CHAPTER

CUDA™ Memories

5

CHAPTER CONTENTS

INTRODUCTION

So far, we have learned to write a CUDA™ kernel function that is executed by a massive number of threads. The data to be processed by these threads are first transferred from the host memory to the device global memory. The threads then access their portion of the data from the global memory using their block IDs and thread IDs. We have also learned more details of the assignment and scheduling of threads for execution. Although this is a very good start, these simple CUDA kernels will likely achieve only a small fraction of the potential speed of the underlying hardware. The poor performance is due to the fact that global memory, which is typically implemented with dynamic random access memory (DRAM), tends to have long access latencies (hundreds of clock cycles) and finite access bandwidth. Although having many threads available for execution can theoretically tolerate long memory access latencies, one can easily run into a situation where traffic congestion in the global memory access paths prevents all but a few threads from making progress, thus rendering some of the streaming multiprocessors (SMs) idle. In order to circumvent such congestion, CUDA provides a number of additional methods for accessing memory that can remove the majority of data requests to the global memory. In this chapter, you will learn to use these memories to boost the execution efficiency of CUDA kernels.

#高延时+有限带宽

#停等.

traffic congestion 流量拥堵. 泛通拥堵.
rendering 渲染, 呈现, 使…呈现出…样的状态.
circumvent. 规避, 绕开

replicated. 复制. 重复.
single-precision floating-point. 单精度浮点数
datum 单个数据.

5.1 IMPORTANCE OF MEMORY ACCESS EFFICIENCY

We can illustrate the effect of memory access efficiency by calculating the expected performance level of the matrix multiplication kernel code shown in Figure 4.6, replicated here in Figure 5.1. The most important part of the kernel in terms of execution time is the *for* loop that performs dot product calculations. In every iteration of this loop, two global memory accesses are performed for one floating-point multiplication and one floating-point addition. Thus, the ratio of floating-point calculation to the global memory access operation is 1 to 1, or 1.0. We will refer to this ratio as the *compute to global memory access (CGMA) ratio*, defined as the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program.

The CGMA ratio has major implications on the performance of a CUDA kernel. For example, the NVIDIA® G80 supports 86.4 gigabytes per second (GB/s) of global memory access bandwidth. The highest achievable floating-point calculation throughput is limited by the rate at which the input data can be loaded from the global memory. With 4 bytes in each single-precision floating-point datum, one can expect to load not more than 21.6 (86.4/4) giga single-precision data per second. With a CGMA ratio of 1.0, the matrix multiplication kernel will execute at no more than 21.6 billion floating-point operations per second (gigaflops), as each floating

#计算输出矩阵的单个像素的值.

M访存 + N访存 / 乘法 + 加法 = 1

#字节单位.

#有效数据数

#由CGMA和内存的读写带宽 => 反向推出当前支持的计算强度.

#pointers. #这里width是矩阵的宽度.

```
__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 index 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];
                            ①              ②
Pd[Row*Width+Col] = Pvalue;
}
```

FIGURE 5.1

Matrix multiplication kernel using multiple blocks (see Figure 4.6).

respectable 相当不错的 / 值得尊敬的.

fraction 分数, 小部分

on-chip.

GPU中直接集成在芯片上的资源和组件.

point operation requires one single-precision global memory datum. Although 21.6 gigaflops is a respectable number, it is only a tiny fraction of the peak performance of 367 gigaflops for the G80. We will need to increase the CGMA ratio to achieve a higher level of performance for the kernel.

5.2 CUDA DEVICE MEMORY TYPES

CUDA supports several types of memory that can be used by programmers to achieve high CGMA ratios and thus high execution speeds in their kernels. Figure 5.2 shows these CUDA device memories. At the bottom of the figure, we see global memory and constant memory. These types of memory can be written (W) and read (R) by the host by calling application programming interface (API) functions. We have already introduced global memory in Chapter 3. The constant memory supports short-latency, high-bandwidth, read-only access by the device when all threads simultaneously access the same location.

