

苹果妖

Anything that can go wrong will go wrong !

博客园 首页 新随笔 联系 管理 订阅 XML

随笔- 45 文章- 0 评论- 18

CUDA ---- Branch Divergence and Unrolling Loop

Avoiding Branch Divergence

有时，控制流依赖于thread索引。同一个warp中，一个条件分支可能导致很差的性能。通过重新组织数据获取模式可以减少或避免warp divergence ([该问题的解释请查看warp解析篇](#))。

The Parallel Reduction Problem

我们现在要计算一个数组N个元素的和。这个过程用CPU编程很容易实现：

```
int sum = 0;
for (int i = 0; i < N; i++)
    sum += array[i];
```

那么如果Array的元素非常多呢？应用并行计算可以大大提升这个过程的效率。鉴于加法的交换律等性质，这个求和过程可以以元素的任意顺序来进行：

- 将输入数组切割成很多小的块。
- 用thread来计算每个块的和。
- 对这些块的结果再求和得最终结果。

数组的切割主旨是，用thread求数组中按一定规律配对的的两个元素和，然后将所有结果组合成一个新的数组，然后再再次求配对两元素和，多次迭代，直到数组中只有一个结果。

比较直观的两种实现方式是：

1. Neighbored pair：每次迭代都是相邻两个元素求和。
2. Interleaved pair：按一定跨度配对两个元素。

下图展示了解题过程，对于有N个元素的数组，这个过程需要N-1次求和，log(N)步。Interleaved pair的跨度是半个数组长度。

昵称：苹果妖
园龄：3年1个月
粉丝：22
关注：1
[+加关注](#)

2017年9月						
日	一	二	三	四	五	六
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
1	2	3	4	5	6	7

搜索

找找看

谷歌搜索

常用链接

[我的随笔](#)
[我的评论](#)
[我的参与](#)
[最新评论](#)
[我的标签](#)
[更多链接](#)

我的标签

[c/c++\(16\)](#)
[CUDA\(15\)](#)
[并行计算\(15\)](#)
[linux\(14\)](#)
[机器学习\(7\)](#)
[深度学习\(6\)](#)
[数据库\(4\)](#)
[sqlserver\(4\)](#)
[windows\(4\)](#)
[error\(3\)](#)
[更多](#)

随笔分类

[c/c++\(21\)](#)
[CUDA\(15\)](#)
[error\(6\)](#)
[java\(1\)](#)
[linux\(8\)](#)
[MachineLearning\(7\)](#)

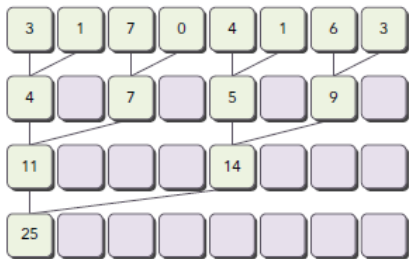
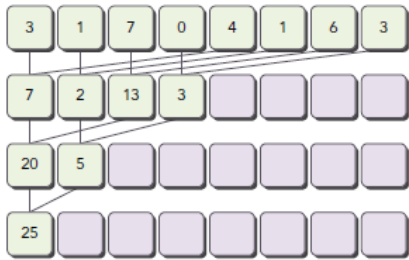


FIGURE 3-19



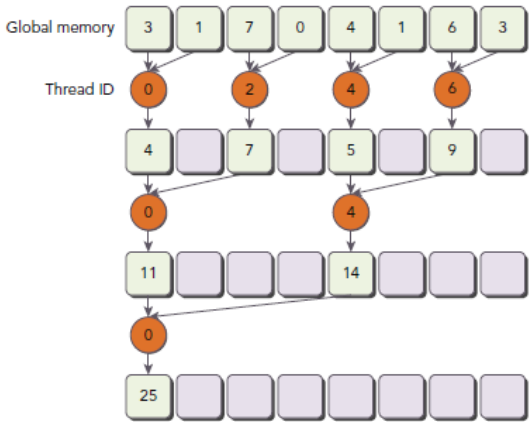
下面是用递归实现的interleaved pair代码（host）：

```
int recursiveReduce(int *data, int const size) {  
    // terminate check  
    if (size == 1) return data[0];  
    // renew the stride  
    int const stride = size / 2;  
    // in-place reduction  
    for (int i = 0; i < stride; i++) {  
        data[i] += data[i + stride];  
    }  
    // call recursively  
    return recursiveReduce(data, stride);  
}
```

