苹果妖

Anything that can go wrong will go wrong !

CUDA ---- Kernel性能调节

Exposing Parallelism

这部分主要介绍并行分析，涉及掌握nvprof的几个metric参数，具体的这些调节为什么会影响性能会在后续博文解释。

代码准备

下面是我们的kernel函数sumMatrixOnGPUD：

```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int NX, int NY) {
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;
    if (ix < NX && iy < NY) {
        C[idx] = A[idx] + B[idx];
    }
}
```

我们指定一个比较大的数据矩阵，包含16384个元素：

```
int nx = 1<<14;
int ny = 1<<14;
```

下面的代码用来配置main函数的参数，也就是block的维度配置：

```
if (argc > 2) {
    dimx = atoi(argv[1]);
    dimy = atoi(argv[2]);
}
dim3 block(dimx, dimy);
dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y);
```

编译：

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

Checking Active Warps with nvprof

在做各项数据比较的时候需要有个基准，这里使用四个block配置的时间消耗作为基准观察，分别为（32，32）（32,16）（16,32）和（16,16），本文开始时有提到，第一个参数是x象限维度，第二个参数是y象限维度。

下面是几种配置的时间消耗输出结果：

```
$ ./sumMatrix 32 32
sumMatrixOnGPU2D <<< (512,512), (32,32) >>> elapsed 60 ms
$ ./sumMatrix 32 16
sumMatrixOnGPU2D <<< (512,1024), (32,16) >>> elapsed 38 ms
```

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)

```
$ ./sumMatrix 16 32
sumMatrixOnGPU2D <<< (1024,512), (16,32) >>> elapsed 51 ms
$ ./sumMatrix 16 16
sumMatrixOnGPU2D <<< (1024,1024),(16,16) >>> elapsed 46 ms
```

比较这几个结果，不难发现，最慢的是第一个（32,32），最快的是第二个（32,16），这里可以猜测的到的是，拥有更多的block并行性更好。这个猜测可以使用nvprof 的achieved_occupancy这个metric参数来验证。该参数的定义公式在上一篇博文有介绍，实际上就是指每个SM在每个cycle能够达到的最大active warp数目占总warp的比例。下面是使用该参数后得到的结果：

```
$ nvprof --metrics achieved_occupancy ./sumMatrix 32 32
sumMatrixOnGPU2D <<<(512,512), (32,32)>>> Achieved Occupancy 0.501071
$ nvprof --metrics achieved_occupancy ./sumMatrix 32 16
sumMatrixOnGPU2D <<<(512,1024), (32,16)>>> Achieved Occupancy 0.736900
$ nvprof --metrics achieved_occupancy ./sumMatrix 16 32
sumMatrixOnGPU2D <<<(1024,512), (16,32)>>> Achieved Occupancy 0.766037
$ nvprof --metrics achieved_occupancy ./sumMatrix 16 16
sumMatrixOnGPU2D <<<(1024,1024),(16,16)>>> Achieved Occupancy 0.810691
```

从上面的输出可以得知两件事儿：

1. 由于第二个配置比第一个有更多的block，device就会达到更多active warp（跟鸡蛋放在多个篮子的道理差不多）。也就是第二个性能优于第一个的原因。
2. 第四个的achieved Occupancy最高，但是却不是最快的，因此，较高的achieved Occupancy并不一定就意味着更好的性能，也就是说还有更多的因素影响着GPU的性能。

checking memory operations with nvprof

对于C[idx] = A[idx] + B[idx]来说共有三个memory操作：两个memory load和一个memory store。要查看这些操作的效率可以使用nvprof的两个metric参数，如果想要查看memory的throughput，则可使用gld_throughput：

```
$ nvprof --metrics gld_throughput./sumMatrix 32 32
sumMatrixOnGPU2D <<<(512,512), (32,32)>>> Global Load Throughput 35.908GB/s
$ nvprof --metrics gld_throughput./sumMatrix 32 16
sumMatrixOnGPU2D <<<(512,1024), (32,16)>>> Global Load Throughput 56.478GB/s
$ nvprof --metrics gld_throughput./sumMatrix 16 32
sumMatrixOnGPU2D <<<(1024,512), (16,32)>>> Global Load Throughput 85.195GB/s
$ nvprof --metrics gld_throughput./sumMatrix 16 16
sumMatrixOnGPU2D <<<(1024,1024),(16,16)>>> Global Load Throughput 94.708GB/s
```

