CHAPTER

CUDA Threads 4

CHAPTER CONTENTS

Tip: transparent. 透明，对程序员不可见

INTRODUCTION

Fine-grained, data-parallel threads are the fundamental means of parallel execution in CUDA™. As we explained in Chapter 3, launching a CUDA kernel function creates a grid of threads that all execute the kernel function. That is, the kernel function specifies the C statements that are executed by each individual thread created when the kernel is launched at runtime. This chapter presents more details on the organization, resource assignment, and scheduling of threads in a grid. A CUDA programmer who understands these details is well equipped to write and to understand high-performance CUDA applications.

#Tip：如何分配grid中的线程资源.

4.1 CUDA THREAD ORGANIZATION

Because all threads in a grid execute the same kernel function, they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into a two-level hierarchy using unique coordinates—blockIdx (for block index) and threadIdx (for thread index)—assigned to them by the CUDA runtime system. The blockIdx and threadIdx appear as built-in, preinitialized variables that can be accessed within kernel functions.

Fine-grained 细粒度

the appropriate portion of data.
数据中的相应部分

preinitialized. 预初始化

respectively 分别，各自

consists of 由...组成

CUDA Runtime System.

Q1：所谓的"build-in"和"preinitialized"是什么.

线程通过 Runtime 被唯一分配 BlockIdx 和 ThreadIdx

不需要用户显式定义.

Q2：同时"build-in"的 gridDim 和 blockDim.

这两个变量本身是不固定的.

都是在线程启动的时候通过<<<grid, block>>>确定

或者说是在 kernel 启动的过程中固定.

Tip：不同的kernel函数执行 / 不同的指定.

之间可以是不同的.

Q3：<<<grid, block>>>如何对应到 Figure 4.1

1). 图中展示的情况都是 1D 的情况，也可认为是标量

2). grid / block 本身是支持 3D 分配的.

<<<(grid.x, grid.y, grid.z), (~)>>>. # 元组.

如果是 1D 分配 / 标量情况，直接传递数值也 OK 的.

3). blockDim 就是[Thread block 0 ~ Thread block N-1]

这样的编号，通过索引查找

When a thread executes the kernel function, references to the `blockIdx` and `threadIdx` variables return the coordinates of the thread. Additional built-in variables, `gridDim` and `blockDim`, provide the dimension of the grid and the dimension of each block respectively.

Figure 4.1 shows a simple example of CUDA thread organization. The grid in this example consists of N thread blocks, each with a `blockIdx.x` value that ranges from 0 to $N - 1$. Each block, in turn, consists of M threads, each with a `threadIdx.x` value that ranges from 0 to $M - 1$. All blocks at the grid level are organized as a one-dimensional (1D) array; all threads within each block are also organized as a 1D array. Each grid has a total of $N*M$ threads.

The black box of each thread block in Figure 4.1 shows a fragment of the kernel code. The code fragment uses the `threadID = blockIdx.x * blockDim.x + threadIdx` to identify the part of the input data to read from and the part of the output data structure to write to. Thread 3 of Block 0 has a `threadID` value of $0*M + 3 = 3$. Thread 3 of Block 5 has a `threadID` value of $5*M + 3$.

#参考数组.

Assume a grid has 128 blocks ($N = 128$) and each block has 32 threads ($M = 32$). In this example, access to `blockDim` in the kernel returns 32. There are a total of $128*32 = 4096$ threads in the grid. Thread 3 of Block 0 has a threaded value of $0*32 + 3 = 3$. Thread 3 of Block 5 has a threaded value of $5*32 + 3 = 163$. Thread 15 of Block 102 has a threaded value of 3279. The reader should verify that every one of the 4096 threads has its own unique threaded value. In Figure 4.1, the kernel code uses threadID variable to index into the `input[]` and the `output[]` arrays. If we assume that both arrays are declared with 4096 elements, then each thread will take one of the input elements and produce one of the output elements.

In general, a grid is organized as a 2D array of blocks. Each block is organized into a 3D array of threads. The exact organization of a grid is determined by the execution configuration provided at kernel launch. The first parameter of the execution configuration specifies the dimensions of the grid in terms of number of blocks. The second specifies the dimensions of each block in terms of number of threads. Each such parameter is a `dim3` type, which is essentially a C `struct` with three unsigned integer fields: x, y, and z. Because grids are 2D arrays of block dimensions, the third field of the grid dimension parameter is ignored; it should be set to 1 for clarity. The following host code can be used to launch the kernel whose organization is shown in Figure 4.1:

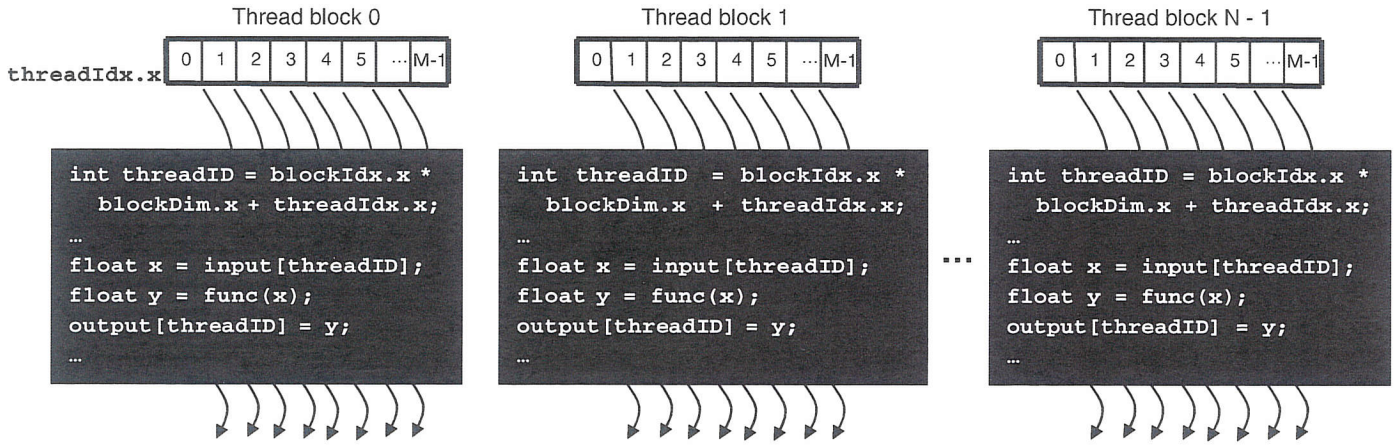


FIGURE 4.1

Overview of CUDA thread organization.

scalar values. 标量.

Q: gridDim的变化范围 [1, 65535].

网格Grid中每个维度(x,y,z)的线程块数量最大值.

Thread Blocks = gridDim.x * gridDim.y * gridDim.z

```
dim3 dimGrid(128, 1, 1);
dim3 dimBlock(32, 1, 1);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

The first two statements initialize the execution configuration parameters. Because the grid and the blocks are 1D arrays, only the first dimension of `dimBlock` and `dimGrid` are used. The other dimensions are set to 1. The third statement is the actual kernel launch. The execution configuration parameters are between <<< and >>>. Note that scalar values can also be used for the execution configuration parameters if a grid or block has only one dimension; for example, the same grid can be launched with one statement:

```
KernelFunction<<<128, 32>>>(...);
```

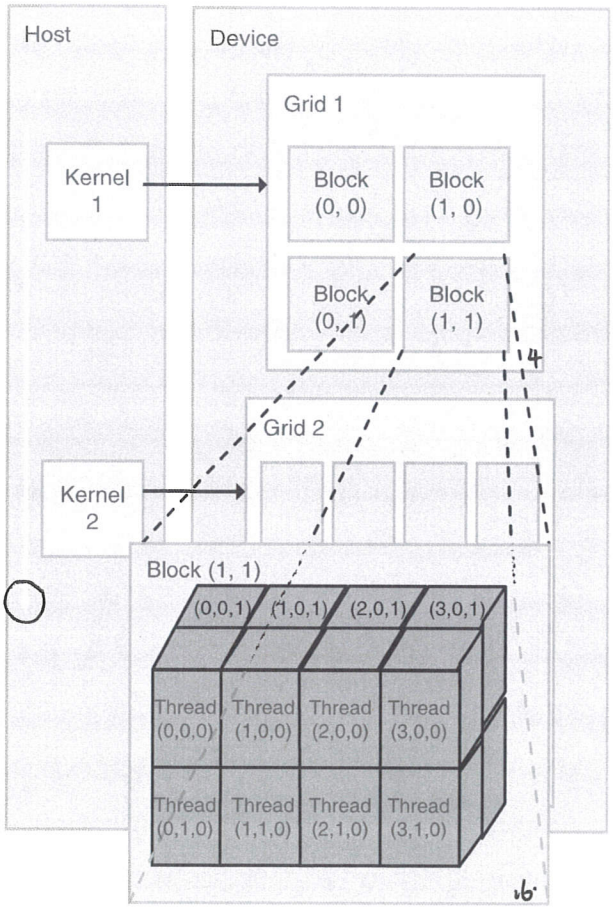
The values of `gridDim.x` and `gridDim.y` can range from 1 to 65,535. The values of `gridDim.x` and `gridDim.y` can be calculated based on other variables at kernel launch time. Once a kernel is launched, its dimensions cannot change. All threads in a block share the same `blockIdx` value. The `blockIdx.x` value ranges between 0 and `gridDim.x - 1`, and the `blockIdx.y` value between 0 and `gridDim.y - 1`.

Figure 4.2 shows a small 2D grid that is launched with the following host code:

```
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

The grid consists of four blocks organized into a 2×2 array. Each block in Figure 4.2 is labeled with (`blockIdx.x`, `blockIdx.y`); for example, Block(1,0) has `blockIdx.x` = 1 and `blockIdx.y` = 0.

In general, blocks are organized into 3D arrays of threads. All blocks in a grid have the same dimensions. Each `threadIdx` consists of three components: the *x* coordinate `threadIdx.x`, the *y* coordinate `threadIdx.y`, and the *z* coordinate `threadIdx.z`. The number of threads in each dimension of a block is specified by the second execution configuration parameter given at the kernel launch. With the kernel, this configuration parameter can be accessed as a predefined `struct` variable, `blockDim`. The total size of a block is limited to 512 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 512. For example, (512, 1, 1), (8, 16, 2), and (16, 16, 2) are all allowable `blockDim` values, but (32, 32, 1) is not allowable because the total number of threads would be 1024.



#kernel函数串行.

FIGURE 4.2

A multidimensional example of CUDA grid organization.

Figure 4.2 also illustrates the organization of threads within a block. In this example, each block is organized into 4×2×2 arrays of threads. Because all blocks within a grid have the same dimensions, we only need to show one of them. Figure 4.2 expands block (1, 1) to show its 16 threads; for example, thread (2, 1, 0) has `threadIdx.x` = 2, `threadIdx.y` = 1, and `threadIdx.z` = 0. Note that, in this example, we have 4 blocks of 16 threads each, with a grand total of 64 threads in the grid. We have used these small numbers to keep the illustration simple. Typical CUDA grids contain thousands to millions of threads.

more sophisticated usage of 更复杂的使用 /更高级的用法.

conceptually 从概念上看.

square tiles. 方形网格.

abbreviate 缩写,简化

4.2 USING blockIdx AND threadIdx

From the programmer's point of view, the main functionality of blockIdx and threadIdx variables is to provide threads with a means to distinguish among themselves when executing the same kernel. One common usage for threadIdx and blockIdx is to determine the area of data that a thread is to work on. This was illustrated by the simple matrix multiplication code in Figure 3.11, where the dot product loop uses threadIdx.x and threadIdx.y to identify the row of **Md** and column of **Nd** to work on. We will now cover more sophisticated usage of these variables.

