











# 谭升的博客

人工智能基础

?

?

×

A	B	C	D	E	F	G	H	I	J	M
WHY YOU'RE NOT LOSING WEIGHT										
WAKE UP TIME		6 AM		7 AM		8 AM				
DAILY MEALS		1		2		3				
HOURS OF SLEEP		 5		 6		 7				
AGE		18-25		26-35		36-55				
DAILY WATER INTAKE		 		 		 +				
YOUR BMI		40+		30+		25-30				
FASTING SCHEDULE		16:8		12:12		14:10				
TAKE THE TEST										

## 【CUDA 基础】3.5 展开循环

2018-04-19 | [CUDA](#) | [Freshman](#) | 0 |

**Abstract:** 本文介绍循环展开技术，在归约的基础上继续加速。

**Keywords:** 展开归约，归约，模板函数

### 展开循环

博客从CSDN那边截流了一些流量，现在网站访问突然增多到让我有点不适应，于是，想想还是别总盯着流量看吧，注意文章质量，同时保证一定的更新，从数学到算法，最后到实现，优化，这些都做好，估计访问量会更多了。

到时候我就可以挂广告了，然后挣了钱吃煎饼就可以加个鸡蛋了！去网吧也能叫饮料了。。想想还有点小

激动，感觉自己都有点膨胀了。

今天我们来做循环展开，GPU喜欢确定的东西，像前面讲解执行模型和线程束的时候，明确的指出，GPU没有分支预测能力，所有每一个分支他都是执行的，所以在内核里尽量别写分支，分支包括啥，包括if当然还有for之类的循环语句。

如果你不知道为啥for算分支语句我给你写个简单到不能运行的例子：

```
1  for (itn i=0;i<tid;i++)
2  {
3      // to do something
4
5  }
```

如果上面这段代码出现在内核中，就会有分支，因为一个线程束第一个线程和最后一个线程tid相差32（如果线程束大小是32的话）那么每个线程执行的时候，for终止时完成的计算量都不同，这就有人要等待，这也就产生了分支。

循环展开是一个尝试通过减少分支出现的频率和循环维护指令来优化循环的技术。

上面这句属于书上的官方说法，我们来看看例子，不止并行算法可以展开，传统串行代码展开后效率也能一定程度的提高，因为省去了判断和分支预测失败所带来的迟滞。

先来c++ 入门循环

```
1  for (int i=0;i<100;i++)
2  {
3      a[i]=b[i]+c[i];
4  }
```

这个是最传统的写法，这个写法在各种c++教材上都能看到，不做解释，如果我们进行循环展开呢？

```
1  for (int i=0;i<100;i+=4)
2  {
3      a[i+0]=b[i+0]+c[i+0];
4      a[i+1]=b[i+1]+c[i+1];
5      a[i+2]=b[i+2]+c[i+2];
6      a[i+3]=b[i+3]+c[i+3];
7  }
```

没错，是不是很简单，修改循环体的内容，把本来循环自己搞定的东西，我们自己列出来了，这样做的好处，从串行较多来看是减少了条件判断的次数。

但是如果你把这段代码拿到机器上跑，其实看不出来啥效果，因为现代编译器把上述两个不同的写法，编译成了类似的机器语言，也就是，我们这不循环展开，编译器也会帮我们做。

不过值得注意的是：*目前CUDA的编译器还不能帮我们做这种优化，人为的展开核函数内的循环，能够非常大的提升内核性能*

在CUDA中展开循环的目的还是那两个：

1. 减少指令消耗
2. 增加更多的独立调度指令

来提高性能

如果这种指令

```
1  a[i+0]=b[i+0]+c[i+0];
2  a[i+1]=b[i+1]+c[i+1];
3  a[i+2]=b[i+2]+c[i+2];
4  a[i+3]=b[i+3]+c[i+3];
```

被添加到CUDA流水线上，是非常受欢迎的，因为其能最大限度的提高指令和内存带宽。

下面我们就在前面归约的例子继续挖掘性能，看看是否能得到更高的效率。

## 展开的归约

前面在[避免分支](#)的博客中,我们的内核函数reduceInterleaved 核函数中每个线程块只处理对应那部分的数据，我们现在的一个想法是能不能用一个线程块处理多块数据，其实这是可以实现的，如果在对这块数据进行求和前（因为总是一个线程对应一个数据）使用每个线程进行一次加法，从别的块取数据，相当于先做一个向量加法，然后再归约，这样将会用一句指令，完成之前一般的计算量，这个性价比看起来太诱人了。

