GPU Series – III

CUDA COMPILATION PROCESS

Host Side

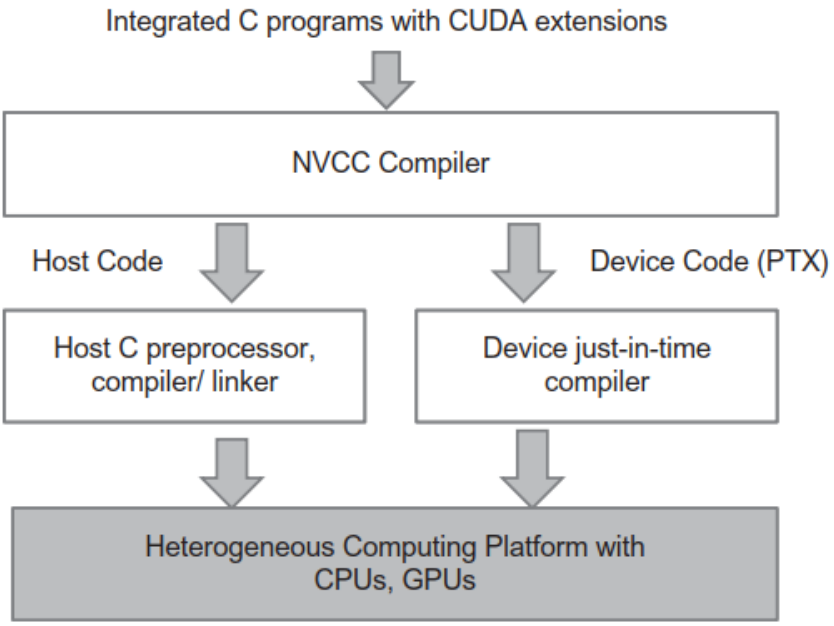

FIGURE 2.3

Overview of the compilation process of a CUDA C Program.

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

Each CUDA source file can have a mixture of both host and device code. Here's a brief overview of the compilation process.

Some terminology before we dive deeper

NVCC – NVIDIA C Compiler is a CUDA C compiler.
Host – CPU, **Device** – GPU.
Host Code: The portion of the CUDA program that runs on the CPU. It is traditional C code.
Device Code: The portion of the CUDA program that runs on the GPU.
Kernel: A function written in CUDA that runs on the GPU and is executed by multiple threads in parallel.
PTX (Parallel Thread Execution): An intermediate representation of the device code, generated by the NVCC.
JIT Compiler (Just-In-Time Compiler): A compiler that converts PTX code into binary code that can be executed by the GPU.
Grid: A collection of thread blocks that execute a kernel function on the GPU.
Thread Block: A group of threads that execute together and can share data through shared memory.
Global Memory: The main memory on the GPU, accessible by all threads.
Shared Memory: Fast memory accessible by all threads within a block.
Registers: The fastest type of memory available to each thread for temporary storage.
Constant and Texture Memory: Special types of read-only memory optimized for specific access patterns.

Source Code Structure: A CUDA source file typically contains both host and device code. Host code is written in standard C/C++ and runs on the CPU, while device code, which includes kernel functions, is marked with CUDA-specific keywords and runs on the GPU.

NVCC Compilation

The NVIDIA CUDA Compiler (NVCC) processes the CUDA source file, separating the host and device code. The host code is compiled using a standard C/C++ compiler, while the device code is compiled into PTX code.

Host Compilation:

- The host code is compiled by a standard C preprocessor, compiler, and linker.
- This part of the code remains largely unchanged from traditional C/C++ programs.

Device Compilation:

- The device code is compiled into PTX, an intermediate assembly-like language that represents the device code.
- PTX code is designed to be compatible across different GPU architectures, allowing for flexibility and optimization.

Host Code Compilation

The host code is compiled using the host C/C++ compiler. This process involves preprocessing, compiling, and linking to generate the host executable. The compiled host code includes calls to CUDA runtime functions that manage the device code execution.

