CUDA ON ARM PLATFORM—存储单元（一）&矩阵乘法

NVIDIA企业级开发者社区 何琨

AGENDA

CUDA并行计算基础

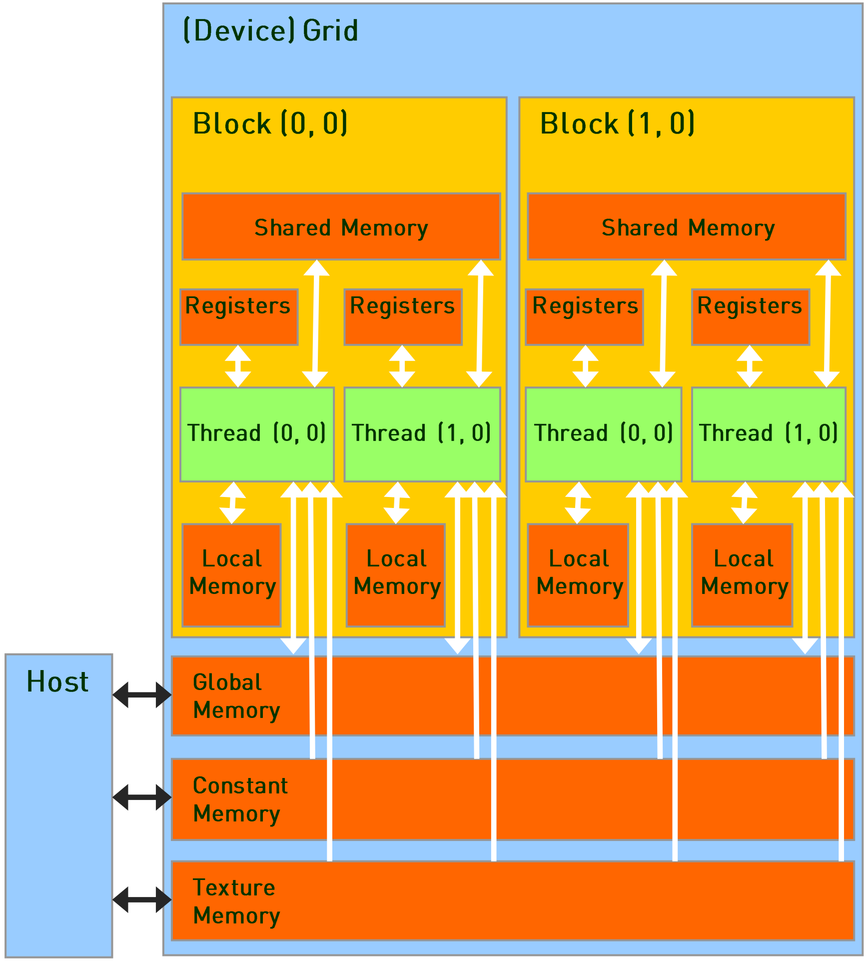
- GPU的存储单元
- GPU存储单元的分配与释放
- 数据的传输
- 矩阵相乘

GPU的存储单元

Each thread can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory
- Read only per-grid texture memory

- The host
  global constant texture



MEMORY ALLOCATION / RELEASE

CPU memory:

- malloc()
- memset()
- free()

GPU memory:

- cudaMalloc()
- cudaMemset()
- cudaFree()