上代码

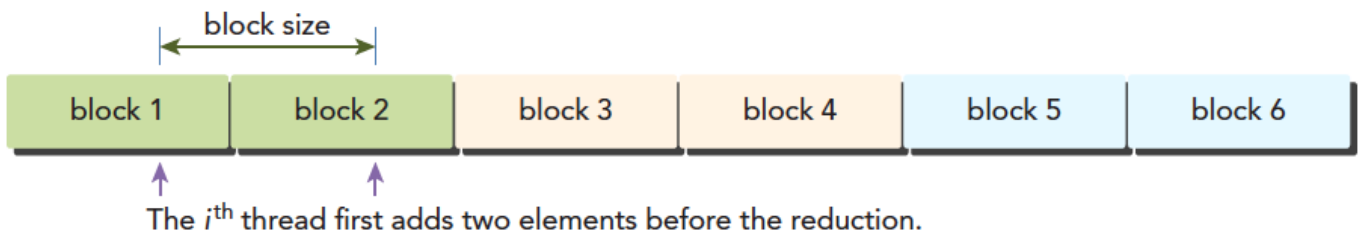
```
1  __global__ void reduceUnroll2(int * g_idata,int * g_odata,unsigned int n)
2  {
3      //set thread ID
4      unsigned int tid = threadIdx.x;
5      unsigned int idx = blockDim.x*blockIdx.x*2+threadIdx.x;
6      //boundary check
```

```

7      if (tid >= n) return;
8      //convert global data pointer to the
9      int *idata = g_idata + blockIdx.x*blockDim.x*2;
10     if(id<math>x+blockDim.x</math><math>n</math>)
11     {
12         g_idata[id<math>x</math>]+=g_idata[id<math>x+blockDim.x</math>];
13
14     }
15     __syncthreads();
16     //in-place reduction in global memory
17     for (int stride = blockDim.x/2; stride>0 ; stride >>=1)
18     {
19         if (tid <stride)
20         {
21             idata[tid] += idata[tid + stride];
22         }
23         //synchronize within block
24         __syncthreads();
25     }
26     //write result for this block to global mem
27     if (tid == 0)
28         g_odata[blockIdx.x] = idata[0];
29
30 }

```

这里面的第二句，第四句，在确定线程块对应的数据的位置的时候有个乘2的偏移量，



**FIGURE 3-25**

这就是第二句，第四句指令的意思，我们只处理红色的线程块，而旁边白色线程块我们用

```

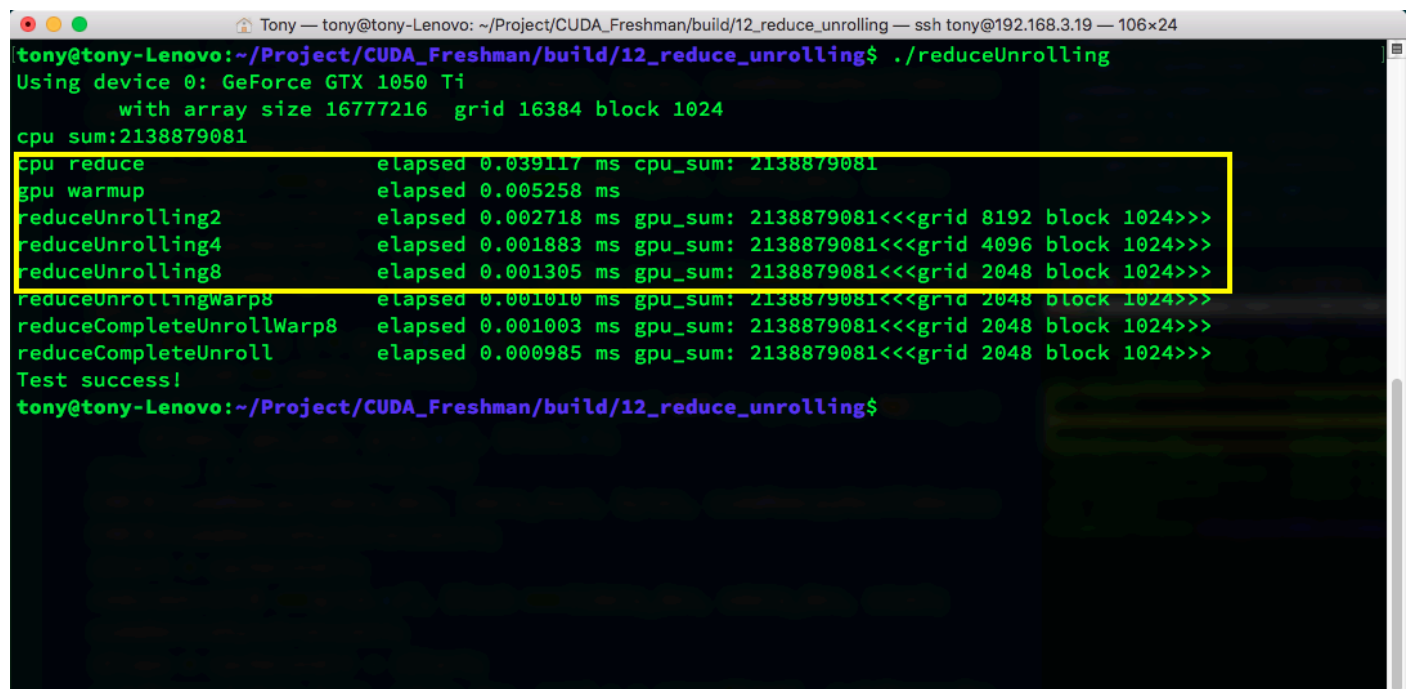
1  if(id<math>x+blockDim.x</math><math>n</math>)
2  {
3      g_idata[id<math>x</math>]+=g_idata[id<math>x+blockDim.x</math>];
4  }

