

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

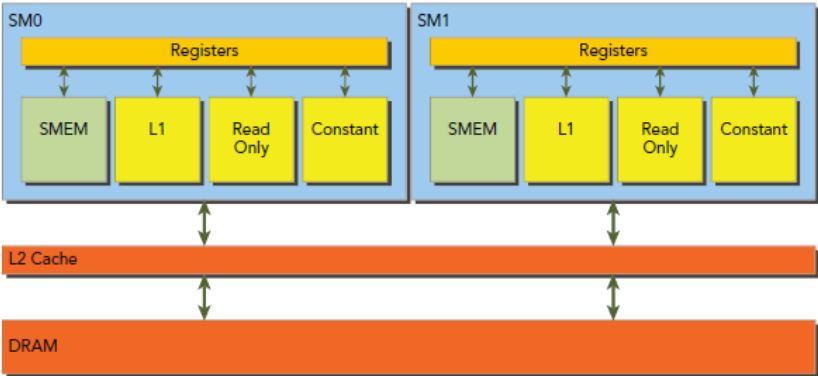
CUDA ---- Memory Access

Memory Access Patterns

大部分device一开始从global Memory获取数据，而且，大部分GPU应用表现会被带宽限制。因此最大化应用对global Memory带宽的使用时获取高性能的第一步。也就是说，global Memory的使用就没调节好，其它的优化方案也获取不到什么大效果,下面的内容会涉及到不少L1的知识，这部分了解下就好，L1在Maxwell之后就不用了，但是cache的知识点是不变的。

Aligned and Coalesced Access

如下图所示，global Memory的load/store要经由cache，所有的数据会初始化在DRAM，也就是物理的device Memory上，而kernel能够获取的global Memory实际上是一块逻辑内存空间。Kernel对Memory的请求都是由DRAM和SM的片上内存以128-byte和32-byte传输解决的。



所有获取global Memory都要经过L2 cache，也有许多还要经过L1 cache，主要由GPU的架构和获取模式决定的。如果L1和L2都被使用，那么Memory的获取是以128-byte为单位传输的，如果只使用L2，则以32-byte为单位传输，在允许使用L1的GPU中（Maxwell已经彻底不使用L1，原本走L1都换成走texture cache），L1是可以在编译期被显示使用或禁止的。

由上文可知，L1 cache中每一行是128bytes，这些数据映射到device Memory上的128位对齐的块。如果warp中每个thread请求一个4-byte的值，那么每次请求会要求获取128 bytes值，正好契合cache line大小和device Memory segment大小。

因此，我们在设计代码的时候，有两个特征需要注意：

- 1. Aligned Memory access 对齐
- 2. Coalesced Memory access 连续

当要获取的Memory首地址是cache line的倍数时，就是Aligned Memory Access，如果是非对齐的，就会导致浪费带宽。至于Coalesced Memory Access则是warp的32个thread请求的是连续的内存块。

下图就是很好的符合了连续和对齐原则，只有128-byte Memory传输的消耗：

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

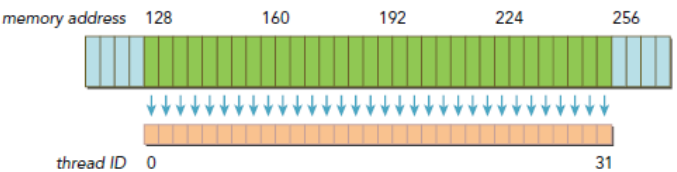
我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

我的标签

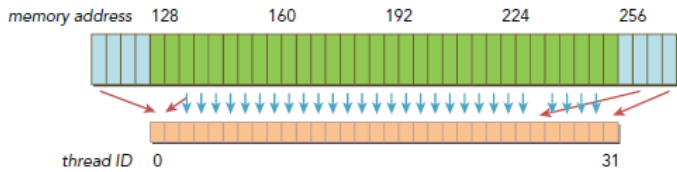
c/c++(16)
CUDA(15)
并行计算(15)
linux(14)
机器学习(7)
深度学习(6)
数据库(4)
sqlserver(4)
windows(4)
error(3)
更多