Registers and shared memory in Figure 5.2 are on-chip memories. Variables that reside in these types of memory can be accessed at very high speed in a highly parallel manner. Registers are allocated to individual threads; each thread can only access its own registers. A kernel function

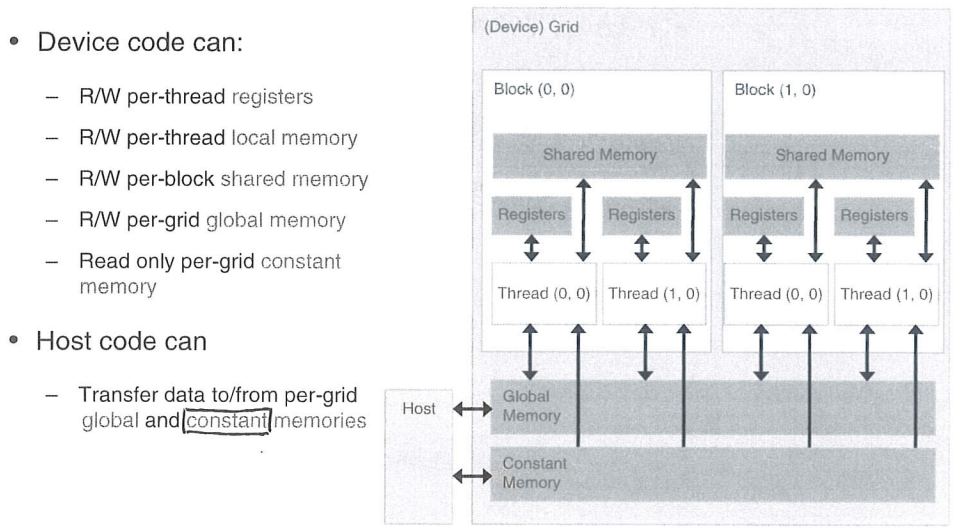


FIGURE 5.2

Overview of the CUDA device memory model.

visibility 能见度

duration 持续时间 / 时间长度

with in a kernel's invocation 在一个kernel函数的调用过程中.

Q. 生效范围.

① A single thread only

② All threads of a block

③ All threads of all grids.

Q. 生命周期

① Within a kernel's invocation.

② Throughout the entire application.

Q. Table 5.1 如何理解.

表头. Memory：变量在GPU中存储的物理位置.

内容.

Q1：为什么数组单独处理.

一般变量(标量)较小并且大小固定，而数组可能超出寄存器容量.

并且数组必须是通过索引动态访问到元素.

Tip：如果数组很小，还是会存在寄存器的.

Q2. Local Memory是哪层.

Local Memory本身是逻辑上的概念，还是映射到 Global Memory 实现的. (HBM，片外存储)

线程自己寄存器不足时，存储线程私有数据/数组.

typically uses registers to hold frequently accessed variables that are private to each thread. Shared memory is allocated to thread blocks; all threads in a block can access variables in the shared memory locations allocated to the block. Shared memory is an efficient means for threads to cooperate by sharing their input data and the intermediate results of their work. By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable.

Table 5.1 presents the CUDA syntax for declaring program variables into the various types of device memory. Each such declaration also gives its declared CUDA variable a scope and lifetime. Scope identifies the range of threads that can access the variable: by a single thread only, by all threads of a block, or by all threads of all grids. If the scope of a variable is a single thread, a private version of the variable will be created for every thread; each thread can only access its private version of the variable. For example, if a kernel declares a variable whose scope is a thread and it is launched with 1 million threads, then 1 million versions of the variable will be created so each thread initializes and uses its own version of the variable.

Lifetime specifies the portion of the program's execution duration when the variable is available for use: either within a kernel's invocation or throughout the entire application. If a variable's lifetime is within a kernel invocation, it must be declared within the kernel function body and will be available for use only by the kernel's code. If the kernel is invoked several times, the contents of the variable are not maintained across these invocations. Each invocation must initialize the variable in order to use them. On the other hand, if a variable's lifetime is throughout the entire

Table 5.1 CUDA Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

automatic scalar variables. 自动标量变量.
线程局部临时变量.

access congestions. 访问拥堵

collaborate with 与--合作

Automatic：线程局部变量.
生命周期 => kernel

application, it must be declared outside of any function body. The contents of the variable are maintained throughout the execution of the application and are available to all kernels.

