樂不思蜀
我留，我走，我是一个停顿～

CUDA性能优化----kernel调优(nvprof工具的使用) CUDA性能优化----bank conflicts of shared memory

CUDA性能优化----内存篇(二)<coalescing access> & <bank conflicts>问题

2017-01-16 17:06:58 | 分类： HPC&CUDA优化 | 标签： hpc cuda gpu

1、引言

在CUDA性能优化----内存篇(一) 一文中提到了关于global memory 和shared memory的几种内存优化方式，例如coalesced memory access、避免bank conflicts等，本文主要对这几种方式做进一步的分析和学习。由于本人知识和能力的局限性，本篇博文会持续改正和更新。

一个warp包含32个threads。warp是调度和执行的基本单位，half-warp是存储器操作的基本单位，这两个非常重要。在分支的时候，warp大显身手，有合并访问和bank conflict的时候half-warp当仁不让。

每个bank的带宽为32bit = 4byte= 4 char = 1 int = 1float;

只要half-warp中的线程访问的数据在同一个段中，就可以满足合并访问条件。

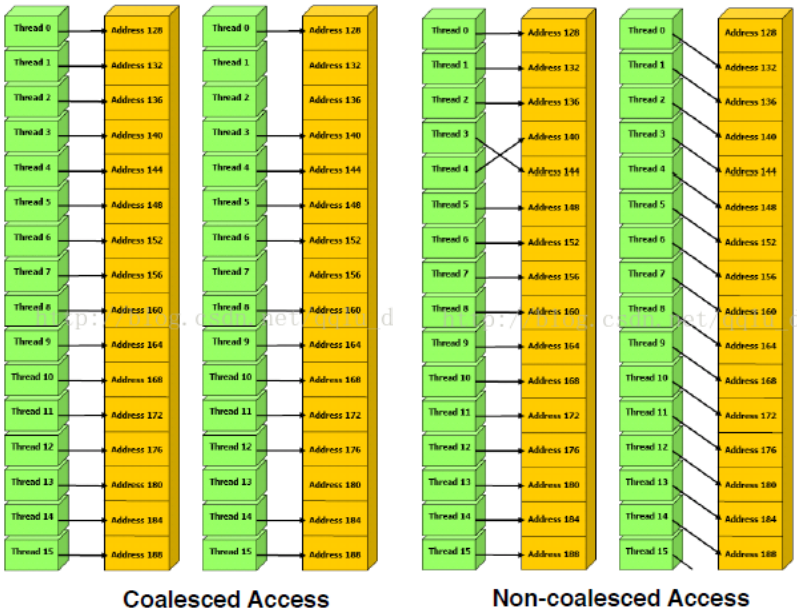
2、coalesced memory access

Global memory是cuda中最常见的存储类型，又叫做Device memory，位于Host主机区域上，它的生命周期是在整个Grid里面，大约具有500个cycle latency。global memory没有被缓存，因此，使用正确的存取模式来获得最大的内存带宽，更为重要，尤其是如何存取昂贵的device memory。

因为对Global memory访问没有缓存，因此显存的性能对GPU至关重要。为

极大的影响效率。此外，多个half-warp的读写操作如果能够满足合并访问（coalesced access），那么多次访存操作会被合并成一次完成，从而提高访问效率。

在cuda并行程序中，尽量用Coalesing accessing的策略来最大化带宽bandwidth。什么是Coalesing accessing呢？如图所示：



对于一个架构的芯片，一个MC（memory controller）两个DRAM chip，如果bus width是32bit，burst length是4的话，那么能够达到最大利用率的一次访存粒度就是32bit * 4 * 2 = 32Byte。如果request size = 64Byte，那么就发射连续的两次访存请求；如果是128Byte，就发射4次。

比如在GT200中，每个MC下属32bit*2的DRAM，然后DRAM的最大Burst长度是8，所以，每个MC最佳访问粒度是， 64bit*8=64Byte 。而GT200有8个MC，所以一次最佳性能，并且对齐的访问，其粒度应该是64Byte*8=512Byte。

而Warp一次访问的最小力度是，32bit*32=128Byte，即，一个Half-warp访存刚好是64Byte，所以一个连续地址空间的Half-warp访存会映射到一个单独的MC上。而如果使用Vector4.float32/int32的格式，那么一个Warp正好可以产生128Byte*4=512Byte的访存粒度！所以合并存储器访问可以最大性能的优化CUDA程序，这即是Coalesced访问模式。每组16 Threads同时访问连续且对齐的64/128 Byte称为Coalesced访问模式，这是达到带宽的理路峰值的必要条件。

There are two characteristics of device memory accesses that you should strive for when optimizing your application:

? Aligned memory accesses

? Coalesced memory accesses

To maximize global memory throughput, it is important to organize memory operations to be both aligned and coalesced.

