GPU Series – IV

CUDA COMPILATION PROCESS

Device Side

CUDA provides its own functions for managing memory on the GPU. These functions are similar to the standard C **malloc** and **free** functions but are designed to work with the device's global memory.

cudaMalloc(): Allocates memory on the device.

```
Syntax:

cudaError_t cudaMalloc(void** devPtr, size_t size);

void **devPtr – Address of a pointer to the allocated object(Cast to ‘void**’)
size – Size of the allocated object in bytes.
```

void **devPtr – Address of a pointer to the allocated object(Cast to ‘void**’)
size – Size of the allocated object in bytes.

cudaFree(): Frees previously allocated memory.

```
Syntax:

cudaError_t cudaFree(void* devPtr);

void **devPtr – Pointer to the memory to be freed.
```

void **devPtr – Pointer to the memory to be freed.

Understanding cudaMalloc and cudaFree:

The first parameter to ***cudaMalloc*** is the address of a pointer that will be set to point to the allocated memory on the device. The address of this pointer should be cast to ***void**** because cudaMalloc expects a generic pointer. This allows the function to write the address of the allocated memory into the pointer variable. The second parameter specifies the **size** of the data to be allocated in bytes, similar to the traditional malloc function in C.

Dereferencing a device memory pointer in host code can cause exceptions or other run-time errors. Therefore, once the host code has allocated device memory, it must manage data transfers between the host and device using **cudaMemcpy**.

cudaMemcpy(): Transfers data between host and device memory.

```
Syntax:
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind);
```

dst – Pointer to the destination.
src- Pointer to the source.
count – Number of bytes to copy.
kind - Type/direction of transfer

Direction of transfer:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Why do we need to transfer the data?

The CPU and GPU have separate memory spaces. The CPU operates in the system's main memory (RAM), while the GPU has its own dedicated memory. To utilize the GPU for computations, data must be explicitly transferred from the CPU's memory to the GPU's memory.

When you allocate memory on the GPU using cudaMalloc, this memory is not directly accessible by the CPU. Similarly, memory allocated on the CPU is not directly accessible by the GPU. cudaMemcpy is used to copy data from one memory space to the other, enabling the CPU to send data to the GPU for processing and retrieve the results after computation.

What is the difference between malloc and cudaMalloc, free and cudaFree()?

malloc is used to allocate memory in the main memory (RAM) of the computer, which the CPU can directly access. This function is essential for programs running on the CPU to dynamically allocate memory during their execution. When you use malloc, the CPU can easily read from and write to this memory without any additional steps.

cudaMalloc: cudaMalloc allocates memory on the GPU's memory, specifically designed for the GPU to use. This is crucial for programs running on the GPU, allowing them to have their own dedicated memory space. Unlike malloc, memory allocated with cudaMalloc cannot be directly accessed by the CPU. Instead, data must be transferred between the CPU and GPU to share information.

free: free is used to release memory that was previously allocated by malloc on the CPU. Once the CPU program no longer needs this memory, free returns it to the system, making it available for other programs to use. This helps manage memory efficiently and prevents memory leaks in CPU applications.

cudaFree: cudaFree serves a similar purpose for the GPU. It frees memory that was allocated by cudaMalloc on the GPU. When a GPU program no longer requires the memory, cudaFree releases it, ensuring that the GPU's memory resources are managed efficiently and preventing memory leaks in GPU applications.

malloc and free handle memory in the ***CPU's*** RAM, whereas **cudaMalloc and cud**aFree manage memory in the ***GPU's*** memory. This distinction is vital because the CPU and GPU have separate memory spaces that require different management techniques.

malloc and free are used in the context of ***CPU programs***. When writing software that runs on the CPU, these functions help manage memory dynamically. On the other hand, **cudaMalloc and cudaFree** are used in the ***context of GPU programs***. They are essential for managing the memory needs of software that runs on the GPU.

Memory allocated by **malloc** is directly accessible by the **CPU**. This means the CPU can easily manipulate this memory as needed. In contrast, memory allocated by **cudaMalloc** is directly accessible by the **GPU**. For the **CPU** to access this memory, data transfers between the **CPU** and **GPU** are necessary, adding an extra step in the process.

