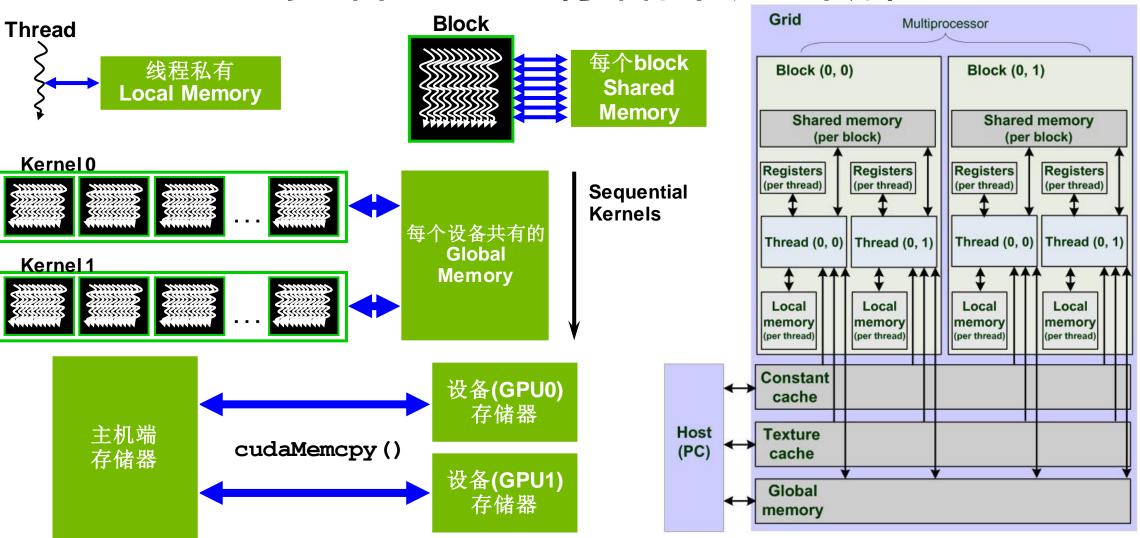




利用共享存储单元优化应用

- 共享存储单元详解
- 共享内存的Bank conflict
- 利用共享存储单元进行矩阵相乘

多种CUDA存储单元详解



多种CUDA存储单元详解

Access speed

- Register file
- Shared Memory
- Constant Memory
- Texture Memory
- Local Memory and Global Memory

多种CUDA存储单元详解

Shared Memory:

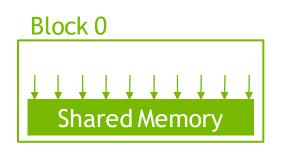
The only two types of memory that actually reside on the GPU chip are register and shared memory.

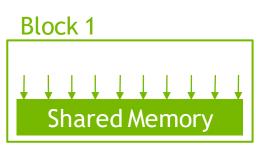
所以,Shared Memory是目前最快的可以让多个线程沟通的地方。

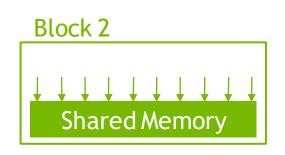
那么,就有可能会出现同时有很多线程访问Shared Memory上的数据。

为了克服这个同时访问的瓶颈,Shared Memory被分成32个逻辑块(banks)

Successive sections of memory are assigned to successive banks

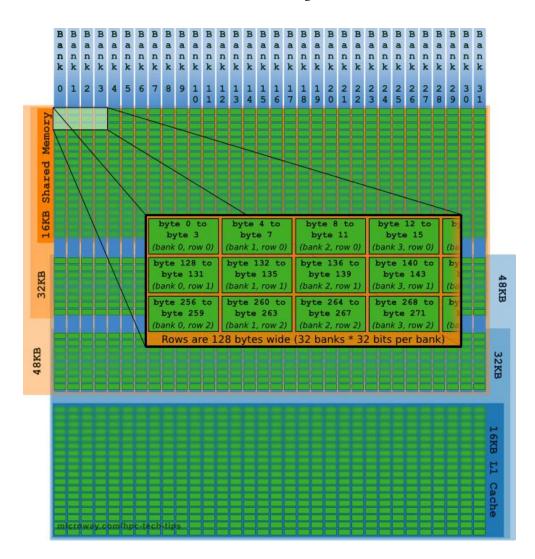






Shared Memory:

