【CUDA】grid、block、thread的关系及thread索引的计算_hujingshuang-CSDN博客

https://blog.csdn.net/hujingshuang/article/details/53097222

None Sun Nov, 22 01:24

由于项目需要用到GPU，所以最近开始学习CUDA编程模型，刚开始接触，先搞清楚线程关系和内存模型是非常重要的，但是发现书上和许多博客关于线程这些关系没讲明白，所以就着自己的理解，做点笔记，欢迎讨论。

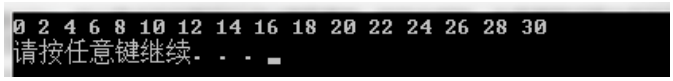
这篇文章针对于已经了解过了CUDA线程的相关知识，最好已经动手写过CUDA C的代码，而对并行线程感到迷惑，不知道怎么计算线程索引的读者，如果没接触过，那么先看看书，敲两段代码跑跑，如果你理解了那么恭喜你，如果还有疑惑，那么再来看看这篇文章，或许有帮助。

首先，认识一下线程。(⊙o⊙)…虽然画成这样总是感觉有点怪怪的，但是你应该已经见怪不怪了吧~

下面我们来看一段代码，其功能是对两个数组求和，并保存到另一个数组，很简单吧~

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <iostream>

using namespace std;

// 二：线程执行代码
__global__ void vector_add(float* vec1, float* vec2, float* vecres, int length) {
    int tid = threadIdx.x;
    if (tid < length) {
        vecres[tid] = vec1[tid] + vec2[tid];
    }
}

int main() {
    const int length = 16;                                      // 数组长度为16
    float a[length], b[length], c[length];                      // host中的数组
    for (int i = 0; i < length; i++) {                          // 初始赋值
        a[i] = b[i] = i;
    }
    float* a_device, *b_device, *c_device;                      // device中的数组

    cudaMalloc((void**)&a_device, length * sizeof(float));      // 分配内存
    cudaMalloc((void**)&b_device, length * sizeof(float));
    cudaMalloc((void**)&c_device, length * sizeof(float));

    cudaMemcpy(a_device, a, length * sizeof(float), cudaMemcpyHostToDevice);    // 将host数组的值拷贝给device数组
    cudaMemcpy(b_device, b, length * sizeof(float), cudaMemcpyHostToDevice);

    // 一：参数配置
    dim3 grid(1, 1, 1), block(length, 1, 1);                    // 设置参数
    vector_add<<<grid,block>>>(a_device, b_device, c_device, length);           // 启动kernel

    cudaMemcpy(c, c_device, length * sizeof(float), cudaMemcpyDeviceToHost);    // 将结果拷贝到host

    for (int i = 0; i < length; i++) {                          // 打印出来方便观察
        cout << c[i] << ' ';
    }
    cout << endl;

    system('pause');
    return 0;
}
```

运行结果：



结果是对的，也是我们所能预料到的。那么现在我们来分析代码中注释的**一、二**处究竟该怎么来写。

首先，我们要明白，上面的代码计算的是两个一维向量的和。由于数组大小是**16**，所以我们使用了**16**个线程来计算。

```
dim3 grid(1, 1, 1), block(length, 1, 1);                    // 设置参数
```

先说grid，在这段代码中，我们设置参数为线程格（grid）中只有一个一维的block，该block的x维度上有16个，这个应该一下就看出来啦。因为grid(x,y,z)中的x=1,y=1,z=1，即各个维度均为1，所以是一维的，数量为x*y*z=1*1*1=1。如果没明白，再看两个例子：

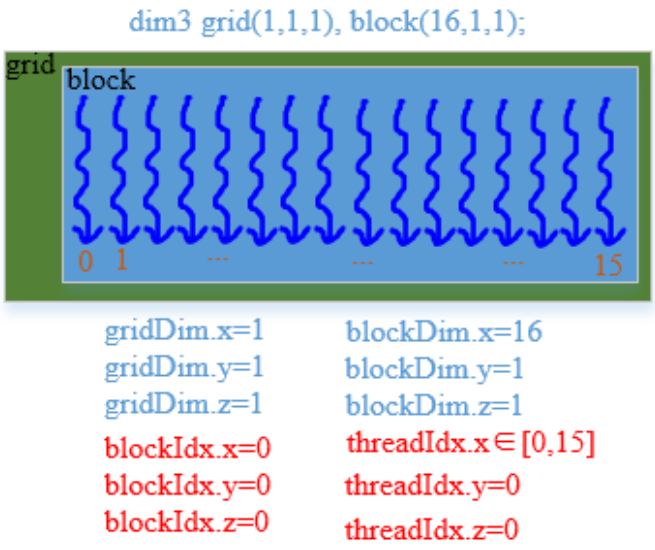
```
dim3 grid1(2, 1, 1); // x=2, y=1, z=1
dim3 grid2(4, 2, 1); // x=4, y=2, z=1
dim3 grid3(2, 3, 4); // x=2, y=3, z=4
```

可以知道，grid1是一维的（因为y,z维度是1），grid2是二维的（因为z维度是1），grid3是三维的，且grid1,grid2,grid3中分别有2、8、24个block。

同理，对于线程块（block），我们知道之前的代码中，block中存在16个线程，且该线程块维度是一维的，因为block(x,y,z)中x=length=16,y=1,z=1。

我画个图来帮助理解，大概就是这样子的：