随笔分类

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



下图则没有遵守连续和对齐原则，有三次传输消耗发生，一次是从偏移地址0开始，一次是从偏移地址256开始，还有一次是从偏移128开始，而这次包含了大部分需要的数据，另外两次则有很多数据并不是需要的，而导致带宽浪费。



一般来讲，我们应该这样优化传输效率：使用最少的传输次数来满足最大的获取内存请求。当然，需要多少传输，多大的吞吐都是跟CC有关的。

Global Memory Reads

在SM中，数据运送是要经过下面三种cache/buffer的，主要依赖于要获取的device Memory种类：

- 1. L1/L2 cache
- 2. Constant cache
- 3. Read-only cache

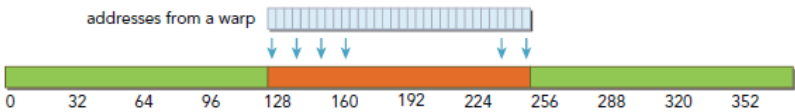
L1/L2是默认路径，另外两条路需要应用显示的说明，一般这样做都是为了提升性能（写CUDA代码的时候，可以先都使用global Memory，然后根据需要慢慢调节，使用一些特殊的内存来提升性能）。Global Memory的load操作是否经过L1cache可以有下面两个因素决定：

- 1. Device compute capability
- 2. Compiler options

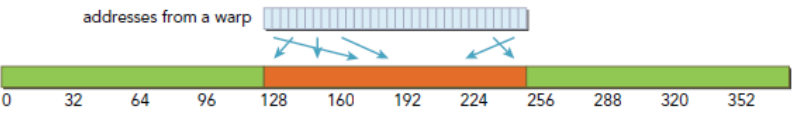
默认情况下，L1是被开启的，-Xptxas -dlcm=cg可以用来禁用L1。L1被禁用后，所有去L1的都直接去L2了。当L2未命中时，就直接去DRAM。所有Memory transaction可能请求一个，两个或者四个segment，每个segment是32 bytes。当然L1也可以被显式的开启-Xptxas -dlcm=ca，此时，所有Memory请求都先走L1，未命中则去L2。在Kepler K10，K20和K20x系列GPU，L1不在用来cache global Memory，L1的唯一用途就是来cache由于register spill放到local Memory的那部分register。

Cache Loads

我们以默认开启L1为例，说明下对齐和连续，下图是理想的情况，连续且对齐，warp中所有thread的Memory请求都落在同一块cache line（128 bytes），只有一次传输消耗，没有任何多余的数据被传输，bus使用效率百分百。



下图是对齐但线程ID和地址不是连续一一对应的情况，不过由于所有数据仍然在一个连续对齐的块中，所有依然没有额外的传输消耗，我们仍然只需要一次128 bytes的传输就能完成。



下图则是非连续未对齐的情况，数据落在了两个128-byte的块中，所以就有两个128-byte的传输消耗，而其中有一半是无效数据，bus使用是百分之五十。



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(5851)
- 2. CUDA ---- Warp解析(4400)
- 3. CUDA ---- GPU架构（Fermi、Kepler）(2607)
- 4. CUDA ---- Memory Model(2221)
- 5. CUDA ---- Stream and Event(2101)

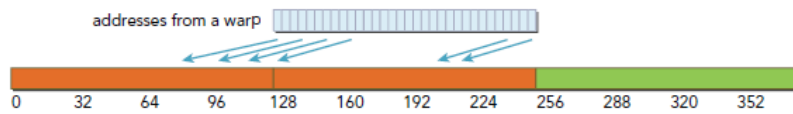
评论排行榜

- 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)

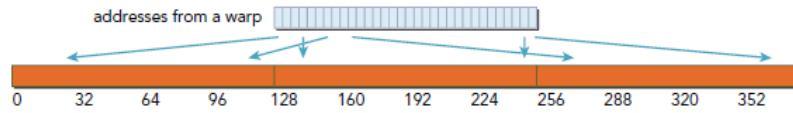
推荐排行榜

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

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



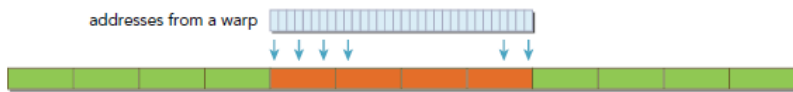
下图是最坏的情况，同样是请求32个4 bytes数据，但是每个地址分布的相当不规律，我们只想要需要的那128 bytes数据，但是，实际上下图这样的分布，却需要 $N \in (0, 32)$ 个cache line，也就是N次数据传输消耗。



