks**

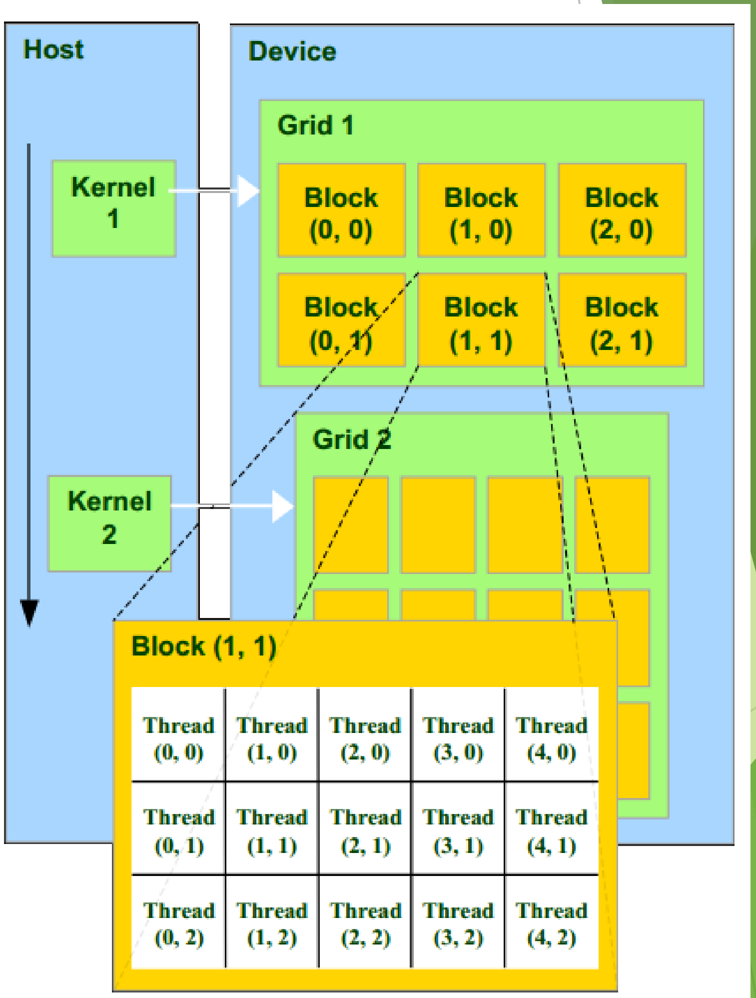


- Threads within a block cooperate via shared memory
- Threads in different blocks cannot cooperate

CUDA Programming Model (6/7)

CUDA Programming Model：

- A kernel is executed by a grid of thread blocks
  - Block can be 1D or 2D or 3D.
- A thread block is a batch of threads
  - Thread can be 1D or 2D or 3D.
  - Data can be shared through shared memory
  - Execution synchronization
  - But threads from different blocks can't cooperate.



CUDA Programming Model (7/7)

Memory Model：

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

Heterogeneous computing: NVIDIA CUDA Compiler (NVCC)

- NVCC separates CPU and GPU source code into two parts.
  - For host codes, NVCC invokes typical C compiler like GCC, Intel C compiler, or MS C compiler.
  - All the device codes are compiled by NVCC.
    - The extension of device source files should be “.cu”.
- All executable with CUDA code requires：
  - CUDA core library (cuda)
  - CUDA runtime library (cudart)

CUDA C/C++ Basic

GPU Memory Allocation/Release

- Memory allocation on GPU
  - cudaMalloc(void **pointer, size_t nbytes)
- Preset value for specific memory area
  - cudaMemset(void *pointer, int value, size_t count)
- Release memory allocation
  - cudaFree(void *pointer)

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

Data Copies

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

Function Qualifiers

- __global__ : invoked from within host (CPU) code, also called ‘kernel’
- __device__ : called from other GPU functions, cannot be called from host (CPU) code
- __host__ : can only be executed by CPU, called from host

Variable Qualifiers (GPU code)

- __device__
  - Stored in device memory (large capacity, high latency, uncached)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- __shared__
  - Stored in on-chip shared memory (SRAM, low latency)
  - Allocated by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: duration of thread block

CUDA Built-in Device Variables

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

Launching Kernels

- kernel<<<dim3 grid, dim3 block>>>(…);

