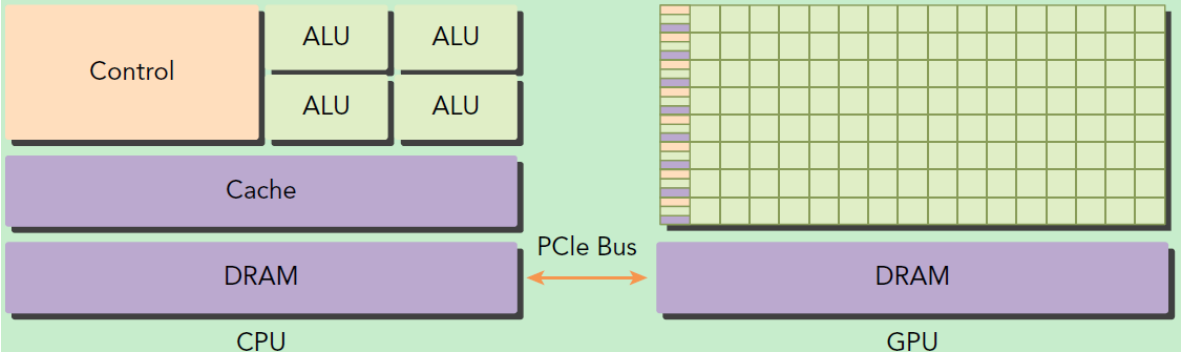
学习笔记1——向量矩阵相加

GPU并不是一个独立运行的计算平台，而需要与CPU协同工作，也可以把GPU看成是CPU的协处理器，因此当在说GPU并行计算时，其实是指的基于CPU+GPU的异构计算架构。在异构计算架构中，GPU与CPU通过PCIe总线连接在一起进行协同工作，CPU所在位置称为为主机端（host），而GPU所在位置称为设备端（device），如下图所示。



可以看到GPU包括更多的运算核心，其特别适合数据并行的计算密集型任务，如大型矩阵运算，而CPU的运算核心较少，但是其可以实现复杂的逻辑运算，因此其适合控制密集型任务。另外，CPU上的线程是重量级的，上下文切换开销大，但是GPU由于存在很多核心，其线程是轻量级的。因此，基于CPU+GPU的异构计算平台可以优势互补，CPU负责处理逻辑复杂的串行程序，而GPU重点处理数据密集型的并行计算程序，从而发挥最大功效。

在给出CUDA的编程实例之前，这里先对CUDA编程模型中的一些概念及基础知识做个简单介绍。CUDA编程模型是一个异构模型，需要CPU和GPU协同工作。在CUDA中，host和device是两个重要的概念，用host指代CPU及其内存，而用device指代GPU及其内存。CUDA程序中既包含host程序，又包含device程序，它们分别在CPU和GPU上运行。同时，host与device之间可以进行通信，这样它们之间可以进行数据拷贝。典型的CUDA程序的执行流程如下：

1) 分配host内存，并进行数据初始化；
2) 分配device内存，并从host将数据拷贝到device上；
3) 调用CUDA的 kernel 函数在device上完成指定的运算；
4) 将device上的运算结果拷贝到host上；
5) 释放device和host上分配的内存。

上面流程中最重要的一个过程是调用CUDA的 kernel 函数来执行并行计算，kernel是CUDA中一个重要的概念，kernel是在device上线程中并行执行的函数， kernel 函数用 `__global__` 符号声明，在调用时需要用 `<<<grid, block>>>` 来指定kernel要执行的线程数量，在CUDA中，每一个线程都要执行 kernel 函数，并且每个线程会分配一个唯一的线程号thread ID，这个ID值可以通过 kernel 函数的内置变量 `threadIdx` 来获得。

由于GPU实际上是异构模型，所以需要区分 host 和 device上 的代码，在CUDA中是通过函数类型限定词开区别 host 和 device 上的函数，主要的3个函数类型限定词如下：