```
dim3 grid(1, 1, 1), block(length, 1, 1);                    // 设置参数
```

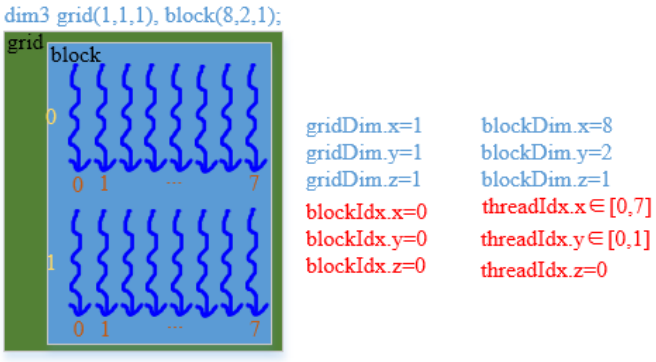


OK，我想这下应该就清楚了，就是一个一维的block（此处只有x维度上存在16个线程）。所以，内建变量只有一个在起作用，就是threadIdx.x，它的范围是[0,15]。因此，我们在计算线程索引是，只用这个内建变量就行了（其他的为0，写了也不起作用）：

```
// 二：线程执行代码
__global__ void vector_add(float* vec1, float* vec2, float* vecres, int length) {
    int tid = threadIdx.x;              // 只使用了threadIdx.x
    if (tid < length) {
        vecres[tid] = vec1[tid] + vec2[tid];
    }
}
```

OK，看到这里，你可能还是不大明白什么一维二维的，我们再来看一个：

```
dim3 grid(1, 1, 1), block(8, 2, 1);                    // 设置参数
```



根据上面的介绍，我们知道这个线程格只有一个一维的线程块，该线程块内的线程是二维的，x的维度为8，y的维度为2，共有8*2=16个线程，如果要用这16个线程来计算数组的累加，当然是可以的，但是我们这里需要改动一下线程执行代码中的索引计算方式了。

```
// 二：线程执行代码
__global__ void vector_add(float* vec1, float* vec2, float* vecres, int length) {
    int tid = threadIdx.y * blockDim.x +  threadIdx.x;  // 使用了threadIdx.x, threadIdx.x, blockDim.x
    if (tid < length) {
        vecres[tid] = vec1[tid] + vec2[tid];
    }
}
```

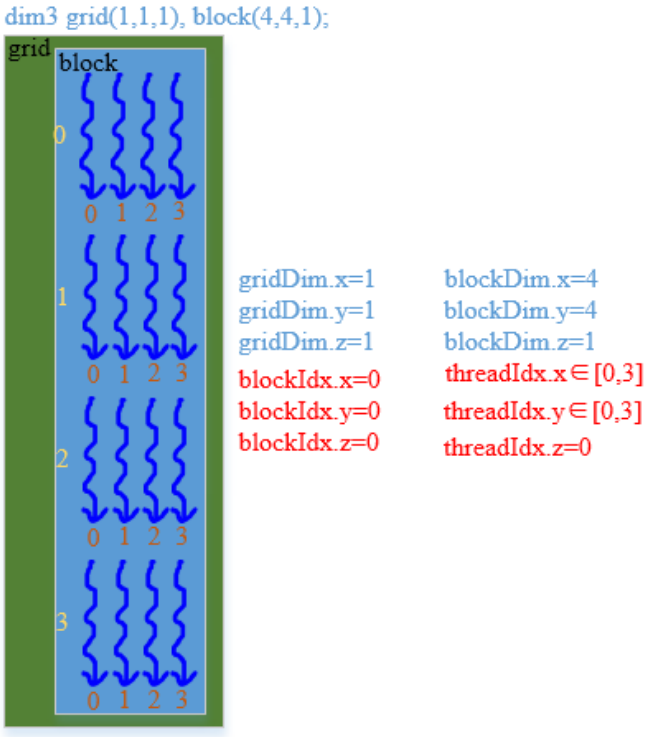
我们一定要有并行思想，这里有16个线程，kernel启动后，每个线程都有自己的索引号，比如某个线程位于grid中哪个维度的block（即blockIdx.x,block.y,block.z），又位于该block的哪个维度的线程（即threadIdx.x,threadIdx.y,threadIdx.z），利用这些线程索引号映射到对应的数组下

标，我们要做的工作就是将保证这些下标不重复（如果重复的话，那就惨了），最初那种一维的计算方式就不行了。因此，通过使用threadIdx，blockDim来进行映射（偏移）。blockDim.x=8,blockDim.y=2，如上代码。

其实，我感觉有些我不能用文字准确、清晰的描述出来，所以咯，我们再来一个例子吧，我相信，多看一看，多想一想就明白了。

