CPU、GPU、CUDA，CuDNN 简介

https://luisstruggle.github.io/blog/GPU_CPU_CUDA.html

LuisStruggle Learning and communication Sun Nov, 22 03:46

发表于 2018-04-05 | 分类于 技术篇

1，CPU (Central Processing Unit) 即中央处理器
2，GPU (Graphics Processing Unit) 即图形处理器
3，GPGPU全称General Purpose GPU，即通用计算图形处理器。其中第一个“GP”通用目的（GeneralPurpose）而第二个“GP”则表示图形处理（GraphicProcess）

CPU虽然有多核，但总数没有超过两位数，每个核都有足够大的缓存和足够多的数字和逻辑运算单元，并辅助有很多加速分支判断甚至更复杂的逻辑判断的硬件。

GPU的核数远超CPU，被称为众核（NVIDIA Fermi有512个核）。每个核拥有的缓存大小相对小，数字逻辑运算单元也少而简单（GPU初始时在浮点计算上一直弱于CPU）。

从结果上导致CPU擅长处理具有复杂计算步骤和复杂数据依赖的计算任务，如分布式计算，数据压缩，人工智能，物理模拟，以及其他很多很多计算任务等。

GPU由于历史原因，是为了视频游戏而产生的（至今其主要驱动力还是不断增长的视频游戏市场），在三维游戏中常常出现的一类操作是对海量数据进行相同的操作，如：对每一个顶点进行同样的坐标变换，对每一个顶点按照同样的光照模型计算颜色值。GPU的众核架构非常适合把同样的指令流并行发送到众核上，采用不同的输入数据执行。

当程序员为CPU编写程序时，他们倾向于利用复杂的逻辑结构优化算法从而减少计算任务的运行时间，即Latency。
当程序员为GPU编写程序时，则利用其处理海量数据的优势，通过提高总的数据吞吐量（Throughput）来掩盖Lantency。

其中绿色的是计算单元，橙红色的是存储单元，橙黄色的是控制单元。

GPU采用了数量众多的计算单元和超长的流水线，但只有非常简单的控制逻辑并省去了Cache。而CPU不仅被Cache占据了大量空间，而且还有有复杂的控制逻辑和诸多优化电路，相比之下计算能力只是CPU很小的一部分。

CUDA(Compute Unified Device Architecture)，是英伟达公司推出的一种基于新的并行编程模型和指令集架构的通用计算架构，它能利用英伟达GPU的并行计算引擎，比CPU更高效的解决许多复杂计算任务。

使用CUDA的好处就是透明。根据摩尔定律GPU的晶体管数量不断增多，硬件结构必然是不断的在发展变化，没有必要每次都为不同的硬件结构重新编码，而CUDA就是提供了一种可扩展的编程模型，使得已经写好的CUDA代码可以在任意数量核心的GPU上运行。如下图所示，只有运行时，系统才知道物理处理器的数量。

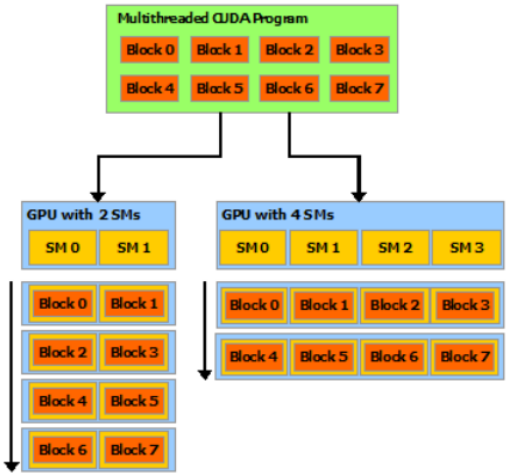


图 1.5: 自动可扩展性

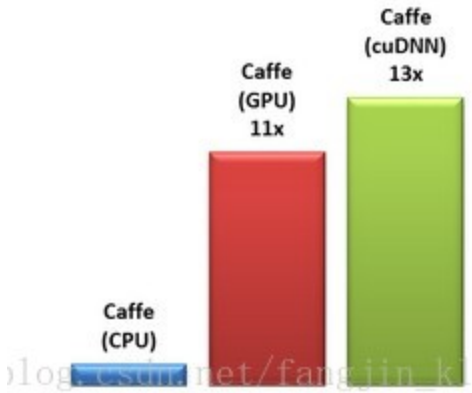
NVIDIA cuDNN是用于深度神经网络的GPU加速库。它强调性能、易用性和低内存开销。NVIDIA cuDNN可以集成到更高级别的机器学习框架中，如加州大学伯克利分校的流行CAFFE软件。简单的，插入式设计可以让开发人员专注于设计和实现神经网络模型，而不是调整性能，同时还可以在GPU上实现高性能现代并行计算。

