Nvidia Assignment01: 解题思路与优化过程

谢泽辉

题干： The provided file contains a simple kernel and full code to measure its performance, please optimize this kernel to maximize its performance. The data dimensions are to remain the same, ie. you can assume that they will not change during our testing and that it is acceptable for your optimizations to rely upon them. You can change the kernel itself in any way you like, naturally the mathematical operation for each output element mustbe the same. You can also change the kernel launch configuration in any way you like, i.e. you can select the grid and block dimensions as you see fit. However, do not modify the for-loop that launches the kernel 'nreps' times (line 40) since this loop is simply there to average the elapsed time over several launches. Please provide a brief summary of the optimizations you apply, a sentence or two for each optimization is sufficient.

作者	系统	环境	GitHub
谢泽辉	Windows 10.0.19042	CUDA 10.1, NVIDIA Driver 446.47, VS2019, Nsight 2020	Mrphase

原始代码

```
__global__ void kernel_A(float* g_data, int dimx, int dimy)
{
    int ix = blockIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = iy * dimx + ix;

    float value = g_data[idx];

    if (ix % 2)
    {
        value += sqrtf(logf(value) + 1.f);
    }
    else
    {
        value += sqrtf(cosf(value) + 1.f);
    }

    g_data[idx] = value;
}

elapsed_time_ms = timing_experiment(kernel_A, d_data, h_data , nbytes,
    dimx, dimy, nreps, 1, 512);
```

初步分析

通过 Nsight compute 分析可见 SM 与 Memory 占用都在60%以下，初步判断问题类型为Occopuancy Bottleneck。 其中ix = blockIdx.x; 且 int idx = iy * dimx + ix; 当设置blocksize(dim.x, dim.y)=(1,512) 时，工作线

程数量为iy * 1* blockIdx.x， 尝试更改 blocksize(dim.x, dim.y)=(512,1)，ix = blockIdx.x* blockDim.x + threadIdx.x; 可提高占用率。

```
__global__ void kernel__Divergence1(float* g_data, int dimx, int dimy)
{
    int ix = blockIdx.x* blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = iy * dimx + ix;

    float value = g_data[idx];

    value += (ix % 2? sqrtf(logf(value) + 1.f) : sqrtf(cosf(value) + 1.f));

    g_data[idx] = value;
}
elapsed_time_ms = timing_experiment(kernel__Divergence1, d_data, h_data,
nbytes,//zehui
    dimx, dimy, nreps, 512, 1);
```

减少warp divergence

使用两个kernel分别计算 sqrtf(logf(value) + 1.f) 与 sqrtf(cosf(value) + 1.f); 两个核函数中，每次移动步长为2。

```
__global__ void kernel__Divergence2_log(float* g_data, int dimx, int dimy)
{
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = iy * dimx + (2 * ix+1);

    float value = g_data[idx];

    value +=  sqrtf(logf(value) + 1.f);

    g_data[idx] = value;
}

__global__ void kernel__Divergence2_cos(float* g_data, int dimx, int dimy)
{
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = iy * dimx + (2 * ix );

    float value = g_data[idx];

    value += sqrtf(cosf(value) + 1.f);

    g_data[idx] = value;
}
elapsed_time_ms = timing_experiment_Divergence2( d_data, h_data, nbytes,//zehui
    dimx, dimy, nreps, 512, 1);
```

修改测试函数, 其中，Grid size 减小为原来的1/2， 先后发射两个kernel。

```
float timing_experiment_Divergence2( float* d_data, float* h_data, int nbytes,
    int dimx, int dimy, int nreps, int blockx, int blocky)
{
    //....
    dim3 block(blockx, blocky);
    dim3 grid(dimx / block.x/2, dimy / block.y);

    //....
        for (int i = 0; i < nreps; i++) {    // do not change this loop, it's not
part of the algorithm - it's just to average time over several kernel launches
        kernel__Divergence2_log << <grid, block >> > (d_data, dimx, dimy);
        kernel__Divergence2_cos << <grid, block >> > (d_data, dimx, dimy);
        }
    //....
}
```

使用查表法

灵感来源于Leetcode的打表法，对于输入/输出结果数量可控时，可预先计算输出结果，在按照条件查询，减少计算量。

观察输入数据的生成， 发现输入数据范围为 10-265 的整数 故输出只有 2*255 种，考虑到输入数据量为2 x 1024 x 2 x 1024， 满足查表法使用条件。