PTX to Binary Translation

Once the PTX code is generated, it is further compiled into binary code by a Just-In-Time (JIT) compiler at runtime. This step is specific to the target GPU architecture, optimizing the code for the particular hardware it will run on.

Linking and Executable Generation

The host executable, which contains both the host code and the embedded PTX code, is generated. This executable can manage and launch the device code on the GPU.

Execution Flow of CUDA Program

1. **Memory Allocation**:
   - Allocate memory on the GPU for the data required by the computation.
   - This involves using CUDA runtime functions like cudaMalloc.
2. **Data Transfer**:
   - Transfer data from the CPU (host) memory to the GPU (device) memory using functions like cudaMemcpy.
3. **Kernel Launch**:
   - Launch a kernel, specifying the number of threads and blocks. The kernel runs on the GPU, with each thread executing a part of the computation.
   - The kernel launch syntax includes configuration parameters defining the grid and block dimensions.
4. **Device Synchronization**:
   - Ensure that all threads complete execution before proceeding. This is achieved using functions like ***`cudaDeviceSynchronize`***.

5. **Data Transfer Back**:

   - Transfer the results from the GPU memory back to the CPU memory.
   - This again uses `cudaMemcpy`.

6. **Memory Deallocation**:

   - Free the allocated memory on the GPU to avoid memory leaks. This involves using `cudaFree`.

Execution of Device code Summarized:

- When a kernel function (parallel device code) is called or launched, it is executed by a large number of threads on a device.
- All the threads that are generated by a kernel launch are collectively called a grid.
- These threads are the primary vehicle of parallel execution in a CUDA platform.
- When all threads of a kernel complete their execution, the corresponding grid terminates, the execution continues on the host until another kernel is launched.

CPU Execution and GPU execution do not overlap.

What is a threads? – https://cthecosmos.com/2022/10/26/threads/

Let us follow through this by an example of a vector addition kernel.

Vector addition kernel in CUDA is equivalent to "Hello World" in sequential programming.

Traditional vector addition:
Part 1: Allocate memory for vectors A, B, and C in the host (CPU) memory. Initialize the vectors A and B with the input values.
Part 2: Iterate through each element of the vectors A and B. Compute the sum of corresponding elements and store the result in vector C.
Part 3: The result vector C is now available in the host memory for further processing or output.

CUDA Vector Addition:
Part 1: Allocate space in device memory to hold copies of A, B and C vectors and copies the vectors from the host.
Part 2: Launches parallel execution of the actual vector additional kernel on the device.
Part 3: Copies the sum vector C from device memory back to host memory and frees the vectors in device memory.

The DRAM on any GPU is the global memory. In order to execute a kernel on a device, the programmer needs to allocate global memory on the device and transfer the needed data from host to allocated device memory.

In this article, we will see how the vector addition is done normally. In the upcoming one, we will dive into CUDA code for Vector addition kernel.

Traditional Vector Addition (***Using Static Memory***):

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>

// #define DISPLAY

#define SIZE 1000000

void initVectors(int *pointerToVectorArray){

    for(int i=0;i<SIZE;i++)
    {
```

```
        *(pointerToVectorArray+i) = rand();
    }


}

void addVectors(int *pointerForVectorA, int *pointerForVectorB, int *pointerForResultVector)
{
    for(int i=0;i<SIZE;i++)
    {
        *(pointerForResultVector + i) =  *(pointerForVectorA + i) + *(pointerForVectorB + i);
    }

}

void printVector(int *pointerToVectorArray)
{
    for(int i=0;i<SIZE;i++)

    {
        printf(" %d\t",*(pointerToVectorArray+i));

    }
    printf("\n");

}

int main(){

    // Allocate Memory for  VectorA and Vector B and initialize them with 0
    int vectorA[SIZE]={0};
    int vectorB[SIZE]={0};
    int resultVector[SIZE]={0};

    // For timing
    clock_t start, end;
    double cpu_time_used;
    srand(time(NULL));


