BASICS

18 May 2025 16:26

KERNEL

- A function written to run on GPU
- we call it once, but it runs many time in parallel, once for each thread

THREAD

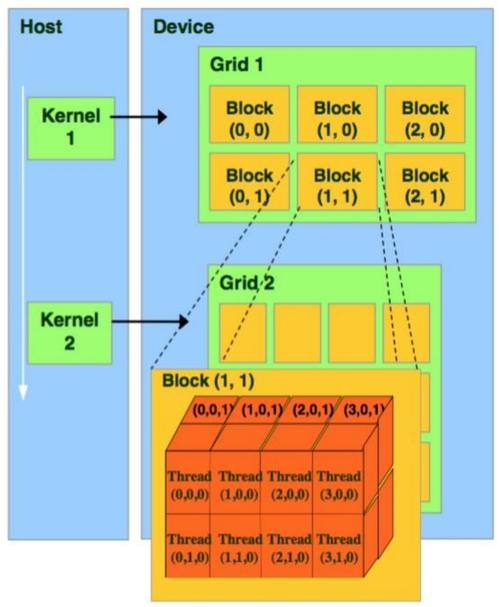
- Smallest unit of execution on the GPU
- Each thread executes one instance of the kernel function
- Threads have their own unique ID and private memory.

BLOCK

- A block is a group of threads
- Threads in a block can
  - → Share shared memory
  - → Synchronize with each other using barriers
- All threads in a block are scheduled together on one Streaming Multiprocessor of the GPU
- you might have 256 threads in a block, & they all can work together on a piece of the data

GRID

- A grid is a group of blocks
- The kernel launches a grid, which contains many blocks, and each block contains many threads.



Thread ID

Local Thread ID

int i = threadIdx.x

This is the thread ID within a block

Global Thread ID

for 1 Dimension int idx = blockIdx.x * blockDim.x + threadIdx.x ;

for 2 Dimension int x = blockIdx.x * blockDim.x + threadIdx.x ;
int y = blockIdx.y * blockDim.y + threadIdx.y

Similarly for 3 Dimensions

7.37. Execution Configuration

Any call to a `__global__` function must specify the *execution configuration* for that call. The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the device, as well as the associated stream (see CUDA Runtime for a description of streams).

The execution configuration is specified by inserting an expression of the form `<<< Dg, Db, Ns, S >>>` between the function name and the parenthesized argument list, where:

- `Dg` is of type `dim3` (see dim3) and specifies the dimension and size of the grid, such that `Dg.x * Dg.y * Dg.z` equals the number of blocks being launched;
- `Db` is of type `dim3` (see dim3) and specifies the dimension and size of each block, such that `Db.x * Db.y * Db.z` equals the number of threads per block;
- `Ns` is of type `size_t` and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory; this dynamically allocated memory is used by any of the variables declared as an external array as mentioned in `__shared__`; `Ns` is an optional argument which defaults to 0;
- `S` is of type `cudaStream_t` and specifies the associated stream; `S` is an optional argument which defaults to 0.

As an example, a function declared as

```
__global__ void Func(float* parameter);
```

must be called like this:

```
Func<<< Dg, Db, Ns >>>(parameter);
```

dim3 → integer vector type used to specify dimensions, any components left unspecified is initialized to 1

eg. 1-D configuration

Func <<< 4 , 256 >>> (parameter)

so we have a grid with 4 blocks
and each block has 256 threads
so we have ⇒ 4x256 threads in total

Total threads = 1024

now let's say if our array size is 1000
i.e. int N = 1000

then idx = 0 to 999 → do useful work
idx = 1000 to 1023 → must be ignored

for block 0 (blockIdx.x = 0)
idx = 0 x blockDim.x + threadIdx.x = 0 x 256 + threadIdx.x ⇒ 0 to 255

for block 1 (blockIdx.x = 1)
idx = 1 x 256 + threadIdx.x ⇒ 256 to 511

for block 2 ⇒ 512 to 767

for block 3 ⇒ 768 to 1023

So that's why we keep a check if(idx < N)

eg. 2-D Execution configuration

dim3 grid (16,16); // 16x16 = 256 blocks
dim3 block (32,32); // 32x32 = 1024 threads per block

Func <<< grid , block >>> (parameters)

CUDA Built-In Variables

Thread Level

Variable	Description	Value Range
threadIdx.x	thread index within the block in X-dimension	0 to blockDim.x − 1
threadIdx.y	" " " " " " Y-dimension	0 to blockDim.y − 1
threadIdx.z	" " " " " " Z-dimension	0 to blockDim.z − 1

Block Level

Variable	Description	Value Range
blockIdx.x	Block index within the grid in X dimension	0 to gridDim.x − 1
blockIdx.y	" " " " " " Y dimension	0 to gridDim.y − 1
blockIdx.z	" " " " " " Z dimension	0 to gridDim.z − 1

Dimensions

blockDim.x	No. of threads per block in X dimension
blockDim.y	" " " " " " Y dimension
blockDim.z	" " " " " " Z dimension
gridDim.x	No. of blocks in grid in X dimension
gridDim.y	" " " " " " Y dimension
gridDim.z	" " " " " " Z dimension

Vector Addition in CUDA

```cpp
#include<iostream>
#include<cuda_runtime.h>
using namespace std;

// my kernel
__global__ void MyFunc(float *A, float *B, float *C, int N){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx<N){
        C[idx] = A[idx] + B[idx];
    }
}

int main(){
    // declaring the size of the arrays
    int N = 1000;
    size_t size = N * sizeof(float);

    // creating arrays on host(cpu) and thus alloting the space as well
    // in c++
    // float *h_A = new float[N];
    // float *h_B = new float[N];
    // float *h_C = new float[N];

    // in c
    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C = (float*)malloc(size);

    // filling the arrays with values
    for(int i=0;i<N;i++){
        h_A[i] = i;
        h_B[i] = 2*i;
    }

    // allocating memory on devic(gpu)
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A,size);
    cudaMalloc(&d_B,size);
    cudaMalloc(&d_C,size);

    // copying arrays from host do device
    cudaMemcpy(d_A ,h_A ,size ,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B ,h_B ,size ,cudaMemcpyHostToDevice);

    // fixing threadsperblock to 256
    // and selecting value of blockspergrid such that its divisible
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1)/threadsPerBlock;

    // calling the kernel
    MyFunc<<<blocksPerGrid,threadsPerBlock>>>(d_A, d_B, d_C ,N);

    // copying the result from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cout<<"A + B = C"<<endl;

    // pritning the result
    for(int i=0;i<N;i++){
        cout<<h_A[i]<<" + "<<h_B[i]<<" = "<<h_C[i]<<endl;
    }

    // deallocation in c++
    // delete[] h_A;
    // delete[] h_B;
    // delete[] h_C;

    // in c
    free(h_A);
    free(h_B);
    free(h_C);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
```

```
Running NVIDIA GTX TITAN X in FUNCTIONAL mode...
Compiling...
Executing...
A + B = C
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
3 + 6 = 9
4 + 8 = 12
5 + 10 = 15
6 + 12 = 18
7 + 14 = 21
8 + 16 = 24
9 + 18 = 27
10 + 20 = 30
11 + 22 = 33
12 + 24 = 36
13 + 26 = 39
14 + 28 = 42
15 + 30 = 45
16 + 32 = 48
```

Snipping Tool

Screenshot copied to clipboard
Automatically saved to screenshots