不难看到，第四个拥有最高的load throughput，但是却比第二个慢（第二个也就是第四个的一半），所以，较高的load throughput也不一定就有较高的性能。之后讲到memory transaction时会具体分析这种现象的原因，简单说，就是高load throughput有可能是一种假象，如果需要的数据在memory中存储格式未对齐不连续，会导致许多额外的不必要的load操作，所以本文中的efficiency会这么低。

然后，我们可以使用nvprof的gld_efficiency来度量load efficiency，该metric参数是指我们确切需要的global load throughput与实际得到global load memory的比值。这个metric参数可以让我们知道，APP的load操作利用device memory bandwidth的程度：

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profili.....
--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(5853)
2. CUDA ---- Warp解析(4401)
3. CUDA ---- GPU架构（Fermi、Kepler）(2608)
4. CUDA ---- Memory Model(2221)
5. CUDA ---- Stream and Event(2103)

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)

```
$ nvprof --metrics gld_efficiency ./sumMatrix 32 32
sumMatrixOnGPU2D <<<(512,512), (32,32)>>> Global Memory Load Efficiency 100.00%
$ nvprof --metrics gld_efficiency ./sumMatrix 32 16
sumMatrixOnGPU2D <<<(512,1024), (32,16)>>> Global Memory Load Efficiency 100.00%
$ nvprof --metrics gld_efficiency ./sumMatrix 16 32
sumMatrixOnGPU2D <<<(1024,512), (16,32)>>> Global Memory Load Efficiency 49.96%
$ nvprof --metrics gld_efficiency ./sumMatrix 16 16
sumMatrixOnGPU2D <<<(1024,1024),(16,16)>>> Global Memory Load Efficiency 49.80%
```

从上述结果可知，最后两个的load efficiency只是前两个的一半。这也可以解释，为什么较高的throughput和较高的Occupancy却没有产生较好的性能。尽管最后两个的load操作数目要多不少（因为二者throughput较高），但是他们的load effecitiveness却低不少（由于efficiency较低）。

观察最后两个可以发现，他们block的x象限配置是warp的一半，由前文推测知，该象限应该保持为warp大小的整数倍。关于其具体原因将在后续博文详细解释。

Exposing More Parallelism

我们现在可以得出一个结论就是blockDim.x应该是warp大小的整数倍。这样做是很容易就提升了load efficiency。现在，我们可能还有其他疑惑，比如：

- 继续调整blockDim.x是否会继续增加load throughput？
- 还有其他方法能增大并行性吗？

现在，我们重新整一个基准数据出来，这两个问题可以从这个基准分析个大概：

```
$ ./sumMatrix 64 2
sumMatrixOnGPU2D <<<(256,8192), (64,2) >>> elapsed 0.033567 sec
$ ./sumMatrix 64 4
sumMatrixOnGPU2D <<<(256,4096), (64,4) >>> elapsed 0.034908 sec
$ ./sumMatrix 64 8
sumMatrixOnGPU2D <<<(256,2048), (64,8) >>> elapsed 0.036651 sec
$ ./sumMatrix 128 2
sumMatrixOnGPU2D <<<(128,8192), (128,2)>>> elapsed 0.032688 sec
$ ./sumMatrix 128 4
sumMatrixOnGPU2D <<<(128,4096), (128,4)>>> elapsed 0.034786 sec
$ ./sumMatrix 128 8
sumMatrixOnGPU2D <<<(128,2048), (128,8)>>> elapsed 0.046157 sec
$ ./sumMatrix 256 2
sumMatrixOnGPU2D <<<(64,8192), (256,2)>>> elapsed 0.032793 sec
$ ./sumMatrix 256 4
sumMatrixOnGPU2D <<<(64,4096), (256,4)>>> elapsed 0.038092 sec
$ ./sumMatrix 256 8
sumMatrixOnGPU2D <<<(64,2048), (256,8)>>> elapsed 0.000173 sec
Error: sumMatrix.cu:163, code:9, reason: invalid configuration argument
```