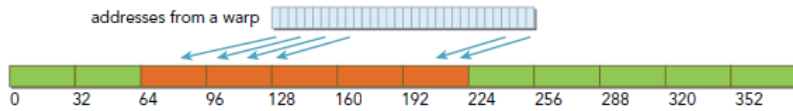
CPU的L1 cache是根据时间和空间局部性做出的优化，但是GPU的L1仅仅被设计成针对空间局部性而不包括时间局部性。频繁的获取L1不会导致某些数据驻留在cache中，只要下次用不到，直接删。

Uncached Loads

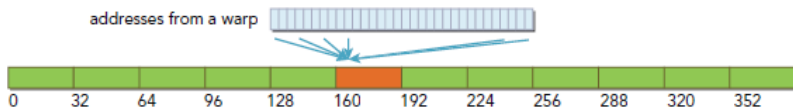
这里就是指不走L1但是还是要走L2，也就是cache line从128-byte变为32-byte了。依然以上文warp 32个thread每个4 bytes请求，总计128 bytes为例，下图是理想的对齐且连续情形，所有的128 bytes都落在四块32 bytes的块中。



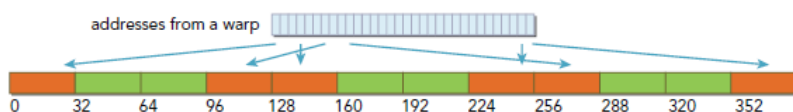
下图请求没有对齐，请求落在了160-byte范围内，bus有效使用率是百分之八十，相对使用L1，性能要好不少。



下图是所有thread都请求同一块数据的情形，bus有效使用率为4bytes/32bytes=12.5%，依然要比L1表现好。



下图是情况最糟糕的，数据非常分散，但是由于所请求的128 bytes落在了多个以32 bytes为单位的segment中，因此无效的数据传输要少的多。



Example of Misaligned Reads

内存获取模式一般都是有应用的实现和算法来决定的，一些情况下，要满足连续内存是非常难的。但是对于对齐来说，是有一些方法来帮助应用实现的。

下面以代码来检验上述知识，kernel中多了一个k索引，是用来配置偏移地址的，通过他就可以配置对齐情况，只有在load两个数组A和B时才会使用k。对C的写操作则继续使用原来的代码，从而保证写操作 保持很好的对齐。

```
__global__ void readOffset(float *A, float *B, float *C, const int n, int offset) {
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int k = i + offset;
    if (k < n) C[i] = A[k] + B[k];
}
```

下面是main代码，offset默认是零：

```
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);
// set up array size
int nElem = 1<<20; // total number of elements to reduce
printf(" with array size %d\n", nElem);
size_t nBytes = nElem * sizeof(float);
// set up offset for summary
int blocksize = 512;
int offset = 0;
if (argc>1) offset = atoi(argv[1]);
if (argc>2) blocksize = atoi(argv[2]);
// execution configuration
dim3 block (blocksize,1);
dim3 grid ((nElem+block.x-1)/block.x,1);
// allocate host memory
float *h_A = (float *)malloc(nBytes);
float *h_B = (float *)malloc(nBytes);
float *hostRef = (float *)malloc(nBytes);
float *gpuRef = (float *)malloc(nBytes);
// initialize host array
initialData(h_A, nElem);
memcpy(h_B, h_A, nBytes);
// summary at host side
sumArraysOnHost(h_A, h_B, hostRef, nElem, offset);
// allocate device memory
float *d_A, *d_B, *d_C;
cudaMalloc((float**)&d_A, nBytes);
cudaMalloc((float**)&d_B, nBytes);
cudaMalloc((float**)&d_C, nBytes);
// copy data from host to device
cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_A, nBytes, cudaMemcpyHostToDevice);
// kernel 1:
double iStart = seconds();
warmup <<< grid, block >>> (d_A, d_B, d_C, nElem, offset);
cudaDeviceSynchronize();
double iElaps = seconds() - iStart;
printf("warmup <<< %4d, %4d >>> offset %4d elapsed %f sec\n",
grid.x, block.x,
offset, iElaps);
iStart = seconds();
readOffset <<< grid, block >>> (d_A, d_B, d_C, nElem, offset);
cudaDeviceSynchronize();
iElaps = seconds() - iStart;
printf("readOffset <<< %4d, %4d >>> offset %4d elapsed %f sec\n",
grid.x, block.x,
offset, iElaps);
// copy kernel result back to host side and check device results
cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost);
checkResult(hostRef, gpuRef, nElem-offset);
// copy kernel result back to host side and check device results
cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost);
checkResult(hostRef, gpuRef, nElem-offset);
// copy kernel result back to host side and check device results
cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost);
checkResult(hostRef, gpuRef, nElem-offset);
// free host and device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
free(h_A);
free(h_B);
// reset device
cudaDeviceReset();