上述讲的这类问题术语叫reduction problem。Parallel reduction（并行规约）是指迭代减少操作，是并行算法中非常关键的一种操作。

Divergence in Parallel Reduction

这部分以neighbored pair为参考研究：



在这个kernel里面，有两个global memory array，一个用来存放数组所有数据，另一个用来存放部分和。所有block独立的执行求和和操作。__syncthreads（关于同步，请看前文）用来保证每次迭代，所有的求和操作都做完了，然后进

windows(5)
任务后记(3)
算法(4)

随笔档案

- 2016年9月 (2)
- 2015年8月 (1)
- 2015年7月 (1)
- 2015年6月 (11)
- 2015年5月 (7)
- 2014年11月 (1)
- 2014年10月 (3)
- 2014年9月 (2)
- 2014年8月 (12)
- 2014年7月 (5)

最新评论

- 1. Re:CUDA ---- CUDA库简介
@dongxiao92什么卡？可能是false dependences的问题。 ...
--吉祥1024
- 2. Re:CUDA ---- CUDA库简介
@dongxiao92先列再行，注意source和destination。 ...
--吉祥1024
- 3. Re:CUDA ---- CUDA库简介
博主，表8.4 Formatting Conversion with cuSPARSE中行csc bsr列是不是有错呢？是不是应该是bsr2csc
--dongxiao92
- 4. Re:CUDA ---- CUDA库简介
博主，我想请教一个问题。我希望使用multi stream的方式挖掘cublas的并行性。我在每次调用cublasgemm方法前都使用cublasSetStream设置了stream，但profil.....
--dongxiao92
- 5. Re:主机找不到vmnet1和vmnet8
红框勾选了没用啊，不一会就自动把勾去掉了，然后网络中心也找不到vmnet8和vmnet1。。。求助啊
--杰·维斯布鲁克

阅读排行榜

- 1. CUDA ---- Shared Memory(5854)
- 2. CUDA ---- Warp解析(4403)
- 3. CUDA ---- GPU架构（Fermi、Kepler）(2608)
- 4. CUDA ---- Memory Model(2222)
- 5. CUDA ---- Stream and Event(2104)

评论排行榜

- 1. CUDA ---- Hello World From GPU(6)
- 2. CUDA ---- CUDA库简介(5)
- 3. CUDA ---- 线程配置(4)
- 4. CUDA ---- Branch Divergence and Unrolling Loop(2)
- 5. 主机找不到vmnet1和vmnet8(1)

推荐排行榜

- 1. CUDA ---- Warp解析(2)
- 2. CUDA ---- Memory Access(2)
- 3. CUDA ---- Memory Model(1)

入下一步迭代。

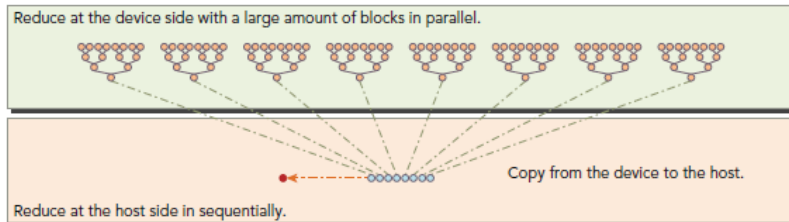


```
__global__ void reduceNeighbored(int *g_idata, int *g_odata, unsigned int n) {
    // set thread ID
    unsigned int tid = threadIdx.x;
    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x;
    // boundary check
    if (idx >= n) return;
    // in-place reduction in global memory
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
        if ((tid % (2 * stride)) == 0) {
            idata[tid] += idata[tid + stride];
        }
        // synchronize within block
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```



- 4. CUDA ---- GPU架构 (Fermi、 Kepler) (1)
- 5. CUDA ---- CUDA库简介(1)

因为没有办法让所有的block同步，所以最后将所有block的结果送回host来进行串行计算，如下图所示：



main代码：



```
int main(int argc, char **argv) {
    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);
    printf("%s starting reduction at ", argv[0]);
    printf("device %d: %s ", dev, deviceProp.name);
    cudaSetDevice(dev);
    bool bResult = false;
    // initialization
    int size = 1<<24; // total number of elements to reduce
    printf(" with array size %d ", size);
    // execution configuration
    int blocksize = 512; // initial block size
    if(argc > 1) {
        blocksize = atoi(argv[1]); // block size from command line argument
    }
    dim3 block (blocksize,1);
    dim3 grid ((size+block.x-1)/block.x,1);
    printf("grid %d block %d\n",grid.x, block.x);
    // allocate host memory
    size_t bytes = size * sizeof(int);
    int *h_idata = (int *) malloc(bytes);
    int *h_odata = (int *) malloc(grid.x*sizeof(int));
    int *tmp = (int *) malloc(bytes);
    // initialize the array
    for (int i = 0; i < size; i++) {
        // mask off high 2 bytes to force max number to 255
    }
```

