HALMSTAD UNIVERSITY

CUDA C Presentation & Demo

Embedded Intelligent Systems Languages and Tools: Oral Presentation

31.01.2024

Table of Content

- **Introduction**
- Heterogenous Computing
- Blocks
- Threads
- Indexing
- Shared Memory and Sync Threads
- Device Management
- CUDA Demo

What is CUDA?

- **CUDA Architecture**
  - ☐ Expose GPU computing for general purpose.
  - ☐ Retain performance.

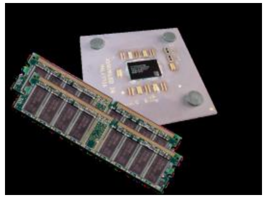
- **CUDA C**
  - ☐ Based on industry-standard C.
  - ☐ Small set of extensions to enable heterogenous programming.
  - ☐ Straightforward APIs to manage devices and memory.

Table of Content

- Introduction
- **Heterogenous Computing**
- Blocks
- Threads
- Indexing
- Shared Memory and Sync Threads
- Device Management
- CUDA Demo

Heterogenous Computing

- **Hardware Terminologies**
  - ☐ *Host* : The CPU and its memory (host memory).
  - ☐ *Device* : The GPU and its memory (device memory).



Host

Device

- **Processing Flow**
  - ☐ Serial part of the code usually run on the ***Host***, but not necessary.
  - ☐ To utilize the GPU's full performance potential, the parallel part of the code is run over the *Device*.
  - ☐ Nvidia compiler nvcc is used to compile C/CUDA-based programs.

CUDA C Programming Sample

- **C Programming**

```
int main(void)  {
        printf("Hello World!\n“);
        return 0;
}
```

- **CUDA C Programming**

```
__global__ void newkernel(void) {
}

int main(void)  {
        newkernel<<<1,1>>>();
        printf("Hello World!\n“);
        return 0;
}
```

① ②

Called from the host and executed on the device

Memory Management

- **Simple CUDA APIs for handling device memory**
  - ❑ `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- **Allocate space for device copies assigned variables**
  - ❑ `cudaMalloc((void **)&a, size);`
- **Copy inputs to device**
  - ❑ `cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);`
- **Cleanup**
  - ❑ `cudaFree(d_a);`

Table of Content

- Introduction
- Heterogenous Computing
- **Blocks**
- Threads
- Indexing
- Shared Memory and Sync Threads
- Device Management
- CUDA Demo

Running in Parallel

- **The `add()` kernel**

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- **Instead of executing `add()` once, execute N times in parallel on the device**

```
        add<<< 1, 1 >>>();

        add<<< N, 1 >>>();
```

Block-based Parallelism

- **With `add()` running in parallel we can do vector addition**

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- **By using `blockIdx.x` to index into the array, each block handles a different element of the array**
- **On the device, each block can execute in parallel:**

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

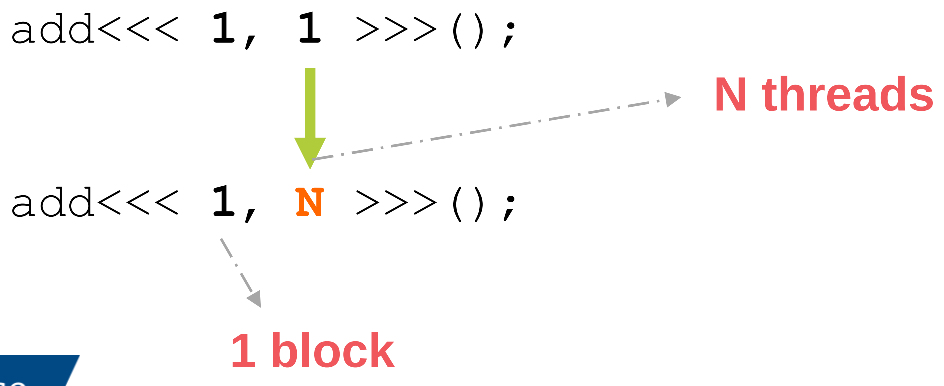
Table of Content

CUDA Threads

- **A block can be split into parallel threads**
- **Let's change `add()` to use parallel *threads* instead of parallel *blocks***

```
__global__ void add(int *a, int *b, int *c) {
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- **We use `threadIdx.x` instead of `blockIdx.x`**
- **We can execute `add()` with N threads**



```
add<<< 1, 1 >>>();

add<<< 1, N >>>();
```

Table of Content

Indexing using Blocks and Threads

- **Let’s adapt `add()` to use both *threads* and *blocks* at the same time**
- **Use the built-in variable `blockDim.x` for threads per block**

