Introduction to CUDA Architecture and Programming

Putt Sakdhnagool
putt.sakdhnagool@nectec.or.th
NECTEC

Some slides are adopted from "CUDA C/C++ Basics" by Cyril Zeller, NVIDIA: http://www.int.washington.edu/PROGRAMS/12-2c/week3/clark_01.pdf
Some contents are adopted "An Even Easier Introduction to CUDA" by Mark Harris, NVIDIA: https://devblogs.nvidia.com/even-easier-introduction-cuda/

Prerequisites

- You should have some experience is C/C++ programming
- Don’t need knowledge in GPU programming
- Don’t need knowledge in Graphic programming
- Parallel programming knowledge is a plus but not necessary

Outlines

- Short Introduction to CUDA (10 mins)
- Setups (10 mins)
- Hands-on Labs (1 hr 10 mins)
  - Lab01: Say Hello to CUDA
  - Lab02: CUDA in Actions.

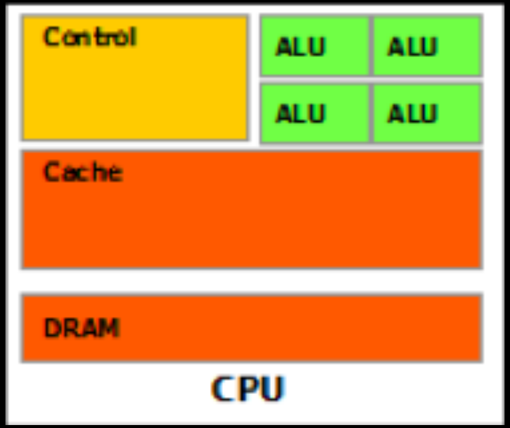
What is CUDA ?

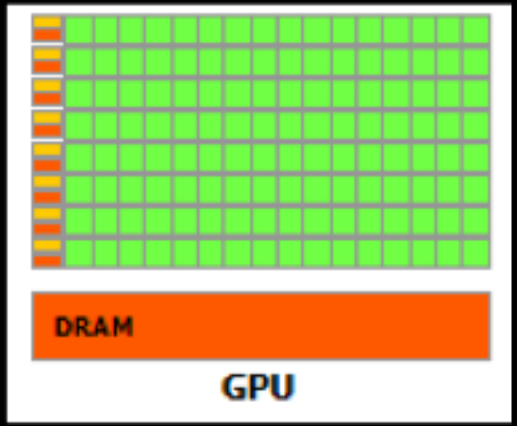
- CUDA
  - Platform and programming model for GPUs
  - Expose GPUs for general purpose computing

- This session introduces CUDA C/C++

- CUDA C/C++
  - C/C++ language extension for programming and managing GPUs
  - APIs to manage GPUs (devices), memory, etc.

