苹果妖

Anything that can go wrong will go wrong !

CUDA ---- Warp解析

Warp

逻辑上，所有thread是并行的，但是，从硬件的角度来说，实际上并不是所有的thread能够在同一时刻执行，接下来我们将解释有关warp的一些本质。

Warps and Thread Blocks

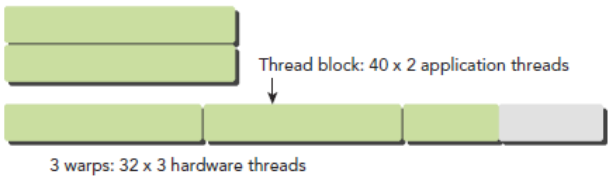
warp是SM的基本执行单元。一个warp包含32个并行thread，这32个thread执行于SMIT模式。也就是说所有thread执行同一条指令，并且每个thread会使用各自的data执行该指令。

block可以是一维二维或者三维的，但是，从硬件角度看，所有的thread都被组织成一维，每个thread都有个唯一的ID(ID的计算可以在之前的博文查看)。

每个block的warp数量可以由下面的公式计算获得：

$$WarpsPerBlock = ceil\left(\frac{ThreadsPerBlock}{warpSize}\right)$$

一个warp中的线程必然在同一个block中，如果block所含线程数目不是warp大小的整数倍，那么多出的那些thread所在的warp中，会剩余一些inactive的thread，也就是说，即使凑不够warp整数倍的thread，硬件也会为warp凑足，只不过那些thread是inactive状态，需要注意的是，即使这部分thread是inactive的，也会消耗SM资源。



Warp Divergence

控制流语句普遍存在于各种编程语言中，GPU支持传统的，C-style，显式控制流结构，例如if…else,for,while等等。

CPU有复杂的硬件设计可以很好的做分支预测，即预测应用程序会走哪个path。如果预测正确，那么CPU只会有很小的消耗。和CPU对比来说，GPU就没那么复杂的分支预测了（CPU和GPU这方面的差异的原因不是我们关心的，了解就好，我们关心的是由这差异引起的问题）。

这样我们的问题就来了，因为所有同一个warp中的thread必须执行相同的指令，那么如果这些线程在遇到控制流语句时，如果进入不同的分支，那么同一时刻除了正在执行的分之外，其余分支都被阻塞了，十分影响性能。这类问题就是warp divergence。

请注意，warp divergence问题只会发生在同一个warp中。

下图展示了warp divergence问题：

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

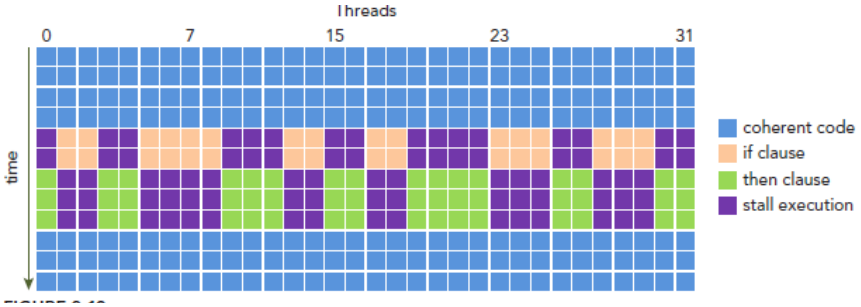
我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

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)



为了获得最好的性能，就需要避免同一个warp存在不同的执行路径。避免该问题的方法很多，比如这样一个情形，假设有两个分支，分支的决定条件是thread的唯一ID的奇偶性：

```
__global__ void mathKernel1(float *c) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float a, b;
    a = b = 0.0f;
    if (tid % 2 == 0) {
        a = 100.0f;
    } else {
        b = 200.0f;
    }
    c[tid] = a + b;
}
```

一种方法是，将条件改为以warp大小为步调，然后取奇偶，如下：

```
__global__ void mathKernel2(void) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float a, b;
    a = b = 0.0f;
    if ((tid / warpSize) % 2 == 0) {
        a = 100.0f;
    } else {
        b = 200.0f;
    }
    c[tid] = a + b;
}
```

