CUDA编程

1 基础

1.1 stream流

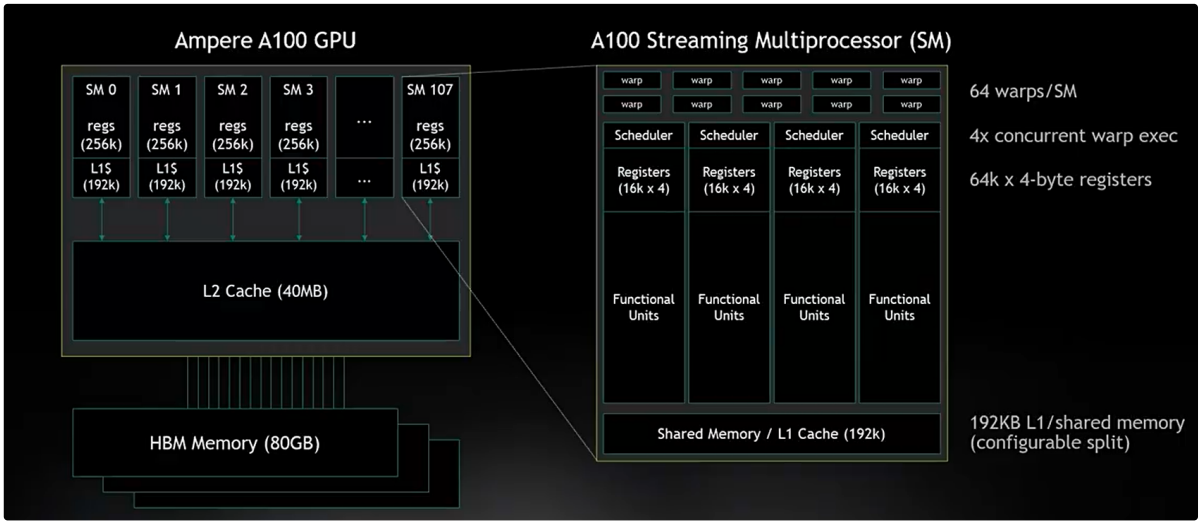
流可以理解为一系列**异步**的GPU操作，这些操作按照主机代码确定的顺序在GPU上执行。
阻塞流与非阻塞流：空流，即默认流，是隐式流，在相同的CUDA上下文中**空流与其它所有阻塞流是同步执行的。**使用cudaStreamCreate()函数创建的非空流都是阻塞流，就是说这些流会被空流阻塞。
例：主机创建了2个阻塞流stream1和stream2，主机启动三个核函数，kernel1被stream1启动，kernel2被空流启动，kernel3被stream2启动，启动代码如下：

```
kernel1<<<1, 1, 0, stream1>>>();
kernel2<<<1, 1>>>();
kernel3<<<1, ,1, 0, stream2>>>();
kernel4<<<1, ,1, 0, stream2>>>();
```

三个核函数，**相对主机而言都是异步执行的**，但是**相对GPU而言是串行执行的。**

1. 同一个流计算是同步执行的
2. **空流阻塞**：kernel2必须等到kernel1执行完成之后才能执行，kernel3必须等到kernel2执行完成之后才能执行
3. 如果去掉kernel2的启动代码，在设备计算资源足够的情况下，支持Hyper-Q时kernel1与kernel3可以并发执行，否则kernel3需等待kernel1执行完成。

1.2 内存



GPU的内存结构和CPU类似，但也存在一些区别，GPU的内存中可读写的有：寄存器(registers)、Local memory、共享内存（shared memory）和全局内存（global memory），只读的有：常量内存（constant memory）和纹理内存（texture memory）。每个线程都有独立的寄存器和Local memory，同一个block的所有线程共享一个共享内存，全局内存、常量内存和纹理内存是所有线程都可访问的。全局内存、常量内存和纹理内存对程序的优化有特殊作用。

1.2.1 寄存器

与CPU不同，GPU的每个SM（流多处理器）有成千上万个寄存器，在GPU技术简介中已经提到，SM类似于CPU的核，每个SM拥有多个SP（流处理器），所有的工作都是在SP上处理的，GPU的每个SM可能有8~192个SP，这就意味着，SM可同时运行这些数目的线程。

寄存器是每个线程私有的，并且GPU没有使用寄存器重命名机制，而是致力于为每一个线程都分配真实的寄存器，CUDA上下文切换机制非常高效，几乎是零开销。无论是CPU还是GPU，通过寄存器的优化方式都会使程序的执行速度得到很大提高。

1.2.2 Local memory

Local memory和寄存器类似，也是线程私有的，访问速度比寄存器稍微慢一点。事实上，是由编译器在寄存器全部使用完的时候自动分配的。在优化程序的时候可以考虑减少block的线程数量以使每个线程有更多的寄存器可使用，这样可减少Local memory的使用，从而加快运行速度。