CPUs vs GPUs





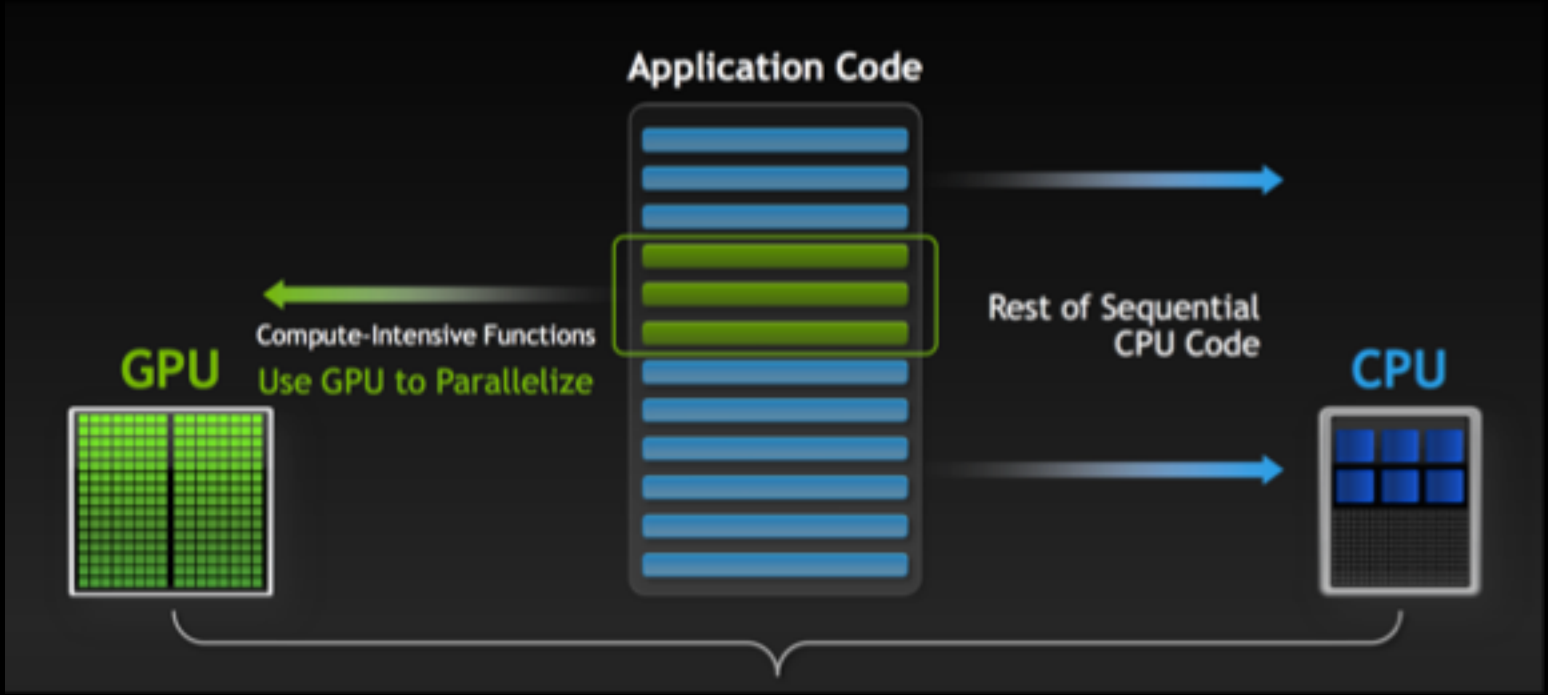
- Optimized for **latency**
  - Get job done ***fast***.
- **Intel Xeon Scalable**: 28 cores running at 2.1 Ghz
- Lower Memory Bandwidth
  - But larger memory size
- Suitable for generic workloads

- Optimized for **throughput**
  - Get ***more*** jobs done
- **Nvidia V100**: 5120 cores running at 1.2 Ghz
- Higher Memory Bandwidth
  - But smaller memory size
- Suitable for highly parallel workloads.

Source: CUDA Programming Guide, NVIDIA, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Heterogeneous Computing

- CPU (host) + GPUs (devices) Computing
  - **Offloading** parallel computation to GPUs.



Source: Introduction to CUDA, Mike Clark, NVIDIA, http://www.int.washington.edu/PROGRAMS/12-2c/week3/clark_01.pdf

00: Setting Up

Setup tutorial from git repository

Connecting to GPU Machine

Windows

- Open SSH Client: MobaXterm (https://mobaxterm.mobatek.net/)
- Click on **Session**
- In **SSH Session**
  - Remote host: 10.34.13.250
  - Check **specify username**
  - Type ***username*** in the textbox
  - Click OK
- Provide password when prompt

MacOS / Linux

- In Terminal, type

`$> ssh` *`username`*`@158.108.32.164`

- Provide password when prompt

Monitoring GPUs

- nvidia-smi: command line utility for managing and monitoring GPUs

```
$> nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.46                 Driver Version: 390.46                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla M2050         Off  | 00000000:14:00.0 Off |                    0 |
| N/A   N/A    P0    N/A /  N/A |      0MiB /  2622MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla M2050         Off  | 00000000:15:00.0 Off |                    0 |
| N/A   N/A    P0    N/A /  N/A |      0MiB /  2622MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Installing Repository

- Clone git repository using command