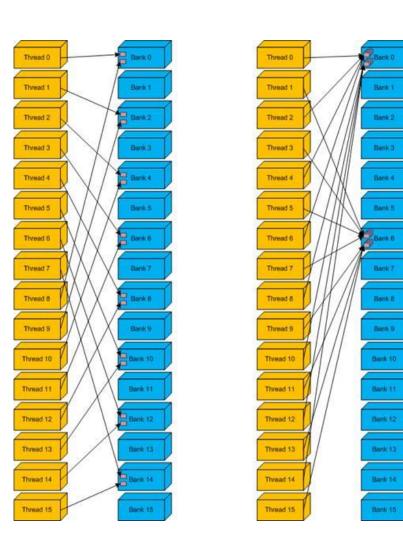
- Shared Memory可以被设置成16KB,32KB, 48KB...剩下的给L1缓存
- 带宽可以使32bit or 64 bit
- 很多线程访问存储器
- 因此,存储器被划分为 banks
- 连续的32-bit 访存被分配到连续的 banks
- 每个bank 每个周期可以响应一个地址
- 如果有多个bank的话可以同时响应更多地址申请





Bank conflict

- 1. 同常量内存一样, 当一个 warp 中的所有线程访问同一地址的共享内存时, 会触发一个广播(broadcast)机制到warp 中所有线程, 这是最高效的。
- 2. 如果同一个 half-warp/warp 中的线程访问同一个 bank 中的不同地址时将发生 bank conflict。
- 3. 每个 bank 除了能广播(broadcast)还可以多播(mutilcast)(计算能力 >= 2.0), 也就是说, 如果一个warp 中的多个线程访问同一个 bank 的同一个地址时(其他线程也没有访问同一个bank 的不同地址)不会发生bank conflict。
- 4. 即使同一个 warp 中的线程 随机的访问不同的 bank, 只要没有访问同一个 bank 的不同地址就不会发生 bank conflict。



Shared memory 跟 registers 一样快 ,如果没有bank冲突的话 快速情况:

- half-warp/warp 内所有线程访问不同 banks, 没有冲突
- half-warp/warp 内所有线程读取同一地址,没有冲突(广播)

慢速情况:

- Bank Conflict: half-warp/warp 内多个线程访问同一个bank
- 访存必须串行化
- 代价 = 多个线程同时访问同一个bank 的线程数最大值

如何避免 Bank conflict

一个线程块

warp 0	0	1	2	3		30	31
warp 1	32	33	34				63
	64	65					95
	96						
warp 31	992						1023

如何避免 Bank conflict

共享内存

	bank 0	bank 1					bank 31
warp 0	0	1	2	3		30	31
warp 1	32	33	34				63
	64	65					95
	96						
warp 31	992						1023

此时是没有 bank conflict

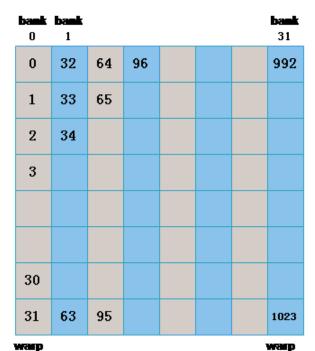
```
int x_id = blockDim.x * blockIdx.x + threadIdx.x; // 列坐标
int y_id = blockDim.y * blockIdx.y + threadIdx.y; //行坐标
int index = y_id * col + x_id;

__shared__ float sData[BLOCKSIZE][BLOCKSIZE];

if (x_id < col && y_id < row)
{
    sData[threadIdx.y][threadIdx.x] = matrix[index];
    __syncthreads();
    matrixTest[index] = sData[threadIdx.y][threadIdx.x];
}
```