```

h_idata[i] = (int)(rand() & 0xFF);
}
memcpy (tmp, h_idata, bytes);
size_t iStart,iElaps;
int gpu_sum = 0;
// allocate device memory
int *d_idata = NULL;
int *d_odata = NULL;
cudaMalloc((void **) &d_idata, bytes);
cudaMalloc((void **) &d_odata, grid.x*sizeof(int));
// cpu reduction
iStart = seconds ();
int cpu_sum = recursiveReduce(tmp, size);
iElaps = seconds () - iStart;
printf("cpu reduce elapsed %d ms cpu_sum: %d\n", iElaps,cpu_sum);
// kernel 1: reduceNeighbored
cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice);
cudaDeviceSynchronize();
iStart = seconds ();
warmup<<<grid, block>>>(d_idata, d_odata, size);
cudaDeviceSynchronize();
iElaps = seconds () - iStart;
cudaMemcpy(h_odata, d_odata, grid.x*sizeof(int), cudaMemcpyDeviceToHost);
gpu_sum = 0;
for (int i=0; i<grid.x; i++) gpu_sum += h_odata[i];
printf("gpu Warmup elapsed %d ms gpu_sum: %d <<<grid %d block %d>>>\n",
iElaps,gpu_sum,grid.x,block.x);
// kernel 1: reduceNeighbored
cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice);
cudaDeviceSynchronize();
iStart = seconds ();
reduceNeighbored<<<grid, block>>>(d_idata, d_odata, size);
cudaDeviceSynchronize();
iElaps = seconds () - iStart;
cudaMemcpy(h_odata, d_odata, grid.x*sizeof(int), cudaMemcpyDeviceToHost);
gpu_sum = 0;
for (int i=0; i<grid.x; i++) gpu_sum += h_odata[i];
printf("gpu Neighbored elapsed %d ms gpu_sum: %d <<<grid %d block %d>>>\n",
iElaps,gpu_sum,grid.x,block.x);
cudaDeviceSynchronize();
iElaps = seconds() - iStart;
cudaMemcpy(h_odata, d_odata, grid.x/8*sizeof(int), cudaMemcpyDeviceToHost);
gpu_sum = 0;
for (int i = 0; i < grid.x / 8; i++) gpu_sum += h_odata[i];
printf("gpu Cmptrnroll elapsed %d ms gpu_sum: %d <<<grid %d block %d>>>\n",
iElaps,gpu_sum,grid.x/8,block.x);
/// free host memory
free(h_idata);
free(h_odata);
// free device memory
cudaFree(d_idata);
cudaFree(d_odata);
// reset device
cudaDeviceReset();
// check the results
bResult = (gpu_sum == cpu_sum);
if(!bResult) printf("Test failed!\n");
return EXIT_SUCCESS;
}

```

初始化数组，使其包含16M元素：

```
int size = 1<<24;
```

kernel配置为1D grid和1D block：

```
dim3 block (blocksize, 1);
dim3 block ((size + block.x - 1) / block.x, 1);
```

编译：

```
$ nvcc -O3 -arch=sm_20 reduceInteger.cu -o reduceInteger
```

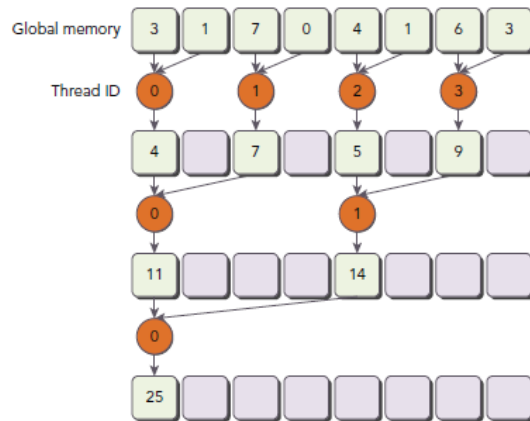
运行：

```
$ ./reduceInteger starting reduction at device 0: Tesla M2070
with array size 16777216 grid 32768 block 512
cpu reduce elapsed 29 ms cpu_sum: 2139353471
gpu Neighbored elapsed 11 ms gpu_sum: 2139353471 <<<grid 32768 block 512>>>
Improving Divergence in Parallel Reduction
```

考虑上节if判断条件：

```
if ((tid % (2 * stride)) == 0)
```

因为这个表达式只对偶数ID的线程为true，所以其导致很高的divergent warps。第一次迭代只有偶数ID的线程执行了指令，但是所有线程都要被调度；第二次迭代，只有四分之一的thread是active的，但是所有thread仍然要被调度。我们可以重新组织每个线程对应的数组索引来强制ID相邻的thread来处理求和操作。如下图所示（注意图中的Thread ID与上一个图的差别）：



新的代码：