```
dim3 grid(1, 1, 1), block(4, 4, 1);                    // 设置参数
```

我们将block改成上面的这样，其线程模型为下图：



当然，kernel函数的代码依然可以不用变动，这个应该想得清楚，还是再写一下吧。

```
// 二：线程执行代码
__global__ void vector_add(float* vec1, float* vec2, float* vecres, int length) {
    int tid = threadIdx.y * blockDim.x +  threadIdx.x;  // 使用了threadIdx.x, threadIdx.x, blockDim.x
    if (tid < length) {
        vecres[tid] = vec1[tid] + vec2[tid];
    }
}
```

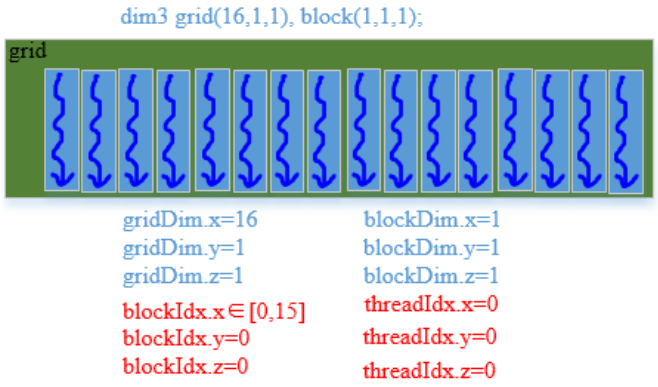
以上内容我们分别介绍了用一维和二维线程来计算一维数组的求和，实际上数组的维度与线程格、线程块和线程的维度并不是那么密不可分的，都可以组合实现，只不过在实现时，良好的参数配置对索引的计算很方便，而且由于grid、block、thread维度的限制，还有warpSize的限制，所以对于较大的数据量来说，我们应该做到心中有数，进行有效的块分解。

现在来看看二维的block，在整个文章中，我只讲解一维、二维的，因为三维的我不知道怎么画图啦，而且不好描述，免得误导大家。

还是上面的一维数组，长度为16。

```
dim3 grid(16, 1, 1), block(1, 1, 1);                    // 设置参数
```

先来个线程模型图，我想大家并不会感到惊讶，绿色的区域表示grid，蓝色的区域表示block，图中有一个grid和16个block，每个block都是一维，而且x维度上只有一个线程的：

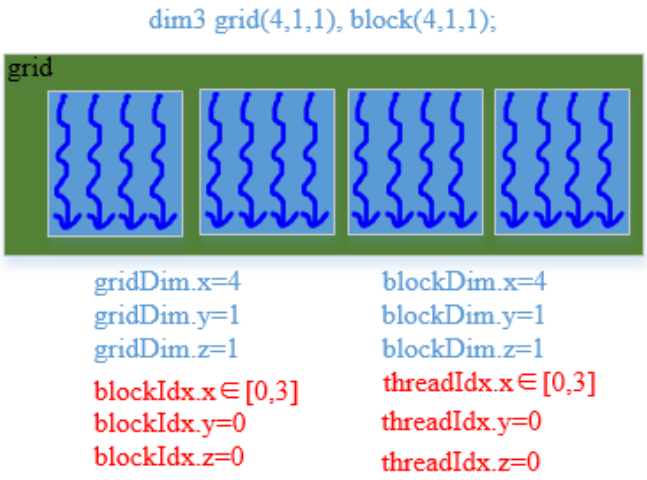


显然，我们的线程索引代码应该为如下：