```

处理掉了，注意我们这里用的是一维线程，也就是说，我们用原来的一半的块的数量，而每一句只添加一小句指令的情况下，完成了原来全部的计算量，这个效果应该是客观的，所以我们来看一下效果之前先看一下调用核函数的部分：

```
1 //kernel 1:reduceUnrolling2
2 CHECK(cudaMemcpy(idata_dev, idata_host, bytes, cudaMemcpyHostToDevice));
3 CHECK(cudaDeviceSynchronize());
4 iStart = cpuSecond();
5 reduceUnroll2 <<<grid.x/2, block >>>(idata_dev, odata_dev, size);
6 cudaDeviceSynchronize();
7 iElaps = cpuSecond() - iStart;
8 cudaMemcpy(odata_host, odata_dev, grid.x * sizeof(int), cudaMemcpyDeviceToHost);
9 gpu_sum = 0;
10 for (int i = 0; i < grid.x/2; i++)
11     gpu_sum += odata_host[i];
12 printf("reduceUnrolling2          elapsed %lf ms gpu_sum: %d<<<grid %d block %d>>>
13     iElaps, gpu_sum, grid.x/2, block.x);
```

这里需要注意由于合并了一半的线程块，这里的网格个数都要对应的减少一半，来看效率



```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/12_reduce_unrolling — ssh tony@192.168.3.19 — 106x24
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/12_reduce_unrolling$ ./reduceUnrolling
Using device 0: GeForce GTX 1050 Ti
    with array size 16777216 grid 16384 block 1024
cpu sum:2138879081
cpu reduce          elapsed 0.039117 ms cpu_sum: 2138879081
gpu warmup          elapsed 0.005258 ms
reduceUnrolling2     elapsed 0.002718 ms gpu_sum: 2138879081<<<grid 8192 block 1024>>>
reduceUnrolling4     elapsed 0.001883 ms gpu_sum: 2138879081<<<grid 4096 block 1024>>>
reduceUnrolling8     elapsed 0.001305 ms gpu_sum: 2138879081<<<grid 2048 block 1024>>>
reduceUnrollingwarp8 elapsed 0.001010 ms gpu_sum: 2138879081<<<grid 2048 block 1024>>>
reduceCompleteUnrollwarp8 elapsed 0.001003 ms gpu_sum: 2138879081<<<grid 2048 block 1024>>>
reduceCompleteUnroll elapsed 0.000985 ms gpu_sum: 2138879081<<<grid 2048 block 1024>>>
Test success!
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/12_reduce_unrolling$
```

相比于[上一篇](#)中的效率，“高到不知道哪里去了”（总能引用名人名言），比最简单的归约算法快了三倍，warmup的代码，不需要理睬。

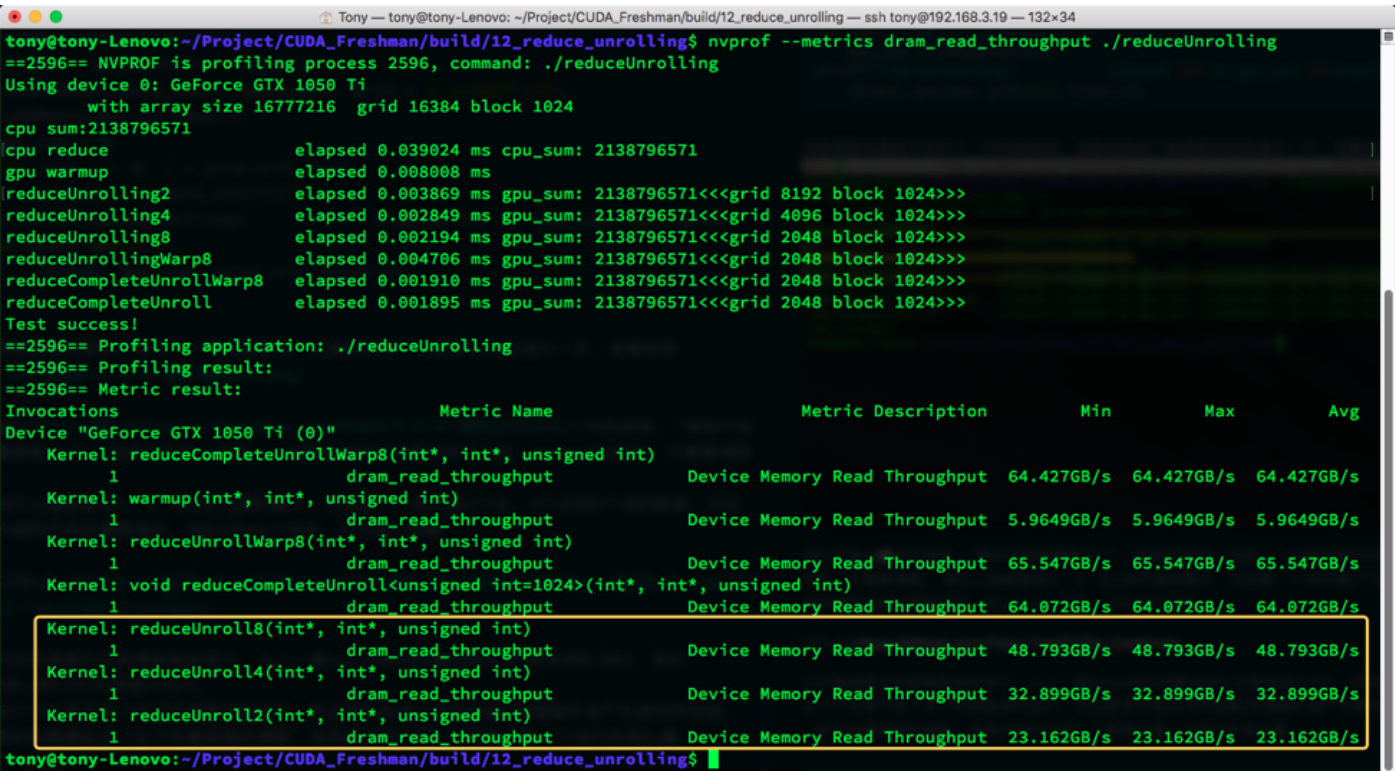
我们上面框里有2，4，8三种尺度的展开，分别是一个块计算2个块，4个块和8个块的数据，对应的调用代码也需要修改，在github上有库，可以自己去看