As shown in Table 5.1, all automatic scalar variables declared in kernel and device functions are placed into registers. We refer to variables that are not arrays as *scalar* variables. The scopes of these automatic variables are within individual threads. When a kernel function declares an automatic variable, a private copy of that variable is generated for every thread that executes the kernel function. When a thread terminates, all of its automatic variables also cease to exist. In Figure 5.1, variables `tx`, `ty`, and `Pvalue` are all automatic variables and fall into this category. Note that accessing these variables is extremely fast and parallel, but one must be careful not to exceed the limited capacity of the register storage in the hardware implementations. We will address this point in Chapter 6.

Automatic array variables are not stored in registers.[1] Instead, they are stored into the global memory and incur long access delays and potential access congestions. The scope of these arrays is, like automatic scalar variables, limited to individual threads. That is, a private version of each automatic array is created for and used by every thread. Once a thread terminates its execution, the contents of its automatic array variables also cease to exist. From our experience, one seldom needs to use automatic array variables in kernel functions and device functions.

#所以它会有一个逻辑上的 Local Memory 的概念。

#不推荐。

If a variable declaration is preceded by the keyword `__shared__` (each __ consists of two _ characters), it declares a shared variable in CUDA. One can also add an optional `__device__` in front of `__shared__` in the declaration to achieve the same effect. Such declarations typically reside within a kernel function or a device function. The scope of a shared variable[2] is within a thread block; that is, all threads in a block see the same version of a shared variable. A private version of the shared variable is created for and used by each thread block during kernel execution. The lifetime of a shared variable is within the duration of the kernel. When a kernel terminates its execution, the contents of its shared variables cease to exist. Shared variables are an efficient means for threads within a block to collaborate with each other. Accessing shared memory is extremely fast

#这里说的是线程块内部和线程块之间的关系。

[1]There are some exceptions to this rule. The compiler may decide to store an automatic array into registers if all accesses are done with constant index values.

[2]The "`extern__shared__SharedArray[]`" notation allows the size of a shared memory array to be determined at runtime. Interested readers are referred to the CUDA Programming Guide for details.

is preceded by 在…之前

A is preceded by B. A之前有B.

Q.在CUDA中使用指针操作 Devire Memory.

相同点：对象本身存储在Devire Memory 中.

法一：cudaMalloc().

在主机端(host)手动进行分配，通过cudaMalloc()的第一个参数，也就是指向该片空间的指针，进行管理.

因为主机控制，所以是动态分配，通过cudaFree()释放

在运行时确定分配的空间大小，更适合大量数据.

法二：直接指向Devire Memory中的变量.

被动分配内存. float* ptr = &globalVar;

静态分配在编译时就必须确定大小，更适合小数据.

这个方法的本质是这样.

程序对数据大小是否已知进行不同处理

↙ ↘

已知 → 静态分配 → 指针直接指向

未知 → 动态分配 → 使用cudaMalloc()返回的参数.

∴书上本身是直接给出了结论，但因果是上面这样.

and highly parallel. CUDA programmers often use shared memory to hold the portion of global memory data that are heavily used in an execution phase of the kernel. One may need to adjust the algorithms used to create execution phases that focus heavily on small portions of the global memory data, as we will demonstrate with matrix multiplication in Section 5.3.

If a variable declaration is preceded by the keyword `__constant__`, it declares a constant variable in CUDA. One can also add an optional `__device__` in front of `__constant__` to achieve the same effect. Declaration of constant variables must be outside any function body. The scope of a constant variable is all grids, meaning that all threads in all grids see the same version of a constant variable. The lifetime of a constant variable is the entire application execution. Constant variables are often used for variables that provide input values to kernel functions. Constant variables are stored in the global memory but are cached for efficient access. With appropriate access patterns, accessing constant memory is extremely fast and parallel. Currently, the total size of constant variables in an application is limited at 65,536 bytes. One may need to break up the input data volume to fit within this limitation, as we will illustrate in Chapter 8.

#会放在constant Memory

A variable whose declaration is preceded only by the keyword `__device__` is a global variable and will be placed in global memory. Accesses to a global variable are slow; however, global variables are visible to all threads of all kernels. Their contents also persist through the entire execution. Thus, global variables can be used as a means for threads to collaborate across blocks. One must, however, be aware of the fact that there is currently no way to synchronize between threads from different thread blocks or to ensure data consistency across threads when accessing global memory other than terminating the current kernel execution. Therefore, global variables are often used to pass information from one kernel invocation to another kernel invocation.

