【CUDA 基础】5.5 常量内存

2018-06-06 | CUDA , Freshman | 0 |

Abstract: 本文介绍另外两种内存——常量内存，只读缓存

Keywords: CUDA常量内存，CUDA只读缓存

常量内存

常量内存

本文介绍常量内存和只读缓存，常量内存是专用内存，他用于只读数据和线程束统一访问某一个数据，常量内存对内核代码而言是只读的，但是主机是可以修改（写）只读内存的，当然也可以读。

注意，常量内存并不是在片上的，而是在DRAM上，而其有在片上对应的缓存，其片上缓存就和一级缓存和共享内存一样， 有较低的延迟，但是容量比较小，合理使用可以提高内和效率，每个SM常量缓存大小限制为64KB。

我们可以发现，所有的片上内存，我们是不能通过主机赋值的，我们只能对DRAM上内存进行赋值。

每种内存访问都有最优与最坏的访问方式，主要原因是内存的硬件结构和底层设计原因，比如全局内存按照连续对去访问最优，交叉访问最差，共享内存无冲突最优，都冲突就会最差，其根本原因在于硬件设计，而我们的常量内存的最优访问模式是线程束所有线程访问一个位置，那么这个访问是最优的。如果要访问不同的位置，就要编程串行了，作为对比，这种情况相当于全局内存完全不连续，共享内存的全部冲突。

数学上，一个常量内存读取成本与线程束中线程读取常量内存地址个数呈线性关系。

常量内存的声明方式：

```
__constant
```

常量内存变量的生存周期与应用程序生存周期相同，所有网格对声明的常量内存都是可以访问的，运行时对主机可见，当CUDA独立编译被使用的，常量内存跨文件可见，这个要后面才会介绍。

初始化常量内存使用一下函数完成

```
cudaError_t cudaMemcpyToSymbol(const void *symbol, const void * src,  size_t count,
```

和我们之前使用的copy到全局内存的函数类似，参数也类似，包含传输到设备，以及从设备读取，kind的默认参数是传输到设备。

使用常量内存实现一维模板

在数值分析中经常使用模板操作，好吧，我们还没学数值分析的这一课，但是说个其他名词你肯定特别熟悉——卷积，一维的卷积就是我们今天要写的例子，所谓模板操作就是把一组数据中中间连续的若干个产生一个输出：

$$\{x-4h, x-3h, x-2h, x-h, x, x+h, x+2h, x+3h, x+4h\}$$

A nine-point stencil is illustrated in Figure 5-18.

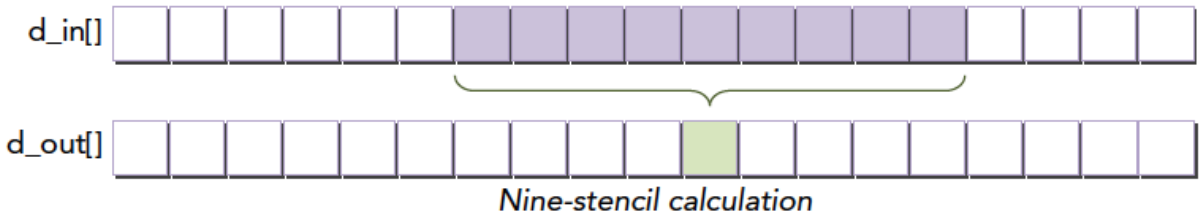


FIGURE 5-18

紫色的输入数据通过某些运算产生一个输出——绿色的块，这个在图像处理里面用处非常多，当然图像中使用的是二维的数据，我们来看一下公式：

$$f'(x) \approx c_3(f(x+4h)-f(x-4h)) + c_2(f(x+3h)-f(x-3h)) \\ + c_1(f(x+2h)-f(x-2h)) + c_0(f(x+h)-f(x-h))$$

此公式计算函数的导数，使用八阶差分来近似.