```
__global__ void reduceNeighboredLess (int *g_idata, int *g_odata, unsigned int n) {
    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x;
    // boundary check
    if (idx >= n) return;
    // in-place reduction in global memory
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
        // convert tid into local array index
        int index = 2 * stride * tid;
        if (index < blockDim.x) {
            idata[index] += idata[index + stride];
        }
        // synchronize within threadblock
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

注意这行代码：

```
int index = 2 * stride * tid;
```

因为步调乘以了2，下面的语句使用block的前半部分thread来执行求和：

```
if (index < blockDim.x)
```

对于一个有512个thread的block来说，前八个warp执行第一轮reduction，剩下八个warp什么也不干；第二轮，前四个warp执行，剩下十二个什么也不干。因此，就彻底不存在divergence了（重申，divergence只发生于同一个warp）。最后的五轮还是会导致divergence，因为这个时候需要执行threads已经凑不够一个warp了。



```
// kernel 2: reduceNeighbored with less divergence
cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice);
cudaDeviceSynchronize();
iStart = seconds();
reduceNeighboredLess<<<grid, block>>>(d_idata, d_odata, size);
cudaDeviceSynchronize();
iElaps = seconds() - iStart;
cudaMemcpy(h_odata, d_odata, grid.x*sizeof(int), cudaMemcpyDeviceToHost);
gpu_sum = 0;
for (int i=0; i<grid.x; i++) gpu_sum += h_odata[i];
printf("gpu Neighbored2 elapsed %d ms gpu_sum: %d <<<grid %d block %d>>>\n", iElaps, gpu_sum, grid.x, block.x);
```



运行结果：

```
$ ./reduceInteger Starting reduction at device 0: Tesla M2070
vector size 16777216 grid 32768 block 512
cpu reduce elapsed 0.029138 sec cpu_sum: 2139353471
gpu Neighbored elapsed 0.011722 sec gpu_sum: 2139353471 <<<grid 32768 block 512>>>
gpu NeighboredL elapsed 0.009321 sec gpu_sum: 2139353471 <<<grid 32768 block 512>>>
```

新的实现比原来的快了1.26。我们也可以使用nvprof的inst_per_warp参数来查看每个warp上执行的指令数目的平均值。

```
$ nvprof --metrics inst_per_warp ./reduceInteger
```

输出，原来的是新的kernel的两倍还多，因为原来的有许多不必要的操作也执行了：

```
Neighbored Instructions per warp 295.562500
NeighboredLess Instructions per warp 115.312500
```

再查看throughput：

```
$ nvprof --metrics gld_throughput ./reduceInteger
```

输出，新的kernel拥有更大的throughput，因为虽然I/O操作数目相同，但是其耗时短：

```
Neighbored Global Load Throughput 67.663GB/s
NeighboredL Global Load Throughput 80.144GB/s
Reducing with Interleaved Pairs
```

Interleaved Pair模式的初始步调是block大小的一半，每个thread处理像半个block的两个数据求和。和之前的图示相比，工作的thread数目没有变化，但是，每个thread的load/store global memory的位置是不同的。

Interleaved Pair的kernel实现：

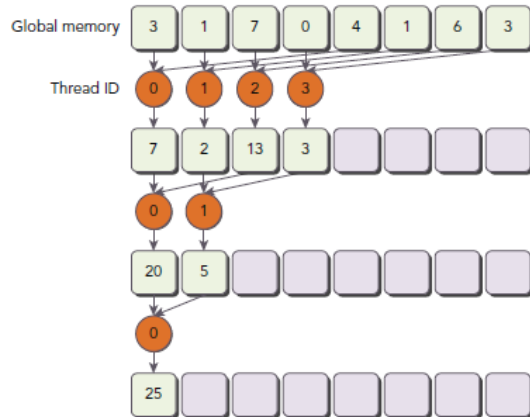


```
/// Interleaved Pair Implementation with less divergence
__global__ void reduceInterleaved (int *g_idata, int *g_odata, unsigned int n) {
// set thread ID
unsigned int tid = threadIdx.x;
unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
// convert global data pointer to the local pointer of this block
int *idata = g_idata + blockIdx.x * blockDim.x;
// boundary check
if (idx >= n) return;
// in-place reduction in global memory
```

```

for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
    if (tid < stride) {
        idata[tid] += idata[tid + stride];
    }
    __syncthreads();
}
// write result for this block to global mem
if (tid == 0) g_odata[blockIdx.x] = idata[0];
}

```



注意下面的语句，步调被初始化为block大小的一半：

```
for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
```