#其实可以和"__constant__"的情况合并到一起看.

Note that there is a limitation on the use of pointers with CUDA variables declared in device memory. In general, pointers are used to point to data objects in global memory. There are two typical ways in which pointer usage arises in kernel and device functions. First, if an object is allocated by a host function, the pointer to the object is initialized by `cudaMalloc()` and can be passed to the kernel function as a parameter; for example, the parameters **Md**, **Nd**, and **Pd** in Figure 5.1 are such pointers. The second type of usage is to assign the address of a variable declared in the global memory to a pointer variable; for example, the statement `{float*ptr = &GlobalVar;}` in a kernel function assigns the address of `GlobalVar` into the automatic pointer variable `ptr`.

#指针本质上是对内存的随机索引,其它的缓存层次也完全可以支持指针的.

intrinsic 内在的，固有本质

partition 分区，划分

draws on 借鉴，吸引

analogy 类比，比喻

term 专业术语.

criterion 标准/依据.

[1]. 专业术语"tile"借用了一个比喻.

全局内存的数据通常可以用子集覆盖.

5.3 A STRATEGY FOR REDUCING GLOBAL MEMORY TRAFFIC

We have an intrinsic tradeoff in the use of device memories in CUDA. Global memory is large but slow, whereas the shared memory is small but fast. A common strategy is to partition the data into subsets called *tiles* such that each tile fits into the shared memory. The term *tile* draws on the analogy that a large wall (i.e., the global memory data) can often be covered by tiles (i.e., subsets that each can fit into the shared memory). An important criterion is that the kernel computations on these tiles can be done independently of each other. Note that not all data structures can be partitioned into tiles given an arbitrary kernel function.

The concept of tiling can be illustrated with the matrix multiplication example. Figure 5.3 shows a small example of matrix multiplication using multiple blocks. It corresponds to the kernel function in Figure 5.1. This example assumes that we use four 2×2 blocks to compute the **Pd** matrix. Figure 5.3 highlights the computation done by the four threads of block (0, 0). These four threads compute $\mathbf{Pd}_{0,0}$, $\mathbf{Pd}_{1,0}$, $\mathbf{Pd}_{0,1}$, and $\mathbf{Pd}_{1,1}$. The accesses to the **Md** and **Nd** elements by thread (0, 0) and thread (1, 0) of block (0, 0) are highlighted with black arrows.

Figure 5.4 shows the global memory accesses done by all threads in block(0,0). The threads are listed in the horizontal direction, with the time of access increasing downward in the vertical direction. Note that each

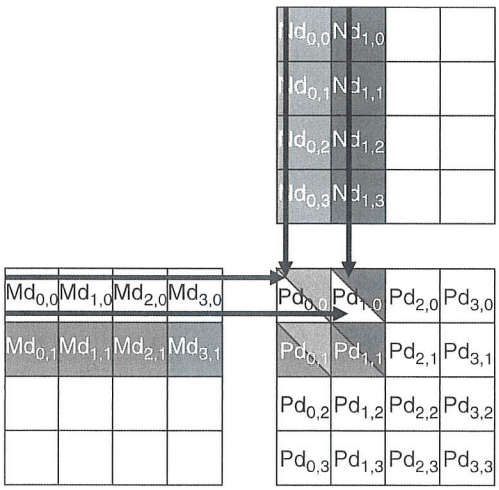


FIGURE 5.3

A small example of matrix multiplication using multiple blocks.

overlap 重叠

verify 验证，确认

potential reduction 可能的降低

be proportional to 与…成比例

#从M,N矩阵的单个元素角度.

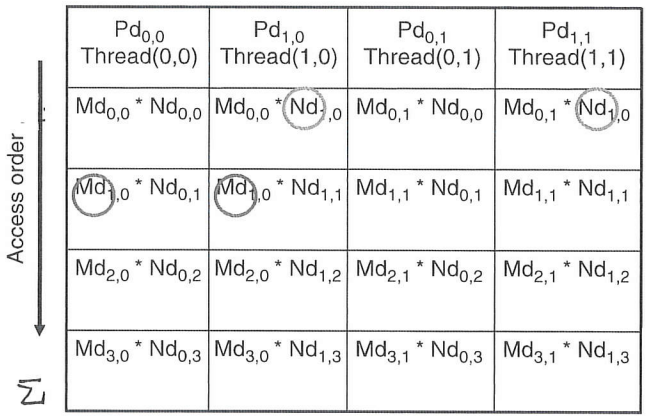


FIGURE 5.4

Global memory accesses performed by threads in block(0,0).