    // Initialize Vector A and Vector B with Random values.
    initVectors(vectorA);
    initVectors(vectorB);


    printf("\nInitialized Vector A and Vector B with Random values\n");

    start = clock();

    // Vector Addition
    addVectors(vectorA,vectorB,resultVector);

    end = clock();


    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Execution time: %f seconds\n", cpu_time_used);

    //For Displaying Vectors
    #ifdef DISPLAY
    printf("\nVector A: \n");
```

```
    printVector(vectorA);

    printf("\nVector B: \n");
    printVector(vectorB);

    printf("\nResult Vector: \n");
    printVector(resultVector);
    #endif
    return 0;
}
```

Using Dynamic Memory:

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>

//#define DISPLAY

#define SIZE 100000000

void initVectors(int *pointerToVector){

    // Initialize values
    for(int i=0;i<SIZE;i++)
    {
        *(pointerToVector+i) = rand();
    }

}

void addVectors(int *pointerForVectorA, int *pointerForVectorB, int *pointerForResultVector)
{
    for(int i=0;i<SIZE;i++)
    {
        *(pointerForResultVector + i) =  *(pointerForVectorA + i) + *(pointerForVectorB + i);
    }

}

void printVector(int *pointerToVector)
{
    for(int i=0;i<SIZE;i++)

    {
        printf(" %d\t",*(pointerToVector+i));

    }
    printf("\n");

}

int main(){


    // Declare Vector A and Vector B
    int *vectorA=NULL;
    int *vectorB=NULL;
    int *resultVector=NULL;

    // For timing
    clock_t start, end;
    double cpu_time_used;
    srand(time(NULL)); // To see the random Dealy
```

```
    // Initialize Vector A and Vector B with Random values.

    // Allocate memory
    vectorA = (int*)malloc(SIZE*(sizeof(int)));
    vectorB = (int*)malloc(SIZE*(sizeof(int)));
    resultVector = (int*)malloc(SIZE*(sizeof(int)));


    initVectors(vectorA);
    initVectors(vectorB);


    printf("\nInitialized Vector A and Vector B with Random values\n");


    start = clock();

    // Vector Addition
    addVectors(vectorA,vectorB,resultVector);

    end = clock();



    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Execution time: %f seconds\n", cpu_time_used);

    //For Displaying Vectors
    #ifdef DISPLAY
    printf("\nVector A: \n");
    printVector(vectorA);

    printf("\nVector B: \n");
    printVector(vectorB);

    printf("\nResult Vector: \n");
    printVector(resultVector);
    #endif

    free(vectorA);
    free(vectorB);
    free(resultVector);

    return 0;
}
```

Why Static and Dynamic codes?

Parallel processing works for huge amount of datasets. For vector addition, if we use arrays, we would be using stack, which would be limited to a certain number on any device. If we use Dynamic memory, i.e, using pointers, we would be using heap, whose area is much more greater than Stack.

Output of Stack :

For 1000:

```
Initialized Vector A and Vector B with Random values
Execution time: 0.000003 seconds
```

For 10000

```
Initialized Vector A and Vector B with Random values
Execution time: 0.000024 seconds
```

For 100000

```
Initialized Vector A and Vector B with Random values
Execution time: 0.000167 seconds
```

For 1000000

```
Segmentation fault
```

For Dynamic:

For 1000:

```
Initialized Vector A and Vector B with Random values
Execution time: 0.000050 seconds
```

For 10000

```
Initialized Vector A and Vector B with Random values
Execution time: 0.000058 seconds
~
```

For 100000

```
Initialized Vector A and Vector B with Random values
Execution time: 0.000438 seconds
~
```

For 1000000

```
Initialized Vector A and Vector B with Random values
Execution time: 0.004210 seconds
~
```

For 100000000

```
Initialized Vector A and Vector B with Random values
Execution time: 0.442185 seconds
~
```

Now you see why we will be comparing it with Dynamic memory.

~~ To be Continued ~~

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