1.2.3 共享内存

共享内存允许同一个block中的线程读写这一段内存，但线程无法看到也无法修改其它block的共享内存。共享内存缓冲区驻留在物理GPU上，所以访问速度也是很快的。事实上，共享内存的速度几乎在所有的GPU中都一致（而全局内存在低端显卡的速度只有高端显卡的1/10），因此，在任何显卡中，除了使用寄存器，还要更有效地使用共享内存。

共享内存的存在就可使运行线程块中的多个线程之间相互通信。共享内存的一个应用场景是线程块中多个线程需要共同操作某一数据。共享内存和 `__syncthreads()` 搭配使用，它是线程同步函数，调用这个函数确保在线程块中所有的线程都执行完 `__syncthreads()` 之前的代码，在执行后面的代码，当然，这会损失一定性能。

1.2.4 常量内存

常量内存，通过它的名字就可以猜到它是只读内存。常量内存其实只是全局内存的一种虚拟地址形式，并没有特殊保留的常量内存块。内存的大小为64KB。常量内存可以在编译时申明为常量内存，使用修饰符 `__constant__` 申明，也可以在运行时通过主机端定义为只读内存。常量只是从GPU内存的角度而言的，CPU在运行时可以通过调用cudaCopyToSymbol来改变常量内存中的内容。

1.2.5 全局内存

GPU的全局内存之所以是全局内存，主要是因为GPU与CPU都可以对它进行写操作，任何设备都可以通过PCI-E总线对其进行访问。在多GPU系统同，GPU之间可以不通过CPU直接将数据从一块GPU卡传输到另一块GPU卡上。在调用核函数之前，使用cudaMemcpy函数就是把CPU上的数据传输到GPU的全局内存上。

1.2.6 纹理内存

和常量内存一样，纹理内存也是一种只读内存，在特定的访问模式中，纹理内存能够提升程序的性能并减少内存流量。纹理内存最初是为图形处理程序而设计，不过同样也可以用于通用计算。由于纹理内存的使用非常特殊，有时使用纹理内存是费力不讨好的事情。因此，对于纹理内存，只有在应用程序真正需要的时候才对其进行了解。主要应该掌握全局内存、共享内存和寄存器的使用。

1.3 kernel(核函数)

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，同一个网格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，而网格又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。

CUDA程序的执行流程中，最重要的一步是调用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_size,block_size >>>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。

利用<< >>执行配置可以指定线程在GPU上调度运行的方式。执行配置的第一个值是网格(grid)维度，也就是启动块(block)的数目。第二个值是块(block)维度，也就是每个块中线程(thread)的数目。

同一个块中的线程之间可以相互协作，不同块内的线程不能协作。对于一个给定的问题，可以使用不同的网格和块布局来组织你的线程。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。

2 性能评估和优化

2.1 带宽峰值和算力峰值

带宽峰值计算(Maximum memory clock显存频率， memory bus width显存位宽)：

$$PeakMBW = M_{clk} * M_{bus} * N_{ddr}/8, (M_{clk}为显存频率，M_{bus}为显存位宽)$$

例T4：

$$PeakMBW = 0.5 * 256 * 20/8 = 320GB/s$$

计算显存带宽时要留意显存类型(Memory Type)，对于GDDR1/2/3/4显存，其数据总线都是采用的DDR技术(通过差分时钟在上升沿和下降沿都进行数据传输，其一个周期传输两次数据，相当于SDRAM频率的2倍)，故其显存等效频率=显存频率×2；而GDDR5则不同，它有两条数据总线，相当于Rambus的QDR技术，传输能力相当于SDRAM频率的4倍，所以显存等效频率=显存频率×4.

算力峰值计算：

$$PeakFLOPS = F_{clk} * N_{SM} * T_{ins} * 2$$

其中，F_{clk}为GPU核心运算频率，N_{SM}为$GPU\ SM$数量，T_{ins}为特定数据类型指令吞吐，乘2是因为乘加视作两次浮点运算

例A100：

$$PeakFLOPS = 1.41 * 108 * 64 * 2 = 19491GFLOPS$$

尽量接近理论峰值，**合理选择需要性能优化的部分。不要花力气去优化无足轻重的部分，找到性能瓶颈，然后再开始。**降低非运算类的比例，使得指令更多地用于计算。算法定下来后可以分析带宽占用和SM使用率的大致关系，与实际profile结果比较，就可以知道是带宽还是ALU使用上存在瓶颈。例：一个算法处理xGB global memory里的数据要yG Flop，若：