#说明缓存的前提条件.

thread accesses four elements of **Md** and four elements of **Nd** during its execution. Among the four threads highlighted, there is a significant overlap in terms of the **Md** and **Nd** elements they access; for example, thread(0,0) and thread(1,0) both access $\mathbf{Md}_{1,0}$ as well as the rest of row 0 of **Md**. Similarly, $\text{thread}_{1,0}$ and $\text{thread}_{1,1}$ both access $\mathbf{Nd}_{1,0}$ as well as the rest of column 1 of **Nd**.

The kernel in Figure 5.1 is written so both thread(0,0) and thread(1,0) access these **Md** row 0 elements from the global memory. If we can somehow manage to have thread(0,0) and thread(1,0) collaborate so these **Md** elements are only loaded from global memory once, we can reduce the total number of accesses to the global memory by half. In general, we can see that every **Md** and **Nd** element is accessed exactly twice during the execution of block(0,0); therefore, if all four threads could collaborate in their accesses to global memory, we could potentially reduce the traffic to the global memory by half.

#eg.以书中的4×4矩阵为例.实际上每个元素要访问4次的.

The reader should verify that the potential reduction in global memory traffic in the matrix multiplication example is proportional to the dimension of the blocks used. With $N \times N$ blocks, the potential reduction of global memory traffic would be N. That is, if we use 16×16 blocks, we could potentially reduce the global memory traffic to 1/16 through collaboration between threads.

We now present an algorithm where threads collaborate to reduce the traffic to the global memory. The basic idea is to have the threads collaboratively load **Md** and **Nd** elements into the shared memory before they

delineated 描述的，勾画的.

Q: Figure 5.3 中元素的编号.

$Md_{i,j}$: $Md_{0,0} \rightarrow Md_{3,0} \rightarrow Md_{3,1}$ 列优先存储

$Nd_{i,j}$: $Nd_{0,0} \rightarrow Nd_{3,0} \rightarrow Nd_{1,3}$. 列优先. 图 同理.

individually use these elements in their dot product calculation. Keep in mind that the size of the shared memory is quite small and one must be careful not to exceed the capacity of the shared memory when loading these **Md** and **Nd** elements into the shared memory. This can be accomplished by dividing the **Md** and **Nd** matrices into smaller tiles. The size of these tiles is chosen so they can fit into the shared memory. In the simplest form, the tile dimensions equal those of the block, as illustrated in Figure 5.5.

In Figure 5.5, we divide **Md** and **Nd** into 2×2 tiles, as delineated by the thick lines. The dot product calculations performed by each thread are now divided into phases. In each phase, all threads in a block collaborate to load a tile of **Md** and a tile of **Nd** into the shared memory. This is done by having every thread in a block to load one **Md** element and one **Nd** element into the shared memory, as illustrated in Figure 5.6. Each row of Figure 5.6 shows the execution activities of a thread. (Note that time now progresses from left to right.) We only need to show the activities of threads in block(0,0); the other blocks all have the same behavior. The shared memory array for the **Md** elements is called **Mds**. The shared memory array for the **Nd** elements is called **Nds**. At the beginning of Phase 1, the four threads of block(0,0) collaboratively load a tile of **Md** into shared memory; thread(0,0) loads $\mathbf{Md}_{0,0}$ into $\mathbf{Mds}_{0,0}$, thread(1,0) loads $\mathbf{Md}_{1,0}$ into $\mathbf{Mds}_{1,0}$, thread(0,1) loads $\mathbf{Md}_{0,1}$ into $\mathbf{Mds}_{0,1}$, and thread(1,1) loads $\mathbf{Md}_{1,1}$ into $\mathbf{Mds}_{1,1}$. Look at the

同一个block中的所有线程 eg. block(0,0)
<Md0,0, Md1,0, Md0,1, Md1,1>
总之就是M, N矩阵中左上角的四个元素.