代码：

```
int main(int argc, char **argv) {
// set up device
int dev = 0;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
printf("%s using Device %d: %s\n", argv[0],dev, deviceProp.name);
// set up data size
int size = 64;
int blocksize = 64;
if(argc > 1) blocksize = atoi(argv[1]);
if(argc > 2) size = atoi(argv[2]);
printf("Data size %d ", size);
// set up execution configuration
dim3 block (blocksize,1);
dim3 grid ((size+block.x-1)/block.x,1);
printf("Execution Configure (block %d grid %d)\n",block.x, grid.x);
```

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)

```
// allocate gpu memory
float *d_C;
size_t nBytes = size * sizeof(float);
cudaMalloc((float**)&d_C, nBytes);
// run a warmup kernel to remove overhead
size_t iStart,iElaps;
cudaDeviceSynchronize();
iStart = seconds();
warmingup<<<grid, block>>> (d_C);
cudaDeviceSynchronize();
iElaps = seconds() - iStart;
printf("warmup <<< %4d %4d >>> elapsed %d sec \n",grid.x,block.x, iElaps );
// run kernel 1
iStart = seconds();
mathKernel1<<<grid, block>>>(d_C);
cudaDeviceSynchronize();
iElaps = seconds() - iStart;
printf("mathKernel1 <<< %4d %4d >>> elapsed %d sec \n",grid.x,block.x,iElaps );
// run kernel 3
iStart = seconds();
mathKernel2<<<grid, block>>>(d_C);
cudaDeviceSynchronize();
iElaps = seconds () - iStart;
printf("mathKernel2 <<< %4d %4d >>> elapsed %d sec \n",grid.x,block.x,iElaps );
// run kernel 3
iStart = seconds ();
mathKernel3<<<grid, block>>>(d_C);
cudaDeviceSynchronize();
iElaps = seconds () - iStart;
printf("mathKernel3 <<< %4d %4d >>> elapsed %d sec \n",grid.x,block.x,iElaps);
// run kernel 4
iStart = seconds ();
mathKernel4<<<grid, block>>>(d_C);
cudaDeviceSynchronize();
iElaps = seconds () - iStart;
printf("mathKernel4 <<< %4d %4d >>> elapsed %d sec \n",grid.x,block.x,iElaps);
// free gpu memory and reset divece
cudaFree(d_C);
cudaDeviceReset();
return EXIT_SUCCESS;
}
```

编译运行：

```
$ nvcc -O3 -arch=sm_20 simpleDivergence.cu -o simpleDivergence
$./simpleDivergence
```

输出：

```
$ ./simpleDivergence using Device 0: Tesla M2070
Data size 64 Execution Configuration (block 64 grid 1)
Warmingup elapsed 0.000040 sec
mathKernel1 elapsed 0.000016 sec
mathKernel2 elapsed 0.000014 sec
```

我们也可以直接使用nvprof (之后会详细介绍) 这个工具来度量性能：

$ nvprof --metrics branch_efficiency ./simpleDivergence

输出为：

```
Kernel: mathKernel1(void)
1 branch_efficiency Branch Efficiency 100.00% 100.00% 100.00%
Kernel: mathKernel2(void)
1 branch_efficiency Branch Efficiency 100.00% 100.00% 100.00%
```

Branch Efficiency的定义如下：

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

$$Branch\ Efficiency = 100 \times \left(\frac{\#\ Branches - \#\ Divergent\ Branches}{\#\ Branches} \right)$$

到这里你应该在奇怪为什么二者表现相同呢，实际上当我们的代码很简单，可以被预测时，CUDA的编译器会自动帮助优化我们的代码。稍微提一下GPU分支预测（理解的有点晕，不过了解下就好），这里，一个被称为预测变量的东西会被设置成1或者0，所有分支都会得到执行，但是只有预测值为1时，才会得到执行。当条件状态少于某一个阈值时，编译器会将一个分支指令替换为预测指令，因此，现在回到自动优化问题，一份较长的代码就会导致warp divergence了。

可以使用下面的命令强制编译器不优化（貌似不怎么管用）：

```
$ nvcc -g -G -arch=sm_20 simpleDivergence.cu -o simpleDivergence
```

Resource Partitioning

一个warp的context包括以下三部分：

1. Program counter
2. Register
3. Shared memory

再次重申，在同一个执行context中切换是没有消耗的，因为在整个warp的生命期内，SM处理的每个warp的执行context都是on-chip的。

每个SM有一个32位register集合放在register file中，还有固定数量的shared memory，这些资源都被thread瓜分了，由于资源是有限的，所以，如果thread比较多，那么每个thread占用资源就叫少，thread较少，占用资源就较多，这需要根据自己的要求作出一个平衡。