预先生成结果表

输入 table 是长度为 510 的一维数组，分为两部分：sqrtf(cosf(value) + 1.f); 的计算结果保存至 [0-254], sqrtf(logf(value) + 1.f); 的计算结果保存至 [255-590]。

```
__global__ void kernel__GeneTable(float* table, int max_lut_val, int niterations)
{
    int ix = threadIdx.x;

    if (blockIdx.x == 0)
    {
        float value = ix + 10;
            value += sqrtf(logf(value) + 1.f);
        table[ix + max_lut_val] = value;
    }
    else
    {
        float value = ix + 10;
            value += sqrtf(cosf(value) + 1.f);
        table[ix] = value;
    }
```

```
}
kernel__GeneTable << <2, max_lut_val >> > (d_table, max_lut_val,  nreps);
```

查表

每个线程检查表中查找结果。 由于Timing_experiment（）函数会修改结果10次，但是生成的表是静态的，在调用timing_experiment（）之后，两种方法的输出会略有差异。

```
__global__ void kernel__Table(float* g_data, float* table, int max_lut_val, int
dimx, int dimy) {
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = iy * dimx + ix;
    if ((ix < dimx) && (iy < dimy)) {
        int tableID = (int)roundf(g_data[idx] - 10);
        if (tableID < max_lut_val) {
            tableID = (ix % 2) ? (tableID + max_lut_val) : tableID;
            g_data[idx] = table[tableID];
        }
    }
}
timing_experiment_Table(d_data, d_table, 255, nbytes, dimx, dimy, nreps, 512, 1);
```

向量化内存访问（Vectorized Memory Access）

参考： Vectorized Memory Access

使用向量加载和存储，以帮助提高带宽利用率，同时减少已执行指令的数量。

以下代码中每个线程处理 KERNEL_X_DIM2 * KERNEL_Y_DIM2 共4租数据， Grid, block size 相应缩小1/2

```
#define KERNEL_X_DIM2 2
#define KERNEL_Y_DIM2 2
__global__ void kernel__device_copy_vector2(float* g_data, const int dimx, const
int dimy) {
    const int ix = blockIdx.x * blockDim.x + threadIdx.x;
    const int iy = blockIdx.y * blockDim.y + threadIdx.y;
    const int idx = KERNEL_Y_DIM2 * iy * dimx + ix;
    float2* f2_data = reinterpret_cast<float2*>(g_data);
    float2 values0 = f2_data[idx + 0 * dimx];
    float2 values1 = f2_data[idx + 1 * dimx];
    //compute
    values0.x += sqrtf(cosf(values0.x) + 1.f);
    values0.y += sqrtf(logf(values0.y) + 1.f);
    values1.x += sqrtf(cosf(values1.x) + 1.f);
    values1.y += sqrtf(logf(values1.y) + 1.f);
    //restore anwser
    f2_data[idx + 0 * dimx] = values0;
```

```
    f2_data[idx + 1 * dimx] = values1;
}
elapsed_time_ms = timing_experiment(kernel__device_copy_vector2,
d_data,h_data,nbytes,
    dimx / KERNEL_X_DIM2, dimy / KERNEL_Y_DIM2, nreps, 256, 1);
```

输出

```
allocated 16.00 MB on GPU
allocated 16.00 MB on CPU

input: print_list:
 51 45 200 142 235 118 224 184 92 154 ...

============================kernel_A (origional)============================
print_list:
 53.4068 67.3596 204.203 166.555 235.619 142.202 229.336 209.047 97.3885 178.709
...
 A:      1.12 ms
CUDA: no error

============================kernel__Divergence1_Occopuancy=========================
===
print_list:
 53.4068 67.3596 204.203 166.555 235.619 142.202 229.336 209.047 97.3885 178.709
...
 kernel__Divergence1_Occopuancy:      0.23 ms
CUDA: no error
res correct

============================kernel__Divergence2============================
print_list:
 53.4068 67.3596 204.203 166.555 235.619 142.202 229.336 209.047 97.3885 178.709
...
 kernel__Divergence2:      0.24 ms
CUDA: no error
res correct

============================kernel__Table============================
===h_data[i] = 10.f + rand() % 256;, range from 10-265 unique values==========
===Since timing_experiment() function modify restult 10 times,
But our table is static, this mehtod may not output exact same value after call
timing_experiment()===
print_list:
 53.2859 65.2678 204.144 162.465 235.431 138.432 229.237 209.516 97.2736 174.479
...
 kernel__Table:      0.24 ms
CUDA: no error

============================kernel_kernel__device_copy_vector2 Vectorized Memory
```

```
Access============================
print_list:
 53.4068 67.3596 204.203 166.555 235.619 142.202 229.336 209.047 97.3885 178.709
...
 kernel__device_copy_vector2:      0.12 ms
CUDA: no error
res correct
```