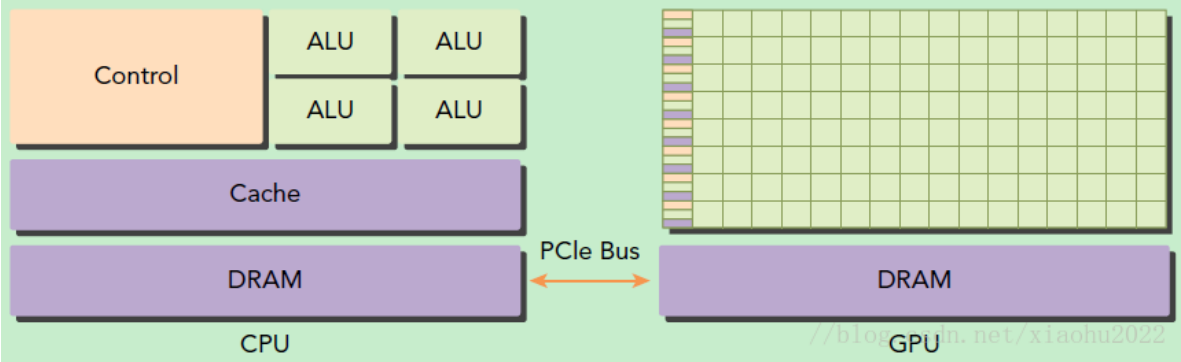
CUDA

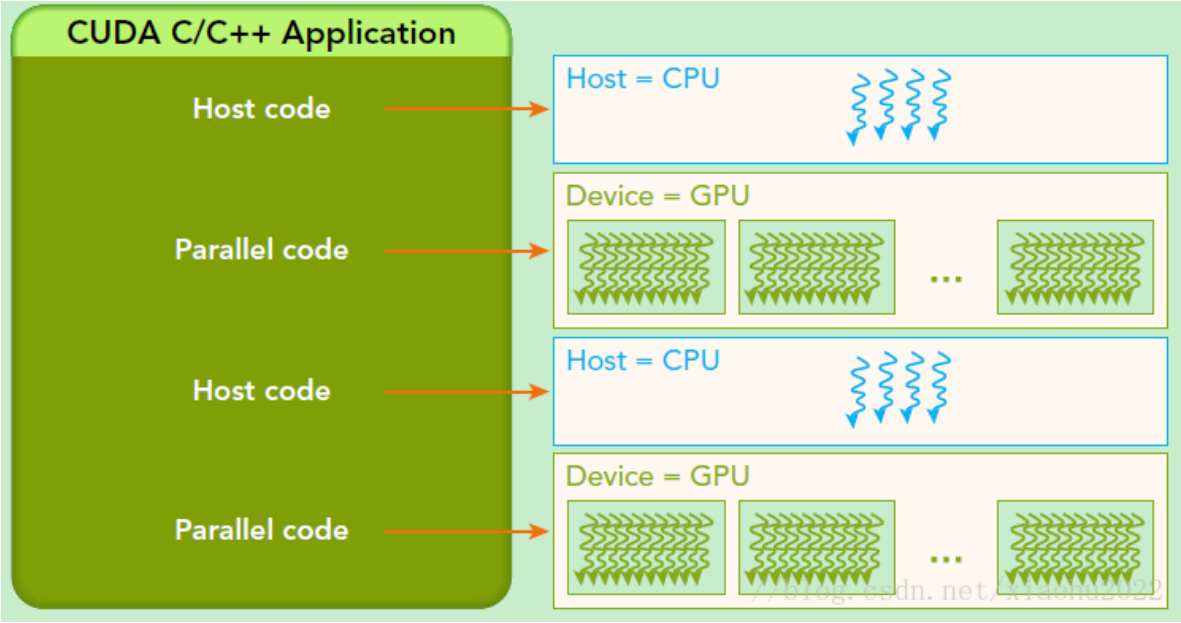
$1 概述

2006年，NVIDIA公司发布了CUDA，CUDA是建立在NVIDIA的CPUs上的一个通用并行计算平台和编程模型，基于CUDA编程可以利用GPUs的并行计算引擎来更加高效地解决比较复杂的计算难题。近年来，GPU最成功的一个应用就是深度学习领域，基于GPU的并行计算已经成为训练深度学习模型的标配。

GPU并不是一个独立运行的计算平台，而需要与CPU协同工作，可以看成是CPU的协处理器，因此当我们在说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总线连接在一起来协同工作，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。



$2 CUDA基本流程

$2.1 CUDA运行过程

在CUDA中，host和device是两个重要的概念，我们用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的核函数在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。

$2.2 kernel介绍

上面流程中最重要的一个过程是**调用CUDA的核函数**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，核函数用**__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核函数，并且每个线程会分配一个唯一的线程号thread ID，这个ID值可以通过核函数的内置变量**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，主要的三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**，此时函数会在device和host都编译。