下面的语句使得第一次迭代时，block的前半部分thread执行相加操作，第二次是前四分之一，以此类推：

```
if (tid < stride)
```

下面是加入main的代码：



```

cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice);
cudaDeviceSynchronize();
iStart = seconds();
reduceInterleaved <<< grid, block >>> (d_idata, d_odata, size);
cudaDeviceSynchronize();
iElaps = seconds() - iStart;
cudaMemcpy(h_odata, d_odata, grid.x*sizeof(int), cudaMemcpyDeviceToHost);
gpu_sum = 0;
for (int i = 0; i < grid.x; i++) gpu_sum += h_odata[i];
printf("gpu Interleaved elapsed %f sec gpu_sum: %d <<<grid %d block %d>>>\n", iElaps, gpu_sum, grid.x, block.x);

```



运行输出：



```

$ ./reduce starting reduction at device 0: Tesla M2070
with array size 16777216 grid 32768 block 512
cpu reduce elapsed 0.029138 sec cpu_sum: 2139353471
gpu Warmup elapsed 0.011745 sec gpu_sum: 2139353471 <<<grid 32768 block 512>>>
gpu Neighbored elapsed 0.011722 sec gpu_sum: 2139353471 <<<grid 32768 block 512>>>
gpu NeighboredL elapsed 0.009321 sec gpu_sum: 2139353471 <<<grid 32768 block 512>>>
gpu Interleaved elapsed 0.006967 sec gpu_sum: 2139353471 <<<grid 32768 block 512>>>

```



这次相对第一个kernel又快了1.69，比第二个也快了1.34。这个效果主要由global memory的load/store模式导致的（这部分知识将在后续博文介绍）。

UNrolling Loops

loop unrolling 是用来优化循环减少分支的方法，该方法简单说就是把本应在多次loop中完成的操作，尽量压缩到一次loop。循环体展开程度称为loop unrolling factor（循环展开因子），loop unrolling对顺序数组的循环操作性能有很大影响，考虑如下代码：

```
for (int i = 0; i < 100; i++) {  
    a[i] = b[i] + c[i];  
}
```

如下重复一次循环体操作，迭代数目将减少一半：


```
for (int i = 0; i < 100; i += 2) {  
    a[i] = b[i] + c[i];  
    a[i+1] = b[i+1] + c[i+1];  
}
```

从高级语言层面是无法看出性能提升的原因的，需要从low-level instruction层面去分析，第二段代码循环次数减少了一半，而循环体两句语句的读写操作的执行在CPU上是可以同时执行互相独立的，所以相对第一段，第二段性能要好。


Unrolling 在CUDA编程中意义更重。我们的目标依然是通过减少指令执行消耗，增加更多的独立指令来提高性能。这样就会增加更多的并行操作从而产生更高的指令和内存带宽（bandwidth）。也就提供了更多的eligible warps来帮助hide instruction/memory latency。

Reducing with Unrolling

在前文的reduceInterleaved中，每个block处理一部分数据，我们给这数据起名data block。下面的代码是reduceInterleaved的修正版本，每个block，都是以两个data block作为源数据进行操作，（前文中，每个block处理一个data block）。这是一种cyclic partitioning：每个thread作用于多个data block，并且从每个data block中取出一个元素处理。



```
__global__ void reduceUnrolling2 (int *g_idata, int *g_odata, unsigned int n) {  
    // set thread ID  
    unsigned int tid = threadIdx.x;  
    unsigned int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;  
  
    // convert global data pointer to the local pointer of this block  
    int *idata = g_idata + blockIdx.x * blockDim.x * 2;  
  
    // unrolling 2 data blocks  
    if (idx + blockDim.x < n) g_idata[idx] += g_idata[idx + blockDim.x];  
    __syncthreads();  
  
    // in-place reduction in global memory  
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {  
        if (tid < stride) {  
            idata[tid] += idata[tid + stride];  
        }  
        // synchronize within threadblock  
        __syncthreads();  
    }  
  
    // write result for this block to global mem  
    if (tid == 0) g_odata[blockIdx.x] = idata[0];  
}
```