MEMORY COPY BETWEEN CPU AND GPU

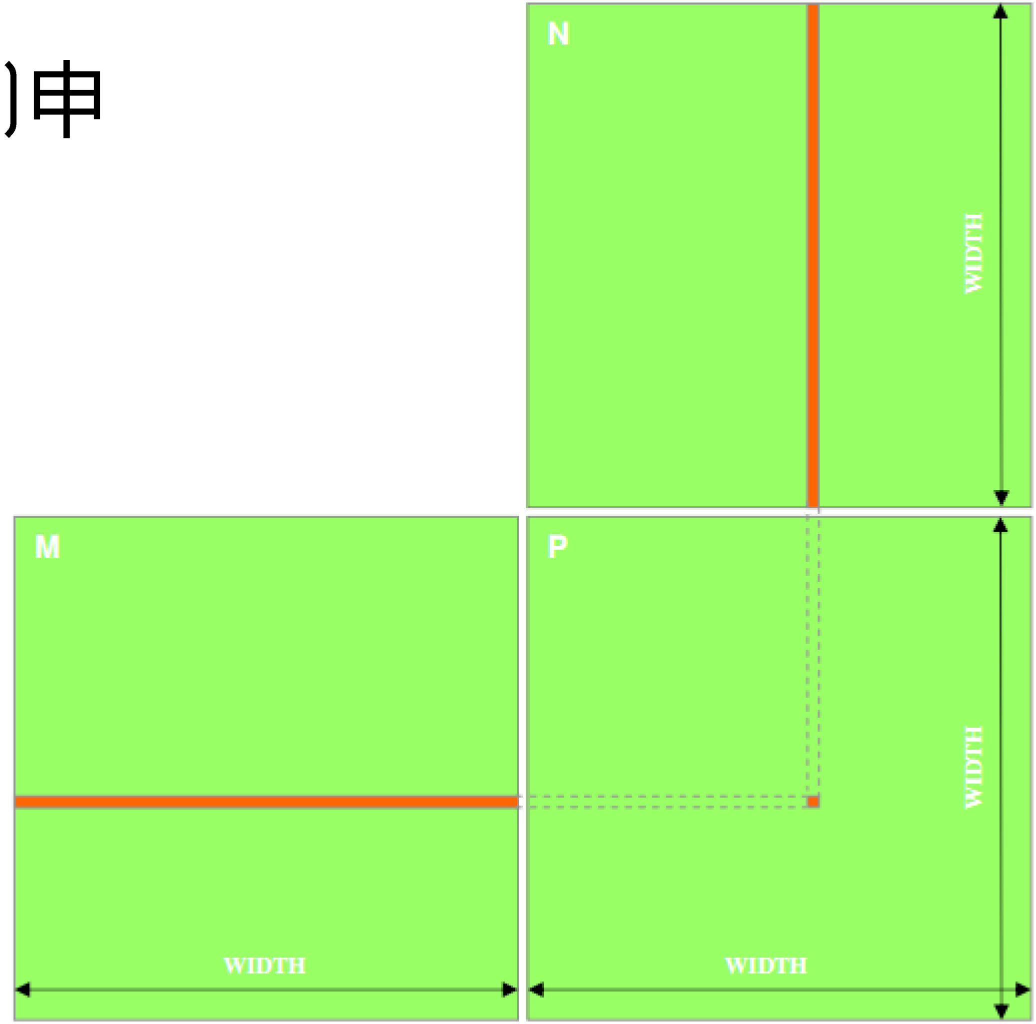
- cudaMemcpy(void *dst, const void *src, size_t count, cudaMemcpyKind kind)
  - dst: destination memory address
  - src: source memory address
  - count: size in bytes to copy
  - kind: direction of the copy
- cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
  - cudaMemcpyHostToHost

GPU的存储单元的分配与释放

当我们要为一个方阵M(m*m)申请GPU的存储单元时：

```
cudaMalloc((void **) &d_m, sizeof(int)*m*m);
```