1. 实测带宽是峰值50%，200G/s，浮点性能是峰值10%，400GFlops，说明带宽是瓶颈。另外，带宽虽是瓶颈，但离100%较远，说明有些地方的延迟隐藏得不够好，导致单元有时处于闲置状态。
2. 实测带宽占用25%，100G/s，浮点性能50%，2000GFlops，计算比带宽与算法相比高很多，说明浮点计算有较大浪费，应该想办法精简计算量。
3. 实测带宽80%，320G/s，浮点性能10%，400GFlops，那说明有些数据可能被多次获取而且没能cache，或者是内存访问合并较差，带宽浪费很多。
   这些判断要反馈回需求限定和精简计算量的步骤，进行相应调整。比如带宽瓶颈就把一些简单的计算现算，计算瓶颈就把一些能存下来的存下来。当然，这只是一个方面，具体性能指标和反馈方式还有很多，必须全盘考虑才能得到比较准确的分析。一个简单的判断是，如果只有一个指标接近峰值，其他都离峰值很远，那优化空间一般很大，如果大多数性能指标都能到峰值的50%~80%，那一般空间就比较有限了，想再优化就需要非常全面具体的微调。如果所有指标都离峰值比较远，说明存在严重的延迟隐藏不足，或是运行参数不合理，需要重新安排程序结构。

2.2 GPU常用优化方法

2.2.1 CUDA指令优化

避免复杂的耗时指令：

1. 除法取模操作，使用移位操作，替换昂贵的除法和取模操作：整数除法和取模操作特别耗时，并应该尽量避免或用位操作来替换：若n为2的指数，则 `i/n=i>>log2(n),i%n=i&(n-1)`
2. 倒数平方根：单精度的倒数平方根应该显式的调用rsqrtf()，双精度的应该调用rsqrt()

3. 其他算术指令：避免自动的双精度到单精度的转换。在以下两种情况下，编译器将时不时的插入转换指令，增加额外的执行周期：
   a 函数操作char或者short时，其值通常要转换到int
   b 双精度浮点常量(没有后缀的浮点数如：1.0是双精度，1.0f是单精度)，作为输入进行单精度浮点计算
   对于b，性能的影响外还有精度的影响，注意到这个不同之处在计算能力为2.x的设备上特别明显
4. 数学库：当速度要求大于精度要求时，建议使用快速数学库：有两种类型的运行时数学操作，可以根据其名字来区分，前面有下划线的和前面没有下划线的：
   比如__functionName(),functionaNmae().有下划线的函数将会被直接映射到硬件级别去执行，其速度快，但是精度较低。并且，其吞吐量大。相反，没有下划线的速度慢，但是精度高，吞吐量小

2.2.2 CUDA内存指令优化

高度推荐：将全局内存的访问降到最低：转而尽可能的使用共享内存
内存指令包括了任何的读写，共享、局部、全局内存的操作。
推荐使用restrict关键字修饰参数，这样可以减少指令的个数，从而提高性能

1. 访存方式
   Global memory：尽量让一个warp中的线程访问连续的一个内存块，实现级联访问（合并访问）
   Shared memory：尽量减少bank conflict，让同一个warp中的线程访问不同的bank
2. 数据分块
   Shared memory block：在SM能够支持的情况下，尽量多地利用此资源提高局部重用性
   Register memory：在shared memory之上可以多加一层寄存器层，进一步提高重用性（寄存器的带宽和延迟都优于共享内存）