资源限制了驻留在SM中blcok的数量，不同的device，register和shared memory的数量也不同，就像之前介绍的Fermi和Kepler的差别。如果没有足够的资源，kernel的启动就会失败。

TECHNICAL SPECIFICATIONS	COMPUTE CAPABILITY			
	2.0	2.1	3.0	3.5
Maximum number of threads per block	1,024			
Maximum number of concurrent blocks per multiprocessor	8		16	
Maximum number of concurrent warps per multiprocessor	48		64	
Maximum number of concurrent threads per multiprocessor	1,536		2,048	
Number of 32-bit registers per multiprocessor	32 K		64 K	
Maximum number of 32-bit registers per thread	63		255	
Maximum amount of shared memory per multiprocessor	48 K			

当一个block或得到足够的资源时，就成为active block。block中的warp就称为active warp。active warp又可以被分为下面三类：

1. Selected warp
2. Stalled warp
3. Eligible warp

SM中warp调度器每个cycle会挑选active warp送去执行，一个被选中的warp称为selected warp，没被选中，但是已经做好准备被执行的称为Eligible warp，没准备好要执行的称为Stalled warp。warp适合执行需要满足下面两个条件：

1. 32个CUDA core有空
2. 所有当前指令的参数都准备就绪

例如，Kepler任何时刻的active warp数目必须少于或等于64个（GPU架构篇有介绍）。selected warp数目必须小于或等于4个（因为scheduler有4个？不确定，至于4个是不是太少则不用担心，kernel启动前，会有一个warmup操作，可以使用cudaFree()来实现）。如果一个warp阻塞了，调度器会挑选一个Eligible warp准备去执行。

CUDA编程中应该重视对计算资源的分配：这些资源限制了active warp的数量。因此，我们必须掌握硬件的一些限

制，为了最大化GPU利用率，我们必须最大化active warp的数目。

Latency Hiding

指令从开始到结束消耗的clock cycle称为指令的latency。当每个cycle都有eligible warp被调度时，计算资源就会得到充分利用，基于此，我们就可以将每个指令的latency隐藏于issue其它warp的指令的过程中。

和CPU编程相比，latency hiding对GPU非常重要。CPU cores被设计成可以最小化一到两个thread的latency，但是GPU的thread数目可不是一个两个那么简单。

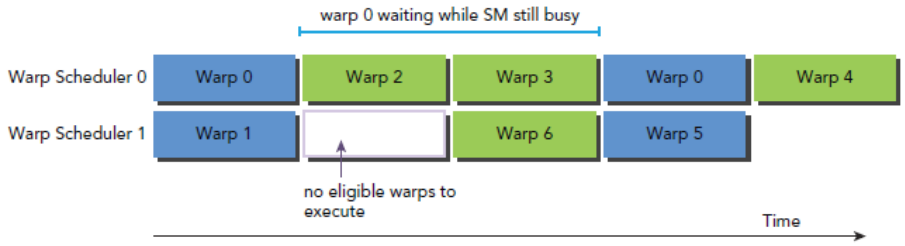
当涉及到指令latency时，指令可以被区分为下面两种：

1. Arithmetic instruction
2. Memory instruction

顾名思义，Arithmetic instruction latency是一个算数操作的始末间隔。另一个则是指load或store的始末间隔。二者的latency大约为：

1. 10-20 cycle for arithmetic operations
2. 400-800 cycles for global memory accesses

下图是一个简单的执行流程，当warp0阻塞时，执行其他的warp，当warp变为eligible时从新执行。



你可能想要知道怎样评估active warps 的数量来hide latency。Little's Law可以提供一个合理的估计：

$$Number\ of\ Required\ Warps = Latency \times Throughput$$

对于Arithmetic operations来说，并行性可以表达为用来hide Arithmetic latency的操作的数目。下表显示了Fermi和Kepler相关数据，这里是以（a + b * c）作为操作的例子。不同的算数指令，throughput（吞吐）也是不同的。