```
$> git clone https://github.com/puttsk/cuda-tutorial.git
$> cd cuda-tutorial
```

- Github:
  https://github.com/puttsk/cuda-tutorial/
- Online Document:
  https://cuda-tutorial.readthedocs.io/en/latest/

01: Say Hello to CUDA

Writing your first CUDA program

Hello World!! C vs CUDA

C	CUDA

`void c_hello(){`
`    printf("Hello World!\n");`
`}`
 	
`int main() {`
`    c_hello()`
`    return 0;`
`}`
	`__global__ void cuda_hello(){`
	`    printf("Hello World from GPU!\n");`
	`}`
	
	`int main() {`
	`    cuda_hello<<<1,1>>>();`
	`    return 0;`
	`}`

Compiling	**Compiling**
`$> g++ hello.c`	`$> nvcc hello.cu`

Hello World!! C vs CUDA

C	CUDA

C

```c
void c_hello(){
    printf("Hello World!\n");
}
```

CUDA

```c
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

Compiling

```
$> nvcc hello.cu
```

What's new?

- `__global__` **specifier**
  - Indicate a function that run on device
  - Known as "**kernels**"
  - The function is called through host code

Hello World!! C vs CUDA

C	CUDA

C

```c
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello()
    return 0;
}
```

CUDA

```cuda
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

What’s new?

- **<<<…>>> kernel execution configuration**
    - A call from **host** to **device** code
    - Called “kernel launch”
    - Will discuss about the parameter (1,1) later.

Compiling

```
$> nvcc hello.cu
```

Hello World!! C vs CUDA

C	CUDA

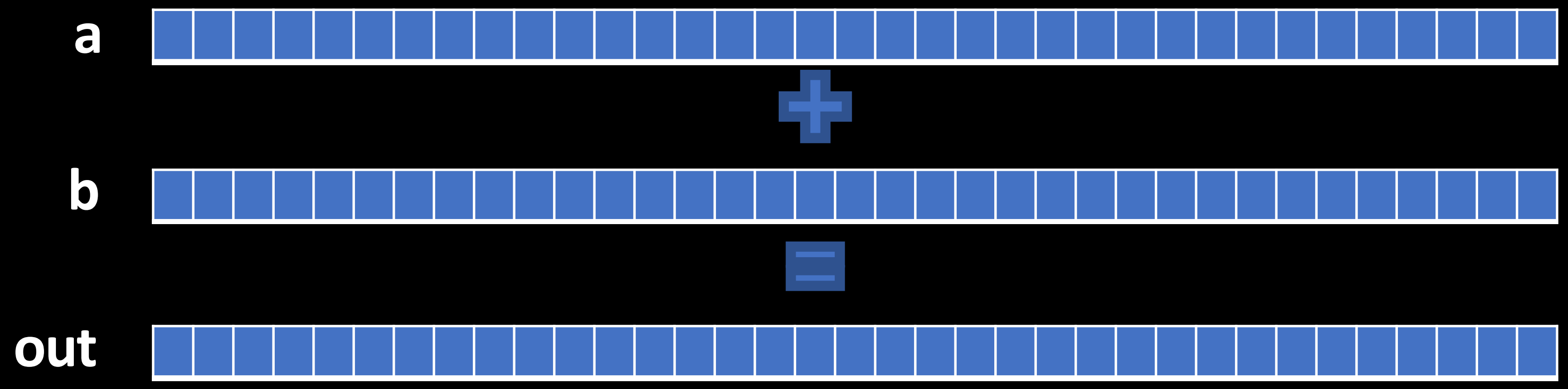
`void c_hello(){`
`    printf("Hello World!\n");`
`}`
 	
Compiling
 	
`$> g++ hello.c`
	`__global__ void cuda_hello(){`
	`    printf("Hello World from GPU!\n");`
	`}`
	
	`int main() {`
	`    cuda_hello<<<1,1>>>();`
	`    return 0;`
	`}`
	
	Compiling
	
	`$> nvcc hello.cu`

What's new?

- **nvcc compiler**
  - New file extension *.cu
  - NVIDIA compiler for compiling GPU program

Say Hello to CUDA: Vector Addition

- GPU is designed for massively parallel work
- We will write CUDA vector addition from C



Vector Addition in CUDA

- Vector addition in C (tutorial01/vector_add.c)

```
#define N 10000000
void vector_add(float *out, float *a, float *b, int n) {
    for(int i = 0; i < n; i++){
        out[i] = a[i] + b[i];
    }
}

void main(){
    a = (float*)malloc(sizeof(float) * N);
    …
    vector_add(out, a, b, N);
    …
    free(a);
}
```

Vector Addition in CUDA

- Compile and run…