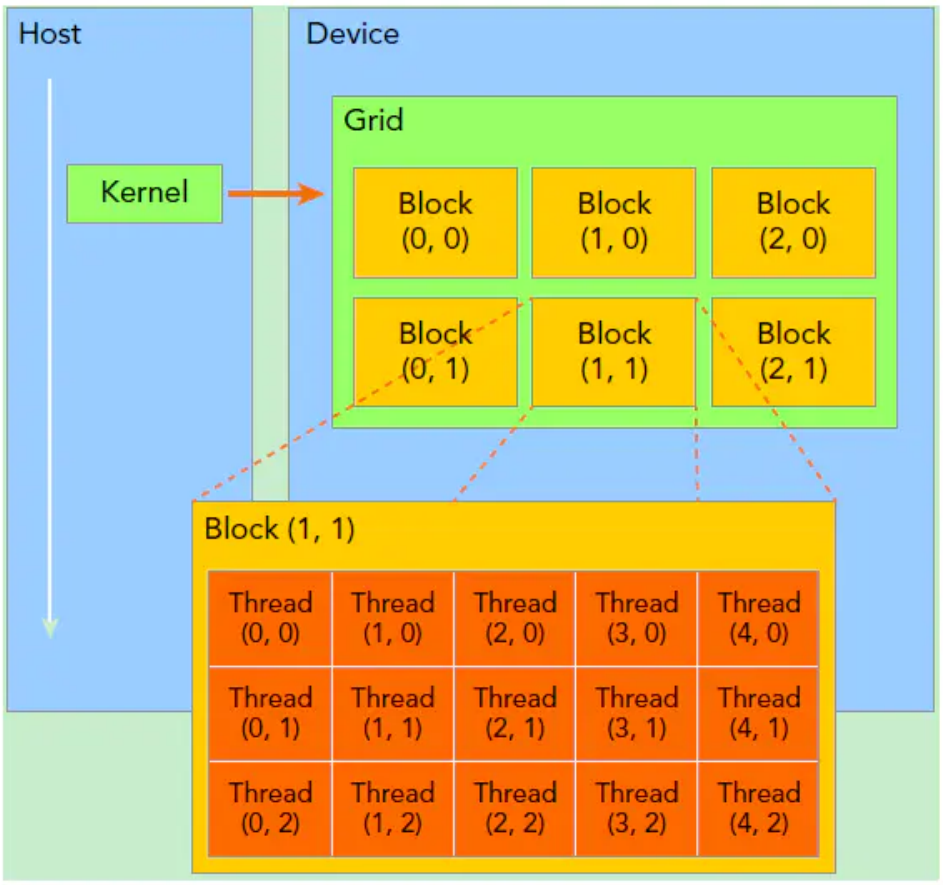
- `__global__`：在device上执行，从host中调用（一些特定的GPU也可以从device上调用），返回类型必须是void，不支持可变参数，不能成为类成员函数。注意用 `__global__` 定义的 kernel 是异步的，这意味着 host 不会等待 kernel 执行完就执行下一步。
- `__device__`：在device上执行，单仅可以从device中调用，不可以和 `__global__` 同时用。
- `__host__`：在host上执行，仅可以从host上调用，一般省略不写，不可以和 `__global__` 同时用，但可和 `__device__` 同时使用，此时函数会在 device 和 host 都编译。

要深刻理解kernel，必须要对 kernel 的线程层次结构有一个清晰的认识。首先GPU上很多并行化的轻量级线程。如下图所示，kernel在device上执行时实际上是启动很多线程：

一个 kernel 所启动的所有线程称为一个 Grid，同一个 Grid 上的线程共享相同的全局内存空间，Grid 是线程结构的第一层次，而 Grid 又可以分为很多 Block，一个Block 里面包含很多Thread，这是第二个层次。

Thread两层组织结构如下图所示，这是一个 Gird 和 Block 均为2-dim的线程组织。Grid 和 Block 都是定义为dim3类型的变量，dim3可以看成是包含3个无符号整数（x，y，z）成员的结构体变量，在定义时，缺省值初始化为1。因此 Gird 和 Block 可以灵活地定义为1-dim，2-dim以及3-dim结构，对于图中结构（主要水平方向为 x 轴），定义的 Gird 和 Block 如下所示，kernel在调用时也必须通过执行配置 <<<Grid, Block>>>来指定 kernel 所使用的线程数及结构。

```
dim3 grid(3, 2);
dim3 block(5, 3);
kernel_fun<<< grid, block >>>(prams...);
```



所以，一个线程需要两个内置的坐标变量（blockIdx，threadIdx）唯一标识，它们都是`dim3`类型变量，其中 blockIdx 指明线程所在 Grid 中的位置，而 threaIdx 指明线程所在 block 中的位置，如图中的 Thread (1,1) 满足：