Github:[https://github.com/Tony-Tan/CUDA\\_Freshman](https://github.com/Tony-Tan/CUDA_Freshman)

可见直接展开对效率影响非常大，从上一篇naive的归约算法的0.01降低到0.002，然后到0.0013可见其威力巨大。

这个不光是节省了多余的线程块的运行，而且更多的独立内存加载/存储操作会产生更好的性能，更好的隐藏延迟（忘了的看前面的博客，有相关知识介绍）下面我们看一下他们的吞吐量

```
1 nvprof --metrics dram_read_throughput ./reduceUnrolling
```



可见执行效率是和内存吞吐量是呈正相关的

### 完全展开的归约

接着我们的目标是最后那32个线程，因为归约运算是个倒金字塔，最后的结果是一个数，所以每个线程最后64个计算得到一个数字结果的过程，没执行一步，线程的利用率就降低一倍，因为从64到32，然后16。。这样到1的，我们现在像个办法，展开最后的6步迭代（64，32，16，8，4，2，1）使用下面的核函数来展开最后6步分支计算：

```
1 __global__ void reduceUnrollWarp8(int * g_idata,int * g_odata,unsigned int n)
```

```

2  {
3      //set thread ID
4      unsigned int tid = threadIdx.x;
5      unsigned int idx = blockDim.x*blockIdx.x*8+threadIdx.x;
6      //boundary check
7      if (tid >= n) return;
8      //convert global data pointer to the
9      int *idata = g_idata + blockIdx.x*blockDim.x*8;
10     //unrolling 8;
11     if(idx+7 * blockDim.x<n)
12     {
13         int a1=g_idata[idx];
14         int a2=g_idata[idx+blockDim.x];
15         int a3=g_idata[idx+2*blockDim.x];
16         int a4=g_idata[idx+3*blockDim.x];
17         int a5=g_idata[idx+4*blockDim.x];
18         int a6=g_idata[idx+5*blockDim.x];
19         int a7=g_idata[idx+6*blockDim.x];
20         int a8=g_idata[idx+7*blockDim.x];
21         g_idata[idx]=a1+a2+a3+a4+a5+a6+a7+a8;
22
23     }
24     __syncthreads();
25     //in-place reduction in global memory
26     for (int stride = blockDim.x/2; stride>32; stride >>=1)
27     {
28         if (tid <stride)
29         {
30             idata[tid] += idata[tid + stride];
31         }
32         //synchronize within block
33         __syncthreads();
34     }
35     //write result for this block to global mem
36     if(tid<32)
37     {
38         volatile int *vmem = idata;
39         vmem[tid]+=vmem[tid+32];
40         vmem[tid]+=vmem[tid+16];
41         vmem[tid]+=vmem[tid+8];
42         vmem[tid]+=vmem[tid+4];
43         vmem[tid]+=vmem[tid+2];
44         vmem[tid]+=vmem[tid+1];
45

```

```

46         }
47
48         if (tid == 0)
49             g_odata[blockIdx.x] = idata[0];
50
51     }