```

```
return EXIT_SUCCESS;
}
```



编译运行：



```
$ nvcc -O3 -arch=sm_20 readSegment.cu -o readSegment
$ ./readSegment 0
readOffset <<< 32768, 512 >>> offset 0 elapsed 0.001820 sec
$ ./readSegment 11
readOffset <<< 32768, 512 >>> offset 11 elapsed 0.001949 sec
$ ./readSegment 128
readOffset <<< 32768, 512 >>> offset 128 elapsed 0.001821 sec
```



当offset=11时，会导致从A和B load数据时不对齐。其运行时间消耗也是最大的，我们可以使用nvcc的gld_efficiency来检验一下：

```
$ nvprof --devices 0 --metrics gld_efficiency ./readSegment 0
$ nvprof --devices 0 --metrics gld_efficiency ./readSegment 11
$ nvprof --devices 0 --metrics gld_efficiency ./readSegment 128
```

输出：

```
Offset 0: gld_efficiency 100.00%
Offset 11: gld_efficiency 49.81%
Offset 128: gld_efficiency 100.00%
```

可以看到offset=11时，效率减半，可以预见其吞吐必然很高，也可以使用gld_transactions来检验：

```
$ nvprof --devices 0 --metrics gld_transactions ./readSegment $OFFSET
```

输出为：

```
Offset 0: gld_transactions 65184
Offset 11: gld_transactions 131039
Offset 128: gld_transactions 65744
```

然后我们使用-Xptxas -dlcm=cg来禁用L1，看一下直接使用L2的表现：

```
$ ./readSegment 0
readOffset <<< 32768, 512 >>> offset 0 elapsed 0.001825 sec
$ ./readSegment 11
readOffset <<< 32768, 512 >>> offset 11 elapsed 0.002309 sec
$ ./readSegment 128
readOffset <<< 32768, 512 >>> offset 128 elapsed 0.001823 sec
```

从该结果看出，未对齐的情况更糟糕了，然后看下gld_efficiency：

```
Offset 0: gld_efficiency 100.00%
Offset 11: gld_efficiency 80.00%
Offset 128: gld_efficiency 100.00%
```

因为L1被禁用后，每次load操作都是以32-byte为单位而不是128，所以无用数据会减少非常多。

这里未对齐反而情况变糟是一种特例，高Occupancy情况下，uncached会帮助提升bus有效使用率，而对于未对齐的情况，无用数据的传输将明显减少。

Read-Only Cache

最开始，read-only cache是用来为texture Memory load服务的，对于CC3.5以上，该cache可以替换L1（Maxwell之后，L1的功能完全就被这个cache取代了）。Read-only cache的单位是32 bytes，一般来讲是比L1要好用得多。

有两种方式来使用read-only cache：

1. Using the function `__ldg`
2. Using a declaration qualifier on the pointer being dereferenced

例如：

```
__global__ void copyKernel(int *out, int *in) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    out[idx] = in[idx];
}
```

改写后：

```
__global__ void copyKernel(int *out, int *in) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    out[idx] = __ldg(&in[idx]);
}
```