$2.3 kernel的线程层次结构

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。

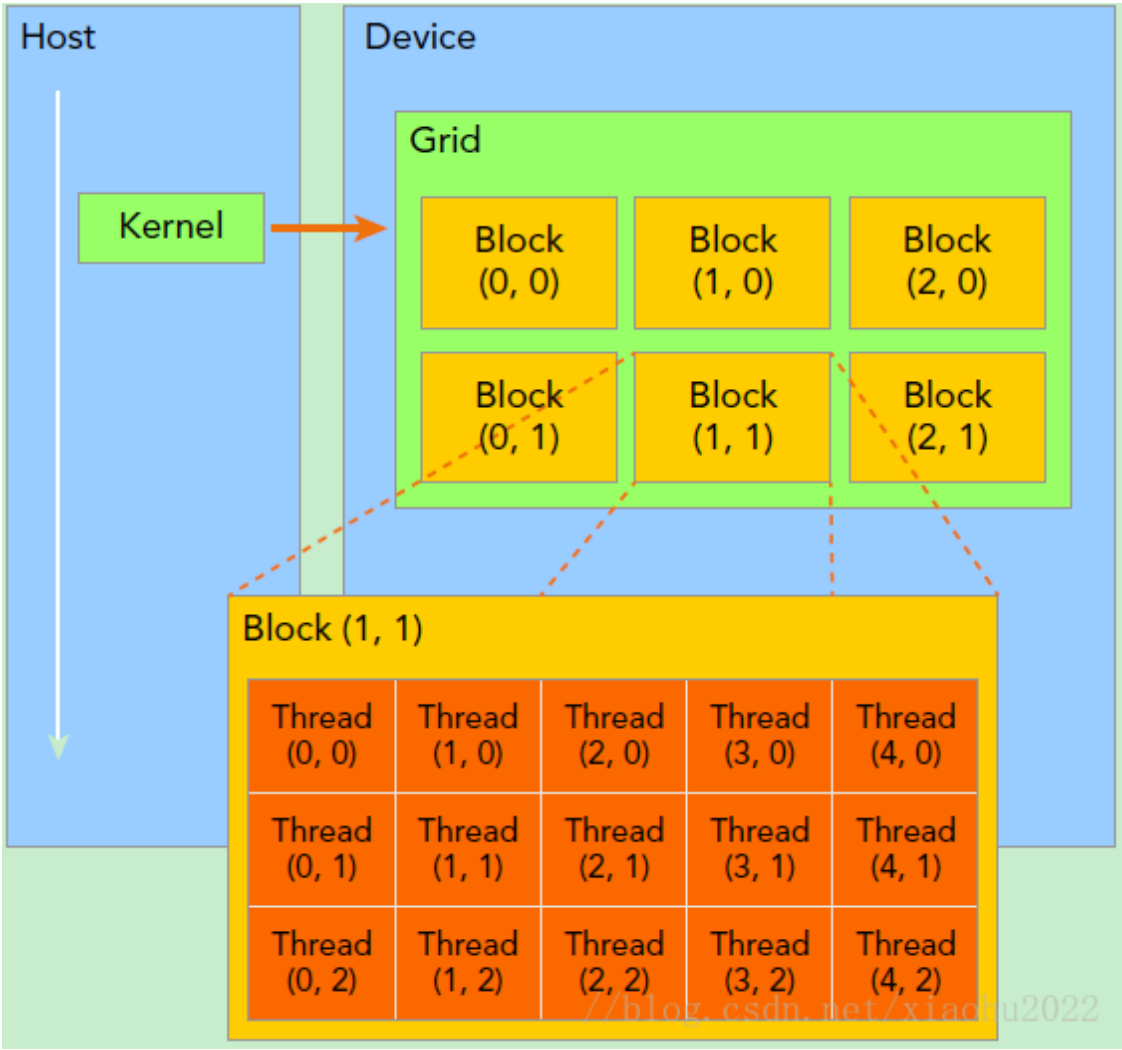
- **第一层次 - 网格（grid）**

首先GPU上很多并行化的轻量级线程。**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网格（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

- **第二层次 - 线程块（block）**

  网格可以分为很多线程块（block），一个线程块里面包含很多线程，这是第二个层次。线程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可以看成是包含三个无符号整数（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grid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grid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...);
```



$2.4 CUDA内存模型

$2.5 CUDA内置函数

- 内存管理API

```
// 在device上分配内存的
cudaError_t cudaMalloc(void** devPtr, size_t size);
/*
* 这个函数和C语言中的malloc类似，但是在device上申请一定字节大小的显存，其中devPtr是
* 指向所分配内存的指针。要释放分配的内存使用cudaFree函数，这和C语言中的free函数对应。
*/
```

- 数据通信的cudaMemcpy函数

```
// host和device之间数据通信
cudaError_t cudaMemcpy(void* dst, const void* src,
                       size_t count, cudaMemcpyKind kind)
/*
* 其中src指向数据源，而dst是目标区域，count是复制的字节数，其中kind控制复制的方向:
* cudaMemcpyHostToHost, cudaMemcpyHostToDevice,
* cudaMemcpyDeviceToHost 及 cudaMemcpyDeviceToDevice
*/
```

Updating | 2019/07/06 Szp