[cuDNN 用户手册（英文）]

CuDNN支持的算法

1. 卷积操作、相关操作的前向和后向过程。
2. pooling的前向后向过程
3. softmax的前向后向过程

4. 激活函数的前向后向过程
    - ReLU
    - sigmoid
    - TANH
5. Tensor转换函数，其中一个Tensor就是一个四维的向量。



Baseline Caffe与用NVIDIA Titan Z 加速cuDNN的Caffe做比较

参考自 一篇不错的CUDA入门博客

开发人员可以通过调用CUDA的API，来进行并行编程，达到高性能计算目的。NVIDIA公司为了吸引更多的开发人员，对CUDA进行了编程语言扩展，如CUDA C/C++,CUDA Fortran语言。注意CUDA C/C++可以看作一个新的编程语言，因为NVIDIA配置了相应的编译器nvcc,CUDA Fortran一样。

如果粗暴的认为C语言工作的对象是CPU和内存条（接下来,称为主机内存），那么CUDA C工作的的对象就是GPU及GPU上的内存（接下来,称为设备内存），且充分利用了GPU多核的优势及降低了并行编程的难度。一般通过C语言把数据从外界读入，再分配数据，给CUDA C，以便在GPU上计算，然后再把计算结果返回给C语言，以便进一步工作，如进一步处理及显示，或重复此过程。

主要概念与名称

1. 主机
   将CPU及系统的内存（内存条）称为主机。

2. 设备
   将GPU及GPU本身的显示内存称为设备。

3. 线程(Thread)

   一般通过GPU的一个核进行处理。（可以表示成一维，二维，三维，具体下面再细说）。

4. 线程块(Block)

   1. 由多个线程组成（可以表示成一维，二维，三维，具体下面再细说）。
   2. 各block是并行执行的，block间无法通信，也没有执行顺序。
   3. 注意线程块的数量限制为不超过65535（硬件限制）。

5. 线程格(Grid)

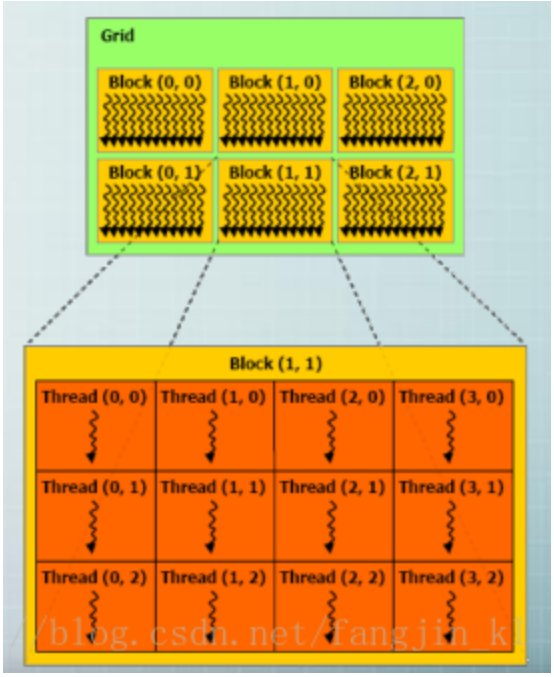
   由多个线程块组成（可以表示成一维，二维，三维，具体下面再细说）。

6. 线程束

   在CUDA架构中，线程束是指一个 **包含32个线程** 的集合，这个线程集合被“编织在一起”并且“步调一致”的形式执行。在程序中的每一行，线程束中的每个线程都将在不同数据上执行相同的命令。