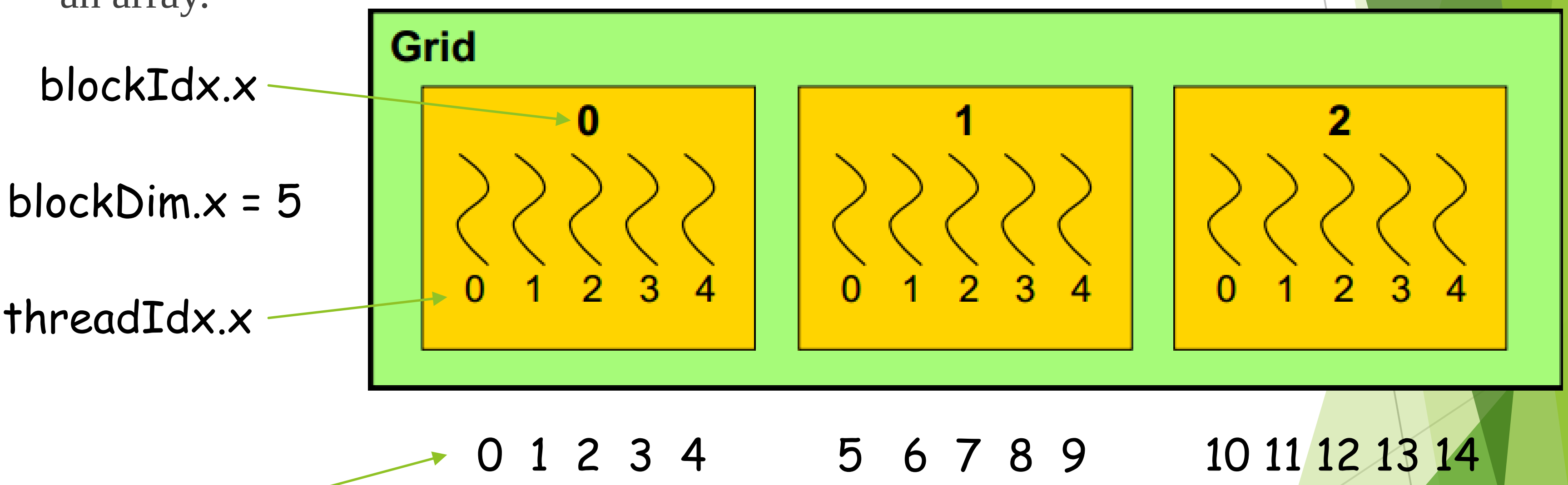
```
dim3 grid(16,16);
dim3 block(16,16);
kernel1<<<grid, block>>>(…);



kernel2<<<32, 512>>>(…);
```

Data Decomposition

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



idx = blockIdx.x*blockDim.x + threadIdx.x

EXAMPLE-1

Number of threads ≥ N

Data Decomposition Example: Increment Array Elements (1) (1/4)

- Increment N-element vector a by scalar b
  - **Each thread only executes ONCE**

CPU program

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



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

CUDA program

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

void main()
{
    …
    dim3 dimBlock(BlockSize);
    dim3 dimGrid(ThreadNum);
    increment_gpu<<<dimGrid,dimBlock>>>(d_a,b,N);
}
```


main_gpu_1.cu

EXAMPLE-1

Number of threads ≥ N

Data Decomposition Example: Increment Array Elements (1) (2/4)

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>
using namespace std;

const int BlockSize = 2;
const int ThreadNum = 5;
const int N = 10;

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

```
int main()
{
    int a[N];
    int *d_a = 0;
    int i = 0;
    const int b = 10;

    cout << "a[N] array before scaling: [";
    for(i=0;i<N;i++)
    {
        a[i] = i;
        cout << a[i] << " ";
    }
    cout << "]" << endl;

    cudaMalloc((void**) &d_a,sizeof(int)*N);
    cudaMemcpy(d_a,a,sizeof(int)*N,cudaMemcpyHostToDevice);

    dim3 dimBlock(BlockSize);
    dim3 dimGrid(ThreadNum);
    increment_gpu<<<dimGrid,dimBlock>>>(d_a,b,N);
    cudaDeviceSynchronize();

    cudaMemcpy(a,d_a,sizeof(int)*N,cudaMemcpyDeviceToHost);

    cout << "a[N] array after scaling: [";
    for(i=0;i<N;i++)
    {
        cout << a[i] << " ";
    }
    cout << "]" << endl;

    return 0;
}
```

Pointer referred to device memory space

1. Allocate an integer array on device's memory

2. Copy the 'a' array's value to 'd_a' (on the device)

main_gpu_1.cu

EXAMPLE-1

Number of threads ≥ N

Data Decomposition Example: Increment Array Elements (1) (3/4)

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>
using namespace std;

const int BlockSize = 2;
const int ThreadNum = 5;
const int N = 10;

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