One limitation of the simple code in Figure 3.11 is that it can only handle matrices of up to 16 elements in each dimension. This limitation comes from the fact that the kernel function does not use blockIdx. As a result, we are limited to using only one block of threads. Even if we used more blocks, threads from different blocks would end up calculating the same **Pd** element if they have the same threadIdx value. Recall that each block can have up to 512 threads. With each thread calculating one element of **Pd**, we can calculate up to 512 **Pd** elements with the code. For square matrices, we are limited to 16×16 because 32×32 requires more than 512 threads per block.

#程序不会自动分辨自己是哪个block,要体现在计算式中.

#硬件会影响逻辑层面的指定.

In order to accommodate larger matrices, we need to use multiple thread blocks. Figure 4.3 shows the basic idea of such an approach. Conceptually, we break **Pd** into square tiles. All the **Pd** elements of a tile are computed by a block of threads. By keeping the dimensions of these **Pd** tiles small, we keep the total number of threads in each block under 512, the maximal allowable block size. In Figure 4.3, for simplicity, we abbreviate threadIdx.x and threadIdx.y as tx and ty. Similarly, we abbreviate blockIdx.x and blockIdx.y as bx and by.

#由硬件结构导致的逻辑层面的分层设计.

Each thread still calculates one **Pd** element. The difference is that it must use its blockIdx values to identify the tile that contains its element before it uses its threadIdx values to identify its element inside the tile. That is, each thread now uses both threadIdx and blockIdx to identify the **Pd** element to work on. This is portrayed in Figure 4.3, where the bx, by, tx, and ty values of threads calculating the **Pd** elements are marked in both x and y dimensions. All threads calculating the **Pd** elements within a tile have the same blockIdx values.

Assume that the dimensions of a block are square and are specified by the variable TILE_WIDTH. Each dimension of **Pd** is now divided into sections of TILE_WIDTH elements each, as shown on the left and top edges of Figure 4.3. Each block handles such a section. Thus, a thread can find the

Q1：每个Pd维度现在被划分为包含2个元素的部分
只是从维度来看，eg. x轴上4个元素切分后为2个，
具体到Pd矩阵内部还是4个元素

x index of its **Pd** element as (bx*TILE_WIDTH + tx) and the y index as (by*TILE_WIDTH + ty). That is, thread (tx, ty) in block (bx, by) is to use row (by*TILE_WIDTH + ty) of **Md** and column (bx*TILE_WIDTH + tx) of **Nd** to calculate the **Pd** element at column (bx*TILE_WIDTH + tx) and row (by*TILE_WIDTH + ty).

Figure 4.4 shows a small example of using multiple blocks to calculate **Pd**. For simplicity, we use a very small TILE_WIDTH value (2) so we can fit the entire example in one picture. The **Pd** matrix is now divided into 4 tiles. Each dimension of **Pd** is now divided into sections of 2 elements. Each block needs to calculate 4 **Pd** elements. We can do so by creating blocks that are organized into 2×2 arrays of threads, with each thread calculating