从上面数据，我们能够分析出来的是：

- 最后一个配置（256,8）不可行，block中总共的thread数目超过了1024，这是GPU的硬件限制。
- 最好的结果是第四个（128，2）。
- 第一个启动了最多的block，但不是最快的。
- 因为第二个与第四个在一个block中拥有相同数目的thread，本应猜测二者有相同的表现，但是实际却是第二个略逊色，所以blockDim.x的大小是很关键的。
- 剩下的相对第四个都有较少的block数目，所以并行规模也是影响性能的关键因素。

现在，我们又得猜测了，拥有block最少的应该会有一个最低的achieved Occupancy吧？而拥有最多block的应该会达到最高的achieved Occupancy吧？为了验证这些想法，我们再看一组数据：

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

```
$ nvprof --metrics achieved_occupancy ./sumMatrix 64 2
sumMatrixOnGPU2D <<<(256,8192), (64,2) >>> Achieved Occupancy 0.554556
$ nvprof --metrics achieved_occupancy ./sumMatrix 64 4
sumMatrixOnGPU2D <<<(256,4096), (64,4) >>> Achieved Occupancy 0.798622
$ nvprof --metrics achieved_occupancy ./sumMatrix 64 8
sumMatrixOnGPU2D <<<(256,2048), (64,8) >>> Achieved Occupancy 0.753532
$ nvprof --metrics achieved_occupancy ./sumMatrix 128 2
sumMatrixOnGPU2D <<<(128,8192), (128,2)>>> Achieved Occupancy 0.802598
$ nvprof --metrics achieved_occupancy ./sumMatrix 128 4
sumMatrixOnGPU2D <<<(128,4096), (128,4)>>> Achieved Occupancy 0.746367
$ nvprof --metrics achieved_occupancy ./sumMatrix 128 8
sumMatrixOnGPU2D <<<(128,2048), (128,8)>>> Achieved Occupancy 0.573449
$ nvprof --metrics achieved_occupancy ./sumMatrix 256 2
sumMatrixOnGPU2D <<<(64,8192), (256,2) >>> Achieved Occupancy 0.760901
$ nvprof --metrics achieved_occupancy ./sumMatrix 256 4
sumMatrixOnGPU2D <<<(64,4096), (256,4) >>> Achieved Occupancy 0.595197
```

看到了吧，（64,2）的achieved Occupancy竟然是最低的，尽管他有最多的block（高中做物理题也是这感觉），它达到了硬件对block数量的限制。

第四个（128,2）和第七个（256,2）拥有拥有差不多的achieved Occupancy。我们对这两个再做一个试验，再次增大，将blockDim.y设置为1，这也减少了block的大小：

```
$ ./sumMatrix 128 1
sumMatrixOnGPU2D <<<(128,16384),(128,1)>>> elapsed 0.032602 sec
$ ./sumMatrix 256 1
sumMatrixOnGPU2D <<<(64,16384), (256,1)>>> elapsed 0.030959 sec
```

这次配置产生了最佳的性能，特别是，（256,1）要比（128,1）要更好，，再次检查achieved Occupancy，load throughput和load efficiency：

```
$ nvprof --metrics achieved_occupancy ./sumMatrix 256 1
$ nvprof --metrics gld_throughput ./sumMatrix 256 1
$ nvprof --metrics gld_efficiency ./sumMatrix 256 1
```

输出：

```
Achieved Occupancy 0.808622
Global Load Throughput 69.762GB/s
Global Memory Load Efficiency 100.00%
```

现在可以看出，最佳配置既不是拥有最高achieved Occupancy也不是最高load throughput的。所以不存在唯一metric来优化计算性能，我么需要从众多metric中寻求一个平衡。

总结

- 在大多数情形下，并不存在唯一的metric可以精确的优化性能。
- 哪个metric或者event对性能的影响大是由kernel具体的代码决定的。
- 在众多相关的metric和event中寻求一个平衡。
- Grid/blcok heuristics（启发）为调节性能提供了不错的切入点。

分类: c/c++,CUDA

标签: CUDA, 并行计算

苹果妖
关注 - 1
粉丝 - 22
+加关注

0 0

« 上一篇：MachineLearning ---- lesson 1
» 下一篇：CUDA ---- Branch Divergence and Unrolling Loop

posted @ 2015-06-01 23:35 苹果妖 阅读(1613) 评论(0) 编辑 收藏