d_m:指向存储在Device端数据的地址的指针
sizeof(int)*m*m：存储在Device端空间的大小

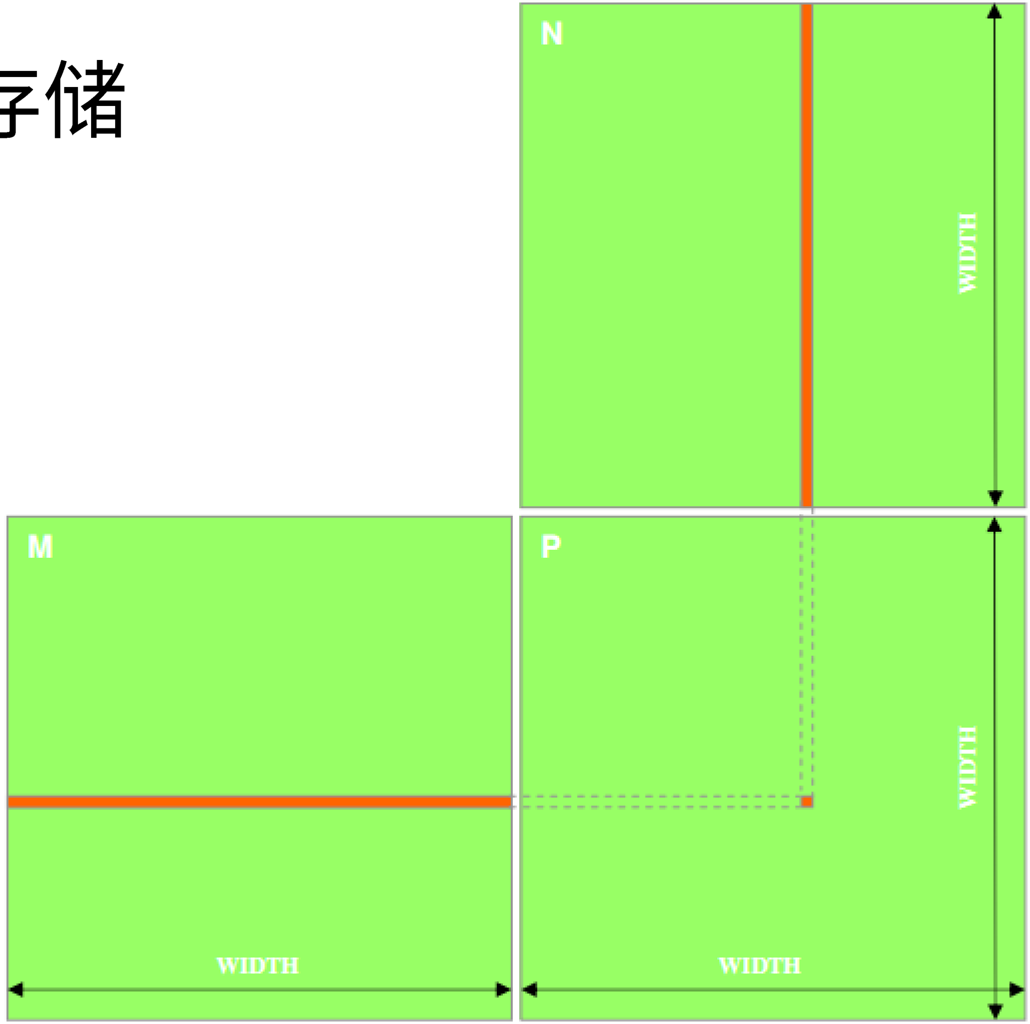


GPU的存储单元的分配与释放

当我们要释放申请的GPU的存储单元时：

```
cudaFree(d_m);
```

d_m:指向存储在Device端数据的地址的指针

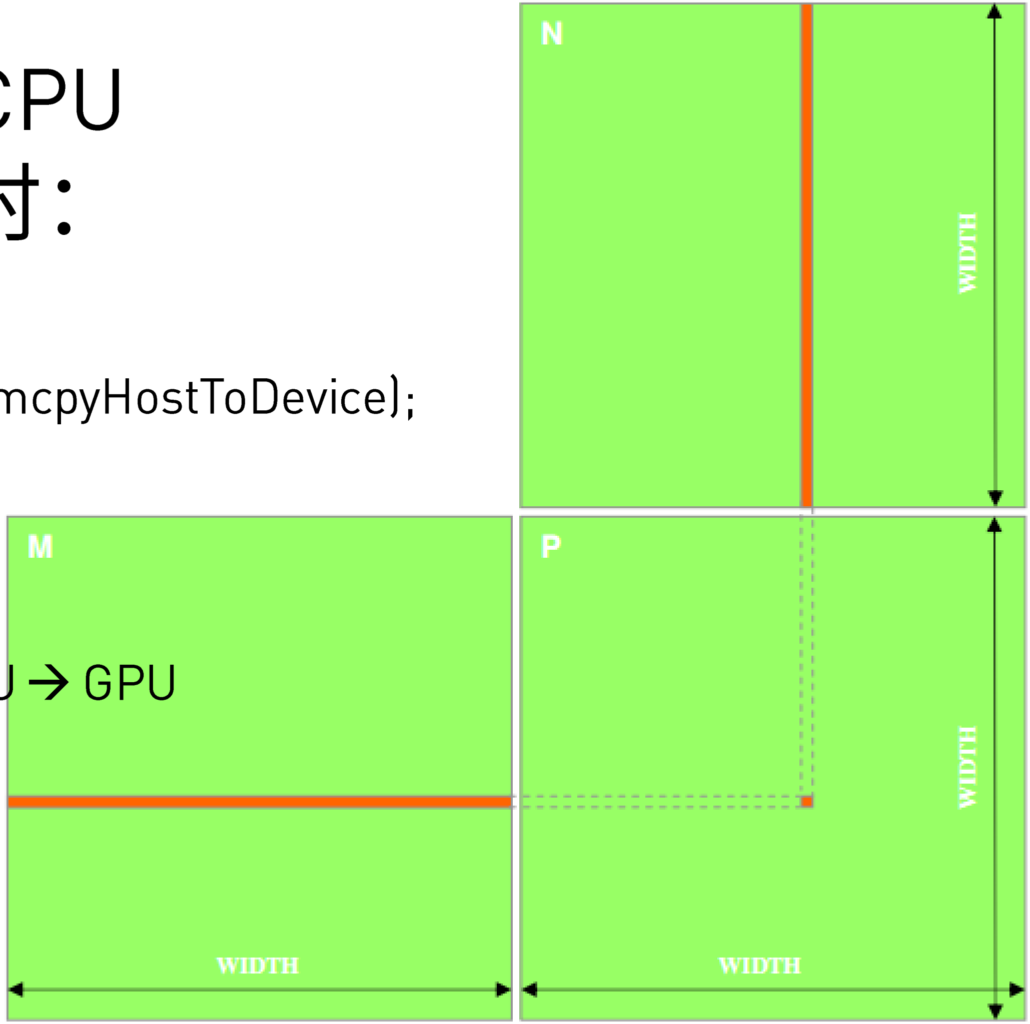


GPU的存储单元的分配与释放

当我们要将准备好的数据从CPU内存传输到GPU的存储单元时：

```
cudaMemcpy(d_m, h_m, sizeof(int)*m*m, cudaMemcpyHostToDevice);
```