注意，这里的参数写的跟书中的不一样，也就是系数 c_0 对应原文 c_3

分析算法，整个算法中我们的 $c_i, 0 < i < 4$ 被所有线程使用，对于某一固定计算我们可以把他声明到寄存器中，但是如果数量较大或者不针对某一组 c 的时候，就需要动态的传递，那么此时，常量内存会是最好的选择，因为其对于核函数的只读属性，线程束内以广播形式访问，也就是32个线程只需要一个内存事务就能完成读取，这样效率是非常高的。

我们观察上面图片，可以发现，每个输入数据要被使用8次，如果每次都从全局内存读，这个延迟过高，所以结合上一篇讲到的共享内存，用共享内存缓存输入数据，得到下面的代码：

```
__global__ void stencil_1d(float * in,float * out)
{
    //1.
    __shared__ float smem[BDIM+2*TEMP_RADIO_SIZE];
    //2.
    int idx=threadIdx.x+blockDim.x*blockIdx.x;
    //3.
    int sidx=threadIdx.x+TEMP_RADIO_SIZE;
    smem[sidx]=in[idx];
    //4.
    if (threadIdx.x<TEMP_RADIO_SIZE)
    {
```

```
        if(idx>TEMP_RADIO_SIZE)
            smem[sidx-TEMP_RADIO_SIZE]=in[idx-TEMP_RADIO_SIZE];
        if(idx<gridDim.x*blockDim.x-BDIM)
            smem[sidx+BDIM]=in[idx+BDIM];

    }
    __syncthreads();
    //5.
    if (idx<TEMP_RADIO_SIZE||idx>=gridDim.x*blockDim.x-TEMP_RADIO_SIZE)
        return;
    float temp=.0f;
    //6.
    #pragma unroll
    for(int i=1;i<=TEMP_RADIO_SIZE;i++)
    {
        temp+=coef[i-1]*(smem[sidx+i]-smem[sidx-i]);
    }
    out[idx]=temp;
    //printf("%d:GPU :%lf,\n",idx,temp);
}
```

完整的代码在github:https://github.com/Tony-Tan/CUDA_Freshman（欢迎随手star😝）

这里我们要面对一个填充问题，无论是观察公式还是图示，我们会发现，如果模板大小是9，那么我们输出的前4个数据是没办法计算的因为要使用第-1，-2，-3，-4位置的数据，最后4个数据也是不能计算的，因为他要使用 $n+1, n+2, n+3, n+4$ 的数据，这些数据也是没有的，为了保证计算过程中访问不会越界，我们把输入数据两端进行扩充，也就是把上面虽然没有，但是要用的数据填充到输入数据中，当我们我们要处理的是中间的某段数据的时候，那么填充位的数据就来自前面的块对应的输入数据，或者后面线程块对应的输入数据，这就是上面代码最困难的地方，这里懂了其他就一马平川了~