```
$> g++ –o vector_add vector_add.c
$> ./vector_add
out[0] = 3.000000
PASSED
```

Vector Addition in CUDA

- Converting vector addition from C to CUDA
- Copy `vector_add.c` to `vector_add.cu`

```
$ cp vector_add.c vector_add.cu
```

- Convert `vector_add()` to GPU kernel

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    for(int i = 0; i < n; i++){
        out[i] = a[i] + b[i];
    }
}
```

- Change `vector_add()` call in `main()` to kernel launch
- Compile and run

```
$> nvcc –o vector_add vector_add.cu
```

Vector Addition in CUDA

- You should get a code similar to this

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    for(int i = 0; i < n; i++){
        out[i] = a[i] + b[i];
    }
}

void main(){
    …
    vector_add<<<1,1>>>(out, a, b, N);
    …
}
```

CUDA Workflow

- The previous example does not work properly. **Why???**
- CPU and GPUs are **separate entities**
  - Both have their own memory space
- GPU **cannot** directly access data in the host/CPU memory
  - and vice versa
- What to do?

CUDA Workflow

- CUDA provides APIs for GPU memory management

- Simple CUDA Workflow
  - Allocate host memory and initialize host data
  - Allocate device memory
  - Transfer data from host to device memory
  - Execute kernels
  - Transfer result from device to host memory

CUDA Workflow

- CUDA provides APIs for GPU memory management
- Simple CUDA Workflow
  - Allocate host memory and initialize host data
  - Allocate device memory
  - Transfer data from host to device memory
  - Execute kernels
  - Transfer result from device to host memory

We have done these steps

CUDA Workflow

- CUDA provides APIs for GPU memory management

- Simple CUDA Workflow
  - Allocate host memory and initialize host data
  - Allocate device memory
  - Transfer data from host to device memory
  - Execute kernels
  - Transfer result from device to host memory

These steps are missing

- How to allocate device memory ?
- How to transfer data between memory space ?

Device Memory Management

- Host and device have separated memory space
  - **Host pointer** points to CPU memory
  - **Device pointer** points to GPU memory
- Allocating device memory
  - `cudaMalloc(void **devPtr, size_t count)`
  - `cudaFree(void *devPtr)`
- Compare to C `malloc()` and `free()`
  - `malloc(size_t count)`
  - `free(void *ptr)`

Memory Transfer

- Transferring data between memory
  - `cudaMemcpy(void *dst, void *src, size_t count,cudaMemcpyKind kind)`
  - `kind` indicates the direction of memory transfer
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost`
- Compare to C `memcpy()`
  - `memcpy(void* dst, void* src, size_t count)`

Back to Vector Addition

- Example for array a

```
void main(){
    float *a, *b, *out;
    a = (float*)malloc(sizeof(float) * N);

    //allocate device memory for a
    float *d_a;
    cudaMalloc((void**)&d_a, sizeof(float) * N);

    //transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

    …
    vector_add<<<1,1>>>(out, d_a, b, N);
    …

    //cleanup after kernel execution
    cudaFree(d_a);
    free(a);
}
```

Back to Vector Addition

- Example for array a

```
void main(){
    float *a, *b, *out;
    a = (float*)malloc(sizeof(float) * N);

    //allocate device memory for a
    float *d_a;
    cudaMalloc((void**)&d_a, sizeof(float) * N);

    //transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

    …
    vector_add<<<1,1>>>(out, d_a, b, N);
    …

    //cleanup after kernel execution
    cudaFree(d_a);
    free(a);
}
```

Allocating device memory for array **a**

Back to Vector Addition

- Example for array a

```
void main(){
    float *a, *b, *out;
    a = (float*)malloc(sizeof(float) * N);

    //allocate device memory for a
    float *d_a;
    cudaMalloc((void**)&d_a, sizeof(float) * N);

    //transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

    …
    vector_add<<<1,1>>>(out, d_a, b, N);
    …

    //cleanup after kernel execution
    cudaFree(d_a);
    free(a);
}
```

Transfer array **a** from host to device memory

Back to Vector Addition

- Example for array a