如何避免 Bank conflict

共享内存



31

此时是有 bank conflict

```
int x_id = blockDim.x * blockIdx.x + threadIdx.x; //列坐标
int y_id = blockDim.y * blockIdx.y + threadIdx.y; //行坐标
int index = y_id * col + x_id;

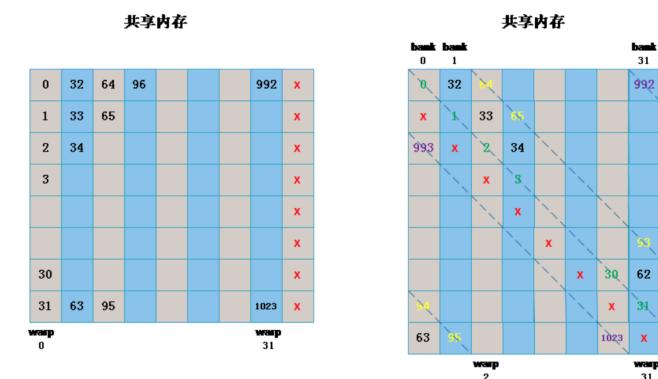
__shared__ float sData[BLOCKSIZE][BLOCKSIZE];

if (x_id < col && y_id < row)
{
    sData[threadIdx.x][threadIdx.y] = matrix[index];
    __syncthreads();
    matrixTest[index] = sData[threadIdx.x][threadIdx.y];
}
```

如何避免 Bank conflict

Memory Padding

warp



如何避免 Bank conflict

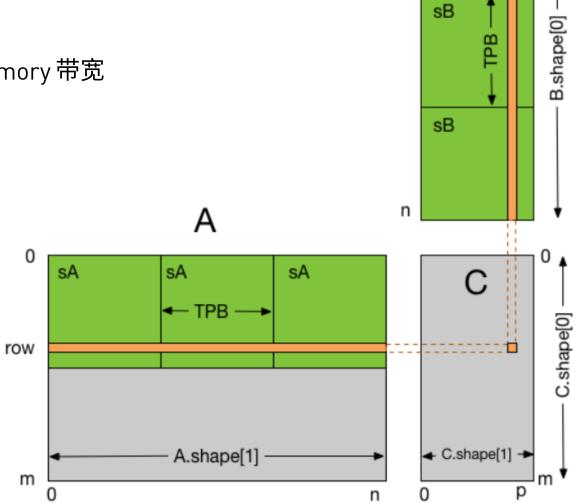
Memory Padding

```
int x_id = blockDim.x * blockIdx.x + threadIdx.x; // 列坐标
int y_id = blockDim.y * blockIdx.y + threadIdx.y; // 行坐标
int index = y_id * col + x_id;

__shared__ float sData[BLOCKSIZE][BLOCKSIZE+1];

if (x_id < col && y_id < row)
{
    sData[threadIdx.x][threadIdx.y] = matrix[index];
    __syncthreads();
    matrixTest[index] = sData[threadIdx.x][threadIdx.y];
}
```

- 每个输入元素被Width 个线程读取
- 使用 shared memory 来减少 global memory 带宽 需求



col p

В

0 sB

假设: M[16][16] * N[16][16] = P[16][16]

Matrix N[16][16]

Matrix M[16][16]



我们要求一个结果矩阵P的维度是16*16 而P矩阵在实际存储单元中,是保存在连续地址的一 维存储空间中,如下图所示

Matrix P

0	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	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

我们申请线程,设置:

gridDim(2,2) blockDim(8,8) 保证每一个线程,负责求出P矩阵中一个位置的值 如下图所示,每个小格子代表一个thread,同一个颜色覆盖的区域代表一个Block。 **坐标示意:(threadIdx.y, thread.x)**