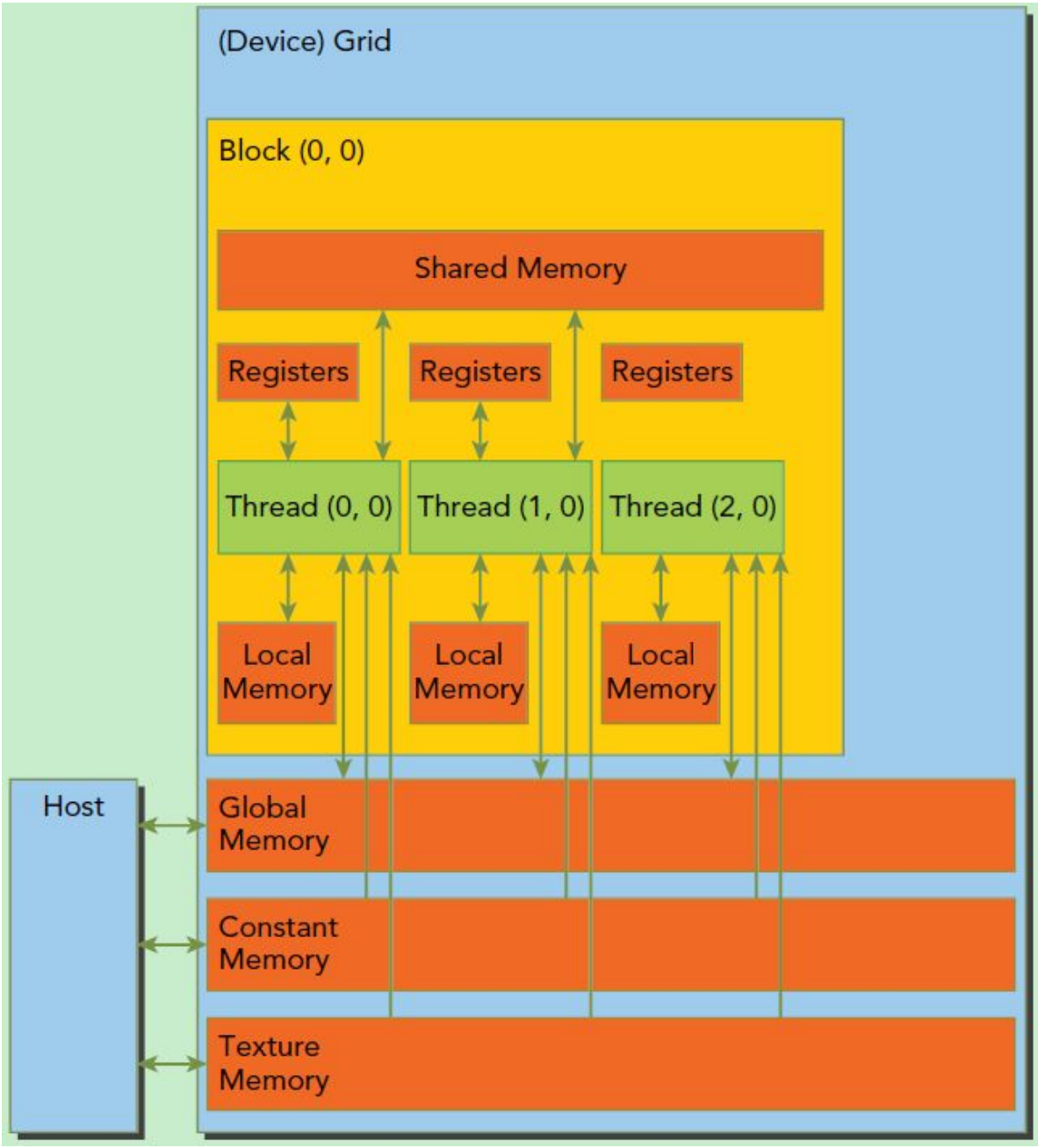
```
threadIdx.x = 1
threadIdx.y = 1
blockIdx.x = 1
blockIdx.y = 1
```