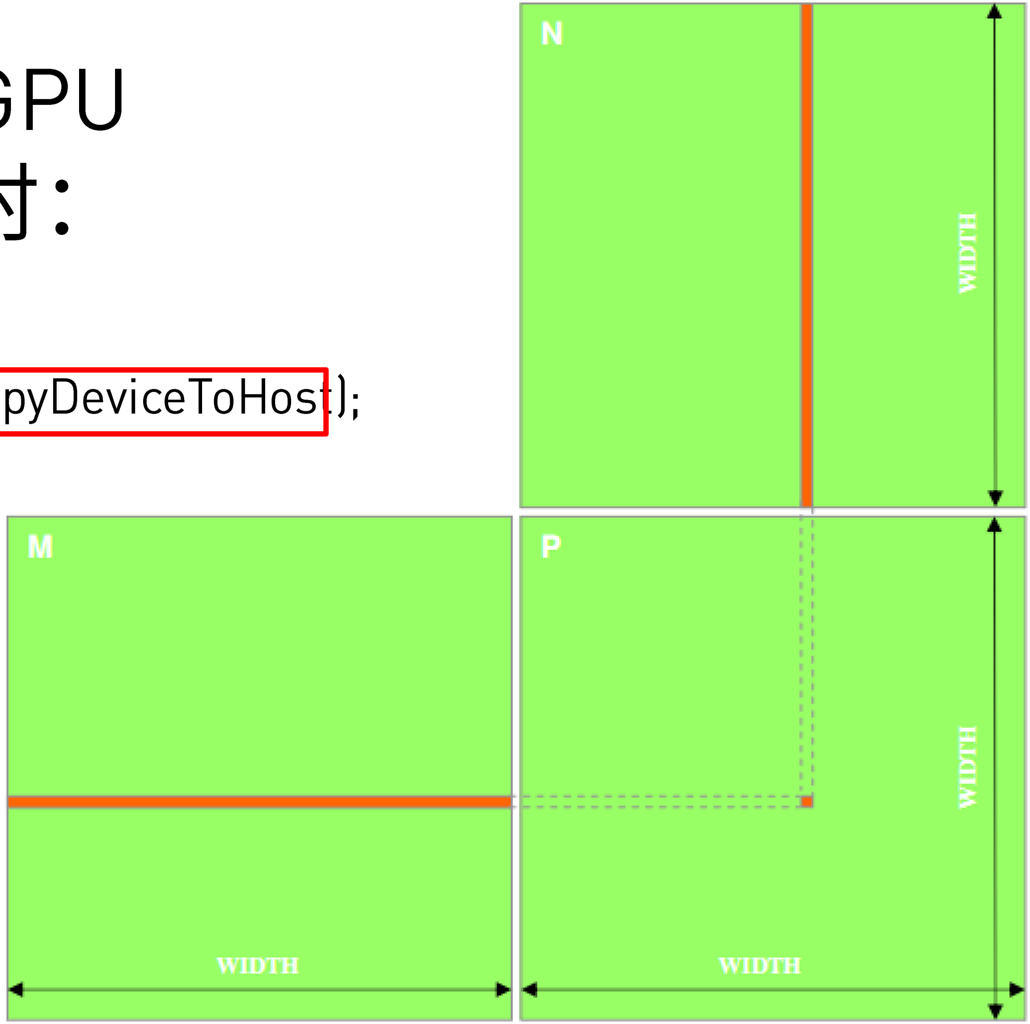
d_m: 传输的目的地，GPU存储单元
h_m: 数据的源地址，CPU存储单元
sizeof(int)*m*m: 数据传输的大小
cudaMemcpyHostToDevice：数据传输的方向，CPU → GPU



GPU的存储单元的分配与释放

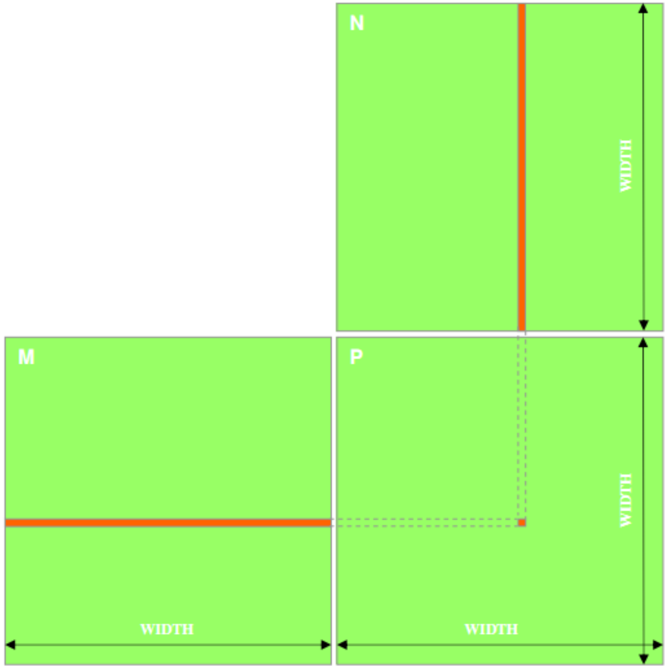
当我们要将准备好的数据从GPU存储单元传输到CPU的内存时：

cudaMemcpy(h_c, d_c, sizeof(int)*m*k, cudaMemcpyDeviceToHost);



矩阵相乘样例

```
void cpu_matrix_mult(int *h_m, int *h_n, int *h_result,
int m, int n, int k) {
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            int tmp = 0.0;
            for (int h = 0; h < n; ++h)
            {
                tmp += h_m[i * n + h] * h_n[h * k + j];
            }
            h_result[i * k + j] = tmp;
        }
    }
}
```