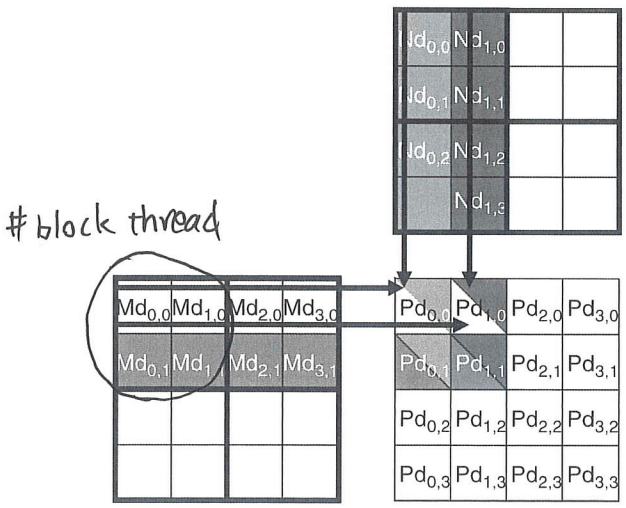


FIGURE 5.5

Tiling **Md** and **Nd** to utilize shared memory.

#Tip: 这里的下标s表示已经被移动到 Shared Memory 中.

#注: 横向看才是一个 thread 中完整的执行流程.

#Tip: 这里线程的执行并不是能一次结束的,是存在等待的

#只从一个线程来看,它的输入仍然是按照行来加载的.

	Phase 1			Phase 2		
$T_{0,0}$	$\mathbf{Md}_{0,0}$ ↓ $Mds_{0,0}$	$\mathbf{Nd}_{0,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	$\mathbf{Md}_{2,0}$ ↓ $Mds_{0,0}$	$\mathbf{Nd}_{0,2}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md}_{1,0}$ ↓ $Mds_{1,0}$	$\mathbf{Nd}_{1,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	$\mathbf{Md}_{3,0}$ ↓ $Mds_{1,0}$	$\mathbf{Nd}_{1,2}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md}_{0,1}$ ↓ $Mds_{0,1}$	$\mathbf{Nd}_{0,1}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	$\mathbf{Md}_{2,1}$ ↓ $Mds_{0,1}$	$\mathbf{Nd}_{0,3}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md}_{1,1}$ ↓ $Mds_{1,1}$	$\mathbf{Nd}_{1,1}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{Md}_{3,1}$ ↓ $Mds_{1,1}$	$\mathbf{Nd}_{1,3}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

FIGURE 5.6

Execution phases of a tiled matrix multiplication.

second column of Figure 5.6. A tile of **Nd** is also loaded in a similar manner, as shown in the third column of Figure 5.6.

After the two tiles of **Md** and **Nd** are loaded into the shared memory, these values are used in the calculation of the dot product. Note that each value in the shared memory is used twice; for example, the $\mathbf{Md}_{1,1}$ value, loaded by $thread_{1,1}$ into $\mathbf{Mds}_{1,1}$, is used twice: once by $thread_{0,1}$ and once by $thread_{1,1}$. By loading each global memory value into shared memory so it can be used multiple times we reduce the number of accesses to the global memory. In this case, we reduce the number of accesses to the global memory by half. The reader should verify that the reduction is by a factor of N if the tiles are $N \times N$ elements.

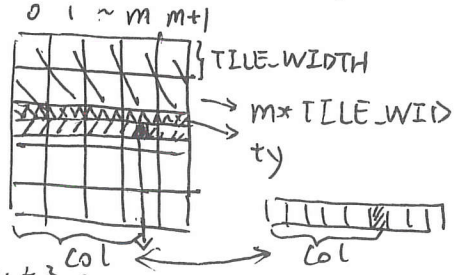
Note that the calculation of each dot product in Figure 5.6 is now performed in two phases shown as Phase 1 and Phase 2. In each phase, products of two pairs of the input matrix elements are accumulated into the `Pvalue` variable. The first phase calculation is shown in the fourth column of Figure 5.6; the second phase is in the seventh column. In general, if an input matrix is of dimension N and the tile size is `TILE_WIDTH`, the dot product would be performed in `N/TILE_WIDTH` phases. The creation of these phases is key to the reduction of accesses to the global memory. With each phase focusing on a small subset of the input matrix values, the threads can collaboratively load the subset into the shared memory and use the values in the shared memory to satisfy their overlapping input needs in the phase.

