GPU Series – V

CUDA COMPILATION PROCESS

Device Side Code

CUDA code involves compiling both host (CPU) and device (GPU) code. The compilation process for CUDA programs is handled by nvcc, the NVIDIA CUDA compiler driver.

This article describes about the CUDA code and CUDA code structure.

The following is a brief overview of the compilation stages for CUDA:

- **Preprocessing:** The preprocessor handles macros, file inclusions (#include), and other preprocessing directives. This stage results in a preprocessed source file.

- **Compilation:** In this stage, the preprocessed source code is compiled into intermediate representation (IR) or assembly code. For host code, this might be standard x86 assembly, while for device code, it's PTX (Parallel Thread Execution) or SASS (Streaming Assembler).

- **Device Linking:** The compiled device code (PTX) is then linked together. This may involve combining multiple kernels or linking with other device libraries.

- **Host Linking:** The host code is compiled and linked. The device code is embedded within the host executable. This stage results in an executable that contains both the CPU and GPU code.

CUDA Code Structure

A CUDA program involves the following steps:

1. **Memory Allocation:** Allocate memory on the GPU for the data needed by the computation.
2. **Memory Transfer:** Transfer data from the host (CPU) memory to the device (GPU) memory.
3. **Kernel Launch:** Launch a CUDA kernel (a function executed on the GPU) with a specified configuration (number of threads and blocks).
4. **Memory Transfer:** Transfer the results from the device memory back to the host memory.
5. **Memory Deallocation:** Free the allocated GPU memory.

To Compile Run a CUDA code, we use: **nvcc programName.cu** and the executable file name.

Code:

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

//#define DISPLAY

#define SIZE 99999999


// This function computes the sum C=A+B. The vector Addition.
// Each thread performs one pair wise addition.
__global__

void vecAddKernel(int *a, int *b, int *c)
{
    int i= blockDim.x * blockIdx.x + threadIdx.x;
    if(i<SIZE)
        c[i]=a[i]*b[i];
}


__global__
void initVectors(int *vectorA, int *vectorB, unsigned long int seed)
{
    int i= blockDim.x * blockIdx.x + threadIdx.x;
    if(i<SIZE)
    {
        curandState state;
        curand_init(seed,i,0,&state);
        vectorA[i] = curand(&state) % 100;
        vectorB[i] = curand(&state) % 100;


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

void cudaAddVectors(int *pointerForVectorA, int *pointerForVectorB, int *pointerForResultVector)
{

        // Allocate Memory
        // Copy data from Host to Device
        // Launch kernel
        // Copy data from Device to Host
        // Free Memory

        int *device_VectorA= NULL;
        int *device_VectorB= NULL;
        int *device_resultVector= NULL;


    long unsigned int size = SIZE *sizeof(int);
    // Allocating memory

    cudaMalloc( (void **) &device_VectorA, size );
    cudaMalloc( (void **) &device_VectorB, size );
    cudaMalloc( (void **) &device_resultVector, size );


    int threadsPerBlock =256;
    int blocksPerGrid = (SIZE + threadsPerBlock -1)/ threadsPerBlock;

    unsigned long seed =time(NULL);
    initVectors<<<blocksPerGrid, threadsPerBlock>>>(device_VectorA, device_VectorB,seed);
    cudaDeviceSynchronize();
    // // Copy Data from Host To Device

    cudaMemcpy(pointerForVectorB, device_VectorB, size, cudaMemcpyDeviceToHost );
    cudaMemcpy( pointerForVectorA,device_VectorA, size, cudaMemcpyDeviceToHost );


    // Launch Kernel
    vecAddKernel<<<blocksPerGrid, threadsPerBlock>>>(device_VectorA, device_VectorB,
device_resultVector);
    cudaDeviceSynchronize();

    // Copy Data from Device to Host
    cudaMemcpy(pointerForResultVector, device_resultVector,size , cudaMemcpyDeviceToHost);

    // Free Memory

    cudaFree(device_VectorA);
    cudaFree(device_VectorB);
    cudaFree( device_resultVector);

}

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=(int *)malloc(SIZE * sizeof(int));;
    int *vectorB=(int *)malloc(SIZE * sizeof(int));;
    int *resultVector=(int *)malloc(SIZE * sizeof(int));;

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

    start = clock();

    // Vector Addition
    cudaAddVectors(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;
}
```

- ***#include <cuda_runtime.h>***: Includes the CUDA runtime library, which provides functions to manage device memory, launch kernels, and synchronize the device.
- ***#include <curand_kernel.h>***: Includes the CURAND library for random number generation on the GPU.

- ***#define SIZE 99999999***: Defines the size of the vectors.

Kernel Functions:

```
void vecAddKernel(int *a, int *b, int *c)
{
   int i= blockDim.x * blockIdx.x + threadIdx.x;
   if(i<SIZE)
      c[i]=a[i]*b[i];
}
```

- ***__global__:*** This qualifier specifies that vecAddKernel is a CUDA kernel function that runs on the GPU and can be called from the host.
- ***int i = blockDim.x * blockIdx.x + threadIdx.x***;: Calculates the global thread index i. Each thread handles one element of the vectors.

  - ***blockDim.x***: Number of threads per block.
  - ***blockIdx.x:*** Block index.
  - ***threadIdx.x***: Thread index within the block.
  - ***if (i < SIZE) c[i] = a[i] * b[i];***: Performs the vector addition for the element at index i. The check ensures that the thread index does not exceed the vector size.