Grid

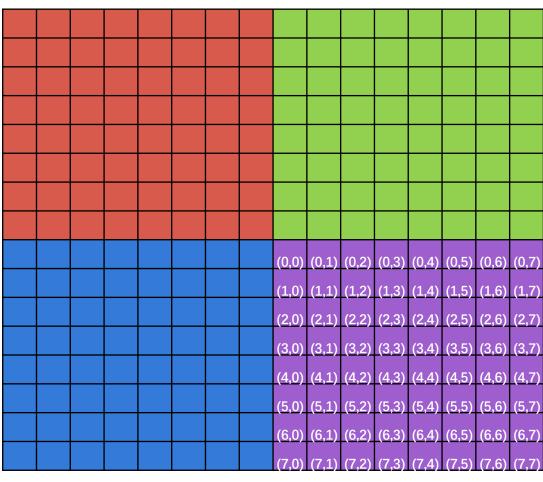
(0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (0,6) (0,7) (0,0) (0,1)(1.0) (1.1) (1.2) (1.3) (1.4) (1.5) (1.6) (1.7) (1.0) (2.0) (2.1) (2.2) (2.3) (2.4) (2.5) (2.6) (2.7) (2.0) (2.1) (2.2) (2.3) (2.4) (2.5) (2.6) (4,0) (4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7) (4,0) (4,1) (5,0) (5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,7) (5,0) (5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,7) (6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6) (6,7) (6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6)(0.0) (0.1) (0.2) (0.3) (0.4) (0.5) (0.6) (0.7) (0.0) (0.1) (0.2) (0.3) (0.4) (0.5) (0.6) (0.7)(1,0) (1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7) (1,0) (1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7)(2,0) (2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7) (2,0) (2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7)(3,0) (3,1) (3,2) (3,3) (3,4) (3,5) (3,6) (3,7) (3,0) (3,1) (3,2) (3,3) (3,4) (3,5) (3,6) (3,7)(4,0) (4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7) (4,0) (4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7)(5,0) (5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,6) (5,7) (5,0) (5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,7)(6,0) (6,1) (6,2) (6,3) (6,3) (6,4) (6,5) (6,6) (6,7) (6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6) (6,7)

那么,下面右图中紫色区域覆盖的block就会负责求出P矩阵中相对应的位置的值(下面左图中紫色区域覆盖的位置)

Matrix P

| 114 | 115 | 116 | 117 | 118 | 119 | 132 | 133 | 134 | 135 162 | 163 | 164 | 166 | 167 178 | 179 | 180 | 181 209 | 210 | 211 | 212 | 213 | 214 | 215 | 216 233 234 | 228 | 229 | 230 | 231

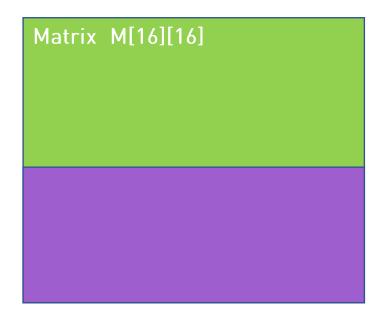
Grid

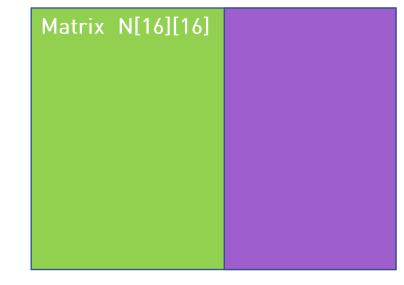


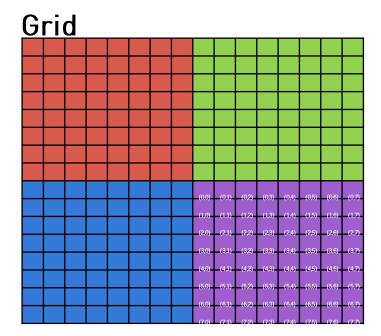
那么,下图Grid中紫色区域的block,将会读取M矩阵中紫色区域的行和N矩阵中紫色区域的列。