TABLE 3-3: SM Parallelism Required to Maintain Full Arithmetic Utilization

GPU MODEL	INSTRUCTION LATENCY (CYCLES)	THROUGHPUT (OPERATIONS/CYCLE)	PARALLELISM (OPERATIONS)
Fermi	20	32	640
Kepler	20	192	3,840

这里的throughput定义为每个SM每个cycle的操作数目。由于每个warp执行同一种指令，因此每个warp对应32个操作。所以，对于Fermi来说，每个SM需要640/32=20个warp来保持计算资源的充分利用。这也就意味着，arithmetic operations的并行性可以表达为操作的数目或者warp的数目。二者的关系也对应了两种方式来增加并行性：

1. Instruction-level Parallelism（ILP）：同一个thread中更多的独立指令
2. Thread-level Parallelism（TLP）：更多并发的eligible threads

对于Memory operations，并行性可以表达为每个cycle的byte数目。

TABLE 3-4: Device Parallelism Required to Maintain Full Memory Utilization

GPU MODEL	INSTRUCTION LATENCY (CYCLES)	BANDWIDTH (GB/SEC)	BANDWIDTH (B/CYCLE)	PARALLELISM (KB)
Fermi	800	144	92	74
Kepler	800	250	96	77

因为memory throughput总是以GB/Sec为单位，我们需要先作相应的转化。可以通过下面的指令来查看device的memory frequency：

```
$ nvidia-smi -a -q -d CLOCK | fgrep -A 3 "Max Clocks" | fgrep "Memory"
```

以Fermi为例，其memory frequency可能是1.566GHz，Kepler的是1.6GHz。那么转化过程为：

$$144\ \text{GB/Sec} \div 1.566\ \text{GHz} \cong 92\ \text{Bytes/Cycle}$$

乘上这个92可以得到上图中的74，这里的数字是针对整个device的，而不是每个SM。

有了这些数据，我们可以做一些计算了，以Fermi为例，假设每个thread的任务是将一个float（4 bytes）类型的数据从global memory移至SM用来计算，你应该需要大约18500个thread，也就是579个warp来隐藏所有的memory latency。

$$74\ \text{KB} \div 4\ \text{bytes/thread} \cong 18{,}500\ \text{threads}$$

$$18{,}500\ \text{threads} \div 32\ \text{threads/warp} \cong 579\ \text{warps}$$

Fermi有16个SM，所以每个SM需要579/16=36个warp来隐藏memory latency。

Occupancy

当一个warp阻塞了，SM会执行另一个eligible warp。理想情况是，每时每刻到保证cores被占用。Occupancy就是每个SM的active warp占最大warp数目的比例：

$$occupancy = \frac{active\ warps}{maximum\ warps}$$

我们可以使用的device篇提到的方法来获取warp最大数目：

```
cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp *prop, int device);
```

然后用maxThreadsPerMultiProcessor来获取具体数值。

grid和block的配置准则：

- 保证block中thrad数目是32的倍数。
- 避免block太小：每个blcok最少128或256个thread。
- 根据kernel需要的资源调整block。
- 保证block的数目远大于SM的数目。
- 多做实验来挖掘出最好的配置。

Occupancy专注于每个SM中可以并行的thread或者warp的数目。不管怎样，Occupancy不是唯一的性能指标，Occupancy达到当某个值是，再做优化就可能不在有效果了，还有许多其它的指标需要调节，我们会在之后的博文继续探讨。

Synchronize

同步是并行编程的一个普遍的问题。在CUDA的世界里，有两种方式实现同步：

1. System-level：等待所有host和device的工作完成
2. Block-level：等待device中block的所有thread执行到某个点

因为CUDA API和host代码是异步的，cudaDeviceSynchronize可以用来停住CUP等待CUDA中的操作完成：

```
cudaError_t cudaDeviceSynchronize(void);
```

因为block中的thread执行顺序不定，CUDA提供了一个function来同步block中的thread。

```
__device__ void __syncthreads(void);
```

当该函数被调用，block中的每个thread都会等待所有其他thread执行到某个点来实现同步。

分类: c/c++,CUDA

标签: CUDA, 并行计算

苹果妖
关注 - 1
粉丝 - 22
+加关注

« 上一篇：CUDA ---- GPU架构（Fermi、Kepler）
» 下一篇：MachineLearning ---- lesson 1

2 0

posted @ 2015-05-31 00:02 苹果妖 阅读(4402) 评论(0) 编辑 收藏