```
void main(){
    float *a, *b, *out;
    a = (float*)malloc(sizeof(float) * N);

    //allocate device memory for a
    float *d_a;
    cudaMalloc((void**)&d_a, sizeof(float) * N);

    //transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

    …
    vector_add<<<1,1>>>(out, d_a, b, N);
    …

    //cleanup after kernel execution
    cudaFree(d_a);
    free(a);
}
```

Passing device pointer **`d_a`** to **`vector_add`** instead of host pointer **`a`**

Back to Vector Addition

- Example for array a

```
void main(){
    float *a, *b, *out;
    a = (float*)malloc(sizeof(float) * N);

    //allocate device memory for a
    float *d_a;
    cudaMalloc((void**)&d_a, sizeof(float) * N);

    //transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

    …
    vector_add<<<1,1>>>(out, d_a, b, N);
    …

    //cleanup after kernel execution
    cudaFree(d_a);
    free(a);
}
```

Freeing device memory

Exercise

- Which array must be transferred before and after kernel execution ?
- Finish GPU vector addition

Running Vector Addition

- See solution in (tutorial01/solutions/vector_add.cu)
- Compile and measure performance.

```
$> nvcc -o vector_add vector_add.cu
$> time ./vector_add
```

Running Vector Addition

- See solution in (tutorial01/solutions/vector_add.cu)
- Compile and measure performance.

```
$> nvcc -o vector_add vector_a
$> time ./vector_add
```

Does not tell much about kernel performance

Profiling Performance

- Use nvprof to profile kernel performance

```
$ nvprof ./vector_add

==6326== Profiling application: ./vector_add
==6326== Profiling result:
Time(%)     Time Calls       Avg       Min       Max Name
97.55% 1.42529s     1 1.42529s 1.42529s 1.42529s vector_add(float*, float*, float*, int)
 1.39% 20.318ms     2 10.159ms 10.126ms 10.192ms [CUDA memcpy HtoD]
 1.06% 15.549ms     1 15.549ms 15.549ms 15.549ms [CUDA memcpy DtoH]
```

02: CUDA in Actions

Parallelizing CUDA program

Going Parallel

- GPU computing is about massive parallelism
  - Previous version uses only one GPU thread.
  - How do we know ?

- Recall the kernel execution configuration

```
vector_add <<< 1 , 1 >>> (d_out, d_a, d_b, N);
```

Going Parallel – Terminology

- Kernel execution configuration tells CUDA runtime how many threads to launch on GPU
  - CUDA organizes threads in a group called "*thread block*"
    - Threads in the same thread block will run on the same processor
  - Kernel can launch multiple thread blocks, which are organized into a "*grid*"
    - Will talk more later…

```
vector_add <<< M , T >>> (d_out, d_a, d_b, N);
```

- Indicates that `vector_add` launches with a grid of M thread blocks, each has T parallel threads

Going Parallel – Thread Block

- Let's start with 1 thread block with 256 threads

```
vector_add <<< 1 , 256 >>> (out, d_a, b, N);
```

- CUDA C/C++ provides built-in variable for thread indices
  - `threadIdx.x` : index of the current thread with in the block
  - `blockDim.x` : the number of threads in the thread block

Going Parallel – Thread Block

- Let’s start with 1 thread block with 256 threads