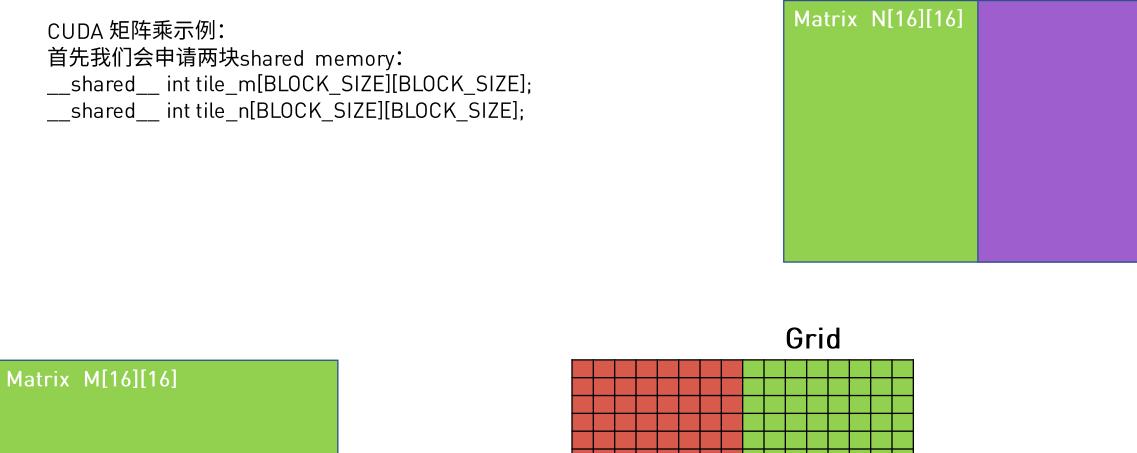
因为,M中的行 和 N中的列将会被多次读取,所以我们就将M中的紫色区域覆盖的行和N中紫色区域覆盖的列存入Shared Memory中来提高访存效率。

因为空间优先,所以要滑动小块,重复利用shared memory的空间









tile_n[8][8]



```
Matrix N[16][16]
   CUDA 矩阵乘示例:
   接下来,Grid中紫色区域覆盖的block中的线程,从₩矩阵中读
   取数值,并放到tile_m中: 此时sub = 0 且 sub < 2(gridDim.x)
        idx = row * n + sub * BLOCK SIZE + threadIdx.x;
        if(row<n && (sub * BLOCK_SIZE + threadIdx.x)<n )
          tile_m[threadIdx.y][threadIdx.x] = d_m[idx];
        else{
          tile_m[threadIdx.y][threadIdx.x] = 0;
                                                    Grid
Matrix M[16][16]
                                     tile_m[8][8]
                                                                                             tile_n[8][8]
```

```
CUDA 矩阵乘示例:
```

```
接下来,Grid中紫色区域覆盖的block中的线程,从M矩阵中读取数值,并放到tile_m中:此时sub = 0且 sub < 2(gridDim.x) idx = row * n + sub * BLOCK_SIZE + threadIdx.x; if(row<n && (sub * BLOCK_SIZE + threadIdx.x)<n ) { tile_m[threadIdx.y][threadIdx.x] = d_m[idx];
```

tile_m[threadIdx.y][threadIdx.x] = 0;