To summarize the differences:

malloc and free: Think of malloc and free as using a regular storage closet in your home. You can store and retrieve items there whenever you need them, without any hassle. This is how the CPU works with memory allocated by malloc.

cudaMalloc and cudaFree: Imagine cudaMalloc and cudaFree as having a special storage unit located across town. You can store items there, but you need to make special trips to access them from your home. This represents the additional steps needed for the CPU and GPU to share memory allocated by cudaMalloc.

A simple usage of cudaMalloc and cudaFree:

```
float *device_variable1;
int size = n * sizeof(float)
cudaMalloc((void**)&device_variable, size);
…
…
cudaFree(device_variable);
```

Kernel Functions and Threading in CUDA

In CUDA, a kernel is a function that runs on the GPU. To execute this function from the host (CPU) code, we use a ***stub function***. Stub functions are essentially the host-side interface to launch the kernels on the GPU. They are not explicitly defined by the programmer but are automatically created by the CUDA compiler during the compilation process.

When a program's host code launches a kernel, the CUDA runtime system generates a grid of threads that are organized into a two-level hierarchy. This hierarchical organization helps manage and execute parallel tasks efficiently on the GPU.

Each grid is organized as an array of thread blocks, referred to as blocks. All blocks in a grid are of the same size, and each block can contain up to 1024 threads. The number of threads per block is specified by the host code when a kernel is launched. This allows the same kernel to be launched with different numbers of threads at various parts of the host code, depending on the specific needs of the computation.

For a grid, the number of threads in a block is available in a built-in variable called blockDim. The blockDim variable is a structure with three unsigned integer fields: x, y, and z. This structure allows programmers to organize threads into 1D, 2D, or 3D configurations, reflecting the dimensionality of the data they are working with.

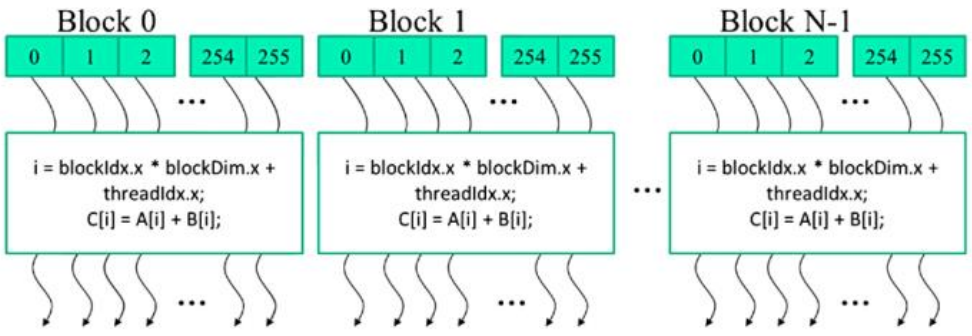

FIGURE 2.11

All threads in a grid execute the same kernel code.

Img Src: Programming Massively Parallel Processors by David B.Kirk, Wen-mei W.Hwu

Example of 1D Thread Block Configuration

The figure illustrates a grid where each thread block is organized as a 1D array of threads, suitable for 1D vector data. Here, the value of blockDim.x specifies the total number of threads in each block, which is 256 in this case.

The number of threads in each dimension of a thread block should be a multiple of 32 due to hardware efficiency reasons.

Thread Indexing

CUDA provides two other built-in variables for thread indexing:

- **threadIdx**: This gives each thread a unique coordinate within a block. For instance, in the above figure, only threadIdx.x is used. The first thread in each block has a value of 0 in threadIdx.x, the second thread has a value of 1, and so on.
- **blockIdx**: This gives all threads in a block a common block coordinate. In the diagram, all threads in the first block have a value of 0 in their blockIdx.x variables, those in the second block have a value of 1, and so forth.

Each thread can combine its **threadIdx** and **blockIdx** values to create a unique global index for itself within the entire grid.

Each thread can calculate its unique global index within the entire grid by combining its **threadIdx** and **blockIdx** values. This is done using the formula:

int i = blockIdx.x * blockDim.x + threadIdx.x;

<u>Understanding threadIdx.x and blockIdx.x:</u>

Imagine you are organizing a large library of books. The library is so big that it needs to be divided into sections and each section into shelves. Here's how we can use this analogy to understand threadIdx.x and blockIdx.x in CUDA.