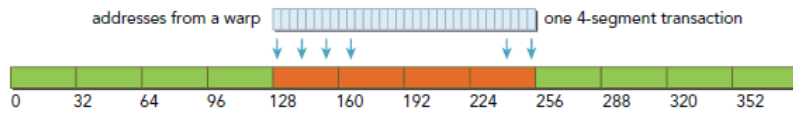
或者使用 `const __restrict__` 来修饰指针。该修饰符帮助nvcc编译器识别non-aliased指针，nvcc会自动使用该non-alias 指针从read-cache读出数据。

```
__global__ void copyKernel(int * __restrict__ out, const int * __restrict__ in) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    out[idx] = in[idx];
}
```

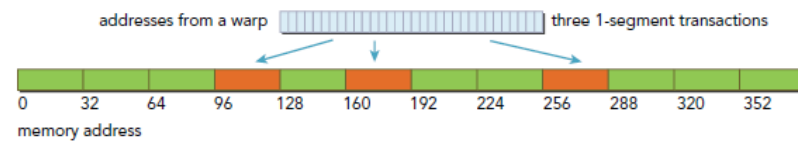
Global Memory Writes

写操作相对要简单的多，L1压根就不使用了。数据只会cache在L2中，所以写操作也是以32bytes为单位的。Memory transaction一次可以是一个、两个或四个segment。例如，如果两个地址落在了同一个128-byte的区域内，但是在不同的两个64-byte对齐的区域，一个四个segment的transaction就会被执行（也就是说，一个单独的4-segment的传输要比两次1-segment的传输性能好）。

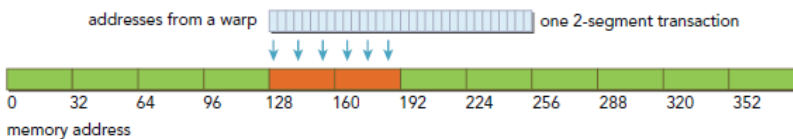
下图是一个理想的情况，连续且对齐，只需要一次4 segment的传输：



下图是离散的情况，会由三次1-segment传输完成。



下图是对齐且地址在一个连续的64-byte范围内的情况，由一次2-segment传输完成：



Example of Misaligned Writes

再次修改代码，load变回使用i，而对C的写则使用k：

```
__global__ void writeOffset(float *A, float *B, float *C, const int n, int offset) {
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int k = i + offset;
    if (k < n) C[k] = A[i] + B[i];
}
```

修改host的计算函数：

```
void sumArraysOnHost(float *A, float *B, float *C, const int n, int offset) {
    for (int idx = offset, k = 0; idx < n; idx++, k++) {
        C[idx] = A[k] + B[k];
    }
}
```

```
}
```

编译运行：

```
$ nvcc -O3 -arch=sm_20 writeSegment.cu -o writeSegment
$ ./writeSegment 0
writeOffset <<< 2048, 512 >>> offset 0 elapsed 0.000134 sec
$ ./writeSegment 11
writeOffset <<< 2048, 512 >>> offset 11 elapsed 0.000184 sec
$ ./writeSegment 128
writeOffset <<< 2048, 512 >>> offset 128 elapsed 0.000134 sec
```

显而易见，Misaligned表现最差，然后查看gld_efficiency：

```
$ nvprof --devices 0 --metrics gld_efficiency --metrics gst_efficiency ./writeSegment $OF
FSET
writeOffset Offset 0: gld_efficiency 100.00%
writeOffset Offset 0: gst_efficiency 100.00%
writeOffset Offset 11: gld_efficiency 100.00%
writeOffset Offset 11: gst_efficiency 80.00%
writeOffset Offset 128: gld_efficiency 100.00%
writeOffset Offset 128: gst_efficiency 100.00%
```

除了offset=11的store外，所有load和store都是百分百。当offset=11时，128-bytes的写请求会被一个4-segment和一个1-segment的传输服务，因此，我们虽然需要写128bytes但是却有160bytes数据被load，从而导致百分之八十的效率。

Array of Structure versus Structure of Arrays

作为C程序员，我们应该熟悉两种组织数据的方式：array of structures (AoS) 和structure of arrays (SoA)。二者的使用是一个有趣的话题，主要是数据排列组织。