其中,最难理解的是这个idx:

```
idx = row * n + sub * BLOCK_SIZE + threadIdx.x;
```

我们以thread(3,4)为例,该线程的索引值:

threadldx.x = 3threadldx.y = 4blockldx.x = 1

blockldx.y = 1

此时,我们sub的迭代步骤为0,即sub=0

row * n: 该线程所在行之前 所有行包含的数据

else{

row * n

sub* BLOCK_SIZE + threadldxx: 该线程要读的数据 在它所在的行中的 位置

```
CUDA 矩阵乘示例:
                                                                Matrix N[16][16]
   接下来,Grid中紫色区域覆盖的block中的线程,从N矩阵中读取
    数值,并放到tile_n中:此时sub = 0且 sub < 2(gridDim.x)
        idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;
        if(col<n && (sub * BLOCK_SIZE + threadIdx.y)<n )
          tile_n[threadIdx.y][threadIdx.x] = d_n[idx];
        else{
          tile_n[threadIdx.y][threadIdx.x] = 0;
                                                     Grid
Matrix M[16][16]
                                     tile_m[8][8]
                                                                                              tile_n[8][8]
```

```
(sub * BLOCK_SIZE + threadIdx.y) * n:
CUDA 矩阵乘示例:
                                                           Matrix N[16][16]
接下来,Grid中紫色区域覆盖的block中的线程,从N矩阵中读取
数值,并放到tile n中:此时sub = 0且 sub < 2(gridDim.x)
    idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;
                                                col :表示该线程要读取的数据在所在
行的位置
    if(col<n && (sub * BLOCK_SIZE + threadIdx.y)<n )
      tile_n[threadIdx.y][threadIdx.x] = d_n[idx];
                                           ldx: 该线程要读取的数据在N中的位置
    else{
      tile n[threadIdx.y][threadIdx.x] = 0;
                                                Grid
  其中,最难理解的还是这个idx:
  idx = (sub * BLOCK SIZE + threadIdx.y) * n + col;
  我们还是以thread(3,4)为例,该线程的索引值:
  threadldx.x = 3
  threadIdx.y = 4
   blockldx.x = 1
   blockldx.y = 1
                                                                                         tile_n[8][8]
   此时,我们sub的迭代步骤为0,即sub=0
```

同步完所有的线程之后,我们紫色区域覆盖的线程块中的 线程对已经读取到的两个tile中的矩阵进行相乘:

```
__syncthreads();
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    tmp += tile_m[threadIdx.y][k] * tile_n[k][threadIdx.x];
}
__syncthreads();</pre>
```

此时还以thread(3,4)为例:

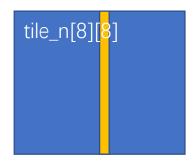
它会将tile_m中index为3的行与tile_n中index为4的列相乘

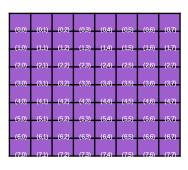
将_{Sub}迭代步骤为₀时计算得到的记过存储在 tmp中

然后同步一下,让所有的线程都完成操作。 以便在下一步中向shared memory中写入新 的数值时不会影响这一步的计算。



此时sub = 0且 sub < 2(gridDim.x)





位置

```
接下来,Grid中紫色区域覆盖的block中的线程,从₩矩阵中读取
                                                           其中,最难理解的是这个idx:
   数值,并放到tile m中: 此时sub = 1且 sub < 2(gridDim.x)
                                                           idx = row * n + sub * BLOCK SIZE + threadIdx.x;
       idx = row * n + sub * BLOCK SIZE + threadIdx.x;
       if(row<n && (sub * BLOCK SIZE + threadIdx.x)<n )
                                                           我们以thread(3,4)为例,该线程的索引值:
                                                           threadIdx.x = 3
         tile_a[threadIdx.y][threadIdx.x] = d_m[idx];
                                                           threadIdx.y = 4
                                                           blockldx.x = 1
       else{
                                                           blockldx.y = 1
         tile a[threadIdx.y][threadIdx.x] = 0;
                                                           此时,我们sub的迭代步骤为1,即sub=1
                                                                     Grid
 row * n:
                    Matrix M[16][16]
                                     ldx: 该线程要读取的数据在M
 该线程所在行之前
                                             中的位置
 所有行包含的数据
      row * n
                          注意这里
                           的变化
                                                      tile_m[8][8]
sub * BLOCK SIZE + threadIdx.x:
该线程要读的数据
在它所在的行中的
                    sub * BLOCK SIZE + threadIdx.x
```

```
接下来,Grid中紫色区域覆盖的block中的线程,从N矩阵中读取数值,并放到tile_n中:此时sub = 1且 sub < 2(gridDim.x) idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col; if(col<n && (sub * BLOCK_SIZE + threadIdx.y)<n ) { tile_n[threadIdx.y][threadIdx.x] = d_n[idx]; } else{
```