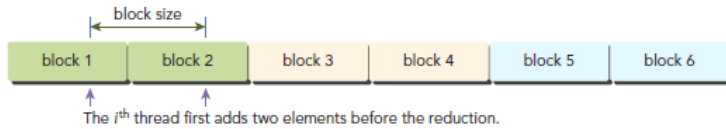


注意下面的语句，每个thread从相邻的data block中取数据，这一步实际上就是将两个data block规约成一个。

```
if (idx + blockDim.x < n) g_idata[idx] += g_idata[idx+blockDim.x];
```

global array index也要相应的调整，因为，相对之前的版本，同样的数据，我们只需要原来一半的thread就能解决问题。要注意的是，这样做也会降低warp或block的并行性（因为thread少啦）：


```
unsigned int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;
int *idata = g_idata + blockIdx.x * blockDim.x * 2;
```



main增加下面代码：

```
cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice);
cudaDeviceSynchronize();
iStart = seconds();
reduceUnrolling2 <<< grid.x/2, block >>> (d_idata, d_odata, size);
cudaDeviceSynchronize();
iElaps = seconds() - iStart;
cudaMemcpy(h_odata, d_odata, grid.x/2*sizeof(int), cudaMemcpyDeviceToHost);
gpu_sum = 0;
for (int i = 0; i < grid.x / 2; i++) gpu_sum += h_odata[i];
printf("gpu Unrolling2 elapsed %f sec gpu_sum: %d <<<grid %d block %d>>>\n", iElaps, gpu_sum, grid.x/2, block.x);
```

由于每个block处理两个data block，所以需要调整grid的配置：

```
reduceUnrolling2<<<grid.x / 2, block>>>(d_idata, d_odata, size);
```

运行输出：

```
gpu Unrolling2 elapsed 0.003430 sec gpu_sum: 2139353471 <<<grid 16384 block 512>>>
```

这样一次简单的操作就比原来的减少了3.42。我们在试试每个block处理4个和8个data block的情况：

```
reduceUnrolling4 : each threadblock handles 4 data blocks
```

```
reduceUnrolling8 : each threadblock handles 8 data blocks
```

加上这两个的输出是：

```
gpu Unrolling2 elapsed 0.003430 sec gpu_sum: 2139353471 <<<grid 16384 block 512>>>
gpu Unrolling4 elapsed 0.001829 sec gpu_sum: 2139353471 <<<grid 8192 block 512>>>
gpu Unrolling8 elapsed 0.001422 sec gpu_sum: 2139353471 <<<grid 4096 block 512>>>
```

可以看出，同一个thread中如果能有更多的独立的load/store操作，会产生更好的性能，因为这样做memory latency能够更好的被隐藏。我们可以使用nvprof的dram_read_throughput来验证：

```
$ nvprof --metrics dram_read_throughput ./reduceInteger
```

下面是输出结果，我们可以得出这样的结论，device read throughput和unrolling程度是正比的：

```
Unrolling2 Device Memory Read Throughput 26.295GB/s
Unrolling4 Device Memory Read Throughput 49.546GB/s
Unrolling8 Device Memory Read Throughput 62.764GB/s
Reducinng with Unrolled Warps
```

__syncthreads是用来同步block内部thread的（请看warp解析篇）。在reduction kernel中，他被用来在每次循环中年那个保证所有thread的写global memory的操作都已完成，这样才能进行下一阶段的计算。

那么，当kernel进行到只需要少于或等32个thread（也就是一个warp）呢？由于我们是使用的SIMT模式，warp内的thread是有一个隐式的同步过程的。最后六次迭代可以用下面的语句展开：

```
if (tid < 32) {
    volatile int *vmem = idata;
    vmem[tid] += vmem[tid + 32];
    vmem[tid] += vmem[tid + 16];
    vmem[tid] += vmem[tid + 8];
    vmem[tid] += vmem[tid + 4];
```

```

    vmem[tid] += vmem[tid + 2];
    vmem[tid] += vmem[tid + 1];
}

```



warp unrolling避免了__syncthreads同步操作，因为这一步本身就没必要。

这里注意下volatile修饰符，他告诉编译器每次执行赋值时必须将vmem[tid]的值store回global memory。如果不这样做的話，编译器或cache可能会优化我们读写global/shared memory。有了这个修饰符，编译器就会认为这个值会被其他thread修改，从而使得每次读写都直接去memory而不是去cache或者register。



```

__global__ void reduceUnrollWarps8 (int *g_idata, int *g_odata, unsigned int n) {
    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x*blockDim.x*8 + threadIdx.x;

    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x*blockDim.x*8;

    // unrolling 8
    if (idx + 7*blockDim.x < n) {
        int a1 = g_idata[idx];
        int a2 = g_idata[idx+blockDim.x];
        int a3 = g_idata[idx+2*blockDim.x];
        int a4 = g_idata[idx+3*blockDim.x];
        int b1 = g_idata[idx+4*blockDim.x];
        int b2 = g_idata[idx+5*blockDim.x];
        int b3 = g_idata[idx+6*blockDim.x];
        int b4 = g_idata[idx+7*blockDim.x];
        g_idata[idx] = a1+a2+a3+a4+b1+b2+b3+b4;
    }
    __syncthreads();

    // in-place reduction in global memory
    for (int stride = blockDim.x / 2; stride > 32; stride >>= 1) {

        if (tid < stride) {
            idata[tid] += idata[tid + stride];
        }

        // synchronize within threadblock
        __syncthreads();
    }

    // unrolling warp
    if (tid < 32) {
        volatile int *vmem = idata;
        vmem[tid] += vmem[tid + 32];
        vmem[tid] += vmem[tid + 16];
        vmem[tid] += vmem[tid + 8];
        vmem[tid] += vmem[tid + 4];
        vmem[tid] += vmem[tid + 2];
        vmem[tid] += vmem[tid + 1];
    }

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