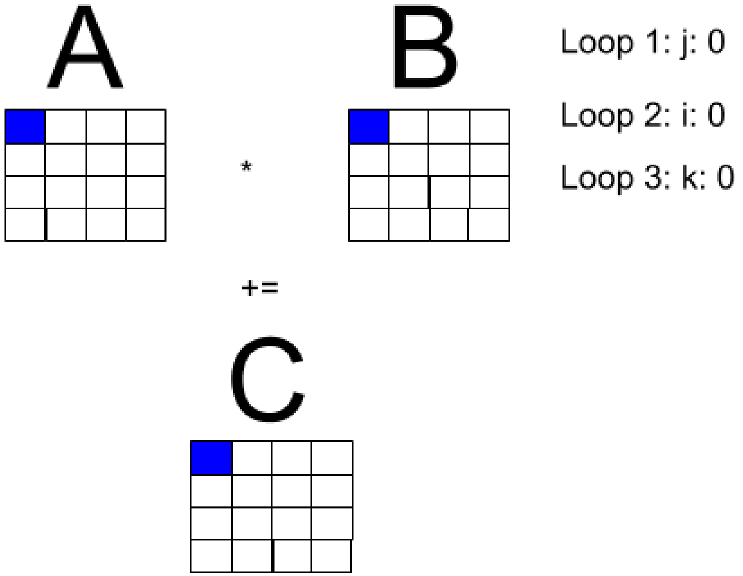


P = M * N
假定 M and N 是方阵

$$
\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}\times b_{11}+a_{12}\times b_{21}+a_{13}\times b_{31} & a_{11}\times b_{12}+a_{12}\times b_{22}+a_{13}\times b_{32} & a_{11}\times b_{13}+a_{12}\times b_{23}+a_{13}\times b_{33} \\ a_{21}\times b_{11}+a_{22}\times b_{21}+a_{23}\times b_{31} & a_{21}\times b_{12}+a_{22}\times b_{22}+a_{23}\times b_{32} & a_{21}\times b_{13}+a_{22}\times b_{23}+a_{23}\times b_{33} \\ a_{31}\times b_{11}+a_{32}\times b_{21}+a_{33}\times b_{31} & a_{31}\times b_{12}+a_{32}\times b_{22}+a_{33}\times b_{32} & a_{31}\times b_{13}+a_{32}\times b_{23}+a_{33}\times b_{33} \end{bmatrix}
$$

矩阵相乘样例

```
void cpu_matrix_mult(int *h_m, int *h_n, int *h_result,
int m, int n, int k) {
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            int tmp = 0.0;
            for (int h = 0; h < n; ++h)
            {
                tmp += h_m[i * n + h] * h_n[h * k + j];
            }
            h_result[i * k + j] = tmp;
        }
    }
}
```



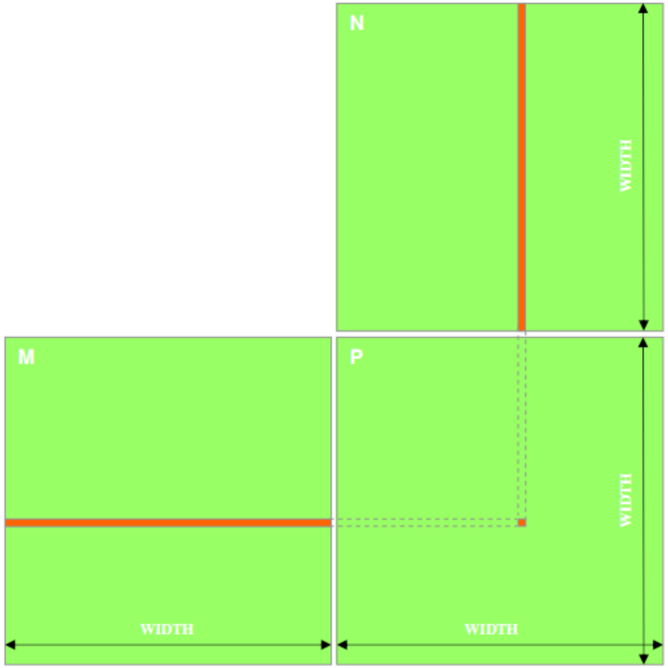
P = M * N
假定 M and N 是方阵

如果，M=N=1000
那么，我们要做1000*1000*1000次： tmp += h_m[i * n + h] * h_n[h * k + j];

矩阵相乘样例

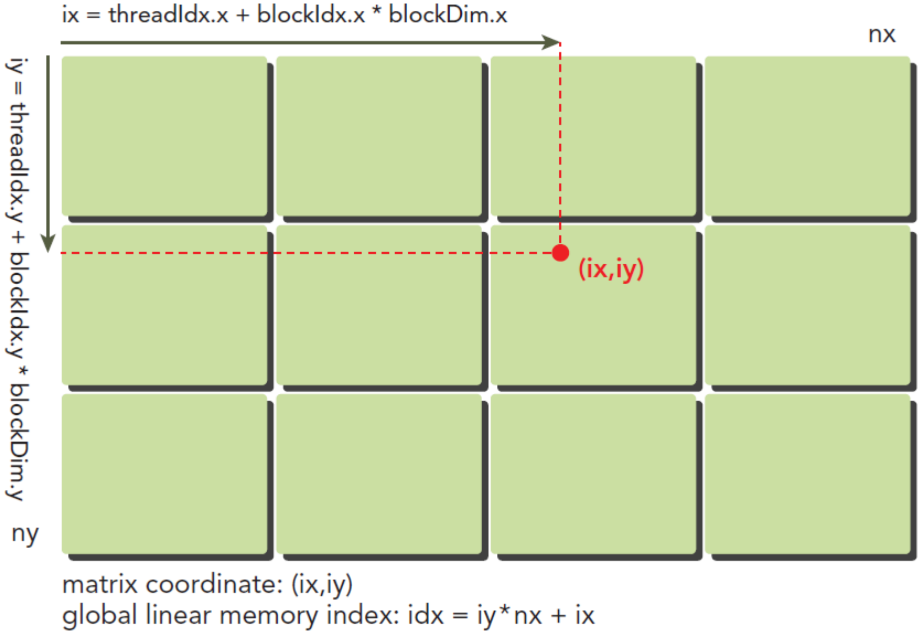
```
void cpu_matrix_mult(int *h_m, int *h_n, int *h_result,
int m, int n, int k) {
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            int tmp = 0.0;
            for (int h = 0; h < n; ++h)
            {
                tmp += h_m[i * n + h] * h_n[h * k + j];
            }
            h_result[i * k + j] = tmp;
        }
    }
}
```