ldx: 该线程要读取的数据在N中的位置

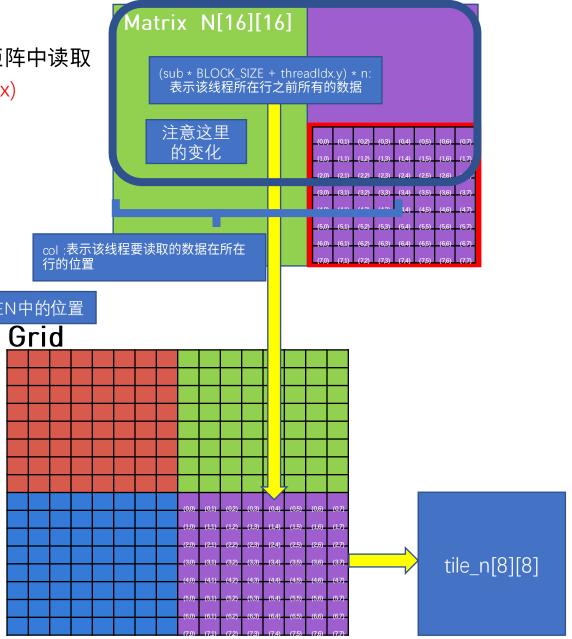
其中,最难理解的还是这个idx:

 $idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;$

我们还是以thread(3,4)为例,该线程的索引值:

tile n[threadIdx.y][threadIdx.x] = 0;

threadIdx.x = 3
threadIdx.y = 4
blockIdx.x = 1
blockIdx.y = 1
此时,我们sub的迭代步骤为1,即sub=1



同步完所有的线程之后,我们紫色区域覆盖的线程块中的 线程对已经读取到的两个tile中的矩阵进行相乘:

```
_syncthreads();
for (int k = 0; k < BLOCK_SIZE; ++k)
{
   tmp += tile_m[threadIdx.y][k] * tile_n[k][threadIdx.x];
}
_syncthreads();</pre>
```

此时还以thread(3,4)为例:

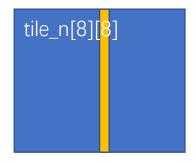
它仍然会将tile_m中index为3的行与tile_n中index为4的列相乘,但是此时tile_m和tile_n中的数据已经发生变化

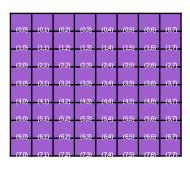
做完累加之后,将计算结果与上一个sub迭 代步骤中的tmp相加

此时,thread(3,4)已经完成它的任务,迭代完所有的步骤。



此时sub = 0且 sub < 2(gridDim.x)





CUDA 矩阵乘示例: 当所有线程完成任务之后,每个线程将计算结果,写入结果矩阵P中对应的位置。

Matrix P

IA	al	IIX	Γ									G	rıd	J															
0	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	31														
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47														
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63														
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79														
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95														
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111														
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127														
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143							(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159							(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175							(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191							(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
192	193	194	195	196	197	198	199	200	201	202	203	4	205	206	207							(4,0)	(4,1)	(4,2)	(4,3)	(4)	(4,5)	(4,6)	(4,7)
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	1						(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239							(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255							(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