当half Warp的16个threads在一次memory transaction中coalesced时，Global memory中的带宽得到了最大的利用。其中，需要注意的是，Device在一次transaction中，从global memory中可以一次读取32-bit，64-bit，128-bit，即是4Byte，8Byte，16Byte。例如：

32 bytes (compute capability 1.2+) - each thread reads a short int.

64 bytes - each thread reads a word: int, float, …

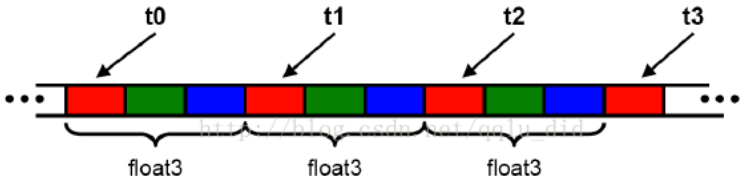
128 bytes - each thread reads a double-word: int2, float2, …

下面有两个实例来说明Global memory中的coalescing access问题：

第一个实例： float3型 Uncoalesced情况

```
__global__ void accessFloat3(float3 *d_in, float3* d_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    float3 a = d_in[index];
    a.x += 2;
    a.y += 2;
    a.z += 2;
    d_out[index] = a;
}
```

在这段代码中，float3类型有12个bytes，不等于要求的4 bytes，8 bytes或16 bytes，half warp读取3个64 bytes中非连续区域，如图：

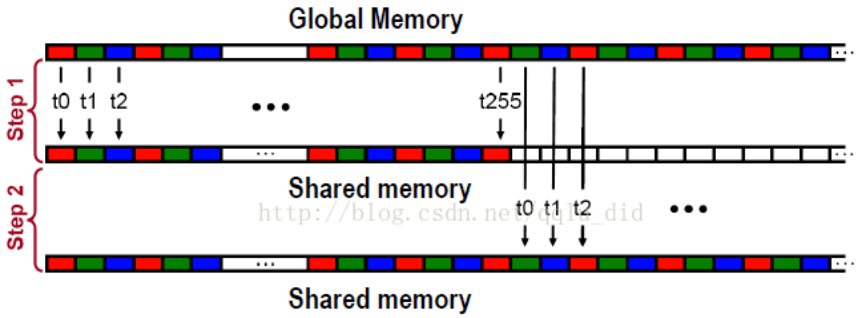


First read

有三种方法可以解决这个问题：

① 使用**shared memory**，也叫做**3-step approach**

假如每个block中使用256个threads，这样一个thread block需要 sizeof(float3)*256 bytes的share memory空间，每个thread读取3个单独的float型，这实质上是指讲输入定义为float型，在核函数里面讲读取在share memory中的float变量转换为float3型并进行操作，最后再转换成float型输出，如图：



Similarly, Step3 starting at offset 512

改进代码如下：

```
{
    int index = 3 * blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float s_data[256*3];
    s_data[threadIdx.x]       = g_in[index];
    s_data[threadIdx.x+256] = g_in[index+256];
    s_data[threadIdx.x+512] = g_in[index+512];
    __syncthreads();
    float3 a = ((float3*)s_data)[threadIdx.x];

    a.x += 2;
    a.y += 2;
    a.z += 2;

    ((float3*)s_data)[threadIdx.x] = a;
    __syncthreads();
    g_out[index]       = s_data[threadIdx.x];
    g_out[index+256] = s_data[threadIdx.x+256];
    g_out[index+512] = s_data[threadIdx.x+512];
}
```

Read the input through SMEM

Compute code is not changed

Write the result through SMEM

如果不好理解的话，假设我们的blockDim=4，取4个float3型变量，我们会发现，每一个thread中输入操作（输出操作一样）为：

Thread 0：

S_data[0]=g_in[0]; S_data[4]=g_in[4]; S_data[8]=g_in[8];