```



因为处理的数据block变为八个，kernel调用变为;

```
reduceUnrollWarps8<<<grid.x / 8, block>>>> (d_idata, d_odata, size);
```


这次执行结果比reduceUnrolling8快1.05，比reduceNeighbourhood快8.65：

```
gpu UnrollWarp8 elapsed 0.001355 sec gpu_sum: 2139353471 <<<grid 4096 block 512>>>
```

nvprof的stall_sync可以用来验证由于__syncthreads导致更少的warp阻塞了：

```
$ nvprof --metrics stall_sync ./reduce
Unrolling8 Issue Stall Reasons 58.37%
UnrollWarps8 Issue Stall Reasons 30.60%
Reducing with Complete Unrolling
```

如果在编译时已知了迭代次数，就可以完全把循环展开。Fermi和Kepler每个block的最大thread数目都是1024，博文中的kernel的迭代次数都是基于blockDim的，所以完全展开循环是可行的。



```
__global__ void reduceCompleteUnrollWarps8 (int *g_idata, int *g_odata,
unsigned int n) {
    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 8 + threadIdx.x;

    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x * 8;

    // unrolling 8
    if (idx + 7*blockDim.x < n) {
        int a1 = g_idata[idx];
        int a2 = g_idata[idx + blockDim.x];
        int a3 = g_idata[idx + 2 * blockDim.x];
        int a4 = g_idata[idx + 3 * blockDim.x];
        int b1 = g_idata[idx + 4 * blockDim.x];
        int b2 = g_idata[idx + 5 * blockDim.x];
        int b3 = g_idata[idx + 6 * blockDim.x];
        int b4 = g_idata[idx + 7 * blockDim.x];
        g_idata[idx] = a1 + a2 + a3 + a4 + b1 + b2 + b3 + b4;
    }
    __syncthreads();

    // in-place reduction and complete unroll
    if (blockDim.x>=1024 && tid < 512) idata[tid] += idata[tid + 512];
    __syncthreads();


    if (blockDim.x>=512 && tid < 256) idata[tid] += idata[tid + 256];
    __syncthreads();

    if (blockDim.x>=256 && tid < 128) idata[tid] += idata[tid + 128];
    __syncthreads();

    if (blockDim.x>=128 && tid < 64) idata[tid] += idata[tid + 64];
    __syncthreads();

    // unrolling warp
    if (tid < 32) {
        volatile int *vsmem = idata;
        vsmem[tid] += vsmem[tid + 32];
        vsmem[tid] += vsmem[tid + 16];
        vsmem[tid] += vsmem[tid + 8];
        vsmem[tid] += vsmem[tid + 4];
        vsmem[tid] += vsmem[tid + 2];
        vsmem[tid] += vsmem[tid + 1];
    }

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



main中调用：


```
reduceCompleteUnrollWarps8<<<grid.x / 8, block>>>(d_idata, d_odata, size);
```

速度再次提升：

```
gpu CmpUnroll8 elapsed 0.001280 sec gpu_sum: 2139353471 <<<grid 4096 block 512>>>
```

Reducing with Template Functions

CUDA代码支持模板，我们可以如下设置block大小：



```
template <unsigned int iBlockSize>
__global__ void reduceCompleteUnroll(int *g_idata, int *g_odata, unsigned int n) {
    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 8 + threadIdx.x;

    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x * 8;

    // unrolling 8
    if (idx + 7*blockDim.x < n) {
        int a1 = g_idata[idx];
        int a2 = g_idata[idx + blockDim.x];
        int a3 = g_idata[idx + 2 * blockDim.x];
        int a4 = g_idata[idx + 3 * blockDim.x];
        int b1 = g_idata[idx + 4 * blockDim.x];
        int b2 = g_idata[idx + 5 * blockDim.x];
        int b3 = g_idata[idx + 6 * blockDim.x];
        int b4 = g_idata[idx + 7 * blockDim.x];
        g_idata[idx] = a1+a2+a3+a4+b1+b2+b3+b4;
    }
    __syncthreads();