7. 核函数（Kernel）

   1. 在GPU上执行的函数通常称为核函数。
   2. 一般通过标识符 `__global__` 修饰，调用通过<<<参数1,参数2>>>，用于说明内核函数中的线程数量，以及线程是如何组织的。
   3. 以线程格（Grid）的形式组织，每个线程格由若干个线程块（block）组成，而每个线程块又由若干个线程（thread）组成。
   4. 是以block为单位执行的。
   5. 只能在主机端代码中调用。
   6. 调用时必须声明内核函数的执行参数。
   7. 在编程时，必须先为kernel函数中用到的数组或变量分配好足够的空间，再调用kernel函数，否则在GPU计算时会发生错误，例如越界或报错，甚至导致蓝屏和死机。




```
/*
 * @file_name HelloWorld.cu  后缀名称.cu
 */

#include <stdio.h>
#include <cuda_runtime.h>  //头文件

//核函数声明，前面的关键字__global__
__global__ void kernel( void ) {
}

int main( void ) {
    //核函数的调用，注意<<<1,1>>>，第一个1，代表线程格里只有一个线程块；第二个1，代表一个线程块里只有一个线程。
    kernel<<<1,1>>>();
    printf( 'Hello, World!\n' );
    return 0;
}
```

dim3结构类型

1. dim3是基于 uint3 定义的矢量类型，相当于由3个unsigned int型组成的结构体。uint3类型有三个数据成员 `unsigned int x; unsigned int y; unsigned int z;`
2. 可使用一维、二维或三维的索引来标识线程，构成一维、二维或三维线程块。
3. dim3结构类型变量用在核函数调用的<<<,>>>中。
4. 相关的几个内置变量

    - threadIdx，顾名思义获取线程 thread 的ID索引；如果线程是一维的那么就取threadIdx.x，二维的还可以多取到一个值threadIdx.y，以此类推到三维threadIdx.z。

- blockIdx，线程块的ID索引；同样有blockIdx.x，blockIdx.y，blockIdx.z。
- blockDim，线程块的维度，同样有blockDim.x，blockDim.y，blockDim.z。
- gridDim，线程格的维度，同样有gridDim.x，gridDim.y，gridDim.z。

5. 对于一维的block，线程的threadID=threadIdx.x。
6. 对于大小为（blockDim.x, blockDim.y）的二维block，线程的threadID=threadIdx.x+threadIdx.y*blockDim.x。
7. 对于大小为（blockDim.x, blockDim.y, blockDim.z）的三维block，线程的threadID=threadIdx.x+threadIdx.y*blockDim.x+threadIdx.z*blockDim.x*blockDim.y。
8. 对于计算线程索引偏移增量为已启动线程的总数。如stride = blockDim.x * gridDim.x; threadId += stride。

函数修饰符

1. `__global__`，表明被修饰的函数在设备上执行，但在主机上调用。
2. `__device__`，表明被修饰的函数在设备上执行，但只能在其他 **device** 函数或者 **global** 函数中调用。

常用的GPU内存函数

cudaMalloc()

1. 函数原型：`cudaError_t cudaMalloc (void **devPtr, size_t size)`。
2. 函数用处：与C语言中的malloc函数一样，只是此函数在GPU的内存你分配内存。
3. 注意事项：

    - 可以将cudaMalloc()分配的指针传递给在设备上执行的函数；
    - 可以在设备代码中使用cudaMalloc()分配的指针进行设备内存读写操作；
    - 可以将cudaMalloc()分配的指针传递给在主机上执行的函数；
    - 不可以在主机代码中使用cudaMalloc()分配的指针进行主机内存读写操作（即不能进行解引用）。

cudaMemcpy()

1. 函数原型：`cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, cudaMemcpyKind kind)`。
2. 函数作用：与c语言中的memcpy函数一样，只是此函数可以在主机内存和GPU内存之间互相拷贝数据。
3. 函数参数：cudaMemcpyKind kind表示数据拷贝方向，如果kind赋值为cudaMemcpyDeviceToHost表示数据从设备内存拷贝到主机内存。

4. 与C中的memcpy()一样，以同步方式执行，即当函数返回时，复制操作就已经完成了，并且在输出缓冲区中包含了复制进去的内容。
5. 相应的有个异步方式执行的函数cudaMemcpyAsync()，这个函数详解请看下面的流一节有关内容。

cudaFree()

1. 函数原型：`cudaError_t cudaFree ( void* devPtr )`。
2. 函数作用：与c语言中的free()函数一样，只是此函数释放的是cudaMalloc()分配的内存。