一个线程块上的线程是放在同一个流式多处理器（SM)上的，但是单个SM的资源有限，这导致线程块中的线程数是有限制的，现代GPUs的线程块可支持的线程数可达1024个。有时候，我们要知道一个线程在blcok中的全局ID，此时就必须还要知道block的组织结构，这是通过线程的内置变量blockDim来获得。它获取线程块各个维度的大小。对于一个2-dim的block (D_x, D_y)，线程 (x, y) 的ID值为 $(x + y * D_x)$，如果是 3-dim 的block (D_x, D_y, D_z)，线程 (x, y, z) 的ID值为 $(x + y * D_x + z * D_x * D_y)$。另外线程还有内置变量 gridDim，用于获得网格块各个维度的大小。

kernel 的这种线程组织结构天然适合vector,matrix等运算，如利用上图 2-dim 结构实现两个矩阵的加法，每个线程负责处理每个位置的两个元素相加，代码如下所示。线程块大小为(16, 16)，然后将NxN大小的矩阵均分为不同的线程块来执行加法运算。

```
// Kernel定义
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    ...
    // Kernel 线程配置
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    // kernel调用
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

此外这里简单介绍一下CUDA的内存模型，如下图所示。可以看到，每个线程有自己的私有本地内存（Local Memory），而每个线程块有包含共享内存（Shared Memory）,可以被线程块中所有线程共享，其生命周期与线程块一致。此外，所有的线程都可以访问全局内存（Global Memory）。还可以访问一些只读内存块：常量内存（Constant Memory）和纹理内存（Texture Memory）。

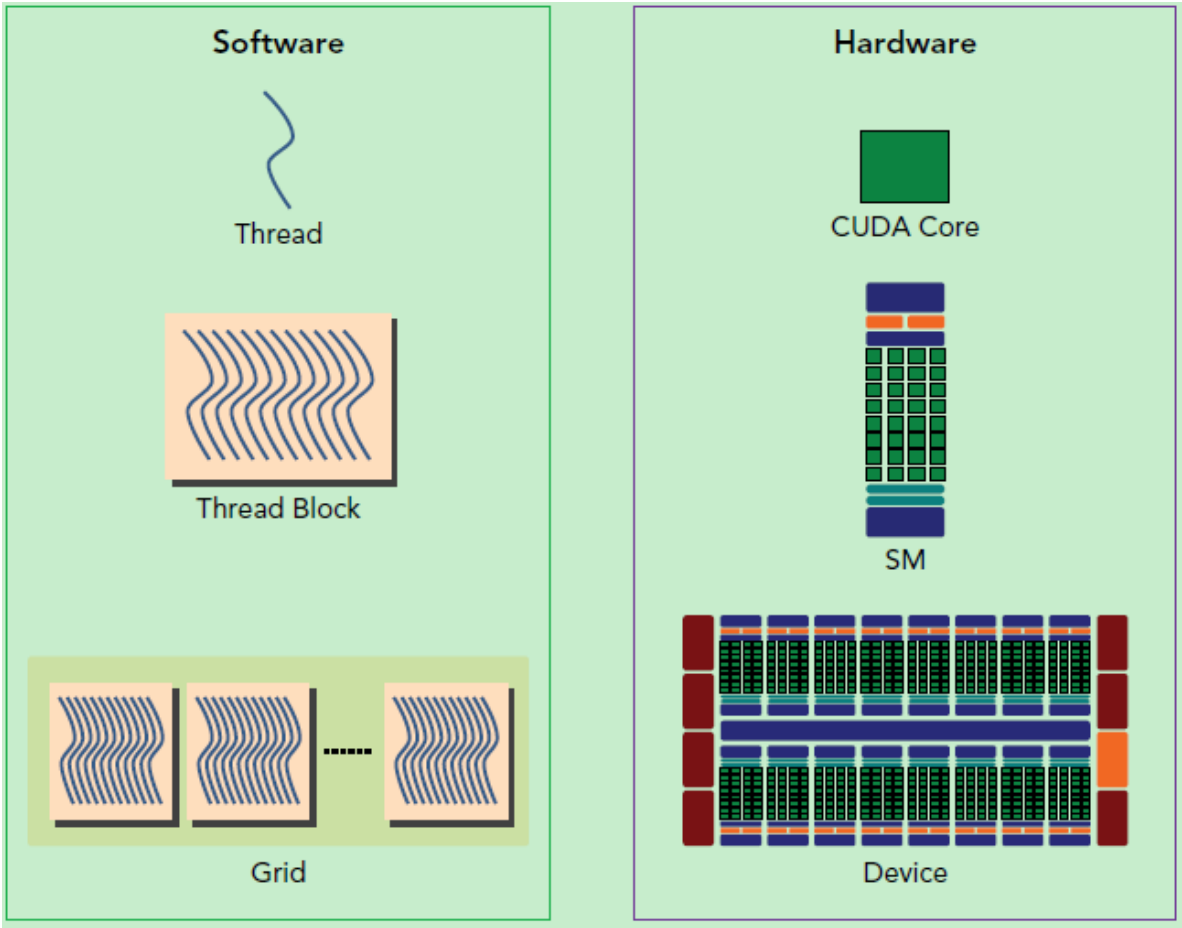


还有重要一点，你需要对GPU的硬件实现有一个基本的认识。上面说到了kernel的线程组织层次，那么一个kernel实际上会启动很多线程，这些线程是逻辑上并行的，但是在物理层却并不一定。这其实和CPU的多线程有类似之处，多线程如果没有多核支持，在物理层也是无法实现并行的。但是好在GPU存在很多CUDA核心，充分利用CUDA核心可以充分发挥GPU的并行计算能力。