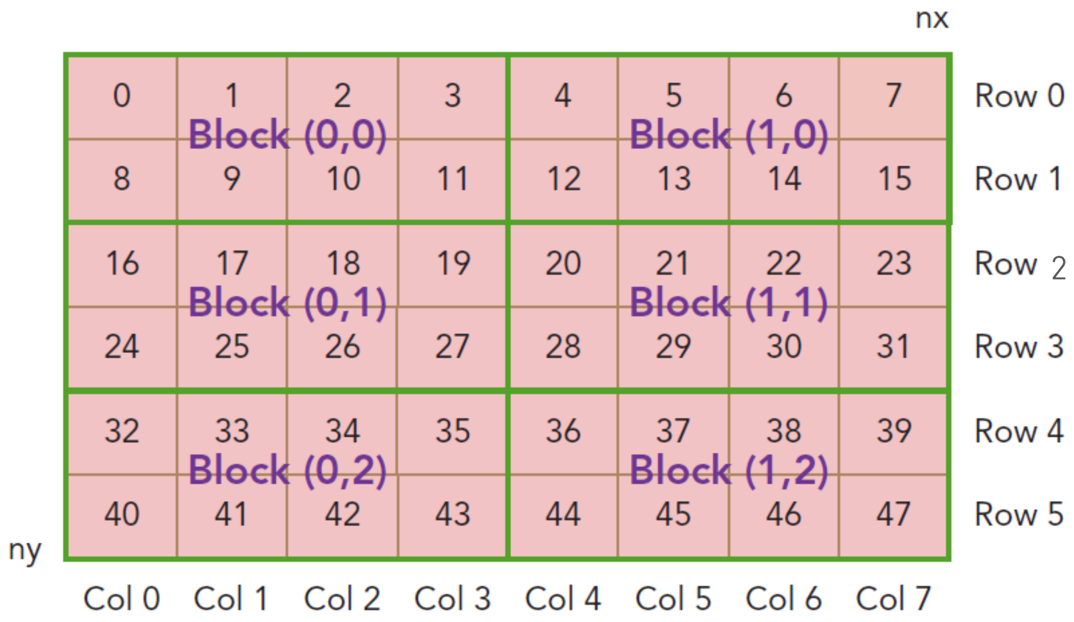
P = M * N
假定 M and N 是方阵



那么，利用CUDA该怎么解决这个问题呢?
空间换时间!