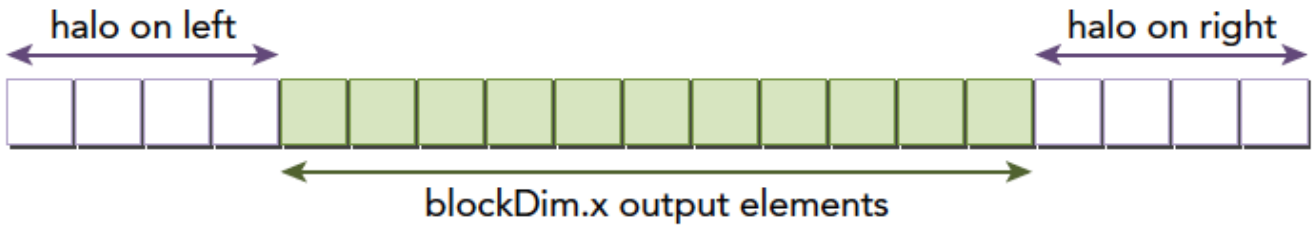
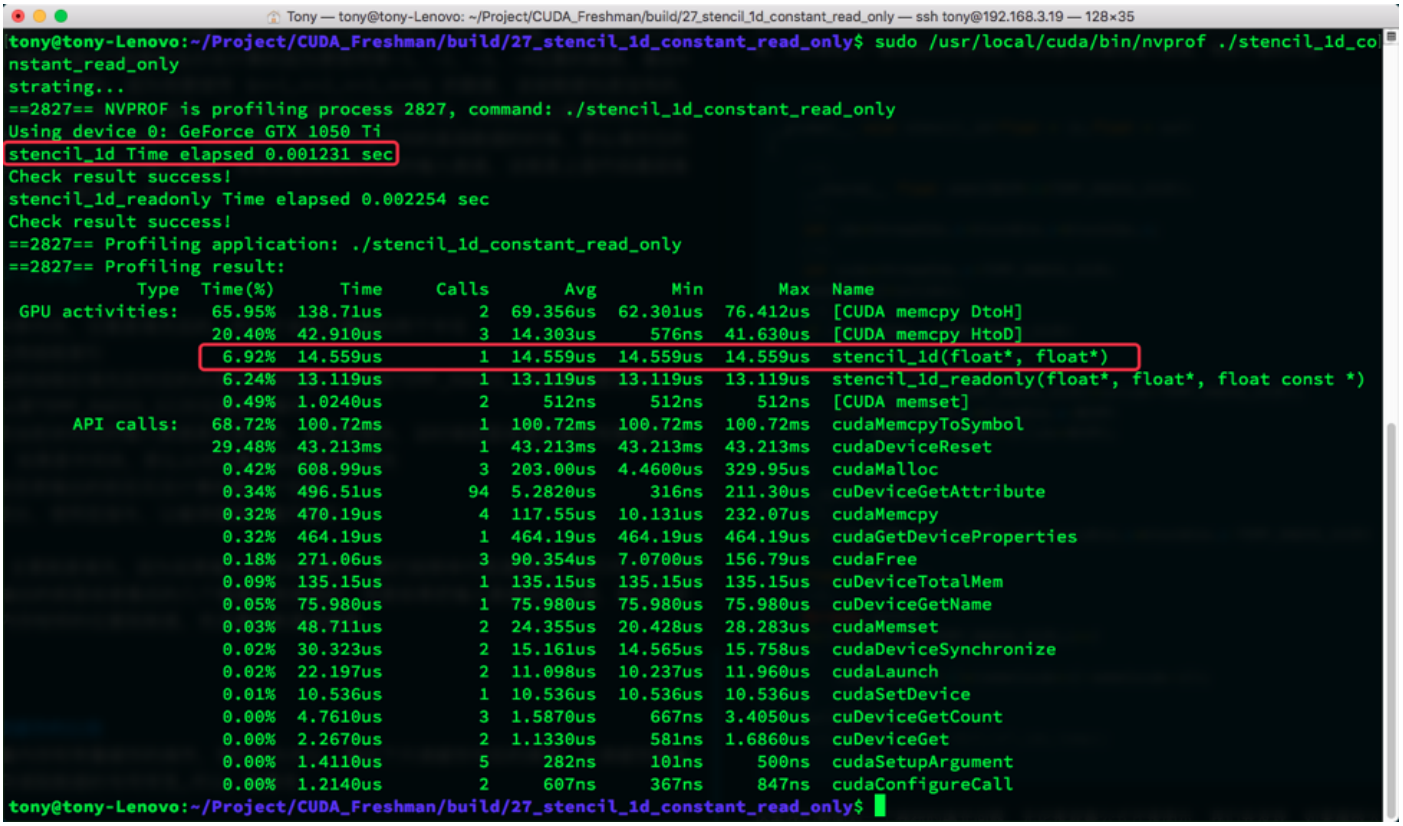


FIGURE 5-19

1. 定义共享内存，注意是填充后的，所以才会加了模板的两个半径
2. 计算全局线程索引

3. 计算当前线程在填充后对应的共享内存的位置，因为前TEMP_RADIO_SIZE数据是填充位，所以我们从第TEMP_RADIO_SIZE位置进行操作。
4. 判断是当前块对应的输入数据是否是首块，或者是尾块，这时候前面或者后面没有数据可以用来填充，如果是中间块，那么从对位置读取数据进行填充
5. 判断是否是输出的前后无法计算的那几个位置
6. 进行差分，使用宏指令，让编译器自己展开循环