1. **Library**: Represents the entire GPU memory space where you want to perform computations.
2. **Sections (Blocks)**:
    - Each section in the library represents a block in CUDA.
    - The sections are numbered sequentially starting from 0. This numbering is similar to the blockIdx.x in CUDA.
    - For example, the first section has a blockIdx.x of 0, the second section has a blockIdx.x of 1, and so on.
3. **Shelves (Threads)**:
    - Each shelf within a section represents a thread in a block.
    - Shelves are also numbered sequentially within each section, starting from 0. This is akin to threadIdx.x in CUDA.
    - For example, the first shelf in any section has a threadIdx.x of 0, the second shelf has a threadIdx.x of 1, and so forth.

    Putting it All Together
    When you are looking for a specific book in the library, you need to know both the section number and the shelf number. Similarly, in CUDA, to identify a specific thread, you need to know both the blockIdx.x and the threadIdx.x.

- **Global Index Calculation**:
    - Just as you combine the section number and the shelf number to find a specific book, in CUDA, you combine blockIdx.x and threadIdx.x to get a unique global index for each thread.
    - The global index is calculated using the formula:

int globalIndex = blockIdx.x * blockDim.x + threadIdx.x;

- This formula helps you determine the exact position of the thread in the entire grid of threads, much like how you would find the exact location of a book in the entire library.

Example(As seen in Figure 2.11)

- Suppose each section (block) can hold 256 shelves (threads).
- If you are in section 2 (blockIdx.x = 2) and looking at shelf 10 (threadIdx.x = 10):
  - The global index of the book would be:

int globalIndex = 2 * 256 + 10 = 522;

- This means the book is at position 522 in the entire library.

In this analogy:

- **Library**: Entire GPU memory space.
- **Sections (Blocks)**: Blocks in CUDA, identified by blockIdx.x.
- **Shelves (Threads)**: Threads within a block, identified by threadIdx.x.
- **Global Index**: Combination of section and shelf number, calculated as blockIdx.x * blockDim.x + threadIdx.x.

By launching a kernel of large number of threads, one can process vectors of large numbers.

It beings with ***__global__***, indicating that the function is a kernel and that it can be called from host function to generate a grid of threads on a device.

There are 3 CUDA C keywords for function declaration:

__ device__ - ***float DeviceFunc()*** – It is executed on device and can only be called from device.

__global__ - ***void KernelFunc()*** – It is executed on device and can only be called from host.

__ host__ - ***float HostFunc()*** – It is executed on host and can only be called from host.

By default, all function in CUDA program are host functions if they don't have any CUDA keywords in their declaration.

All threads execute the same kernel code but different threads will see different values for threadIdx.x, blockIdx.x and blockDim.x variables. In a CUDA kernel function, automatic variables are private to each thread. Each thread in the grid corresponds to one iteration of the original loop.

When the host code launches a kernel, it sets the grid and thread block dimensions via execution configuration parameters.

To launch a kernel, you need to specify the number of blocks and the number of threads per block using the triple angle bracket syntax <<< >>>.

Example:

```
int blockSize = 256;  // Number of threads per block
int gridSize = (N + blockSize - 1) / blockSize;  // Number of blocks
kernelFunc<<<gridSize, blockSize>>>(parameters);
```

This configuration ensures that there are enough threads to process the entire data set. The formula (N + blockSize - 1) / blockSize calculates the number of blocks needed, rounding up to ensure all data is covered.

Summarizing the entire code execution on device side:

- Allocate memory on the GPU using `cudaMalloc`.
- Transfer data from the host to the GPU using `cudaMemcpy`.
- Configure and launch the kernel with the appropriate number of blocks and threads.
- Transfer results back to the host using `cudaMemcpy`.
- Free the GPU memory using `cudaFree`.

Coding the CUDA code for vectorAddition……

~~ To be Continued ~~

Reference: Programming Massively Parallel Processors by David B.Kirk, Wen-mei W.Hwu