GPU硬件的一个核心组件是SM，前面已经说过，SM是英文名是 Streaming Multiprocessor，翻译过来就是流式多处理器。SM的核心组件包括CUDA核心，共享内存，寄存器等，SM可以并发地执行数百个线程，并发能力就取决于SM所拥有的资源数。当一个kernel被执行时，它的 Gird 中的线程块被分配到SM上，一个线程块只能在一个SM上被调度。SM一般可以调度多个线程块，这要看SM本身的能力。那么有可能一个 kernel 的各个线程块被分配多个SM，所以 Gird 只是逻辑层，而SM才是执行的物理层。

SM采用的是SIMT架构，基本的执行单元是线程束（warps），线程束包含32个线程，这些线程同时执行相同的指令，但是每个线程都包含自己的指令地址计数器和寄存器状态，也有自己独立的执行路径。所以尽管线程束中的线程同时从同一程序地址执行，但是可能具有不同的行为，比如遇到了分支结构，一些线程可能进入这个分支，但是另外一些有可能不执行，它们只能死等，因为GPU规定线程束中所有线程在同一周期执行相同的指令，线程束分化会导致性能下降。当线程块被划分到某个SM上时，它将进一步划分为多个线程束，因为这才是SM的基本执行单元，但是一个SM同时并发的线程束数是有限的。这是因为资源限制，SM要为每个线程块分配共享内存，而也要为每个线程束中的线程分配独立的寄存器。所以SM的配置会影响其所支持的线程块和线程束并发数量。

总之，就是 Gird 和 Block 只是逻辑划分，一个 kernel 的所有线程其实在物理层是不一定同时并发的。所以kernel的 Gird 和 Block 的配置不同，性能会出现差异，这点是要特别注意的。还有，由于SM的基本执行单元是包含32个线程的线程束，所以 Block 大小一般要设置为32的倍数。



所有CUDA kernel的启动都是异步的，当CUDA kernel被调用时，控制权会立即返回给CPU。在分配Grid、Block大小时，可以遵循这几点原则：

> 1、保证 Block 中 Thread 数目是32的倍数。这是因为同一个 Block 必须在一个SM内，而SM的 Warp 调度是32个线程一组进行的；
>
> 2、避免 Block 太小：每个 Blcok 最少 128 或 256 个thread；
>
> 3、根据 kernel 需要的资源调整 Block，多做实验来挖掘最佳配置；
>
> 4、保证 Block 的数目远大于 SM 的数目。

在进行CUDA编程前，可以先检查一下自己的GPU的硬件配置，这样才可以有的放矢，可以通过下面的程序获得GPU的配置属性：