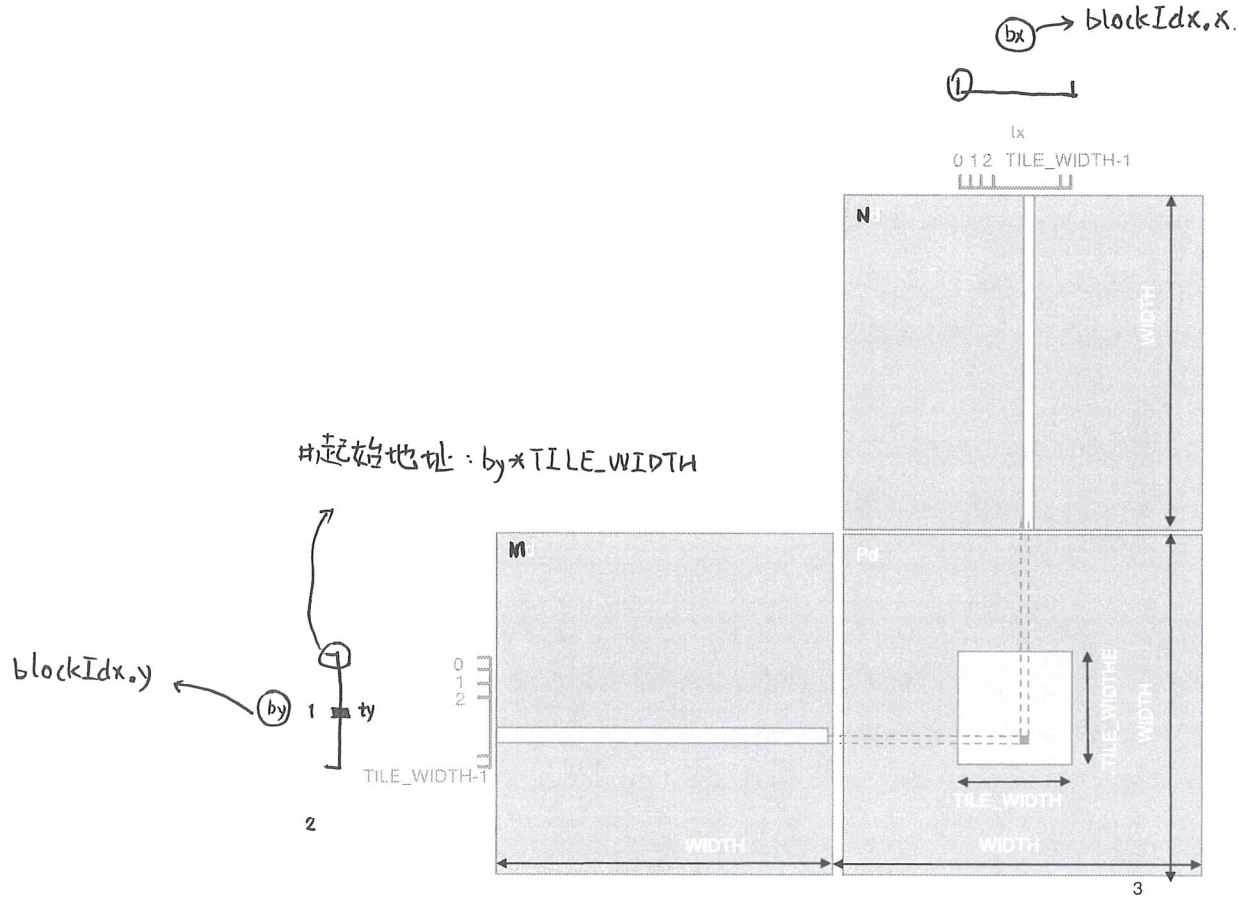


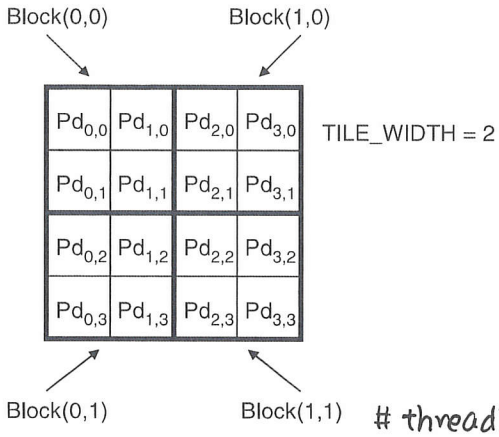
FIGURE 4.3

Matrix multiplication using multiple blocks by tiling **Pd**.

index derivation. 索引推导.

revise 修订,校正

#.Tip:这里是矩阵乘的结果
一个像素 Pd0,0 对应一行一列的乘加结果.
$M_{4,4} \times N_{4,4} = P_{4,4}$



#threadIdx在各自的block中是独立计数
(threadIdx.x, threadIdx.y)

FIGURE 4.4
A simplified example of using multiple blocks to calculate **Pd**.

one **Pd** element. In the example, thread (0, 0) of block (0, 0) calculates $\mathbf{Pd}_{0,0}$, whereas thread (0, 0) of block (1, 0) calculates $\mathbf{Pd}_{2,0}$. It is easy to verify that one can identify the **Pd** element calculated by thread (0, 0) of block (1, 0) with the formula given above: `Pd[bx* TILE_WIDTH + tx][by* TILE_WIDTH + ty] = Pd[1*2 + 0][0*2 + 0] = Pd[2][0]`. The reader should work through the index derivation for as many threads as it takes to become comfortable with the concept.

Once we have identified the indices for the **Pd** element to be calculated by a thread, we also have identified the row (y) index of **Md** and the column (x) index of **Nd** for input values. As shown in Figure 4.3, the row index of **Md** used by thread (`tx`, `ty`) of block (`bx`, `by`) is (`by*TILE_WIDTH + ty`). The column index of **Nd** used by the same thread is (`bx*TILE_WIDTH + tx`). We are now ready to revise the kernel of Figure 3.11 into a version that uses multiple blocks to calculate **Pd**.

Figure 4.5 illustrates the multiplication actions in each thread block. For the small matrix multiplication, threads in block (0, 0) produce four dot products: Thread (0, 0) generates $\mathbf{Pd}_{0,0}$ by calculating the dot product of row 0 of **Md** and column 0 of **Nd**. Thread (1, 0) generates $\mathbf{Pd}_{1,0}$ by calculating the dot product of row 0 of **Md** and column 1 of **Nd**. The arrows of $\mathbf{Pd}_{0,0}$, $\mathbf{Pd}_{1,0}$, $\mathbf{Pd}_{0,1}$, and $\mathbf{Pd}_{1,1}$ shows the row and column used for generating their result value.

Figure 4.6 shows a revised matrix multiplication kernel function that uses multiple blocks. In Figure 4.6, each thread uses its `blockIdx` and `threadIdx` values to identify the row index (`Row`) and the column index

即：在需要计算的矩阵大于此新限制的情况下

4. Tip: 单个线程所需要的是一行&一列的输入.
但同一个block中的线程数据没有重复的部分, ∴ Block有数据共享.

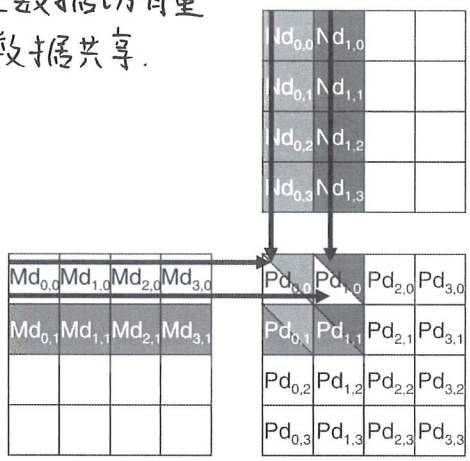


FIGURE 4.5

Matrix multiplication actions of one thread block.