```
// 二：线程执行代码
__global__ void vector_add(float* vec1, float* vec2, float* vecres, int length) {
    int tid = blockIdx.x;
    if (tid < length) {
        vecres[tid] = vec1[tid] + vec2[tid];
    }
}
```

或许你会有疑惑，那么我们再来看一个：

```
dim3 grid(4, 1, 1), block(4, 1, 1);
```



线程索引代码应该为如下：

```
// 二：线程执行代码
__global__ void vector_add(float* vec1, float* vec2, float* vecres, int length) {
    int tid = blockIdx.x * gridDim.x + threadIdx.x;
    if (tid < length) {
        vecres[tid] = vec1[tid] + vec2[tid];
    }
}
```

到现在为止，我觉得你应该有所领悟。如果还是不晓得的话，我想你应该认认真真的看图并动手分析了，图中的每一个块，每一个字都是有它的作用的，你不应该就此放弃。

我依然相信，能用图解决，就不哔哔。就好像你给一个人描述一座宫殿是多么多么的宏伟，富丽堂皇，他并不不会感冒。你就说，嘿大傻，给你瞧瞧我去欧洲玩的教堂，这是照片，不用多说，大傻自己就知道了。

比如，我描述说：

```
dim3 grid(2, 2, 1), block(2, 2, 1);
```

这样肯定不直观，那我再给你一幅示意图：

那么执行代码及索引计算如下：

```
// 二：线程执行代码
__global__ void vector_add(float* vec1, float* vec2, float* vecres, int length) {
    // 在第几个块中 * 块的大小 + 块中的x，y维度（几行几列）
    int tid = (blockIdx.y * gridDim.x + blockIdx.x) * (blockDim.x * blockDim.y) + threadIdx.y * blockDim.y + threadIdx.x;
    if (tid < length) {
        vecres[tid] = vec1[tid] + vec2[tid];
    }
}
```

上面的代码可能要复杂一点，但是你慢慢的会发现这很有趣。

到此，我想讲的就完了。当然对于二维的数组或是三维的数组，我想多看几个例子也就会有体会了。

这里还是忍不住要吐槽

骂人

一下内建变量threadIdx和blockIdx的命名了，每次看到这些内建变量其最后一个字母是x，就会给我一种误会是x维度上的

发火

，我觉得使用threadId和blockId是多么的良好

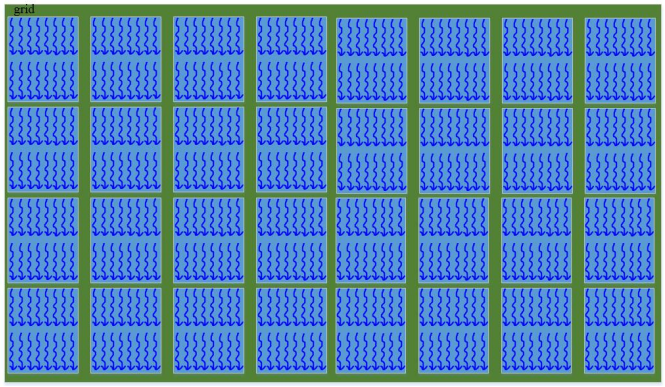
可怜

。当然，胜利的总是API一方，我也只能吐吐槽

快哭了

。

最后再来一发，我给个图，我们来倒推其参数及相关执行代码，如下：



由于上传图片大小限制，由BMP转成JPG格式的了，有点不清晰，但足够看了。

显然参数为：

```
dim3 grid(8, 4, 1), block(8, 2, 1);
```

共有**8*4*8*2=512**个线程，当然在CUDA编程中，这算很少的了。如果是一幅512x512大小的图像做加或点乘之类的运算，随随便便就是几十万的线程数了。
万变不离其宗，其一维的计算方式如下：

```
__global__ void vector_add(float* vec1, float* vec2, float* vecres, int length) {
    // 在第几个块中 * 块的大小 + 块中的x，y维度（几行几列）
    int tid = (blockIdx.y * gridDim.x + blockIdx.x) * (blockDim.x * blockDim.y) + threadIdx.y * blockDim.y + threadIdx.x;
    if (tid < length) {
        vecres[tid] = vec1[tid] + vec2[tid];
    }
}
```

再给出二维的：

```
__global__ void vector_add(float** mat1, float** mat2, float** matres, int width) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < width) {
        matres[x][y] = mat1[x][y] + mat2[x][y];
    }
}
```