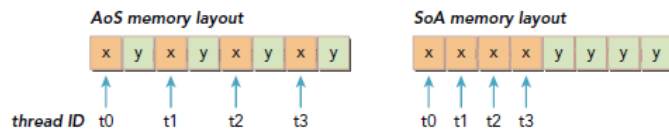
观察下面代码，首先考虑该数据结构集合在使用AoS组织时，是怎样存储的：

```
struct innerStruct {
    float x;
    float y;
};
struct innerStruct myAoS[N]; //每一对x和y的存储，空间上是连续的
```

然后是SoA：

```
struct innerArray {
    float x[N];
    float y[N];
};
struct innerArray moa; //x和y是分别存储的，所有x和y是分别存储在两段不同的连续地址里。
```

下图显示了AoS和SoA在内存中的存储格式，当对x进行操作时，会导致一般的带宽浪费，因为在操作x时，y也会隐式的被load，而SoA的表现就要好得多，因为所有x都是相邻的。



许多并行编程规范里，特别是SIMD-style风格的规范，都更倾向于使用SoA，在CUDA C里，SoA也是非常建议使用的，因为数据已经预先排序连续了。

Example：Simple Math with the AoS Data Layout

```
__global__ void testInnerStruct(innerStruct *data, innerStruct *result, const int n) {
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        innerStruct tmp = data[i];
        tmp.x += 10.f;
        tmp.y += 20.f;
        result[i] = tmp;
    }
}
```

输入长度是1M, #define LEN 1<<20.

初始化数据:

```
void initialInnerStruct(innerStruct *ip, int size) {
    for (int i = 0; i < size; i++) {
        ip[i].x = (float)(rand() & 0xFF) / 100.0f;
        ip[i].y = (float)(rand() & 0xFF) / 100.0f;
    }
    return;
}
```

Main代码:

[View Code](#)

编译运行(Fermi M2070):

```
$ nvcc -O3 -arch=sm_20 simpleMathAoS.cu -o simpleMathAoS
$ ./simpleMathAoS
innerStruct <<< 8192, 128 >>> elapsed 0.000286 sec
```

查看load和store性能:

```
$ nvprof --devices 0 --metrics gld_efficiency,gst_efficiency ./simpleMathAoS
gld_efficiency 50.00%
gst_efficiency 50.00%
```

正如预期那样,都只达到了一般,因为额外那部分消耗都用来load/store 另一个元素了,而这部分不是我们需要的。

Example : Simple Math with the SoA Data Layout

```
__global__ void testInnerArray(InnerArray *data, InnerArray *result, const int n) {
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        float tmpx = data->x[i];
        float tmpy = data->y[i];
        tmpx += 10.f;
        tmpy += 20.f;
        result->x[i] = tmpx;
        result->y[i] = tmpy;
    }
}
```

分配global Memory:

```
int nElem = LEN;
size_t nBytes = sizeof(InnerArray);
```



```
InnerArray *d_A,*d_C;
cudaMalloc((InnerArray **)&d_A, nBytes);
cudaMalloc((InnerArray **)&d_C, nBytes);
```

编译运行：

```
$ nvcc -O3 -arch=sm_20 simpleMathSoA.cu -o simpleSoA
$ ./simpleSoA
innerArray <<< 8192, 128 >>> elapsed 0.000200 sec
```

查看load/store性能：

```
$ nvprof --devices 0 --metrics gld_efficiency,gst_efficiency ./simpleMathSoA
gld_efficiency 100.00%
gst_efficiency 100.00%
```


Performance Tuning

调节device Memory带宽利用性能时，主要是力求达到下面两个目标：


1. **Aligned and Coalesced Memory accesses that reduce wasted bandwidth**
2. **Sufficient concurrent Memory operations to hide Memory latency**

Unrolling Techniques


展开循环可以增加更多的独立的Memory操作，我们在之前博文有详细介绍如何展开loop，考虑之前的redSegment的例子，我们修改下readOffset来使每个thread执行四个独立Memory操作，就像下面那样：




```
__global__ void readOffsetUnroll4(float *A, float *B, float *C,const int n, int offset) {
    unsigned int i = blockIdx.x * blockDim.x * 4 + threadIdx.x;
    unsigned int k = i + offset;
    if (k + 3 * blockDim.x < n) {
        C[i] = A[k]
        C[i + blockDim.x] = A[k + blockDim.x] + B[k + blockDim.x];
        C[i + 2 * blockDim.x] = A[k + 2 * blockDim.x] + B[k + 2 * blockDim.x];
        C[i + 3 * blockDim.x] = A[k + 3 * blockDim.x] + B[k + 3 * blockDim.x];
    }
}
```