下面实例用于解释上面三个函数

```
#include <stdio.h>
#include <cuda_runtime.h>
__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}
int main( void ) {
    int c;
    int *dev_c;
    //cudaMalloc()
    cudaMalloc( (void**)&dev_c, sizeof(int) );
    //核函数执行
    add<<<1,1>>>( 2, 7, dev_c );
    //cudaMemcpy()
    cudaMemcpy( &c, dev_c, sizeof(int),cudaMemcpyDeviceToHost ) ;
    printf( '2 + 7 = %d\n', c );
    //cudaFree()
    cudaFree( dev_c );

    return 0;
}
```

GPU内存分类

全局内存

通俗意义上的设备内存。

共享内存

1. 位置：设备内存。
2. 形式：关键字 **shared** 添加到变量声明中。如 **shared** float cache[10]。

3. 目的：对于GPU上启动的每个线程块，CUDA C编译器都将创建该共享变量的一个副本。线程块中的每个线程都共享这块内存，但线程却无法看到也不能修改其他线程块的变量副本。这样使得一个线程块中的多个线程能够在计算上通信和协作。

常量内存

1. 位置：设备内存
2. 形式：关键字 **constant** 添加到变量声明中。如 **constant** float s[10];。
3. 目的：为了提升性能。常量内存采取了不同于标准全局内存的处理方式。在某些情况下，用常量内存替换全局内存能有效地减少内存带宽。
4. 特点：常量内存用于保存在核函数执行期间不会发生变化的数据。变量的访问限制为只读。NVIDIA硬件提供了64KB的常量内存。不再需要cudaMalloc()或者cudaFree(),而是在编译时，静态地分配空间。
5. 要求：当我们需要拷贝数据到常量内存中应该使用cudaMemcpyToSymbol()，而cudaMemcpy()会复制到全局内存。
6. 性能提升的原因：
    - 对常量内存的单次读操作可以广播到其他的“邻近”线程。这将节约15次读取操作。（为什么是15，因为“邻近”指半个线程束，一个线程束包含32个线程的集合。）
    - 常量内存的数据将缓存起来，因此对相同地址的连续读操作将不会产生额外的内存通信量。

纹理内存

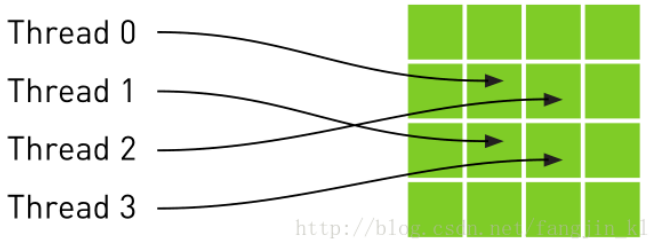
1. 位置：设备内存
2. 目的：能够减少对内存的请求并提供高效的内存带宽。是专门为那些在内存访问模式中存在大量空间局部性的图形应用程序设计，意味着一个线程读取的位置可能与邻近线程读取的位置“非常接近”。如下图：
3. 纹理变量（引用）必须声明为文件作用域内的全局变量。
4. 形式：分为一维纹理内存 和 二维纹理内存。

    - 一维纹理内存

        - 用 `texture<类型>` 类型声明，如 `texture<float> texIn` 。
        - 通过 `cudaBindTexture()` 绑定到纹理内存中。
        - 通过 `tex1Dfetch()` 来读取纹理内存中的数据。
        - 通过 `cudaUnbindTexture()` 取消绑定纹理内存。
    - 二维纹理内存

        - 用 `texture<类型,数字>` 类型声明，如 `texture<float, 2> texIn` 。
        - 通过 `cudaBindTexture2D()` 绑定到纹理内存中。

- 通过 `tex2D()` 来读取纹理内存中的数据。
- 通过 `cudaUnbindTexture()` 取消绑定纹理内存。



固定内存

1. 位置：主机内存。
2. 概念：也称为页锁定内存或者不可分页内存，操作系统将不会对这块内存分页并交换到磁盘上，从而确保了该内存始终驻留在物理内存中。因此操作系统能够安全地使某个应用程序访问该内存的物理地址，因为这块内存将不会破坏或者重新定位。
3. 目的：提高访问速度。由于GPU知道主机内存的物理地址，因此可以通过“直接内存访问DMA（Direct Memory Access)技术来在GPU和主机之间复制数据。由于DMA在执行复制时无需CPU介入。因此DMA复制过程中使用固定内存是非常重要的。
4. 缺点：使用固定内存，将失去虚拟内存的所有功能；系统将更快的耗尽内存。
5. 建议：对cudaMemcpy()函数调用中的源内存或者目标内存，才使用固定内存，并且在不再需要使用它们时立即释放。
6. 形式：通过cudaHostAlloc()函数来分配；通过cudaFreeHost()释放。
7. 只能以异步方式对固定内存进行复制操作。

