به نام خدا

تمرین ۶

علیرضا آخوندی

۹۷۳۱۱۰۷

کد کرنل هر بخش داخل فایل rediction.cu قرار گرفته است، راه اول در تابع کرنل reduce0 ، راه دوم در reduce1 تا راه حل پنجم که در تابع reduce4 قرار گرفته است.

برای اجرای فایل باید دستورات زیر را اجرا کنیم :

```
nvcc reduction.cu -o reduction
./reduction
```

نمونه ای از خروجی برنامه :

```
(base) ┌─alirezaak@ALIPC ~/multicore programming/HW6
└─$ ./reduction
[Matrix Reduction Using CUDA] - Starting...
GPU Device 0: "NVIDIA GeForce GTX 1050" with compute capability 6.1

Array with size (4194304)
serial execution : 9.277000 ms
total serial execution : 29.435999 ms

 num implementation : 1
grid size : 32768 block size : 128 number of threads : 4194304
Elapsed time in msec = 1.813216 and bandwidth 9.252740 GB/s result = 4194304
execution + memory allocations : 47.456997 ms

 num implementation : 2
grid size : 32768 block size : 128 number of threads : 4194304
Elapsed time in msec = 1.073440 and bandwidth 15.629392 GB/s result = 4194304
execution + memory allocations : 5.901000 ms

 num implementation : 3
grid size : 32768 block size : 128 number of threads : 4194304
Elapsed time in msec = 0.891168 and bandwidth 18.826098 GB/s result = 4194304
execution + memory allocations : 5.158000 ms

 num implementation : 4
grid size : 16384 block size : 128 number of threads : 2097152
Elapsed time in msec = 0.464384 and bandwidth 36.127895 GB/s result = 4194304
execution + memory allocations : 4.384000 ms

 num implementation : 5
grid size : 16384 block size : 128 number of threads : 2097152
Elapsed time in msec = 0.309920 and bandwidth 54.134019 GB/s result = 4194304
execution + memory allocations : 4.623000 ms
```

Peak bandwidth مربوط به هر کرنل به ترتیب آمده در اسلاید ها به دست آمده اند. این مقدار داده انتقال داده شده به global memory بر زمان اجرای تابع به دست آمده است. زمان دیگری که دیده می شود نیز زمان اجرای هر

کرنل به همراه زمان انتقال های داده بین فضای آدرس دهی cpu و gpu است.در ابتدای خروجی نیز مدل پردازنده گرافیکی ، سایز ورودی مساله و زمان اجرای سریال برنامه دیده می شود.

برای پردازنده ی Geforce GTX 1050 داریم :

Peak bandwidth = 128 bit * 1752 MHz / 2 = 112.1 GB/s

منبع

کرنل گام اول :
مشکل پیاده سازی اول branching بیش از حد است.همانطور که می دانیم نخ های gpu به صورت SIMD بر روی جریان داده عمل می کنند. حال شرط داخل این حلقه تعداد زیادی از نخ ها را معلق می کند. همین باعث پایین آمدن کارایی می شود.

```
__global__ void reduce0(int *g_idata, int *g_odata)
{

    __shared__ int sdata[BLOCK_SIZE];
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    // printf("%d\n" , i);
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for (unsigned int s = 1; s < blockDim.x; s *= 2)
    {
        if (tid % (2 * s) == 0)
        {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0)
        g_odata[blockIdx.x] = sdata[0];
}
```

کرنل گام دوم :
در این روش برای بهبود نسبت روش قبلی از یک strided index برای پیمایش استفاده می کنیم تا مقدار branching را کاهش دهیم.

```
__global__ void reduce1(int *g_idata, int *g_odata)
{

    __shared__ int sdata[BLOCK_SIZE];
```

```
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for (unsigned int s = 1; s < blockDim.x; s *= 2)
    {
        int index = 2 * s * tid;
        if (index < blockDim.x)
        {
            sdata[index] += sdata[index + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0)
        g_odata[blockIdx.x] = sdata[0];
}
```

کرنل گام سوم :

مشکل کرنل قبلی برخورد در shared memory های بلاک ها است.برای حل این مشکل هر نخ را مسئول پر کردن خانه متناظر با آیدی خود در reduction بزند.

```
__global__ void reduce2(int *g_idata, int *g_odata)
{

    __shared__ int sdata[BLOCK_SIZE];
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
    {
        if (tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0)
        g_odata[blockIdx.x] = sdata[0];
}
```

کرنل چهارم :