N/TILE_WIDTH 因为需要等待完整的一行一列数据后, Pij的计算才能完整

be as important for 与…一样重要

Q. Nds 的计算过程.

Nd[(m*TILE_WIDTH+ty)*Width+Col]



外部:(m*TILE_WIDTH+ty) => ROW.

*Width. 找到前面横向的所有元素.

内部:在找到具体的某行后,+Col.

Q:为什么Mds可以直接用Row计算

而Nds可以用Col计算.

首先,这里是索引访问. Md和Nd是列存储或行存储不会影响它的索引.

影响的只是索引效率.

其次,根据矩阵乘法的规则.左行右列.

但是这个是要动态体现的.

也就是随着循环,m增大. Md读行

Nd读列.如果只是读取一个元素的话.

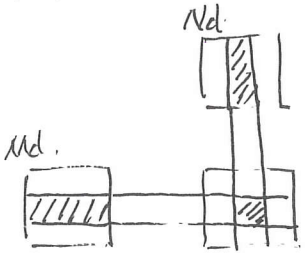
其实直接用ROW,COL都可以.

Q. the tiled kernel function.

在CUDA kernel中实现了Tile策略的函数.

~~Q. 为什么Mds和Nds分别使用Row和Col计算~~

~~这个要从输入状态看,左乘右乘的区别.~~



~~根据数学规律.左行乘右~~

~~所以Nd天生用行算,Nd同理.~~

无论是Md还是Nd,都是行计算.

Q.为什么要执行__syncthreads().

线程同步函数.确保线程块block内部所有线程都在同一位置等待.直到所有线程完成当前操作.eg.完成加载后再计算.

加载与执行的代码虽然是顺序写的但执行不一定

Q.这里Mds[ty][tx]的下标选择.

保证线程与共享内存的位置一一对应

后续线程在执行的时候访问自己对应的数据就是依赖这个下标.

本质上只是人为设计的规定.如果整个程序都是你写.你可以自由定义一套映射方式.

但是使用(tx,ty)最通用.合理.直观.

#Tip:这里ty表示行,tx表示列,结合Figure3下标.

Note also that **Mds** and **Nds** are reused to hold the input values. In each phase, the same **Mds** and **Nds** are used to hold the subset of **Md** and **Nd** elements used in the phase. This allows a much smaller shared memory to serve most of the accesses to global memory. This is because each phase focuses on a small subset of the input matrix elements. Such focused access behavior is called *locality*. When an algorithm exhibits locality, there is an opportunity to use small, high-speed memories to serve most of the accesses and remove these accesses from the global memory. Locality is as important for achieving high performance in multicore central processing units (CPUs) as in many-core graphics processing units (GPUs). We will return to the concept of locality in Chapter 6.

We are now ready to present the tiled kernel function that uses shared memory to reduce the traffic to global memory. The kernel shown in Figure 5.7 implements the phases illustrated in Figure 5.6. In Figure 5.7, Line 1 and Line 2 declare **Mds** and **Nds** as shared memory variables.

#CUDA会自发启动多个线程执行

Width: 矩阵的宽
Width 控制主循环的迭代次数.

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;

7.   float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.   for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collaborative loading of Md and Nd tiles into shared memory
9.     Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.    Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
11.    __syncthreads();