```
__global__
void initVectors(int *vectorA, int *vectorB, unsigned long int seed)
{
   int i= blockDim.x * blockIdx.x + threadIdx.x;
   if(i<SIZE)
   {
      curandState state;
      curand_init(seed,i,0,&state);
      vectorA[i] = curand(&state) % 100;
      vectorB[i] = curand(&state) % 100;


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

- ***curandState state;***: Declares a CURAND state for random number generation.
- ***curand_init(seed, i, 0, &state);***: Initializes the CURAND state with the seed and thread index.
- ***vectorA[i] = curand(&state) % 100; vectorB[i] = curand(&state) % 100;***: Generates random values for vectorA and vectorB using the CURAND state.

```

void cudaAddVectors(int *pointerForVectorA, int *pointerForVectorB, int *pointerForResultVector)
{

      // Allocate Memory
      // Copy data from Host to Device
      // Launch kernel
      // Copy data from Device to Host
      // Free Memory

      int *device_VectorA= NULL;
      int *device_VectorB= NULL;
      int *device_resultVector= NULL;


   long unsigned int size = SIZE *sizeof(int);
   // Allocating memory

   cudaMalloc( (void **) &device_VectorA, size );
   cudaMalloc( (void **) &device_VectorB, size );
   cudaMalloc( (void **) &device_resultVector, size );


   int threadsPerBlock =256;
   int blocksPerGrid = (SIZE + threadsPerBlock -1)/ threadsPerBlock;

   unsigned long seed =time(NULL);
   initVectors<<<blocksPerGrid, threadsPerBlock>>>(device_VectorA, device_VectorB,seed);
   cudaDeviceSynchronize();
   // // Copy Data from Host To Device

   cudaMemcpy(pointerForVectorB, device_VectorB, size, cudaMemcpyDeviceToHost );
   cudaMemcpy( pointerForVectorA,device_VectorA, size, cudaMemcpyDeviceToHost );


   // Launch Kernel
   vecAddKernel<<<blocksPerGrid, threadsPerBlock>>>(device_VectorA, device_VectorB,
device_resultVector);
   cudaDeviceSynchronize();

   // Copy Data from Device to Host
   cudaMemcpy(pointerForResultVector, device_resultVector,size , cudaMemcpyDeviceToHost);

   // Free Memory

   cudaFree(device_VectorA);
   cudaFree(device_VectorB);
   cudaFree( device_resultVector);

}
```

- **Memory Allocation on the Device:**

  - *cudaMalloc((void**)&device_VectorA, size);*
  - *cudaMalloc((void**)&device_VectorB, size);*
  - *cudaMalloc((void**)&device_resultVector, size);*
  - Allocates memory for vectorA, vectorB, and the result vector on the GPU.

Kernel Configuration:

- *int threadsPerBlock = 256;:* Specifies the number of threads per block.
- *int blocksPerGrid = (SIZE + threadsPerBlock - 1) / threadsPerBlock*;: Calculates the number of blocks needed.

Vector Initialization:

- *unsigned long seed = time(NULL)*;: Generates a seed for the random number generator.
- *initVectors<<<blocksPerGrid, threadsPerBlock>>>(device_VectorA, device_VectorB, seed);:* Initializes the vectors on the device.
- *cudaDeviceSynchronize();:* Synchronizes the device, ensuring all preceding CUDA calls are complete.

Copy Data from Device to Host:

- *cudaMemcpy(pointerForVectorB, device_VectorB, size, cudaMemcpyDeviceToHost);*
- *cudaMemcpy(pointerForVectorA, device_VectorA, size, cudaMemcpyDeviceToHost);*
- Copies the initialized vectors back to the host for verification.

Vector Addition Kernel Launch:

- *vecAddKernel<<<blocksPerGrid, threadsPerBlock>>>(device_VectorA, device_VectorB, device_resultVector);*
- Launches the vector addition kernel on the GPU.
- *cudaDeviceSynchronize();:* Synchronizes the device again to ensure the kernel has completed execution.

Copy Result from Device to Host:

- *cudaMemcpy(pointerForResultVector, device_resultVector, size, cudaMemcpyDeviceToHost);*
- Copies the result vector from the device to the host.

Free Allocated Memory on the Device:

- *cudaFree(device_VectorA);*
- *cudaFree(device_VectorB);*
- *cudaFree(device_resultVector);*
- Frees the allocated GPU memory to avoid memory leaks.

```
int main(){

   // Declare Vector A and Vector B
   int *vectorA=(int *)malloc(SIZE * sizeof(int));;
   int *vectorB=(int *)malloc(SIZE * sizeof(int));;
   int *resultVector=(int *)malloc(SIZE * sizeof(int));;

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

```
    start = clock();

    // Vector Addition
    cudaAddVectors(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;
}
```

Memory Allocation on the Host:

- Allocates memory for vectorA, vectorB, and resultVector on the host.

Timing the Execution:

- *clock_t start, end;:* Variables to store the start and end times.
- *start = clock();:* Records the start time.
- *cudaAddVectors(vectorA, vectorB, resultVector);: Calls* the function to perform vector addition on the GPU.
- *end = clock();:* Records the end time.

Calculating Execution Time:

- *cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC*;: Calculates the execution time in seconds.
- *printf("Execution time: %f seconds\n", cpu_time_used);:* Prints the execution time.

Optional Vector Display:

- If DISPLAY is defined, the vectors vectorA, vectorB, and resultVector are printed for verification.

Free Allocated Memory on the Host:

- Frees the allocated host memory to avoid memory leaks.

This gives a brief idea about the basic CUDA code.

~~ To be Continued ~~

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