Thread 1：

S_data[1]=g_in[1]; S_data[5]=g_in[5]; S_data[9]=g_in[9];

Thread 2：

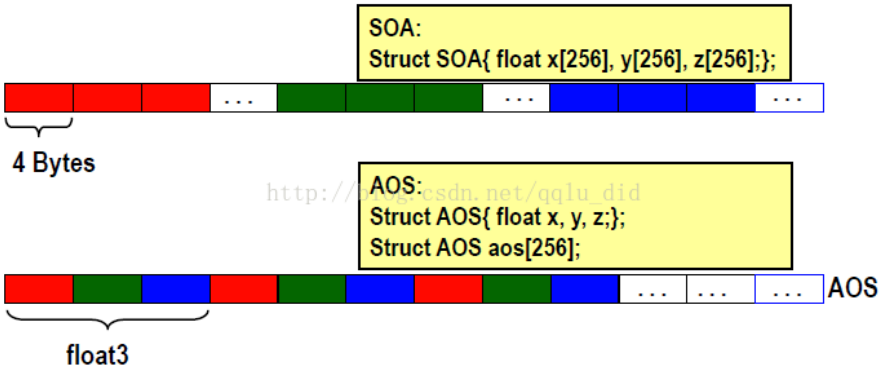
S_data[2]=g_in[2]; S_data[6]=g_in[6]; S_data[10]=g_in[10];

Thread 3：

S_data[3]=g_in[3]; S_data[7]=g_in[7]; S_data[11]=g_in[11];

可以看出，对于每个thread同一时刻（similar step）的数据读入，地址均是连续，这样就达到了coalescing access。

② 使用数组的结构体**(SOA)**来取代结构体的数组**(AOS)**



③ 使用**alignment specifiers**

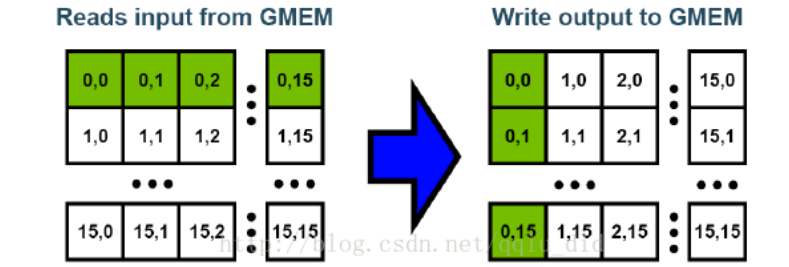
__align__(X), where X = 4, 8, or 16

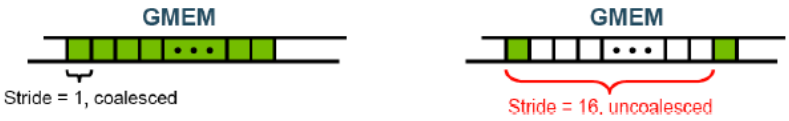
struct __align__(16) { float x; float y; float z; };

尽管这浪费了比较多的空间：

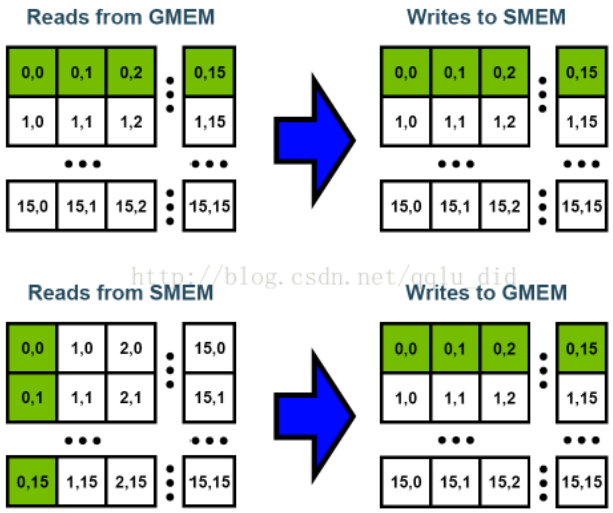
第二个实例： 矩阵转置 Matrix Transpose

一般做法：Uncoalesced Transpose，GMEM为Global memory缩写。





我们发现一般的做法，在写output时，地址是不连续的，即uncoalesced，因此我们利用shared memory存储输入数据，根据转置的关系，来实现coalescing，SMEM为shared memory的缩写，如下图：