```
struct cudaDeviceProp {
    char name[256];             // 识别设备的ASCII字符串（比如，"GeForce GTX 1050"）
    size_t totalGlobalMem;      // 全局内存大小
    size_t sharedMemPerBlock; // 每个block内共享内存的大小
    int regsPerBlock;           // 每个block 32位寄存器的个数
    int warpSize;               // warp大小
    size_t memPitch;            // 内存中允许的最大间距字节数
    int maxThreadsPerBlock;     // 每个Block中最大的线程数是多少
    int maxThreadsDim[3];       // 一个块中每个维度的最大线程数
    int maxGridSize[3];         // 一个网格的每个维度的块数量
    size_t totalConstMem;       // 可用恒定内存量
    int major;                  // 该设备计算能力的主要修订版号
    int minor;                  // 设备计算能力的小修订版本号
    int clockRate;              // 时钟速率
    size_t textureAlignment;  // 该设备对纹理对齐的要求
    int deviceOverlap;          // 一个布尔值，表示该装置是否能够同时进行cudamemcpy()和内核执行
    int multiProcessorCount;  // 设备上的处理器的数量
    int kernelExecTimeoutEnabled;    // 一个布尔值，该值表示在该设备上执行的内核是否有运行时的限制
    int integrated;                  // 返回一个布尔值，表示设备是否是一个集成的GPU（即部分的芯片组、没有独立显卡等）
    int canMapHostMemory;            // 表示设备是否可以映射到CUDA设备主机内存地址空间的布尔值
    int computeMode;                 // 一个值，该值表示该设备的计算模式：默认值，专有的，或禁止的
    int maxTexture1D;                // 一维纹理内存最大值
    int maxTexture2D[2];             // 二维纹理内存最大值
    int maxTexture3D[3];             // 三维纹理内存最大值
    int maxTexture2DArray[3];        // 二维纹理阵列支持的最大尺寸
    int concurrentKernels;           // 一个布尔值，该值表示该设备是否支持在同一上下文中同时执行多个内核
}
```

向量加法以及矩阵加法的 cuda 实现

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

// HandleError： 主要用于检查对应的 CUDA 函数是否运行成功，其他地方可复用
// 初始化一个 cudaError_t 类型变量 err
static void HandleError(cudaError_t err, const char *file, int line){
                        // cudaSuccess=0:API调用返回没有错误;对于查询调用，这还意味着要查询的操作已完成。
                        if(err != cudaSuccess){
                            // cudaGetErrorString(err) ：返回的是 err 变量对应的错误信息，以字符串的形式接收。
                            printf("%s in %s at line %d\n", cudaGetErrorString(err), file, line);
                            exit(EXIT_FAILURE);
                            }
                        }

#define HANDLE_ERROR(err)(HandleError(err, __FILE__, __LINE__))

int getThreadNum(){
    // cudaDeviceProp数据类型针对函式 cudaGetDeviceProperties定义的，cudaGetDeviceProperties函数的功能是取得支持GPU计算的装置的相关属性；
    // 如支持CUDA版本号装置的名称、内存的大小、最大的 thread 数目、执行单元的频率等。
    cudaDeviceProp prop;    //定义prop数据结构
    int count;

    //可以通过 cudaGetDeviceCount 函数获取 CUDA 的设备数
    HANDLE_ERROR(cudaGetDeviceCount(&count));
    printf("gpu num %d\n", count);
    HANDLE_ERROR(cudaGetDeviceProperties(&prop, 0));
    // prop.maxThreadsPerBlock：每个Block中最大的线程数
    printf("max thread num: %d\n", prop.maxThreadsPerBlock);
    // prop.maxGridSize[0]： 一个网格的 0 维度的块数量
    // prop.maxGridSize[1]： 一个网格的 1 维度的块数量
    // prop.maxGridSize[2]： 一个网格的 2 维度的块数量
    printf("max grid dimensions: %d, %d, %d)\n", prop.maxGridSize[0], prop.maxGridSize[1], prop.maxGridSize[2]);
    return prop.maxThreadsPerBlock;
}

// CUDA 核函数:向量相加
__global__ void vevtorAdd(int* a, int* b, int* c, int num){
    // threadIdx.x 一样是 CUDA 内建的变量，它表示的是目前的 thread 编号
    int i = threadIdx.x;
    if(i < num){
        c[i] = a[i] + b[i];
    }
}

// CUDA 核函数:矩阵相加
__global__ void MatAdd(int* A, int* B, int* C, int num)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < num && j < num)
```

```
        C[i][j] = A[i][j] + B[i][j];
}