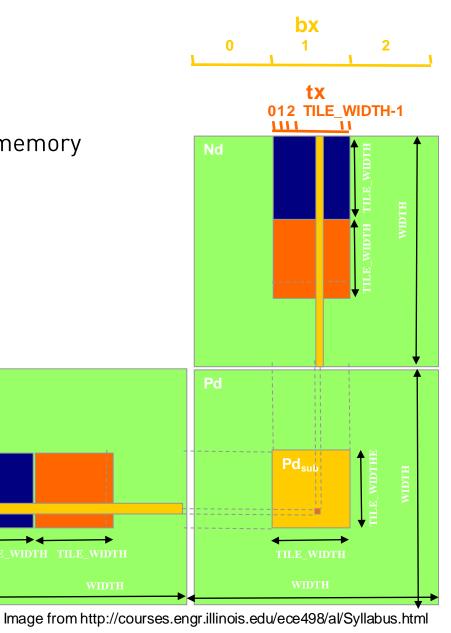
if(row < n && col < n)

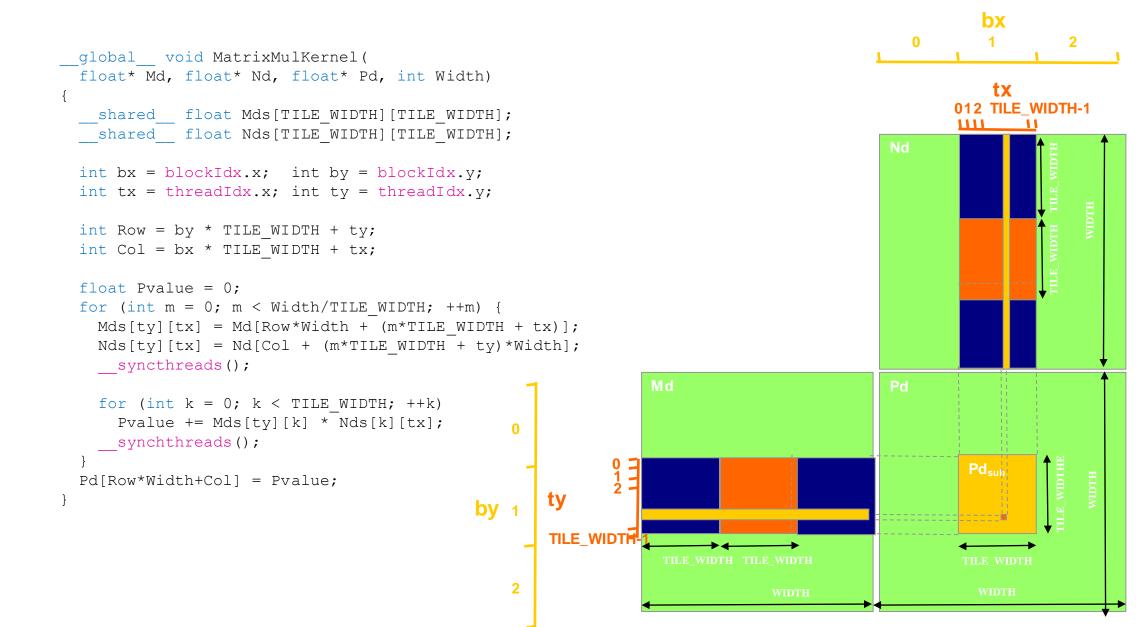
 $d_result[row * n + col] = tmp;$

每次利用shared memory先读取一小块矩阵 每个线程 读入瓦片内 Md 和 Nd 的一个元素存入 shared memory

TILE WIDTH

把 kernel 拆分成多个阶段 每个阶段用 Md 和 Nd 的子集累加Pd 每个阶段有很好的数据局部性





Shared memory 大小的选择

超出shared memory 极限
 P100: 48KB/SM 并且 8 blocks / SM
 6KB/block
 3KB给Nds, 3KB给Mds(3*16*16*4)

所以我们设置多大的tile最合适? 如何更好地利用硬件?

更多资源:

https://developer.nvidia-china.com





https://www.nvidia.cn/developer/comm
unity-training/