原子性

1. 概念：如果操作的执行过程不能分解为更小的部分，我们将满足这种条件限制的操作称为原子操作。
2. 形式：函数调用，如atomicAdd（addr,y)将生成一个原子的操作序列，这个操作序列包括读取地址addr处的值，将y增加到这个值，以及将结果保存回地址addr。

常用线程操作函数

1. 同步方法**syncthreads()，这个函数的调用，将确保线程块中的每个线程都执行完**syscthreads()前面的语句后，才会执行下一条语句。

使用事件来测量性能

1. 用途：为了测量GPU在某个任务上花费的时间。CUDA中的事件本质上是一个GPU时间戳。由于事件是直接在GPU上实现的。因此不适用于对同时包含设备代码和主机代码的混合代码设计。
2. 形式：首先创建一个事件，然后记录事件，再计算两个事件之差，最后销毁事件。如：

```
cudaEvent_t start, stop;
cudaEventCreate( &start );
cudaEventCreate( &stop );
cudaEventRecord( start, 0 );
//do something
cudaEventRecord( stop, 0 );
float   elapsedTime;
cudaEventElapsedTime( &elapsedTime,start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

流

1. 扯一扯：并发重点在于一个极短时间段内运行多个不同的任务；并行重点在于同时运行一个任务。
2. 任务并行性：是指并行执行两个或多个不同的任务，而不是在大量数据上执行同一个任务。
3. 概念：CUDA流表示一个GPU操作队列，并且该队列中的操作将以指定的顺序执行。我们可以在流中添加一些操作，如核函数启动，内存复制以及事件的启动和结束等。这些操作的添加到流的顺序也是它们的执行顺序。可以将每个流视为GPU上的一个任务，并且这些任务可以并行执行。
4. 硬件前提：必须是支持设备重叠功能的GPU。支持设备重叠功能，即在执行一个核函数的同时，还能在设备与主机之间执行复制操作。
5. 声明与创建：声明 `cudaStream_t stream`;，创建 `cudaSteamCreate(&stream)` 。
6. `cudaMemcpyAsync()` ：前面在 `cudaMemcpy()` 中提到过，这是一个以异步方式执行的函数。在调用 `cudaMemcpyAsync()` 时，只是放置一个请求，表示在流中执行一次内存复制操作，这个流是通过参数stream来指定的。当函数返回时，我们无法确保复制操作是否已经启动，更无法保证它是否已经结束。我们能够得到的保证是，复制操作肯定会当下一个被放入流中的操作之前执行。传递给此函数的主机内存指针必须是通过 `cudaHostAlloc()` 分配好的内存。（流中要求固定内存）
7. 流同步：通过 `cudaStreamSynchronize()` 来协调。
8. 流销毁：在退出应用程序之前，需要销毁对GPU操作进行排队的流，调用 `cudaStreamDestroy()` 。

9. 针对多个流：
    - 记得对流进行同步操作。
    - 将操作放入流的队列时，应采用宽度优先方式，而非深度优先的方式，换句话说，不是首先添加第0个流的所有操作，再依次添加后面的第1，2,…个流。而是交替进行添加，比如将a的复制操作添加到第0个流中，接着把a的复制操作添加到第1个流中，再继续其他的类似交替添加的行为。
    - 要牢牢记住操作放入流中的队列中的顺序影响到CUDA驱动程序调度这些操作和流以及执行的方式。

技巧

1. 当线程块的数量为GPU中处理数量的2倍时，将达到最优性能。
2. 核函数执行的第一个计算就是计算输入数据的偏移。每个线程的起始偏移都是0到线程数量减1之间的某个值。然后，对偏移的增量为已启动线程的总数。

来源：https://blog.csdn.net/fangjin_kl/article/details/53906874