```
vector_add <<< 1 , 256 >>> (out, d_a, b, N);
```

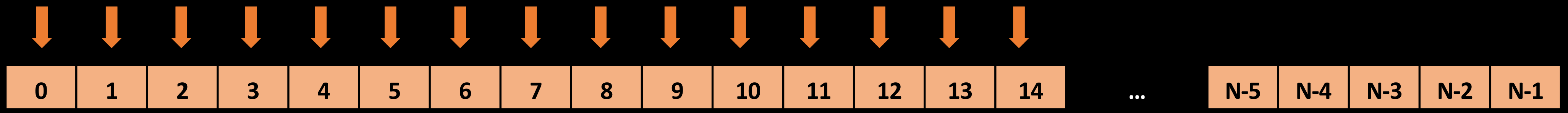
- CUDA C/C++ provides built-in variable for thread indices
  - `threadIdx.x` : index of the current thread with in the block
  - `blockDim.x` : the number of threads in the thread block

- **How to improve GPU vector addition?**
- Try to implement this in vector_add_thread.cu

Going Parallel – Thread Block

- Recall single thread version

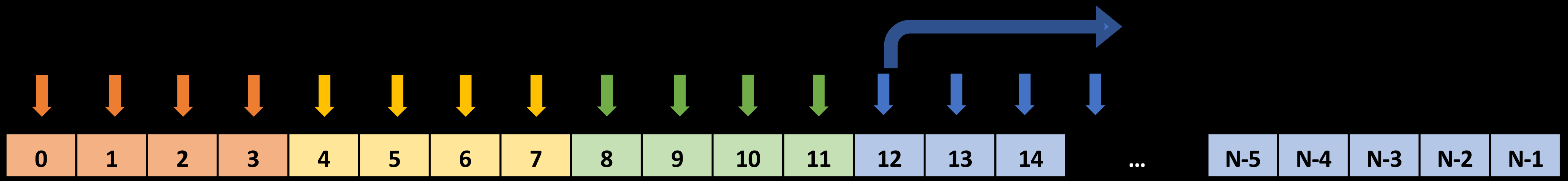
```
for(int i = 0; i < N; i++){
        out[i] = a[i] + b[i];
    }
```



Going Parallel – Thread Block

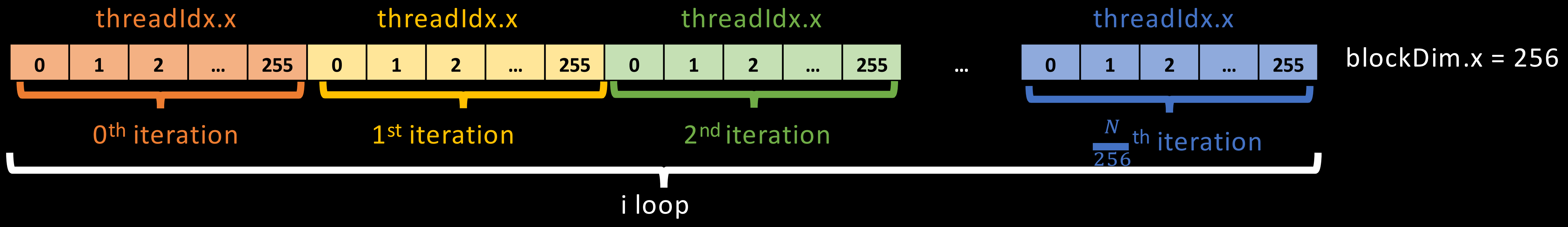
- With 4 threads

```
for(int i = tid; i < N; i+= 4){
        out[i] = a[i] + b[i];
    }
```



Going Parallel – Thread Block

- Ideas



- 0^{th} iteration, the k^{th} thread computes the addition of k^{th} elements.
- 1^{st} iteration, the k^{th} thread computes the addition of $(k+256)^{th}$ elements.
- 2^{st} iteration, the k^{th} thread computes the addition of $(k+512)^{th}$ elements.

Vector Addition - Thread Block

- See solution in tutorial02/solutions/vector_add_thread.cu

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int tid = threadIdx.x;
    for(int i = tid; i < n; i += blockDim.x){
        out[i] = a[i] + b[i];
    }
}

void main(){
    …
    vector_add<<< 1, 256>>>(out, a, b, N);
    …
}
```

Running and Profiling Again

- Compile and measure performance.

```
$ nvcc vector_add_thread.cu -o vector_add_thread
$ nvprof ./vector_add_thread
```

```
==6430== Profiling application: ./vector_add_thread
==6430== Profiling result:
Time(%)     Time Calls       Avg       Min       Max Name
39.18% 22.780ms      1 22.780ms 22.780ms 22.780ms vector_add(float*, float*, float*, int)
34.93% 20.310ms      2 10.155ms 10.137ms 10.173ms [CUDA memcpy HtoD]
25.89% 15.055ms      1 15.055ms 15.055ms 15.055ms [CUDA memcpy DtoH]
```