编译运行（可能需要使用-Xptxas -dlcm=ca来启用L1）：




```
$ ./readSegmentUnroll 0
warmup <<< 32768, 512 >>> offset 0 elapsed 0.001990 sec
unroll4 <<< 8192, 512 >>> offset 0 elapsed 0.000599 sec
$ ./readSegmentUnroll 11
warmup <<< 32768, 512 >>> offset 11 elapsed 0.002114 sec
unroll4 <<< 8192, 512 >>> offset 11 elapsed 0.000615 sec
$ ./readSegmentUnroll 128
warmup <<< 32768, 512 >>> offset 128 elapsed 0.001989 sec
unroll4 <<< 8192, 512 >>> offset 128 elapsed 0.000598 sec
```



我们看到，unrolling技术会对性能有巨大影响，比地址对齐影响还大。对于这类I/O-bound的kernel，提高内存获取的并行性对性能提升的影响，有更高的优先级。不过，我们应该看到，对齐的test比未对齐的test表现依然要好。

Unrolling并不能影响内存操作的总数目（只是影响并行的操作数目），我们可以查看下相关属性：



```
$ nvprof --devices 0 --metrics gld_efficiency,gst_efficiency ./readSegmentUnroll 11
readOffset gld_efficiency 49.69%
readOffset gst_efficiency 100.00%
readOffsetUnroll4 gld_efficiency 50.79%
readOffsetUnroll4 gst_efficiency 100.00%
```

```
$ nvprof --devices 0 --metrics gld_transactions,gst_transactions
./readSegmentUnroll 11
readOffset gld_transactions 132384
readOffset gst_transactions 32928
readOffsetUnroll4 gld_transactions 33152
readOffsetUnroll4 gst_transactions 8064
```

Exposing More Parallelism

这方面就是调整grid和block的配置，下面是加上unrolling后的结果：

```
$ ./readSegmentUnroll 0 1024 22
unroll14 <<< 1024, 1024 >>> offset 0 elapsed 0.000169 sec
$ ./readSegmentUnroll 0 512 22
unroll14 <<< 2048, 512 >>> offset 0 elapsed 0.000159 sec
$ ./readSegmentUnroll 0 256 22
unroll14 <<< 4096, 256 >>> offset 0 elapsed 0.000157 sec
$ ./readSegmentUnroll 0 128 22
unroll14 <<< 8192, 128 >>> offset 0 elapsed 0.000158 sec
```

表现最好的是block配置256 thread的kernel，虽然128thread会增加并行性，但是依然比256少那么一点点性能，这个主要是CC版本对应的资源限制决定的，以本代码为例，Fermi每个SM最多有8个block，每个SM能够并行的warp是48个，当使用128个thread（per block）时，每个block中有4个warp，因为每个SM最多8个block能够同时运行，因此该kernel每个SM最多只能有32个warp，还有16个warp的计算性能没用上，所以性能差了就，可以使用Occupancy来验证下。

参考书：《professional cuda c programming》

分类: [c/c++,CUDA](#)

标签: [c/c++](#), [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](#)

2

0

« 上一篇: [MachineLearning Exercise 4 : Neural Networks Learning](#)

» 下一篇: [CUDA ---- CUDA库简介](#)

posted @ 2015-06-13 15:21 苹果妖 阅读(1116) 评论(0) 编辑 收藏

[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



美团云
MEITUAN CLOUD SERVICES

GPU
云主机

5折起

M60 原价1660 / 月 现价830 / 月

P40 原价3400 / 月 现价2100 / 月

查看详情

最新IT新闻:

- 阿里发布首款防诈骗智能机器人：92%识别率全球最高
 - 苹果第一批AR应用上线 我们帮你试了哪个最好玩
 - 微信扫一扫开发票 可领现金红包
 - 35颗卫星覆盖全球！中国启动北斗三号：要啥GPS？
 - 中国电信5G试验6城市公布：兰州、成都、深圳、雄安、苏州、上海
- » 更多新闻...



JIGUANG | 极光

极光统计

多维洞察用户
增长运营指标

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...