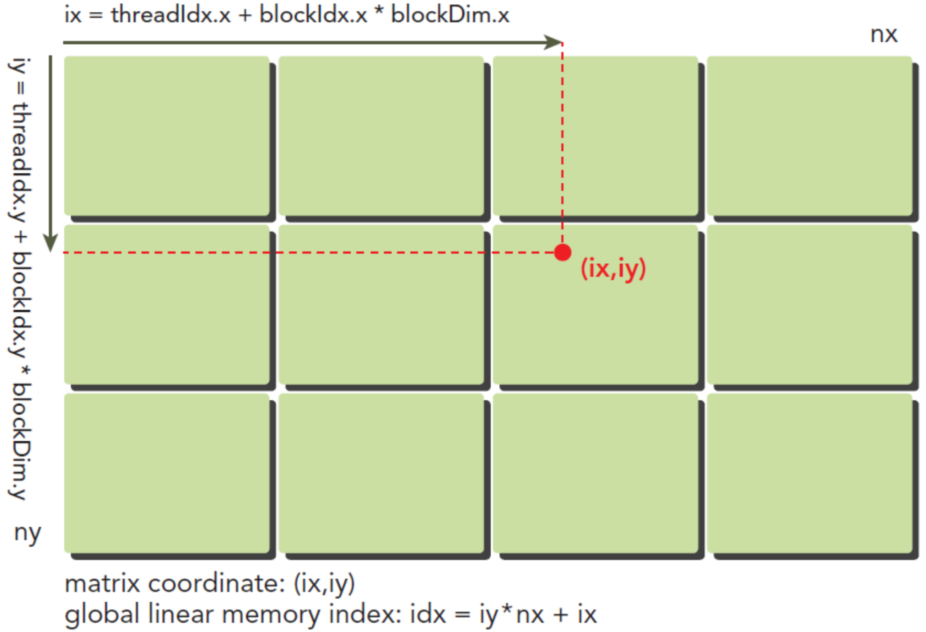
2D GRID AND 2D BLOCKS

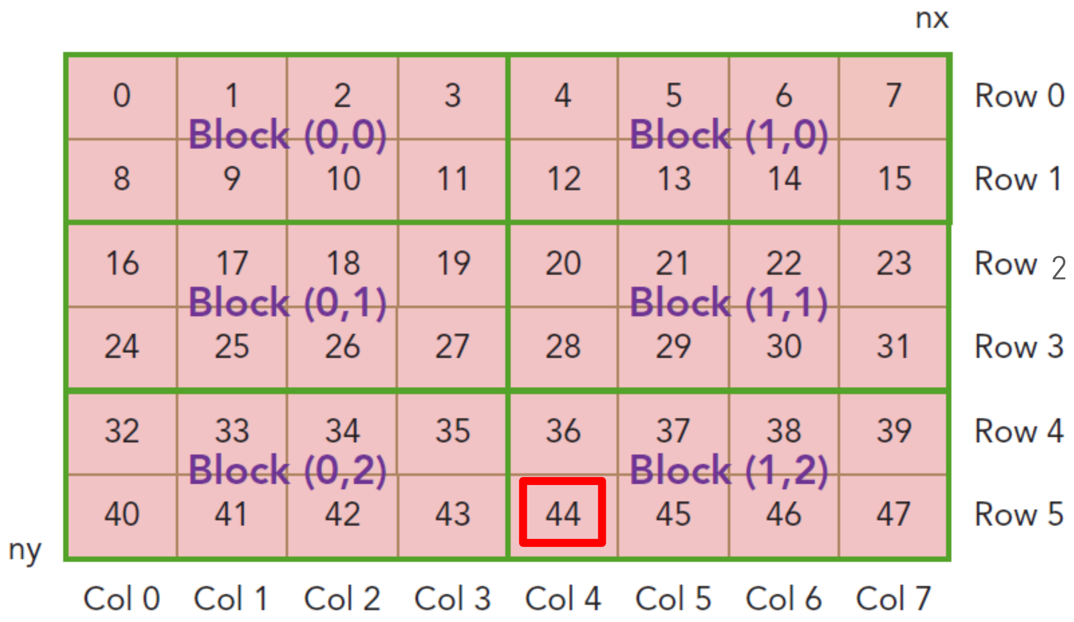





```
__global__ void printThreadIndex(const int nx, const int ny) {
    int ix = threadIdx.x + blockIdx.x * blockDim.x;
    int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy*nx + ix;
    printf("thread_id (%d,%d) block_id (%d,%d) coordinate (%d, %d) "
        "global index %2d \n", threadIdx.x, threadIdx.y, blockIdx.x, blockIdx.y, ix, iy, idx);
}
```

2D GRID AND 2D BLOCKS





int index = (blockIdx.y*blockDim.y+threadIdx.y)*nx+ (blockIdx.x * blockDim.x + threadIdx.x);

Index == (2*2+1)*8+(1*4+0)

矩阵相乘样例

Grid

0,0	1,0	2,0	3,0	0,0	1,0	2,0	3,0	0,0	1,0	2,0	3,0
0,1	1,1	2,1	3,1	0,0	1,1	2,1	3,1	0,1	1,1	2,1	3,1
0,2	1,2	2,2	3,2	0,2	1,2	2,2	3,2	0,2	1,2	2,2	3,2
0,0	1,1	2,0	3,0	0,0	1,0	2,0	3,0	0,0	1,0	2,0	3,0
0,0	1,1	2,1	3,1	0,0	1,1	2,1	3,1	0,1	1,1	2,1	3,1
0,0	1,2	2,2	3,2	0,0	1,2	2,2	3,2	0,2	1,2	2,2	3,2

threadIdx.x = 2
threadIdx.y = 0
blockIdx.x = 1
blockIdx.y = 1
blockDim.x = 4
blockDim.y = 3

每个颜色对应一个block
该线程在grid中所有线程的索引为：

Thread_x = blockIdx.x*blockDim.x+threadIdx.x = 6
Thread_y = blockIdx.y*blockDim.y+threadIdx.y = 3