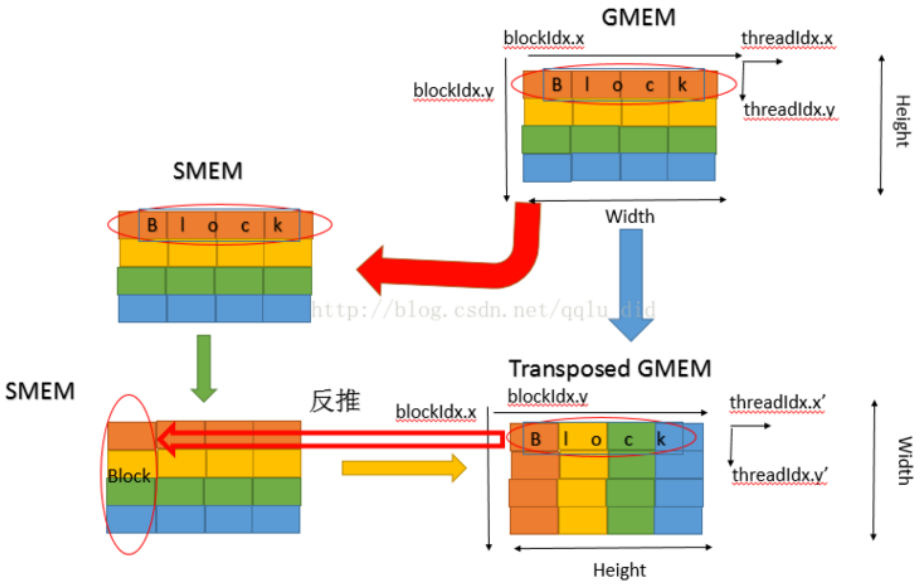


实现代码如下：

```
__global__ void transpose(float *odata, float *idata, int width, int height)
{
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];
    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;
    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;
```

```
    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;
        block[index_block] = idata[index_in];
        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();
    if (xIndex < width && yIndex < height)
        odata[index_out] = block[index_transpose];
}
```

程序的逻辑关系有时还挺绕的，我们以一个4*4矩阵为例，将逻辑关系展示如下：



设dim3 gridDim(4,1), dim3 blockDim(1,4)，以橙色block为例，如输入数据时，将其放入到sharememory中，代码体现在：

```
unsigned int index_in = width * yIndex + xIndex;
unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;
block[index_block] = idata[index_in];
```

接下来的代码实际上是将block的区域给换了,如左下图所示，block换成了一列四种不同颜色的，最终转置的矩阵如右下图所示，从图示可以看出，最终结果的坐标系Height、Width、blockIdx.x、blockIdx.y均对位变换了，这时我们只需要找threadIdx.x'、threadIdx.y'与threadIdx.x、threadIdx.y之间的关系，其实可以看出，一个block里面的坐标系没有发生变换，则threadIdx.x'=threadIdx.x，threadIdx.y'=threadIdx.y，所以代码如下：

```
index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
odata[index_out] = block[index_transpose];
```

总体来说，Global memory中coalescing就是保证其在数据读取或者写入时，使用连续的地址，且地址所存储的变量尺寸为32、64、128 bit，我们常常使用share memory来解决coalescing问题。

bank conflicts 如何产生的?

对GPU来说，local memory是由banks组成的，每个bank是32bit，可视化图如下。bank是实际存

```
Bank    |      1      |      2      |      3      |...
Address |  0  1  2  3 |  4  5  6  7 |  8  9 10 11 |...
Address | 64 65 66 67 | 68 69 70 71 | 72 73 74 75 |...
```

在编程过程中，有静态的shared memory 和动态的shared memory；

静态的shared memory 在程序中定义：__shared__ type shared[SIZE];

动态的shared memory 通过内核函数的每三个参数设置大小： extern __shared__ type shared[];

那么问题来了，为什么 shared memory 存在 bank conflict，而 global memory 不存在? 因为访问 global memory 的只能是 block，而访问 shared memory 的却是同一个 half-warp 中的任意线程。

引用风辰CUDA入门教程的一段话:

Tesla 的每个 SM 拥有 16KB 共享存储器，用于同一个线程块内的线程间通信。为了使一个 half-warp 内的线程能够在一个内核周期中并行访问，共享存储器被组织成 16 个 bank，每个 bank 拥有 32bit 的宽度，故每个 bank 可保存 256 个整形或单精度浮点数，或者说目前的bank 组织成了 256 行 16 列的矩阵。如果一个 half-warp 中有一部分线程访问属于同一bank 的数据，则会产生 bank conflict，降低访存效率，在冲突最严重的情况下，速度会比全局显存还慢，但是如果 half-warp 的线程访问同一地址的时候，会产生一次广播，其速度反而没有下降。在不发生 bank conflict 时，访问共享存储器的速度与寄存器相同。在不同的块之间，共享存储器是毫不相关的。

里面说的很清楚的一点就是每个bank有1KB的存储空间。

Shared memory 是以 4 bytes 为单位分成 banks。因此，假设以下的数据:

```
__shared__ int data[128];
```

那么，data[0] 是 bank 0、data[1] 是 bank 1、data[2] 是 bank 2、...、data[15] 是bank 15，而 data[16] 又回到 bank 0。由于 warp 在执行时是以 half-warp 的方式执行，因此分属于不同的 half warp 的 threads，不会造成 bank conflict。

const int tid = threadIdx.x;

因此，如果程序在存取 shared memory 的时候，使用以下的方式:

```
int number = data[base + tid];
```

那就不会有任何 bank conflict，可以达到最高的效率。但是，如果是以下的方式:

```
int number = data[base + 4 * tid];
```

那么，thread 0 和 thread 4 就会存取到同一个 bank，thread 1 和 thread 5 也是同 样，这样就会造成 bank conflict。在这个例子中，一个 half warp 的 16 个 threads 会有四个threads 存取同一个 bank，因此存取 share memory 的速度会变成原来的 1/4。

CUDA编程中,一个half-warp（16个threads）访问连续的32bit地址,不会有bank conflicts。一个重要的例外是，当多个 thread 存取到同一个 shared memory 的地址时，shared memory 可以将这个地址的 32 bits 数据「广播」到所有读取的 threads，因此不会造成 bank conflict。例如:

```
int number = data[3];
```

这样不会造成 bank conflict，因为所有的 thread 都读取同一个地址的数据。

很多时候 shared memory 的 bank conflict 可以透过修改数据存放的方式来解决。例如，以下的程序:

```
data[tid] = global_data[tid];
...
int number = data[16 * tid];
```

会造成严重的 bank conflict，为了避免这个问题，可以把数据的排列方式稍加修改，把存取方式改成:

```
int column = tid % 16;
data[row * 17 + column] = global_data[tid];
...
int number = data[17 * tid];
```

这样就不会造成 bank conflict 了。

简单的说，矩阵中的数据是按照bank存储的，第i个数据存储在第i个bank中。一个block要访问shared memory，只要能够保证以其中相邻的16个线程一组访问thread，每个线程与bank是一一对应就不会产生bank conflict。否则会产生bank conflict，访存时间成倍增加，增加的倍数由一个bank最多被多少个thread同时访问决定。有一种极端情况，就是所有的16个thread同时访问同一bank时反而只需要一个访问周期，此时产生了一次广播。

下面有一些小技巧可以避免bank conflict 或者提高global存储器的访问速度：

1. 尽量按行操作，需要按列操作时可以先对矩阵进行转置；
2. 划分子问题时，使每个block处理的问题宽度恰好为16的整数倍，使得访存可以按照s_data[tid]=i_data[tid]的形式进行；
3. 使用对齐的数据格式，尽量使用nvidia定义的格式如float3,int2等，这些格式本身已经对齐；
4. 当要处理的矩阵宽度不是16的整数倍时，将其补为16的整数倍，或者用malloctopitch而不是malloc；
5. 利用广播。例如：

```
s_odata[tid] = tid%16 < 8 ? s_idata[tid] : s_idata[15];
```

会产生8路的块访问冲突，而用：

```
s_odata[tid]=s_idata[15];s_odata[tid]= tid%16 < 8 ? s_idata[tid] : s_data[tid];
```

则不会产生块访问冲突。

参考：CUDA性能优化----内存篇(一)

CUDA性能优化----kernel调优(nvprof工具的使用) CUDA性能优化----bank conflicts of shared memory