#16是一个block内部的线程数.
65535是一个grid内部的.

(Col) of the **Pd** element that it is responsible for. It then performs a dot product on the row of **Md** and column of **Nd** to generate the value of the **Pd** element. It eventually writes the **Pd** value to the appropriate global memory location. Note that this kernel can handle matrices of up to 16 × 65,535 elements in each dimension. In the situation where matrices larger than this new limit are to be multiplied, one can divide the **Pd** matrix into submatrices of a size permitted by the kernel. Each submatrix would still be

#62.

#Tip: 这里是从单个元素数量(输出)的角度来讨论.

注[1].

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
 // Calculate the row index of the Pd element and M
 int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
 // Calculate the column idenx of Pd and N
 int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

 float Pvalue = 0;
 // each thread computes one element of the block sub-matrix
 for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

 Pd[Row*Width+Col] = Pvalue;
}
```

FIGURE 4.6

Revised matrix multiplication kernel using multiple blocks.

an ample number of 大量的，足够的.

row major layout. 行优先存局.

TRANSPARENT SCALABILITY.
透明的可扩展性.

a barrier synchronization function
阻塞同步函数

Q. Devide the Pd matrix into submatrices.
将大矩阵拆分为内核允许的大小的子矩阵.
每个矩阵仍然可由 65535 × 65535 个线程块执行.
Tip: 这里的重点，是每个子矩阵单独对应一个 kernel.
kernel 的调用是异步的.
case1: 只有一片 GPU ⇒ 只能串行计算.
case2: 有多块 GPU ⇒ 可以同时计算

Q. 最大的线程块数不能是 65535 × 65535 × 65535 吗.
理论上是支持. gridDim.z 达到 65535 的.
但硬件上可能不支持. 而且大部分输入都是二维.

Q: stub function.
指 host 代码 (CPU端) 启动 CUDA kernel 的函数.
eg. <<<grid, block>>>.
每个被 "__global__" 修饰的 kernel 函数都对应一个 stub f

Q. row major layout.
CUDA 本身没有默认规定的存储方式.
但是来自 C / C++ 的数据一般是默认行优先存储.
同时行优先存储 ⇒ 索引计算方式的固定.

```
// Setup the execution configuration
  dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
  dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

FIGURE 4.7

Revised host code for launching the revised kernel.

processed by an ample number of blocks (65,535 × 65,535). All of these blocks can run in parallel with each other and will fully utilize parallel execution resources of any processors in the foreseeable future.

Figure 4.7 shows the revised host code to be used in the `MatrixMultiplication()` stub function to launch the revised kernel. Note that the `dimGrid` now receives the value of `Width/TILE_WIDTH` for both the *x* dimension and the *y* dimension. The revised code now launches the `MatrixMulKernel()` with multiple blocks. Note that the code treats the Md, Nd, and Pd arrays as 1D array with row major layout. The calculation of the indices used to access Md, Nd, and Pd is the same as that in Section 3.3.

#MatrixMulKernel()内部执行不变，顶层的引用改变了.

4.3 SYNCHRONIZATION AND TRANSPARENT SCALABILITY

#在线程到达同步点前必须阻塞直到全部到达

CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function, `__syncthreads()`. When a kernel function calls `__syncthreads()`, the thread that executes the function call will be held at the calling location until every thread in the block reaches the location. This ensures that all threads in a block have completed a phase of their execution of the kernel before any moves on to the next phase. We will discuss an important use of `__syncthreads()` in Chapter 5.

Barrier synchronization is a simple and popular method of coordinating parallel activities. In real life, we often use barrier synchronization to coordinate parallel activities of multiple persons; for example, assume that four friends go to a shopping mall in a car. They can all go to different stores to buy their own clothes. This is a parallel activity and is much more efficient than if they all remain as a group and sequentially visit the stores to shop for their clothes. However, barrier synchronization is needed before they leave the mall. They have to wait until all four friends have returned to the car before they can leave. Without barrier synchronization, one or more persons could be left at the mall when the car leaves, which could seriously damage their friendship!

imposes 施加.

execution constraints. 执行约束

proximity 接近. 大约

scalable implementation 可扩展的实现

In CUDA, a `__syncthreads()` statement must be executed by all threads in a block. When a `__syncthreads()` statement is placed in an *if* statement, either all threads in a block execute the path that includes the `__syncthreads()` or none of them does. For an *if–then–else* statement, if each path has a `__syncthreads()` statement, then either all threads in a block execute the `__syncthreads()` on the *then* path or all of them execute the *else* path. The two `__syncthreads()` are different barrier synchronization points. If a thread in a block executes the *then* path and another executes the *else* path, then they would be waiting at different barrier synchronization points. They would end up waiting for each other forever.

#死锁的情况.

The ability to synchronize also imposes execution constraints on threads within a block. These threads should execute in close time proximity with each other to avoid excessively long waiting times. CUDA runtime systems satisfy this constraint by assigning execution resources to all threads in a block as a unit; that is, when a thread of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource. This ensures the time proximity of all threads in a block and prevents excessive waiting time during barrier synchronization.

This leads us to a major tradeoff in the design of CUDA barrier synchronization. By not allowing threads in different blocks to perform barrier synchronization with each other, the CUDA runtime system can execute blocks in any order relative to each other because none of them must wait for each other. This flexibility enables scalable implementations as shown in Figure 4.8. In a low-cost implementation with only a few execution resources, one can execute a small number of blocks at the same time

#突显出线程资源分级分配的优势.