```

在unrolling8的基础上，我们对于tid在  $[0, 32]$  之间的线程用这个代码展开

```

1  volatile int *vmem = idata;
2  vmem[tid]+=vmem[tid+32];
3  vmem[tid]+=vmem[tid+16];
4  vmem[tid]+=vmem[tid+8];
5  vmem[tid]+=vmem[tid+4];
6  vmem[tid]+=vmem[tid+2];
7  vmem[tid]+=vmem[tid+1];

```

第一步定义 `volatile int` 类型变量我们先不说，我们先把最后这个展开捋顺一下，当只剩下最后下面三角部分，从64个数合并到一个数，首先将前32个数，按照步长为32，进行并行加法，前32个tid得到64个数字的两两和，存在前32个数字中

接着，到了我们的关键技巧了

然后这32个数加上步长为16的变量，理论上，这样能得到16个数，这16个数的和就是最后这个块的归约结果，但是根据上面`tid<32`的判断条件线程tid 16到31的线程还在运行，但是结果已经没意义了，这一步很重要（这一步可能产生疑惑的另一个原因是既然是同步执行，会不会比如线程17加上了线程33后写入17号的内存了，这时候1号才来加17号的结果，这样结果就不对了，因为我们的CUDA内核从内存中读数据到寄存器，然后进行加法都是同步进行的，也就是17号线程和1号线程同时读33号和17号的内存，这样17号即便在下一步修改，也不影响1号线程寄存器里面的值了），虽然32以内的tid的线程都在跑，但是没进行一步，后面一半的线程结果将没有用途了，

这样继续计算，得到最后的一个有效的结果就是 `tid[0]`。

上面这个过程有点复杂，但是我们自己好好想一想，从硬件取数据，到计算，每一步都分析一下，就能得到实际的结果。

`volatile int` 类型变量是控制变量结果写回到内存，而不是存在共享内存，或者缓存中，因为下一步的计算马上要用到它，如果写入缓存，可能造成下一步的读取会读到错误的结果

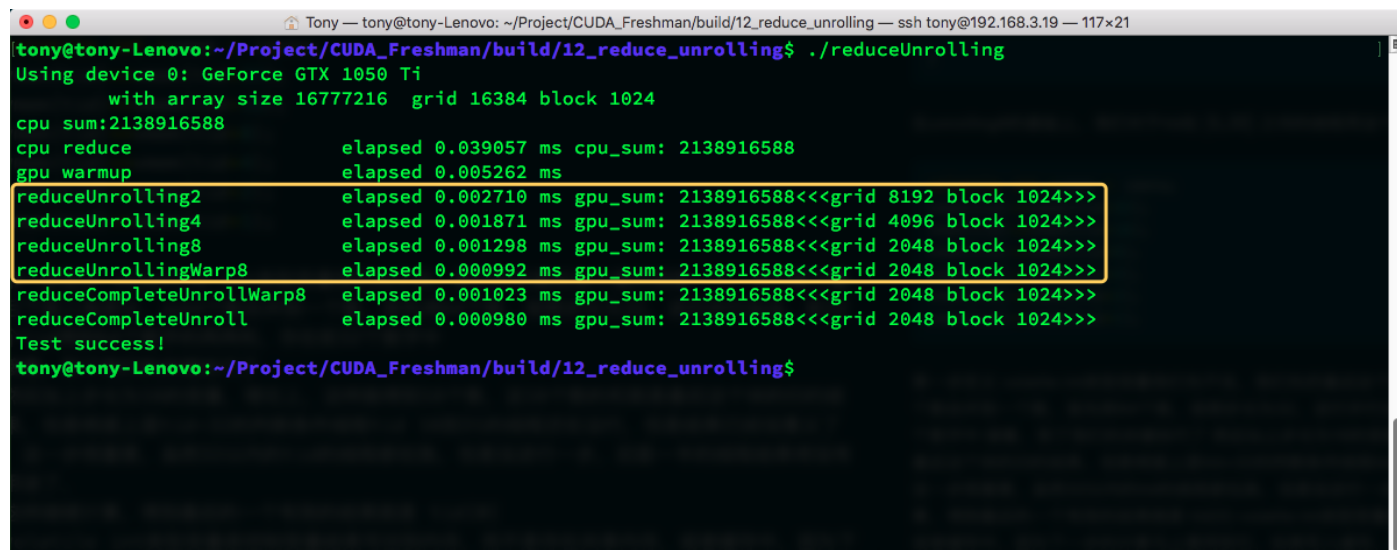
你可能不明白



```
1  vmem[tid]+=vmem[tid+32];
2  vmem[tid]+=vmem[tid+16];
```

tid+16要用到tid+32的结果，会不会有其他的线程造成内存竞争，答案是不会的，因为一个线程束，执行的进度是完全相同的，当执行 tid+32的时候，这32个线程都在执行这步，而不会有任何本线程束内的线程会进行到下一句，详情请回忆CUDA执行模型（因为CUDA编译器是激进的，所以我们必须添加volatile，防止编译器优化数据传输而打乱执行顺序）。