困难在4，主要就是填充，因为如果输入数据当做整体，我们按照串行思路来做，我们可以不填充，而只把输出的前面或者最后的几个数据无效掉就好，但是如果把输入数据分块处理，那么就涉及从全局内存相邻的位置取数据，而且要判断是否越界。



与只读缓存的比较

以上是常量内存和常量缓存的操作，我们作为对比，展示下只读缓存对应的操作，只读缓存拥有从全局内存读取数据的专用带宽,所以，如果内核函数是带宽限制型的，那么这个帮助是非常大的，不同的设备有不同的只读缓存大小，Kepler SM有48KB的只读缓存，只读缓存对于分散访问的更好，当所有线程读取同一地址的时候常量缓存最好，只读缓存这时候效果并不好，只读换粗粒度为32.

实现只读缓存可以使用两种方法

- 使用__ldg函数
- 全局内存的限定指针

使用__ldg()的方法：

```
__global__ void kernel(float* output, float* input) {
...
output[idx] += __ldg(&input[idx]);
...
}
```

使用限定指针的方法：

```
void kernel(float* output, const float* __restrict__ input) {
...
output[idx] += input[idx];
}
```

使用只读缓存需要更多的声明和控制，在代码非常复杂的情况下以至于编译器都没办法保证制度缓存使用是否安全的情况下，建议使用 __ldg()函数，更容易控制。
只读缓存独立存在，区别于常量缓存，常量缓存喜欢小数据，而只读内存加载的数据比较大，可以再非统一模式下访问，我们修改上面的代码，得到只读缓存版本：