در کرنل سوم با جلو رفتن حلقه ، در هر گام تعداد نخ های بیکار دو برابر می شود. بنابراین برای حل این مشکل باید تعداد بلاک ها را نصف کرده و یک دستور load قبل از حلقه را با دو دستور load و یک دستور add جایگزین می کنیم.این کار از هدر رفت نخ ها جلوگیری می کند.

```
__global__ void reduce3(int *g_idata, int *g_odata)
{

    __shared__ int sdata[BLOCK_SIZE];
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x * 2 + threadIdx.x;
    if (i < N)
    {
        sdata[tid] = g_idata[i] + g_idata[i + blockDim.x];
        __syncthreads();
        // do reduction in shared mem
        for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
        {
            if (tid < s)
            {
                sdata[tid] += sdata[tid + s];
            }
            __syncthreads();
        }
        // write result for this block to global mem
        if (tid == 0)
            g_odata[blockIdx.x] = sdata[0];
    }
}
```

کرنل پنجم :

بهبودی که می توان نسبت به روش قبلی داشت کمک کردن instruction overhead است.بعضی از دستورات که دستوراتی فرعی هستند (نه عملیات ریاضی نه عملیات load و store) برای پردازنده ها overhead محسوب می شوند (مانند دستورات ریاضی مربوط آدرس ها و دستورات مربوط به حلقه ها) . هنگامی که در حلقه لوپ مورد نظر به مقدار 32 می‌رسیم در واقع برای ما تنها یک warp باقی می ماند و همانطور که می دانیم برای نخ های داخل یک warp دستورات به صورت SIMD اجرا می‌شوند ، بنابراین بهتر است که ۶ قدم آخر حلقه را باز کنیم و به صورت تکی اجرا کنیم.

```
__global__ void reduce4(int *g_idata, int *g_odata, int size)
{

    __shared__ int sdata[BLOCK_SIZE];
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x * 2 + threadIdx.x;

    if (i < size)
    {
        sdata[tid] = g_idata[i] + g_idata[i + blockDim.x];
        __syncthreads();
        // do reduction in shared mem
        for (unsigned int s = blockDim.x / 2; s > 32; s >>= 1)
        {
            if (tid < s)
            {
                sdata[tid] += sdata[tid + s];
            }
```

```
            __syncthreads();
        }

        if (tid < 32)
        {
            sdata[tid] += sdata[tid + 32];
            __syncthreads();
            sdata[tid] += sdata[tid + 16];
            __syncthreads();
            sdata[tid] += sdata[tid + 8];
            __syncthreads();
            sdata[tid] += sdata[tid + 4];
            __syncthreads();
            sdata[tid] += sdata[tid + 2];
            __syncthreads();
            sdata[tid] += sdata[tid + 1];
        }

        // write result for this block to global mem
        if (tid == 0)
            g_odata[blockIdx.x] = sdata[0];
    }
}
```

جدول زیر حاصل از اجرای برنامه برای ورودی 2^{22} است :

Kernel	Time(ms)	Bandwidth(Gb/s)	Total Time(ms)	Step speedup	Cumulative speedup
1	1.772832	9.463511	55.948997	-	-
2	1.052096	15.946469	14.882000	1.68	1.68
3	0.876768	19.135297	15.698999	1.19	2.02
4	0.453856	36.965946	13.251000	1.93	3.90
5	0.309920	54.134019	6.690000	1.5	5.9

در آخر تسریعی که نسبت به اجرای سریال بر رو cpu گرفته ایم ، با توجه به تصویر اول گزارش ، 37.79 است که البته این زمان اجرا آنها بدون در نظر گرفتن سربار تخصیص حافظه است. حال اگر سربار مربوط به قرار دادن داده ها را نیز در نظر بگیریم تسریع 3.61 را می گیریم.همانطور که دیده می شود قدرت محاسباتی gpu ها تسریع بسیار زیادی نسبت به cpu ها به ما می دهند اما از طرفی سربار انتقال داده تا حد زیادی این تسریع را پایین می آورد.

اگر اندازه ی همین ورودی را چهار برابر کنیم تسریع محاسباتی ما 45.51 خواهد بود اما تسریع کل حتی از مقدار

قبلی نیز کمتر می شود (تقریبا مقدار 2)

اگر اندازه ورودی را بسیار کوچکتر بگیریم ، ممکن است حتی تسریع کوچکتر از یک شود.

در تصویر ابتدای گزارش نیز اندازه گرید ، بلاک و تعداد کل نخ ها آمده است که البته برای اجرای اول کرنل ها است.در واقع با هر فراخوانی کرنل اندازه گرید بر اندازه ی بلاک تقسیم می شود تا به صفر برسد. کر مربوط به این فراخوانی های کرنل نیز در تابع reduction آمده است.