    // in-place reduction and complete unroll
    if (iBlockSize>=1024 && tid < 512) idata[tid] += idata[tid + 512];
    __syncthreads();


    if (iBlockSize>=512 && tid < 256) idata[tid] += idata[tid + 256];
    __syncthreads();

    if (iBlockSize>=256 && tid < 128) idata[tid] += idata[tid + 128];
    __syncthreads();

    if (iBlockSize>=128 && tid < 64) idata[tid] += idata[tid + 64];
    __syncthreads();

    // unrolling warp
    if (tid < 32) {
        volatile int *vsmem = idata;
        vsmem[tid] += vsmem[tid + 32];
        vsmem[tid] += vsmem[tid + 16];
        vsmem[tid] += vsmem[tid + 8];
        vsmem[tid] += vsmem[tid + 4];
        vsmem[tid] += vsmem[tid + 2];
        vsmem[tid] += vsmem[tid + 1];
    }

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



对于if的条件，如果值为false，那么在编译时就会去掉该语句，这样效率更好。例如，如果调用kernel时的blocksize是256，那么，下面的语句将永远为false，编译器会将他移除不予执行：

iBlockSize>=1024 && tid < 512

这个kernel必须以switch-case来调用：



```
switch (blocksize) {
    case 1024:
        reduceCompleteUnroll<1024><<<grid.x/8, block>>>(d_idata, d_odata, size);
        break;
    case 512:
        reduceCompleteUnroll<512><<<grid.x/8, block>>>(d_idata, d_odata, size);
        break;
    case 256:
        reduceCompleteUnroll<256><<<grid.x/8, block>>>(d_idata, d_odata, size);
        break;
    case 128:
        reduceCompleteUnroll<128><<<grid.x/8, block>>>(d_idata, d_odata, size);
        break;
    case 64:
        reduceCompleteUnroll<64><<<grid.x/8, block>>>(d_idata, d_odata, size);
        break;
}
```

各种情况下，执行后的结果为:

KERNEL	TIME (S)	STEP SPEEDUP	CUMULATIVE SPEEDUP
Neighbored (divergence)	0.011722		
Neighbored (no divergence)	0.009321	1.26	1.26
Interleaved	0.006967	1.34	1.68
Unroll 8 blocks	0.001422	4.90	8.24
Unroll 8 blocks + last warp	0.001355	1.05	8.65
Unroll 8 blocks + loop + last warp	0.001280	1.06	9.16
Templatized kernel	0.001253	1.02	9.35

\$nvprof --metrics gld_efficiency,gst_efficiency ./reduceInteger

KERNEL	TIME (S)	LOAD EFFICIENCY	STORE EFFICIENCY
Neighbored (divergence)	0.011722	16.73%	25.00%
Neighbored (no divergence)	0.009321	16.75%	25.00%
Interleaved	0.006967	77.94%	95.52%
Unroll 8 blocks	0.001422	94.68%	97.71%
Unroll 8 blocks + last warp	0.001355	98.99%	99.40%
Unroll 8 blocks + loop + last warp	0.001280	98.99%	99.40%
Templatized the last kernel	0.001253	98.99%	99.40%

分类: [c/c++,CUDA](#)

标签: [CUDA](#), [并行计算](#)

好文要顶

关注我

收藏该文

苹果妖
关注 - 1
粉丝 - 22

+加关注

0

0

« 上一篇 : [CUDA ---- Kernel性能调节](#)
» 下一篇 : [CUDA ---- Dynamic Parallelism](#)

评论

#1楼 2015-06-03 11:00 | xingoo

CUDA更新的速度真是快啊，两年的时间就从5.5更新到8了！

支持(0) 反对(0)

#2楼[楼主] 2015-06-03 14:14 | 苹果妖

@ xingoo
抱歉，我标题有歧义，现在应该是7。

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问](#)网站首页。

- 【推荐】50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】腾讯云上实验室 1小时搭建人工智能应用
- 【推荐】可嵌入您系统的“在线Excel”！SpreadJS 纯前端表格控件
- 【推荐】阿里云“全民云计算”优惠升级

- 最新IT新闻:
- 战斗民族的手机正式杀入中国市场
 - Nest发布家庭安全系统NestSecure 售价499美元
 - 借款乐视网却爽约，贾跃亭家族套现近140个亿去哪儿了？
 - Intel 10nm CannonLake处理器突传噩耗：跳票2018年末
 - Uber冤吗？窃取Waymo商业机密被索赔26亿美元
- » 更多新闻...

- 最新知识库文章:
- Google 及其云智慧
 - 做到这一点，你也可以成为优秀的程序员
 - 写给立志做码农的大学生
 - 架构腐化之谜
 - 学会思考，而不只是编程
- » 更多知识库文章...