然后我们就得到结果了，看看时间：



```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/12_reduce_unrolling — ssh tony@192.168.3.19 — 117x21
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/12_reduce_unrolling$ ./reduceUnrolling
Using device 0: GeForce GTX 1050 Ti
with array size 16777216 grid 16384 block 1024
cpu sum:2138916588
cpu reduce          elapsed 0.039057 ms cpu_sum: 2138916588
gpu warmup          elapsed 0.005262 ms
reduceUnrolling2    elapsed 0.002710 ms gpu_sum: 2138916588<<<grid 8192 block 1024>>>
reduceUnrolling4    elapsed 0.001871 ms gpu_sum: 2138916588<<<grid 4096 block 1024>>>
reduceUnrolling8    elapsed 0.001298 ms gpu_sum: 2138916588<<<grid 2048 block 1024>>>
reduceUnrollingWarp8 elapsed 0.000992 ms gpu_sum: 2138916588<<<grid 2048 block 1024>>>
reduceCompleteUnrollWarp8 elapsed 0.001023 ms gpu_sum: 2138916588<<<grid 2048 block 1024>>>
reduceCompleteUnroll elapsed 0.000980 ms gpu_sum: 2138916588<<<grid 2048 block 1024>>>
Test success!
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/12_reduce_unrolling$
```

又往后退了一位，看起来还是很爽的。

这个展开还有一个节省时间的部分就是减少了5个线程束同步指令 \_\_syncthreads(); 这个指令被我们减少了5次，这个也是非常有效果的。我们来看看阻塞减少了多少

使用命令

```
1  nvprof --metrics stall_sync ./reduceUnrolling
```

```
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/12_reduce_unrolling$ nvprof --metrics stall_sync ./reduceUnrolling
==2743== NVRPROF is profiling process 2743, command: ./reduceUnrolling
Using device 0: GeForce GTX 1050 Ti
    with array size 16777216    grid 16384 block 1024
cpu sum:2139087239
cpu reduce      elapsed 0.039021 ms cpu_sum: 2139087239
==2743== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "warmup(int*, int*, unsigned int)" (6 of 6)...
Replaying kernel "warmup(int*, int*, unsigned int)" (done)
Replaying kernel "reduceUnroll2(int*, int*, unsigned int)" (done)
reduceUnrolling2 elapsed 0.038192 ms gpu_sum: 2139087239<<<grid 8192 block 1024>>>
Replaying kernel "reduceUnroll4(int*, int*, unsigned int)" (done)
reduceUnrolling4 elapsed 0.033406 ms gpu_sum: 2139087239<<<grid 4096 block 1024>>>
Replaying kernel "reduceUnroll8(int*, int*, unsigned int)" (done)
reduceUnrolling8 elapsed 0.029454 ms gpu_sum: 2139087239<<<grid 2048 block 1024>>>
Replaying kernel "reduceUnrollWarp8(int*, int*, unsigned int)" (done)
reduceUnrollingWarp8 elapsed 0.027721 ms gpu_sum: 2139087239<<<grid 2048 block 1024>>>
Replaying kernel "reduceCompleteUnrollWarp8(int*, int*, unsigned int)" (done)
reduceCompleteUnrollWarp8 elapsed 0.027589 ms gpu_sum: 2139087239<<<grid 2048 block 1024>>>
Replaying kernel "void reduceCompleteUnroll<unsigned int=1024>(int*, int*, unsigned int)" (done)
reduceCompleteUnroll elapsed 0.030124 ms gpu_sum: 2139087239<<<grid 2048 block 1024>>>
Test success!
==2743== Profiling application: ./reduceUnrolling
==2743== Profiling result:
==2743== Metric result:
Invocations      Metric Name      Metric Description      Min      Max      Avg
Device "GeForce GTX 1050 Ti (0)"
Kernel: reduceCompleteUnrollWarp8(int*, int*, unsigned int)
1      stall_sync      Issue Stall Reasons (Synchronization)      24.21%      24.21%      24.21%
Kernel: warmup(int*, int*, unsigned int)
1      stall_sync      Issue Stall Reasons (Synchronization)      26.36%      26.36%      26.36%
Kernel: reduceUnrollWarp8(int*, int*, unsigned int)
1      stall_sync      Issue Stall Reasons (Synchronization)      32.76%      32.76%      32.76%
Kernel: void reduceCompleteUnroll<unsigned int=1024>(int*, int*, unsigned int)
1      stall_sync      Issue Stall Reasons (Synchronization)      23.95%      23.95%      23.95%
Kernel: reduceUnroll8(int*, int*, unsigned int)
1      stall_sync      Issue Stall Reasons (Synchronization)      25.98%      25.98%      25.98%
Kernel: reduceUnroll4(int*, int*, unsigned int)
1      stall_sync      Issue Stall Reasons (Synchronization)      43.92%      43.92%      43.92%
Kernel: reduceUnroll2(int*, int*, unsigned int)
1      stall_sync      Issue Stall Reasons (Synchronization)      51.32%      51.32%      51.32%
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/12_reduce_unrolling$
```