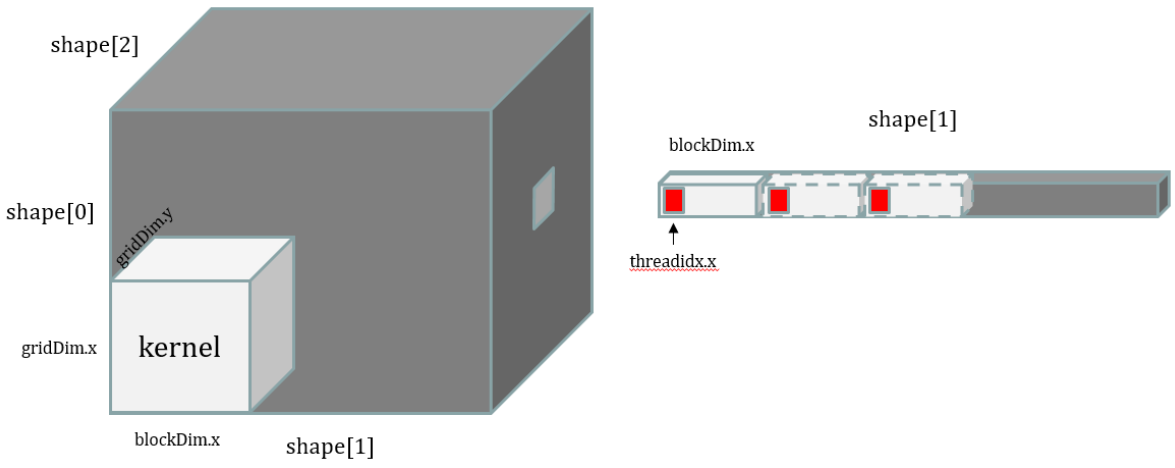
2.2.3 CUDA控制指令优化

if switch do for while，高度推荐：避免在一个warp（32线程）内执行不同的执行路径
任何流控制指令能够显著的影响指令吞吐量：因为其（不执行同一个执行路径）会导致同一个warp内的线程不收敛
当不执行同一个执行路径时，不同的执行路径会被串行化，因为线程束中的所有线程共享同一个程序计数器，这将增加这个warp内的总的指令数
以下需注意：

1. 分支预测，避免一个warp（32线程）内执行不同的分支
2. 循环计数器：有符号vs无符号：在循环中，使用有符号的计数器而不是无符号的计数器
3. 循环中发散线程的同步：高度推荐：避免在一个发散代码中使用__syncthreads(),即，要保证循环中的所有线程都会执行到这个函数，要是有某个线程执行不到这个函数，那就将出错

3 核函数编写示例

编写核函数时，根据数据大小，分配合理的核大小，本例中shape[0]和gridDim.x大小相同，用kernel去计算输入矩阵，得到所需结果。**kernel中所有thread同时执行**。图中线程编号为threadIdx.x的线程将顺序计算红色位置的数据。



以MaxEngine-Lite的softmax计算为例，了解如何创建kernel函数实现cuda编程。cuda编程中求和/最值为归约算法，本例中一并展示。

```
//定义在一个grid中开启的thread实现的计算
//在这个grid中，开了<gridDim.x, gridDim.y>个block
//每个block中，开了blockDim.x个thread
template<typename DType>
__global__ void softmax_simple_kernel(DType *input, DType *output, const int
stride0, const int stride1, const int length, const double temperature, int
log_on){
    using namespace limits;

    //每个blockIdx.y处理gridDim.y上该位置(+=)的有效数据(<)
    for(int idy = blockIdx.y; idy < stride1; idy += gridDim.y){
        //确定处理数据的首地址 in[blockIdx.x, 0, idy]
        DType* in = input + blockIdx.x * stride0 + idy;
        DType* out = output + blockIdx.x * stride0 + idy;

        int tid = threadIdx.x;
        //find max
        //1. find max in length locally
        DType max = MinValue<Dtype>(), sum = 0;
        __shared__ DType temp[256];

        //每个threadIdx.x处理blockDim.x上该位置(+=)的有效数据(<)
        for(int i = tid; i < length; i += blockDim.x){
            if(in[i*stride1] > max){
                max = in[i * stride1];
            }
        }
        temp[tid] = max;
        __syncthreads();
        //2. find block max
        for(int i = blockDim.x >> 1; i>0; i>>=1){
            if(tid < i){
                if(temp[tid] < temp[tid + i]){
                    temp[tid] = temp[tid + i];
                }
            }
             __syncthreads();
        }
        max = temp[0];
        __syncthreads();

        //sum exp
        float sum = 0;
        //1. sum temp
        for(int i = tid; i < length; i += blockDim.x){
            float expval = __expf((float)in[i*stride1] - max);
            sum += expval;
        }
        temp[tid] = sum;
        __syncthreads();

        //2. sum block
        for(int i = blockDim.x >> 1; i > 0; i >>=1){
            if(tid < i>)
                temp[tid] += temp[tid + i];
                __syncthreads();
```

```
        }
        sum = temp[0];
        num = 0;
        for(int i = tid; i < length; i += blockDim.x){
            if(log_on == 0){
                out[i * stride1] = exp(in[i*stride1] - max) / sum;
            }
            else{
                out[i * stride] = in[i*stride1] - (logf(sum) + max);
            }
        }
    }
}


//功能: 输入in [shape[0], shape[1], shape[2]], 对1轴做softmax
template<typename DType>
void Softmax(Stream<gpu> *s, DType* in, DType* out, TShape shape, const double
temperature, int optype){
    using namespace cuda;
    int block_y = shape[2];
    while(block_y > 65535){
        block_y /= 2;
    }
    dim3 num_block(shape[0], block_y);
    int num+thread = maxengine::cuda::kBaseThradNum;
    int log_on = optype == maxengine_op::long_softmax ? 1: 0;
    softmax_simple_kernel<<<num_block, num_thread, num_thread*sizeof(DType), s-
>stream>>> (
        in, out, shape[1]*shape[2], shape[2], shape[1], temperature, log_on);
    MAXENGINE_CUDA_POST_KERNEL_CHECK(softmax_simple_kernel);
}
```