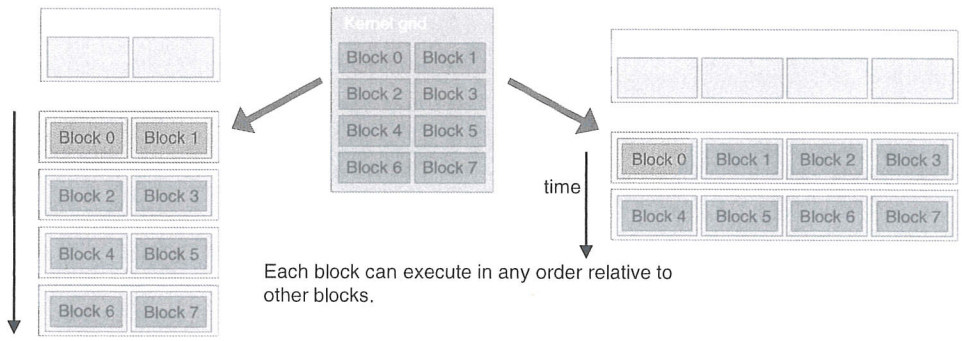


FIGURE 4.8
Transparent scalability for CUDA programs allowed by the lack of synchronization constraints between blocks.

#由于没有同步约束，实现了透明的可扩展性

#. 在一个 grid 中，block 之间没有前后关系.
所以同一个 grid 中的 block 执行数量和顺序任意.

#Tip：无论是在高端/低端 GPU 上执行.
CUDA 代码不需要任何修改.

eg. <<< 1024, 256>>> (...);
1). 低端 GPU：4个 SM，线程块排队.
2). 高端 GPU：80个，更快.

a mobile processor 移动处理.

on a block-by-block basis 按块处理.

simultaneously tracked and scheduled. 同时跟踪和调度.

(shown as executing two blocks a time on the left-hand side of Figure 4.8). In a high-end implementation with more execution resources, one can execute a large number of blocks at the same time (shown as executing four blocks at a time on the right-hand side of Figure 4.8). The ability to execute the same application code at a wide range of speeds allows the production of a wide range of implementations according to the cost, power, and performance requirements of particular market segments. A mobile processor, for example, may execute an application slowly but at extremely low power consumption, and a desktop processor may execute the same application at a higher speed while consuming more power. Both execute exactly the same application program with no change to the code. The ability to execute the same application code on hardware with different numbers of execution resources is referred to as *transparent scalability*, which reduces the burden on application developers and improves the usability of applications.

4.4 THREAD ASSIGNMENT

Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads. These threads are assigned to execution resources on a block-by-block basis. In the current generation of hardware, the execution resources are organized into streaming multiprocessors (SMs); for example, the NVIDIA® GT200 implementation has 30 streaming multiprocessors, 2 of which are shown in Figure 4.9. Up to 8 blocks can be assigned to each SM in the GT200 design as long as there are enough resources to satisfy the needs of all of the blocks. In situations with an insufficient amount of any one or more types of resources needed for the simultaneous execution of 8 blocks, the CUDA runtime automatically reduces the number of blocks assigned to each SM until the resource usage is under the limit. With 30 SMs in the GT200 processor, up to 240 blocks can be simultaneously assigned to them. Most grids contain many more than 240 blocks. The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them.

Figure 4.9 shows an example in which three thread blocks are assigned to each SM. One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled. Hardware resources are required for SMs to maintain the thread, block IDs, and track their execution status. In the GT200 design, up to 1024 threads can be assigned to each SM. This could

residing in. 驻留，存在于.

Latency tolerance. 延迟容忍度.

strictly 严格，确切来说

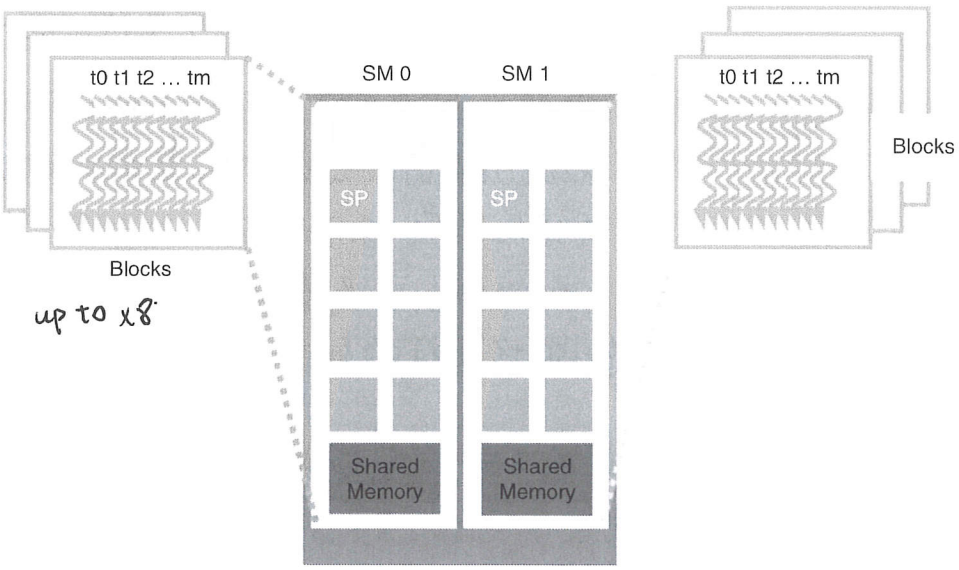


FIGURE 4.9

Thread block assignment to streaming multiprocessors (SMs).