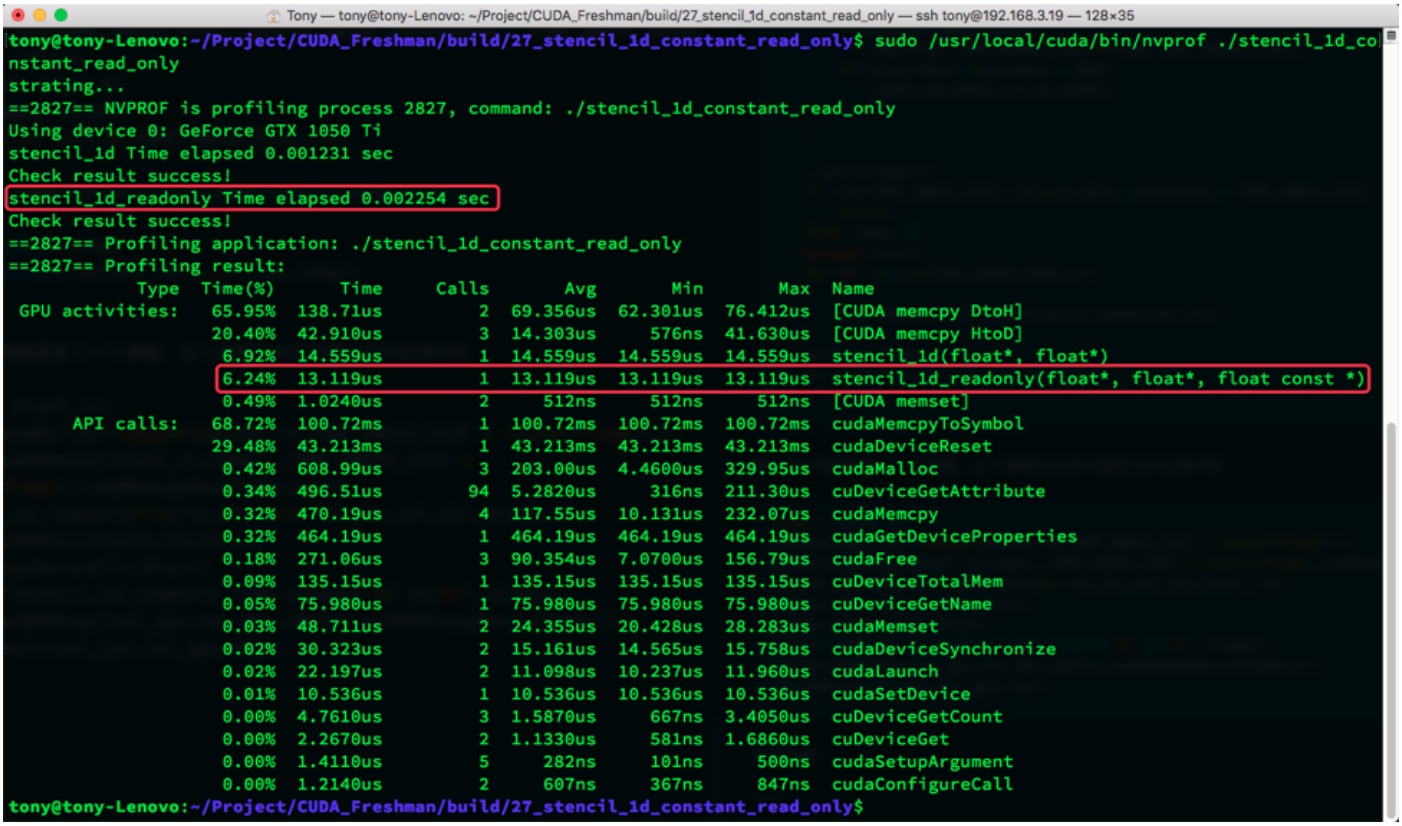
```
__global__ void stencil_1d_readonly(float * in,float * out,const float* __restrict_
{
    __shared__ float smem[BDIM+2*TEMP_RADIO_SIZE];
    int idx=threadIdx.x+blockDim.x*blockIdx.x;
    int sidx=threadIdx.x+TEMP_RADIO_SIZE;
    smem[sidx]=in[idx];

    if (threadIdx.x<TEMP_RADIO_SIZE)

    {
        if(idx>TEMP_RADIO_SIZE)
            smem[sidx-TEMP_RADIO_SIZE]=in[idx-TEMP_RADIO_SIZE];
        if(idx<gridDim.x*blockDim.x-BDIM)
            smem[sidx+BDIM]=in[idx+BDIM];

    }
```

```
    __syncthreads();
    if (idx<TEMP_RADIO_SIZE||idx>=gridDim.x*blockDim.x-TEMP_RADIO_SIZE)
        return;
    float temp=.0f;
    #pragma unroll
    for(int i=1;i<=TEMP_RADIO_SIZE;i++)
    {
        temp+=dcoef[i-1]*(smem[sidx+i]-smem[sidx-i]);
    }
    out[idx]=temp;
    //printf("%d:GPU :%lf,\n",idx,temp);
}
```

唯一的不同就是多了一个参数，这个参数在主机内是定义的全局内存

```
float * dcoef_ro;
CHECK(cudaMalloc((void**)&dcoef_ro,TEMP_RADIO_SIZE * sizeof(float)));
CHECK(cudaMemcpy(dcoef_ro,templ_,TEMP_RADIO_SIZE * sizeof(float),cudaMemcpyHostToDev
stencil_1d_readonly<<<grid,block>>>(in_dev,out_dev,dcoef_ro);
CHECK(cudaDeviceSynchronize());
iElaps=cpuSecond()-iStart;
printf("stencil_1d_readonly Time elapsed %f sec\n",iElaps);
CHECK(cudaMemcpy(out_gpu,out_dev,nBytes,cudaMemcpyDeviceToHost));
checkResult(out_cpu,out_gpu,nxy);
```

执行结果：



总结

常量内存和只读缓存：

- 对于核函数都是只读的
- SM上的资源有限，常量缓存64KB，只读缓存48KB
- 常量缓存对于统一读取（读同一个地址）执行更好
- 只读缓存适合分散读取

相关文章

- 【Julia】整型和浮点型数字
- 【Julia】变量
- 【Julia】开始使用Julia