```
__global__ void add(int *a, int *b, int *c) {
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        c[index] = a[index] + b[index];
}
```

- **For proper indexing, changes need to be made in `main()`**

```
#define N (256 * 256)
#define THREADS_PER_BLOCK 16
add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>();
```

Table of Content

Cooperating Threads

- **Use `__shared__` to declare a variable or array in shared memory**
  - **Data is shared between threads in a block**
  - **Not visible to threads in other blocks**
- **Use `__syncthreads()` as a barrier**
  - **Used to prevent data hazards**

Implementation with Shared Memory

- **Example:**

```
#define BLOCK_SIZE 16
__global__ void add(int *a, int *b, int *c) {
        __shared__ int temp[BLOCK_SIZE];
        int index_1 = threadIdx.x + blockIdx.x * blockDim.x;
        int index_2 = threadIdx.x;
        //Read input elements into shared memory
        temp[index_1] = in[index_2];
}
        //Synchronize, ensure all the data is available
        __syncthreads();
```

Table of Content

Coordinating Host and Device

- **Kernel launches are asynchronous**
  - **Control returns to the CPU immediately**
- **CPU needs to synchronize before consuming the results**

`cudaMemcpy()`	- Blocks the CPU until the copy is complete. - Copy begins when all preceding CUDA calls have completed.
`cudaMemcpyAsync()`	- Asynchronous, does not block the CPU.
`cudaDeviceSynchronize()`	- Blocks the CPU until all preceding CUDA calls have completed.

Device Management

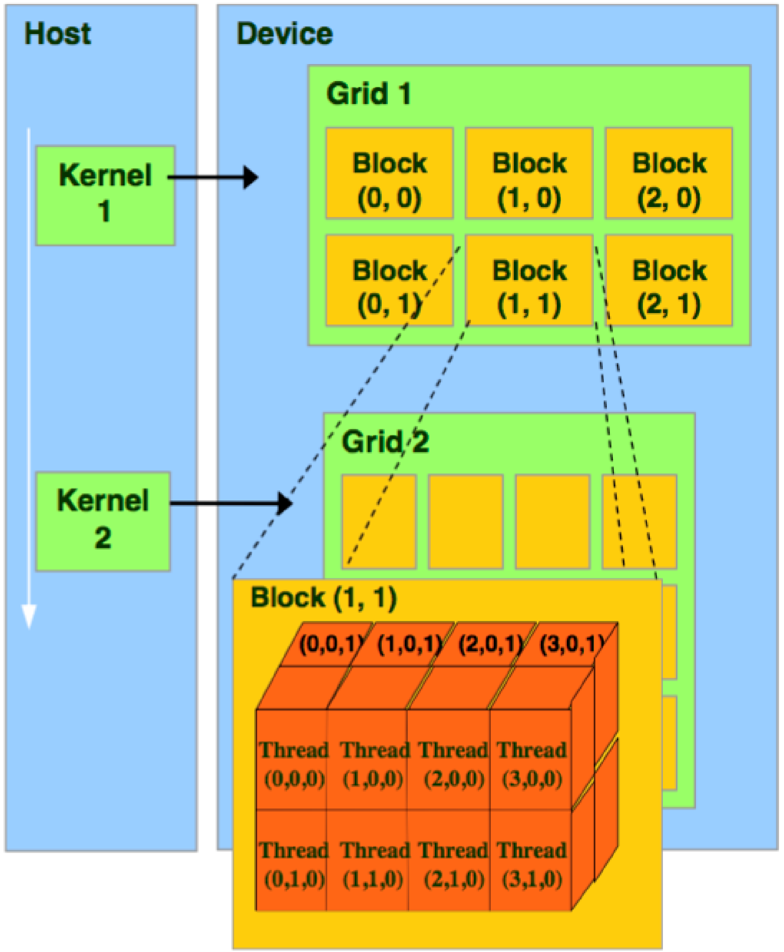
- **Application can query and select GPUs**
  - `cudaGetDeviceCount(int *count)`
  - `cudaSetDevice(int device)`
  - `cudaGetDevice(int *device)`
  - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
- **Multiple host threads can share a device**
- **A single host thread can manage multiple devices**
  - `cudaSetDevice(i)` to select the current device
  - `cudaMemcpy(…)` for peer-to-peer copies

Table of Content

Demo

- **Grids**
  - **Blocks**
    - **Threads**

- **We prepared 2 kernels for adding and multiplying 2 matrices**
- **Compare performance of running these 2 kernels on a GPU vs. CPU**
  - `gpu_matrixadd(int *a, int *b, int *c, int N)`
  - `gpu_matrixmult(int *a, int *b, int *c, int N)`



GPU Hardware General Architecture

Thank you!