从1024的线程总数分配

be in the form of 4 blocks of 256 threads each, 8 blocks of 128 threads each, etc. It should be obvious that 16 blocks of 64 threads each is not possible, as each SM can only accommodate up to 8 blocks. Because the GT200 has 30 SMs, up to 30,720 threads can be simultaneously residing in the SMs for execution. The number of threads that can be assigned to each SM increased from the G80 to the GT200. Each SM in G80 can accommodate 768 threads, and, because the G80 has 16 SMs, up to 12,288 threads can be simultaneously residing in the SMs for execution. The transparent scalability of CUDA allows the same application code to run unchanged on G80 and GT200.

4.5 THREAD SCHEDULING AND LATENCY TOLERANCE

Thread scheduling is strictly an implementation concept and thus must be discussed in the context of specific hardware implementations. In the GT200 implementation, once a block is assigned to a streaming multiprocessor, it is further divided into 32-thread units called *warps*. The size of warps is implementation specific. In fact, warps are not part of the CUDA specification; however, knowledge of warps can be helpful in understanding and optimizing the performance of CUDA applications on particular generations of CUDA devices. The warp is the unit of thread scheduling in SMs.

SM中的线程分配器

consecutive. 连续的.

Q: block, warp, threadIdx 的关系.

一个线程块中的线程被 GPU 按照 warp 的粒度划分

warp：硬件调度单位，GPU以warp为单位执行.

Block：逻辑调度单位，CUDA程序员的代码编写.

eg 假如一个 Block 支持 32N 的线程.

那么CUDA会调用 N个 warp 来执行这一个 block.

eg. 假如一个 Block 支持 4个的线程 (小于32).

那么CUDA仍然会调一个完整的warp来支持一个 block.

即使是浪费 28个线程.

因为 CUDA 中的 Block 是逻辑上独立的执行单位.

合并就破坏这样的独立性.

Tip：逻辑同属一个 Block 对应多个 warp 的情况 (32*N).

如果硬件资源足够，这N个 warp 会同时执行.

如果不足，这N个 warp 之间就会部分串行.

eg. 这里硬件资源比如 SM 支持的 warp 数量.

RegisterFiles, SharedMemory 之类的.

threadIdx，与 Block 同一层面的逻辑上的概念.

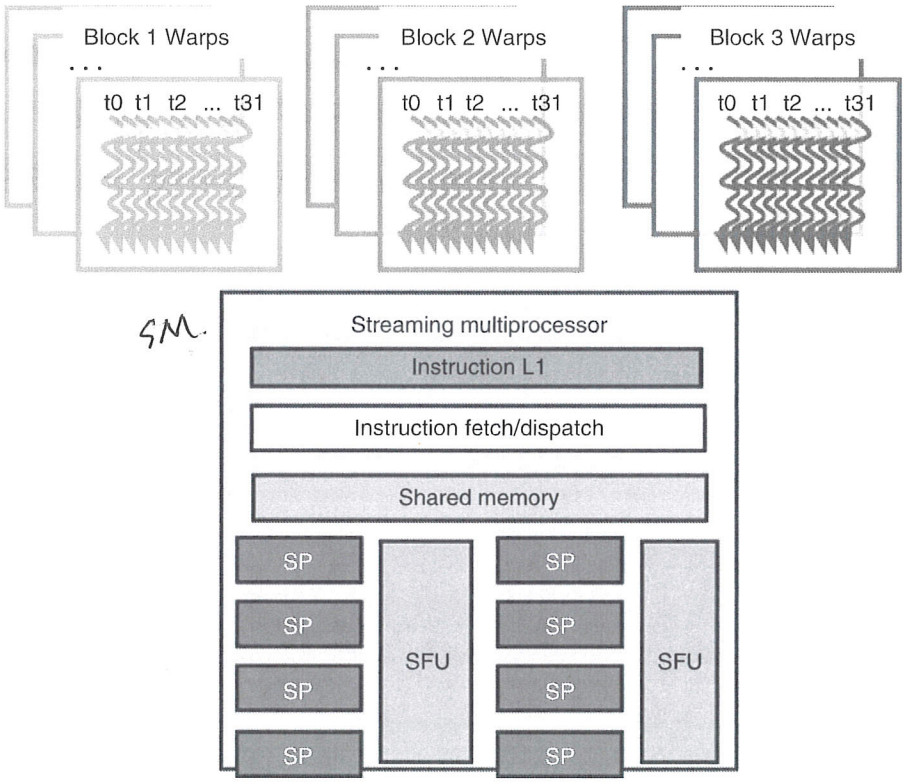


FIGURE 4.10

Blocks partitioned into warps for thread scheduling.

#连续编号.

Figure 4.10 shows the division of blocks into warps in the GT200. Each warp consists of 32 threads of consecutive `threadIdx` values: Threads 0 through 31 form the first warp, threads 32 through 63 the second warp, and so on. In this example, three blocks (Block 1, Block 2, and Block 3) are all assigned to an SM. Each of the three blocks is further divided into warps for scheduling purposes.

We can calculate the number of warps that reside in an SM for a given block size and a given number of blocks assigned to each SM. In Figure 4.10, for example, if each block has 256 threads, then we can determine that each block has 256/32 or 8 warps. With 3 blocks in each SM, we have $8 \times 3 = 24$ warps in each SM. This is, in fact, the maximal number of warps that can reside in each SM in the G80, as there can be no more than 768 threads in each SM, which amounts to $768/32 = 24$ warps. This number increases to 32 warps per SM for the GT200.

$$\frac{block\_size \times block\_num}{32} = warp\_num$$

a legitimate question

一个合法问题.

dedicate. 致力于.

the pros and cons of

…的优缺点,

Q:SM与SP的关系.