Going Parallel – Adding More Blocks

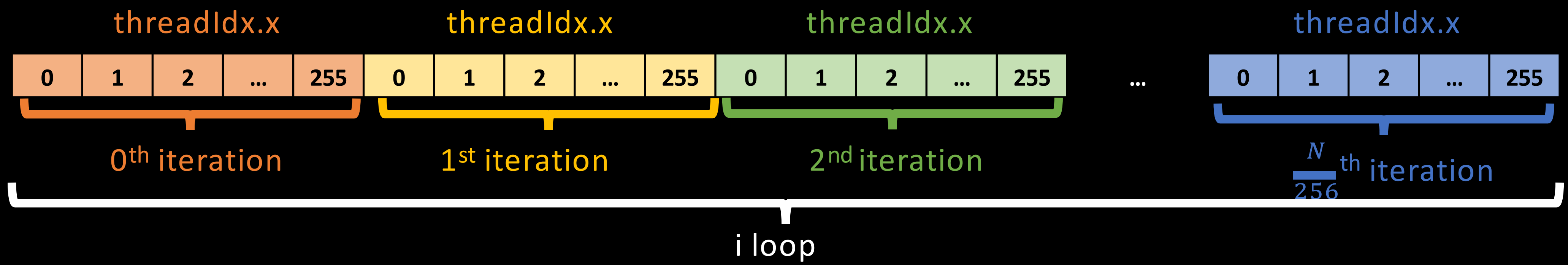
- CUDA GPUs have several parallel processors called **Streaming Multiprocessors** or **SMs**.
  - Each SM has multiple parallel processors
  - Each SM can run multiple concurrent thread blocks.

- We want to use many thread blocks to fully utilize GPU

- CUDA C/C++ provides keyword for block indices
  - `blockIdx.x` : index of the current block in the grid
  - `gridDim.x` : the number of blocks in the grid

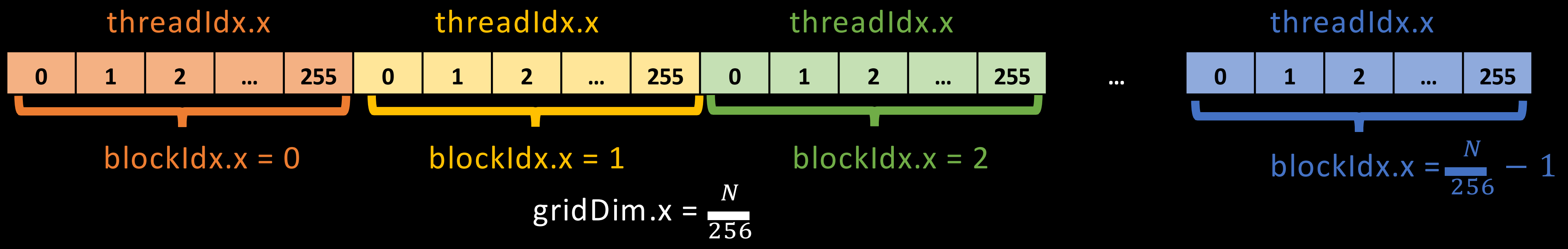
Going Parallel – Adding More Blocks

- Recall idea of single thread block



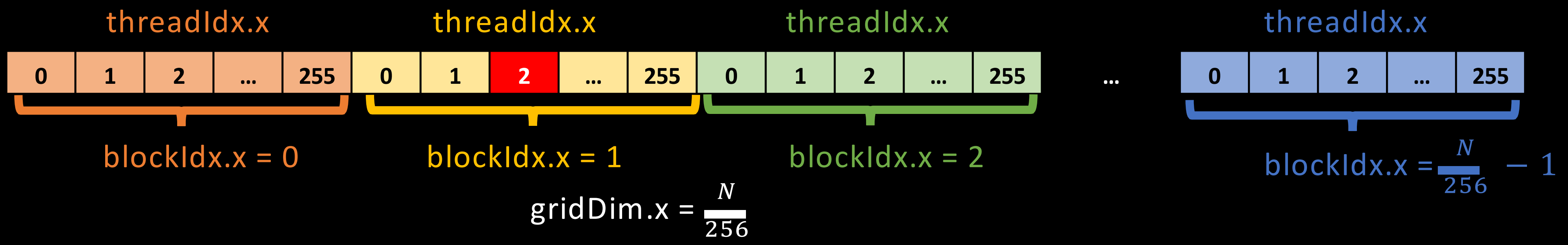
Going Parallel – Adding More Blocks

- Ideas: Create N threads, each thread processes one element.