```
int main()
{
    int a[N];
    int *d_a = 0;
    int i = 0;
    const int b = 10;

    cout << "a[N] array before scaling: [";
    for(i=0;i<N;i++)
    {
        a[i] = i;
        cout << a[i] << " ";
    }
    cout << "]" << endl;

    cudaMalloc((void**) &d_a,sizeof(int)*N);
    cudaMemcpy(d_a,a,sizeof(int)*N,cudaMemcpyHostToDevice);

    dim3 dimBlock(BlockSize);
    dim3 dimGrid(ThreadNum);
    increment_gpu<<<dimGrid,dimBlock>>>(d_a,b,N);
    cudaDeviceSynchronize();

    cudaMemcpy(a,d_a,sizeof(int)*N,cudaMemcpyDeviceToHost);

    cout << "a[N] array after scaling: [";
    for(i=0;i<N;i++)
    {
        cout << a[i] << " ";
    }
    cout << "]" << endl;

    return 0;
}
```

3. Invoke CUDA kernel function
4. Make sure all the tasks are finished
5. Retrieve the results from device.

main_gpu_1.cu

EXAMPLE-1

Number of threads ≥ N

Data Decomposition Example: Increment Array Elements (1) (4/4)

N = 10 ⇦ **total tasks**

BlockSize = 2

ThreadNum = 5 ⇦ **total threads = 2*5**

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
#1	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)

(blockIdx.x , threadIdx.x)

EXAMPLE-1

Number of threads < N

Data Decomposition Example: Increment Array Elements (2) (1/3)

N = 10 **total tasks**

BlockSize = 1

ThreadNum = 5 **total threads = 1*5**

	a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**	**a[5]**	**a[6]**	**a[7]**	**a[8]**	**a[9]**
#1	(0,0)		(0,1)		(0,2)		(0,3)		(0,4)	
#2		(0,0)		(0,1)		(0,2)		(0,3)		(0,4)

stripe size

(blockIdx.x , threadIdx.x)

EXAMPLE-1

Number of threads < N

Data Decomposition Example: Increment Array Elements (2) (2/3)

- Increment N-element vector a by scalar b

CPU program

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


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

CUDA program

```
__global__ void increment_gpu(float *a, float *b, int N)
{
     int TotalThread = gridDim.x*blockDim.x;
     int stripe = N / TotalThread;
     int head = (blockIdx.x*blockDim.x + threadIdx.x)*stripe;
     for(int i = head;i<(head+stripe);i++)
         a[i]=a[i]+b;
}

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

main_gpu_2.cu

EXAMPLE-1

Number of threads < N

Data Decomposition Example: Increment Array Elements (2) (3/3)

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>
using namespace std;

const int BlockSize = 1;
const int ThreadNum = 5;
const int N = 10;

__global__ void increment_gpu(int a[],const int b, const int N)
{
    int i = 0;
    int TotalThread = gridDim.x*blockDim.x;
    int stripe = N / TotalThread;
    int head = (blockIdx.x*blockDim.x + threadIdx.x)*stripe;

    for(i = head;i<(head+stripe);i++)
        a[i]=a[i]+b;
}
```

```
int main()
{
    int a[N];
    int *d_a = 0;
    int i = 0;
    const int b = 10;

    cout << "a[N] array before scaling: [";
    for(i=0;i<N;i++)
    {
        a[i] = i;
        cout << a[i] << " ";
    }
    cout << "]" << endl;

    cudaMalloc((void**) &d_a,sizeof(int)*N);
    cudaMemcpy(d_a,a,sizeof(int)*N,cudaMemcpyHostToDevice);

    dim3 dimBlock(BlockSize);
    dim3 dimGrid(ThreadNum);
    increment_gpu<<<dimGrid,dimBlock>>>(d_a,b,N);
    cudaDeviceSynchronize();

    cudaMemcpy(a,d_a,sizeof(int)*N,cudaMemcpyDeviceToHost);

    cout << "a[N] array after scaling: [";
    for(i=0;i<N;i++)
    {
        cout << a[i] << " ";
    }
    cout << "]" << endl;

    return 0;
}
```

main_gpu_2.cu

Reference

- NVIDIA CUDA Toolkit Documentations (http://docs.nvidia.com/cuda/)
- Recommended learning materials
  - CUDA Programming C/C++ basic (https://www.youtube.com/watch?v=kyL2rj_Se3M)