矩阵相乘样例

当把整个grid对应到一个矩阵的时候，它就像下面的样子：

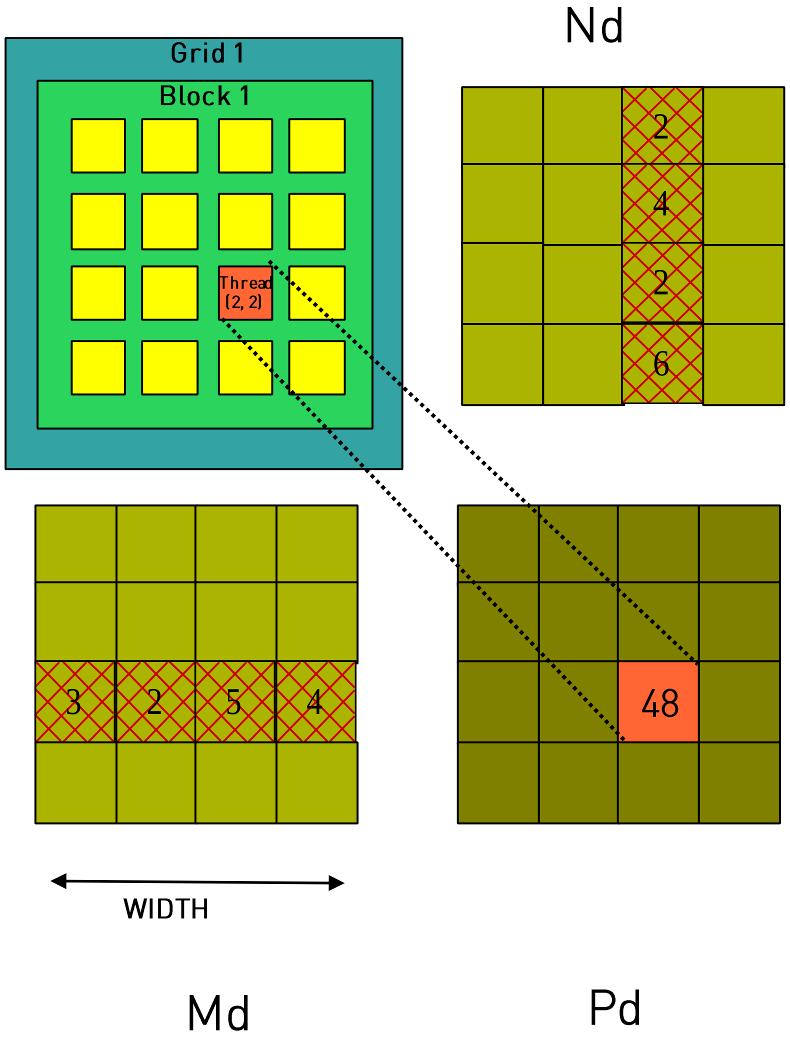
0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0	10,0	11,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	9,1	10,1	11,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2	10,2	11,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3	10,3	11,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4	9,4	10,4	11,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5	9,5	10,5	11,5

而该线程在grid中所有线程的索引正好对应到矩阵的坐标
该线程在grid中所有线程的索引为：

Thread_x = blockIdx.x*blockDim.x+threadIdx.x = 6
Thread_y = blockIdx.y*blockDim.y+threadIdx.y = 3

矩阵相乘样例

- 一个线程grid计算Pd
  - 每个线程计算Pd的一个元素
- 每个线程
  - 读入矩阵Md的一行
  - 读入矩阵Nd的一列
  - 为每对Md和Nd元素执行一次乘法和加法



矩阵相乘样例

计算出当前执行的线程在所有线程中的坐标

读取M矩阵的一行，N矩阵的一列，并做乘积累加

```
__global__ void gpu_matrix_mult(int *M,int *N, int *P,
int m_size, int n_size, int k_size)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    if( col < k_size && row < m_size)
    {
        for(int i = 0; i < n; i++)
        {
            sum += M[row * n_size + i] * N[i *  k_size + col];
        }
        P[row * k_size + col] = sum;
    }
}
```

矩阵相乘样例

```
int main(int argc, char const *argv[])
{
    int m=100;
    int n=100;
    int k=100;

    int *h_a, *h_b, *h_c, *h_cc;
    cudaMallocHost((void **) &h_a, sizeof(int)*m*n);
    cudaMallocHost((void **) &h_b, sizeof(int)*n*k);
    cudaMallocHost((void **) &h_c, sizeof(int)*m*k);
    cudaMallocHost((void **) &h_cc, sizeof(int)*m*k);

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            h_a[i * n + j] = rand() % 1024;
        }
    }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < k; ++j) {
            h_b[i * k + j] = rand() % 1024;
        }
    }

    int *d_a, *d_b, *d_c;
    cudaMalloc((void **) &d_a, sizeof(int)*m*n);
    cudaMalloc((void **) &d_b, sizeof(int)*n*k);
    cudaMalloc((void **) &d_c, sizeof(int)*m*k);

    // copy matrix A and B from host to device memory
    cudaMemcpy(d_a, h_a, sizeof(int)*m*n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, sizeof(int)*n*k, cudaMemcpyHostToDevice);

    unsigned int grid_rows = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;
    unsigned int grid_cols = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;
    dim3 dimGrid(grid_cols, grid_rows);
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

    gpu_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, m, n, k);

    cudaMemcpy(h_c, d_c, sizeof(int)*m*k, cudaMemcpyDeviceToHost);
    //cudaThreadSynchronize();
```

```
__global__ void gpu_matrix_mult(int *a,int *b, int *c, int m, int n, int k)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    if( col < k && row < m)
    {
        for(int i = 0; i < n; i++)
        {
            sum += a[row * n + i] * b[i * k + col];
        }
        c[row * k + col] = sum;
    }
}
```

矩阵相乘样例

- 在算法实现中最主要的性能问题是什么?
- 主要的限制是什么?

更多资源:

https://developer.nvidia-china.com

何琨-Ken

北京 密云

扫一扫上面的二维码图案，加我微信

https://www.nvidia.cn/developer/community-training/


NVIDIA