SP是最小的线程执行单元

一个线程依赖SP来执行所有指令.

如果warp中的线程数超出SM中SP的数量.线程后续会排队等待资源.也是影响warp并行的硬件资源指标.

zero-Overhead. Thread. Scheduling.

零开销线程调度.

正是因为没有线程开销.才能支持多个warp切换的方法.

#相同SP执行资源会轮流给不同的warp执行来，掩盖时延.

A legitimate question is why do we need to have so many warps in an SM if there are only 8 SPs in an SM? The answer is that this is how CUDA processors efficiently execute long-latency operations such as global memory accesses. When an instruction executed by the threads in a warp must wait for the result of a previously initiated long-latency operation, the warp is not selected for execution. Another resident warp that is no longer waiting for results is selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution. This mechanism of filling the latency of expensive operations with work from other threads is often referred to as *latency hiding*.

Note that warp scheduling is also used for tolerating other types of long-latency operations such as pipelined floating-point arithmetic and branch instructions. With enough warps around, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long-latency operations. The selection of ready warps for execution does not introduce any idle time into the execution timeline, which is referred to as *zero-overhead thread scheduling*. With warp scheduling, the long waiting time of warp instructions is hidden by executing instructions from other warps. This ability to tolerate long-latency operations is the main reason why graphics processing units (GPUs) do not dedicate nearly as much chip area to cache memories and branch prediction mechanisms as central processing units (CPUs) do. As a result, GPUs can dedicate more of their chip area to floating-point execution resources.

#block中的线程数量，这里的 8/16/32 指矩阵维度.

We are now ready to do a simple exercise.[1] For matrix multiplication, should we use 8×8, 16×16, or 32×32 thread blocks for the GT200? To answer the question, we can analyze the pros and cons of each choice. If we use 8×8 blocks, each block would have only 64 threads, and we will need 1024/64 = 12 blocks to fully occupy an SM; however, because we are limited to 8 blocks in each SM, we will end up with only 64 × 8 = 512 threads in each SM. This means that the SM execution resources will likely be underutilized because there will be fewer warps to schedule around long-latency operations.

[1]Note that this is an overly simplified exercise. As we will explain in Chapter 5, the usage of other resources such as registers and shared memory must also be considered when determining the most appropriate block dimensions. This exercise highlights the interactions between the limit on the number of thread blocks and the limit on the number of threads that can be assigned to each SM.

compels. 强迫.

in arbitrary order 以任意顺序

on a block-by-block basis. 按块处理.

partitioned. into. 被划分为

The 16×16 blocks give 256 threads per block. This means that each SM can take 1024/256 = 4 blocks. This is within the 8-block limitation. This is a good configuration because we will have full thread capacity in each SM and the maximal number of warps for scheduling around the long-latency operations. The 32×32 blocks exceed the limitation of up to 512 threads per block.

4.6 SUMMARY

The kernel execution configuration defines the dimensions of a grid and its blocks. Unique coordinates in `blockIdx` and `threadIdx` variables allow threads of a grid to identify themselves and their domains. It is the programmer's responsibility to use these variables in kernel functions so the threads can properly identify the portion of the data to process. This model of programming compels the programmer to organize threads and their data into hierarchical and multidimensional organizations.

Once a grid is launched, its blocks are assigned to streaming multiprocessors in arbitrary order, resulting in transparent scalability of CUDA applications. The transparent scalability comes with the limitation that threads in different blocks cannot synchronize with each other. The only safe way for threads in different blocks to synchronize with each other is to terminate the kernel and start a new kernel for the activities after the synchronization point.

Threads are assigned to SMs for execution on a block-by-block basis. For GT200 processors, each SM can accommodate up to 8 blocks or 1024 threads, whichever becomes a limitation first. Once a block is assigned to an SM, it is further partitioned into warps. At any time, the SM executes only a subset of its resident warps for execution. This allows the other warps to wait for long-latency operations without slowing down the overall execution throughput of the massive number of execution units.

4.7 EXERCISES

4.1 A student mentioned that he was able to multiply two 1024×1024 matrices using a tiled matrix multiplication code with 1024 thread blocks on the G80. He further mentioned that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?

Not possible.

4.2 The following kernel is executed on a large matrix, which is tiled into submatrices. To manipulate tiles, a new CUDA programmer has written the following device kernel to transpose each tile in the matrix. The tiles are of size `BLOCK_SIZE` by `BLOCK_SIZE`, and each of the dimensions of matrix A is known to be a multiple of `BLOCK_SIZE`. The kernel invocation and code are shown below. `BLOCK_SIZE` is known at compile time but could be set anywhere from 1 to 20.

```
dim3 blockDim(BLOCK_SIZE,BLOCK_SIZE);
dim3 gridDim(A_width/blockDim.x,A_height/blockDim.y);
BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);

__global__ void
BlockTranspose(float* A_elements, int A_width, int A_height)
{
  __shared__ float blockA[BLOCK_SIZE][BLOCK_SIZE];

  int baseIdx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
  baseIdx += (blockIdx.y * BLOCK_SIZE + threadIdx.y) * A_width;

  blockA[threadIdx.y][threadIdx.x] = A_elements[baseIdx];
  A_elements[baseIdx] = blockA[threadIdx.x][threadIdx.y];
}
```

Out of the possible range of values for `BLOCK_SIZE`, for what values of `BLOCK_SIZE` will this kernel function correctly when executing on the device? baseIdx的+= ∴是1合理.

4.3 If the code does not execute correctly for all `BLOCK_SIZE` values, suggest a fix to the code to make it work for all `BLOCK_SIZE` values.

把+=改成=.