哈哈，又搞笑了，书上的结果和运行结果又不一样，展开后的stall\_sync 指标反而高了，也就是说之前有同步指令的效率更高，哈哈，无解。。可以把锅甩给CUDA编译器。

## 模板函数的归约

根据上面展开最后64个数据，我们可以直接就展开最后128个，256个，512个，1024个，废话不多说，直接上代码，我们这次的目的就是让循环上西天：

```
1  __global__ void reduceCompleteUnrollWarp8(int * g_idata,int * g_odata,unsigned int
2  {
3      //set thread ID
4      unsigned int tid = threadIdx.x;
5      unsigned int idx = blockDim.x*blockIdx.x*8+threadIdx.x;
6      //boundary check
7      if (tid >= n) return;
8      //convert global data pointer to the
9      int *idata = g_idata + blockIdx.x*blockDim.x*8;
10     if(idx+7 * blockDim.x<n)
11     {
12         int a1=g_idata[idx];
13         int a2=g_idata[idx+blockDim.x];
14         int a3=g_idata[idx+2*blockDim.x];
15         int a4=g_idata[idx+3*blockDim.x];
16         int a5=g_idata[idx+4*blockDim.x];
```

```

17         int a6=g_idata[idx+5*blockDim.x];
18         int a7=g_idata[idx+6*blockDim.x];
19         int a8=g_idata[idx+7*blockDim.x];
20         g_idata[idx]=a1+a2+a3+a4+a5+a6+a7+a8;
21
22     }
23     __syncthreads();
24     //in-place reduction in global memory
25     if(blockDim.x>=1024 && tid <512)
26         idata[tid]+=idata[tid+512];
27     __syncthreads();
28     if(blockDim.x>=512 && tid <256)
29         idata[tid]+=idata[tid+256];
30     __syncthreads();
31     if(blockDim.x>=256 && tid <128)
32         idata[tid]+=idata[tid+128];
33     __syncthreads();
34     if(blockDim.x>=128 && tid <64)
35         idata[tid]+=idata[tid+64];
36     __syncthreads();
37     //write result for this block to global mem
38     if(tid<32)
39     {
40         volatile int *vmem = idata;
41         vmem[tid]+=vmem[tid+32];
42         vmem[tid]+=vmem[tid+16];
43         vmem[tid]+=vmem[tid+8];
44         vmem[tid]+=vmem[tid+4];
45         vmem[tid]+=vmem[tid+2];
46         vmem[tid]+=vmem[tid+1];
47
48     }
49
50     if (tid == 0)
51         g_odata[blockIdx.x] = idata[0];
52
53 }

```

内核代码如上，这里用到了tid的大小，和最后32个没用到tid不同的是，这些如果计算完整会有一半是浪费的，而最后32个已经是线程束最小的尺寸了，所以无论后面的数据有没有意义，那些进程都不会停。每一步进行显示的同步，然后我们看结果，哈哈，又又又搞笑了：

```
Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/12_reduce_unrolling — ssh tony@192.168.3.19 — 111x20
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/12_reduce_unrolling$ ./reduceUnrolling
Using device 0: GeForce GTX 1050 Ti
    with array size 16777216  grid 16384 block 1024
cpu sum:2139025811
cpu reduce                elapsed 0.039077 ms cpu_sum: 2139025811
gpu warmup                elapsed 0.005294 ms
reduceUnrolling2          elapsed 0.002721 ms gpu_sum: 2139025811<<<grid 8192 block 1024>>>
reduceUnrolling4          elapsed 0.001870 ms gpu_sum: 2139025811<<<grid 4096 block 1024>>>
reduceUnrolling8          elapsed 0.001294 ms gpu_sum: 2139025811<<<grid 2048 block 1024>>>
reduceUnrollingWarp8      elapsed 0.001009 ms gpu_sum: 2139025811<<<grid 2048 block 1024>>>
reduceCompleteUnrollWarp8 elapsed 0.001001 ms gpu_sum: 2139025811<<<grid 2048 block 1024>>>
reduceCompleteUnroll      elapsed 0.001008 ms gpu_sum: 2139025811<<<grid 2048 block 1024>>>
Test success!
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/12_reduce_unrolling$
```