int main(void){
    // init data
    int num = 10;
    int a[num], b[num], c[num];
    int *a_gpu, *b_gpu, *c_gpu;
    // 初始化向量
    for(int i = 0; i < num; i++){
        a[i] = i;
        b[i] = i * i;
    }

    int mA[num][num], mB[num][num], mC[num][num];
    int *A_GPU, *B_GPU, *C_GPU;

    //初始化矩阵
    for(int i = 0;i < num;i++){
        for(int j = 0;j < num;j++){
            mA[i][j] = 1;
            mB[i][j] = 2;
        }
    }
    printf("Matrix A\n");
    for(int i = 0;i < num;i++){
        for(int j = 0;j < num;j++){
            printf("%d ",mA[i][j]);
        }
        printf("\n");
    }
    printf("\n");

    printf("Matrix B:\n");
    for(int i = 0;i < num;i++){
        for(int j = 0;j < num;j++){
            printf("%d ", mB[i][j]);
        }
        printf("\n");
    }
    printf("\n");

    // 其实直接用 &a_gpu 也是可以的，但是函数原型必须得是 (void **) 型的指针
    // cudaMalloc的第 1 个参数传递的是存储在 cpu 内存中的指针变量的地址，第 2 个参数传递的是欲申请内存的大小
    // cudaMalloc在执行完成后，向这个地址中写入了一个地址值（此地址值是GPU显存里的）。
    HANDLE_ERROR(cudaMalloc((void **)&a_gpu, num * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void **)&b_gpu, num * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void **)&c_gpu, num * sizeof(int)));

    HANDLE_ERROR(cudaMalloc((void **)&A_GPU, num * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void **)&B_GPU, num * sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void **)&C_GPU, num * sizeof(int)));

    // copy data
    // cudaMemcpy用于在主机（Host）和设备（Device）之间往返的传递数据，用法如下：
    // 主机到设备：cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice);
    // 设备到主机：cudaMemcpy(h_A, d_A, nBytes, cudaMemcpyDeviceToHost);
```

```
    // 注意：该函数是同步执行函数，在未完成数据的转移操作之前会锁死并一直占有 CPU 进程的控制
权，所以不用再添加 cudaDeviceSynchronize() 函数
    HANDLE_ERROR(cudaMemcpy(a_gpu, a, num * sizeof(int),
cudaMemcpyHostToDevice));    // 主机到设备
    HANDLE_ERROR(cudaMemcpy(b_gpu, b, num * sizeof(int),
cudaMemcpyHostToDevice));    // 主机到设备

    HANDLE_ERROR(cudaMemcpy(A_GPU, mA, num * sizeof(int),
cudaMemcpyHostToDevice));    // 主机到设备
    HANDLE_ERROR(cudaMemcpy(B_GPU, mB, num * sizeof(int),
cudaMemcpyHostToDevice));    // 主机到设备

    int threadNum = getThreadNum();    // 设置核函数的 thread 数
    int blockNum = 1;    // 设置核函数的 block 数量
    // do add with cuda
    vevtorAdd<<<blockNum, threadNum>>>(a_gpu, b_gpu, c_gpu, num);

    dim3 gridSize(5, 5);
    dim3 blockSize(2, 2);
    MatAdd<<<gridSize, blockSize>>>(A_GPU, B_GPU, C_GPU, num);

    // get data
    HANDLE_ERROR(cudaMemcpy(mC, C_GPU, num * sizeof(int),
cudaMemcpyDeviceToHost));   // 设备到主机

    // visualization
    printf("Matrix Add Result:\n");
    for(int i = 0; i < num; i++)
    {
        for(int j = 0; j < num; j++)
        {
            printf("%d ", mC[i][j]);
        }
        printf("\n");
    }
    printf("\n");

    printf("Vector Add Result:\n");
    for(int i = 0; i < num; i++){
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    return 0;
}
```

CmakeList.txt内容如下：

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
PROJECT(add)
FIND_PACKAGE(CUDA REQUIRED)
CUDA_ADD_EXECUTABLE(add main.cu)
TARGET_LINK_LIBRARIES(add)
```

Linux系统执行指令如下：

```
$ mkdir build && cd build
$ make -j4
$ ./add
```

参考：

[1].https://zhuanlan.zhihu.com/p/34587739

[2].https://zhuanlan.zhihu.com/p/266633373

[3].https://blog.csdn.net/mysniper11/category_1200646.html