Going Parallel – Adding More Blocks

- Ideas: Create N threads, each thread processes one element.



- Finding array index

```
array_idx =   blockIdx.x * blockDim.x + threadIdx.x
array_idx =      (1)      *    (256)     +     (2)         = 258
```

Going Parallel – Adding More Blocks

- How to specify number of block ?
- Kernel execution configuration with multiple thread blocks

```
vector_add <<< M , 256 >>> (out, d_a, b, N);
```

- What should be the value of M ?
  - We want 1 thread to add 1 element of the vector

Going Parallel – Adding More Blocks

- How to specify number of block ?
- Kernel execution configuration with multiple thread blocks

```
vector_add <<< M , 256 >>> (out, d_a, b, N);
```

- What should be the value of M ?
  - We want 1 thread to add 1 element of the vector

```
//Assume N is divisible by 256
int M = N / 256;
vector_add <<< M , 256 >>> (out, d_a, b, N);
```

- Try to implement the rest in vector_add_grid.cu

Vector Addition - Final

- See solution in tutorial02/solutions/vector_add_grid.cu

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // for handling arbitrary vector size
    if (tid < n){
        out[tid] = a[tid] + b[tid];
    }
}

void main(){
    …
    int grid_size = ((N + block_size) / block_size);
    vector_add<<< grid_size, block_size>>>(out, a, b, N);
    …
}
```

Running and Profiling Again

- Compile and measure performance.

```
$ nvcc vector_add_grid.cu -o vector_add_grid
$ nvprof ./vector_add_grid
```

```
==6564== Profiling application: ./vector_add_grid
==6564== Profiling result:
Time(%)     Time Calls       Avg       Min       Max Name
55.65% 20.312ms     2 10.156ms 10.150ms 10.162ms [CUDA memcpy HtoD]
41.24% 15.050ms     1 15.050ms 15.050ms 15.050ms [CUDA memcpy DtoH]
 3.11% 1.1347ms     1 1.1347ms 1.1347ms 1.1347ms vector_add(float*, float*, float*, int)
```

Comparing Performance

Version	Execution Time (ms)	Speedup
1 thread	1425.29	1.00x
1 block (256 threads)	22.78	62.56x
Multiple blocks	1.13	1261.32x

What's Missing

- See CUDA Programming Guide for more information about
  - Single Instruction Multiple Thread (SIMT) concept
  - Compute capability
  - Unified memory
  - Asynchronous execution
  - Texture memory
  - and more…

- CUDA libraries and tools
  - This afternoon

- OpenACC and OpenMP 4.x
  - Directive-based programming model for accelerators

References

- CUDA C/C++ Basics
  - http://www.int.washington.edu/PROGRAMS/12-2c/week3/clark_01.pdf
- An Even Easier Introduction to CUDA
  - https://devblogs.nvidia.com/even-easier-introduction-cuda/
- CUDA Programming Guide
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Q&A

Extras

Hello World!!

- Hello world in CUDA

```
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

- Compiling

```
$nvcc hello.cu
```

What's new?

- `__global__` **specifier**
  - Indicate a function that run on device
  - Known as "**kernels**"
  - The function is called through host code
- **<<<…>>> kernel execution configuration**
  - A call from **host** to **device** code
  - Called "kernel launch"
  - Will discuss about the parameter (1,1) later.
- **`nvcc` compiler**
  - NVIDIA compiler for compiling GPU program, e.g. .cu

Back to Vector Addition

- Need to allocate device memory for a, b, and out
- Example for a

```
void main(){
    float *a, *b, *out;
    float *d_a;

    a = (float*)malloc(sizeof(float) * size);

    //allocate device memory for a
    cudaMalloc((void**)&d_a, sizeof(float) * size);
    …
    vector_add<<<1,1>>>(out, d_a, b, size);
    …

    //cleanup after kernel execution
    cudaFree(d_a);
    free(a);
}
```

What's new?

- **Declaring device pointer**
- **Allocating device memory**
- **Freeing device memory**