似乎速度根本没什么影响，所以我觉得是编译器的锅没错了！他已经帮我们优化这一步了。

## 模板函数的归约

我们看上面这个完全展开的函数，

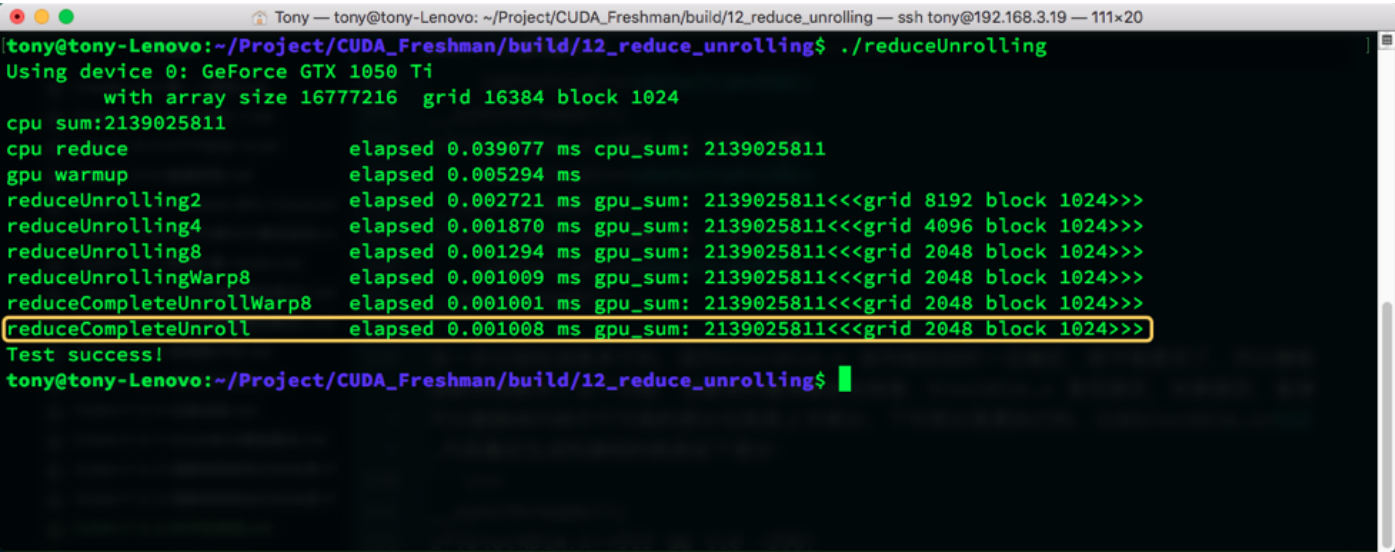
```
1  if(blockDim.x>=1024 && tid <512)
2      idata[tid]+=idata[tid+512];
3  __syncthreads();
4  if(blockDim.x>=512 && tid <256)
5      idata[tid]+=idata[tid+256];
6  __syncthreads();
7  if(blockDim.x>=256 && tid <128)
8      idata[tid]+=idata[tid+128];
9  __syncthreads();
10 if(blockDim.x>=128 && tid <64)
11     idata[tid]+=idata[tid+64];
12 __syncthreads();
```

这一步比较应该是多余的，因为blockDim.x 自内核启动时一旦确定，就不能更改了，所以模板函数帮我解决了这个问题，当编译时编译器会去检查，blockDim.x 是否固定，如果固定，直接可以删除掉内核中不可能的部分也就是上半部分，下半部分是要执行的，比如blockDim.x=512 ,代码最后生成机器码的就是如下部分：

```
1  __syncthreads();
2  if(blockDim.x>=512 && tid <256)
```

```
3      idata[tid]+=idata[tid+256];
4      __syncthreads();
5      if(blockDim.x>=256 && tid <128)
6          idata[tid]+=idata[tid+128];
7      __syncthreads();
8      if(blockDim.x>=128 && tid <64)
9          idata[tid]+=idata[tid+64];
10     __syncthreads();
```

删掉了不可能的部分。  
我们来看下模板函数的效率：



结果是，居然还慢了一些。。书上不是这么说的。。编译器的锅！

小结

我们总结下本文四步优化的指标对比：  
加载效率存储效率：

```
1 nvprof --metrics gld_efficiency,gst_efficiency ./reduceUnrolling
```

算法	时间	加载效率	存储效率
相邻无分化（上一篇）	0.010491	25.01%	25.00%

相邻分化（上一篇）	0.005968	25.01%	25.00%
交错（上一篇）	0.004956	98.04%	97.71%
展开8	0.001294	99.60%	99.71%
展开8+最后的展开	0.001009	99.71%	99.68%
展开8+完全展开+最后的展开	0.001001	99.71%	99.68%
模板上一个算法	0.001008	99.71%	99.68%

虽然和书上结果不太一样，但是指标和效率关系还是很明显的，所以我们今天得出的结论是。。一步一步优化，如果改了代码没效果，那么锅是编译器的，谁让你帮老子优化了还不告诉我！

## 总结

在开发之前最好了解一下编译器特性，以及最关键的一点，一个真理，没有人能一下写出最好的代码，一步一步优化，从0.0104优化到0.001001，近10倍的效率提升，一分钟的程序你可能觉得无所谓但是一个运行一年的程序，有化了十倍，那可是大功一件啊。  
下篇我们继续，锅都是编译器的！

本文作者： 谭升  
本文链接： <https://face2ai.com/CUDA-F-3-5-展开循环/>  
版权声明： 本博客所有文章除特别声明外，均采用 [CC BY-NC-SA 4.0](#) 许可协议。转载请注明出处！

### 相关文章

- [【CUDA 基础】 5.3 减少全局内存访问](#)
- [【Julia】 整型和浮点型数字](#)
- [【Julia】 变量](#)
- [【Julia】 开始使用Julia](#)