12.    for (int k = 0; k < TILE_WIDTH; ++k)
13.      Pvalue += Mds[ty][k] * Nds[k][tx];
14.    __syncthreads();
     }
15.  Pd[Row*Width + Col] = Pvalue;
}
```

#以 TILE_WIDTH 规模的大小写入 Share Mem

#定位到全局某个元素位置.

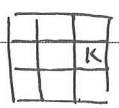
#从Md写入Mds 实现写入Shared Mem的过程.

#遍历这矩阵中所有block.

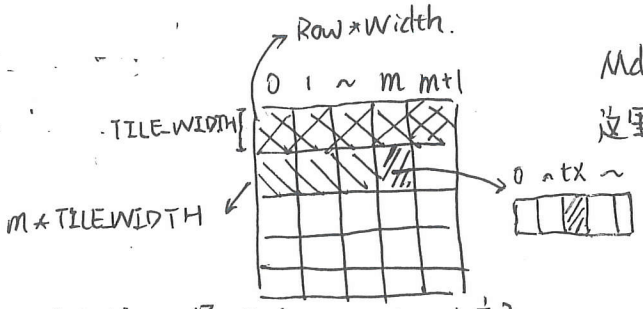
#Tip: 注意这里 line_9, line_10 的赋值.
Mds[i][j] = Md[k]
∴左侧是二维索引, 右侧是一维索引

FIGURE 5.7
Tiled matrix multiplication kernel using shared memories.

eg.



K = row * Width + col



这里的tx是指在一个 tile 内部.

Md[Row * Width + (m * TILE_WIDTH + tx)]
这里仍然是进行一个全局的线性索引
因为 ROW 是全局性, 所以只考虑列.
外部: Row * Width | 内部: tx
偏移: m * TILE_WIDTH

cease to 停止，不再.

cease to exist. 停止存在

Q: Automatic Variables.

有局部作用域&自动生命周期的变量.

只有 Automatic Variables 才能存在寄存器中

因为是当前线程独享的，所以不用考虑同步.

对比 Shared Variables.

Global Variables.

Constant Variables.

都不是单线程独享，所以他们都不能存在寄存器中.

Recall that the scope of shared memory variables is a block. Thus, all threads of a block have access to the same **Mds** and **Nds** arrays. This is important, as all threads in a block must have access to the **Md** and **Nd** values loaded into **Mds** and **Nds** by their peers so they can use these values to satisfy their input needs.

Lines 3 and 4 save the `threadIdx` and `blockIdx` values into automatic variables and thus into registers for fast access. Recall that automatic scalar variables are placed into registers. Their scope is in each individual thread. That is, one private version of `tx`, `ty`, `bx`, and `by` is created by the runtime system for each thread. They will reside in registers that are accessible by one thread. They are initialized with the `threadIdx` and `blockIdx` values and used many times during the lifetime of thread. Once the thread ends, the values of these variables also cease to exist.

Lines 5 and 6 determine the row index and column index of the **Pd** element that the thread is to produce. As shown in Figure 5.8, the column (x) index of the **Pd** element to be produced by a thread can be calculated as `bx*TILE_WIDTH + tx`. This is because each block covers `TILE_WIDTH` elements in the x dimension. A thread in block `bx` would have bx blocks of threads, or `bx*TILE_WIDTH` threads, before it; they cover `bx*TILE_WIDTH` elements of **Pd**. Another `tx` thread within the same block would cover another `tx` element of **Pd**. Thus, the thread with `bx` and `tx` should be responsible for calculating the **Pd** element whose x index is `bx*TILE_WIDTH + tx`. For the example in Figure 5.5, the x index of the **Pd** element to be calculated by thread(1,0) of block(0,1) is 0*2 + 1 = 1. Similarly, the y index can be calculated as `by*TILE_WIDTH + ty`. In Figure 5.5, the y index of the **Pd** element to be calculated by thread(1,0) of block(0,1) is 1*2 + 0 = 2. Thus, the **Pd** element to be calculated by this thread is $\mathbf{Pd}_{1,2}$.

Line 8 of Figure 5.7 marks the beginning of the loop that iterates through all the phases of calculating the final **Pd** element. Each iteration of the loop corresponds to one phase of the calculation shown in Figure 5.6. The `m` variable indicates the number of phases that have already been done for the dot product. Recall that each phase uses one tile of **Md** and one tile of **Nd** elements; therefore, at the beginning of each phase, `m*TILE_WIDTH` pairs of **Md** and **Nd** elements have been processed by previous phases.

Recall that all threads in a grid execute the same kernel function. The `threadIdx` variable allows them to identify the part of the data they are to process. Also recall that the thread with `by = blockIdx.y` and `ty = threadIdx.y` processes row (`by*TILE_WIDTH + ty`) of **Md**, as shown at the left side of Figure 5.8. Line 5 stores this number into the `Row` variable of each thread. Likewise, the thread with `bx = blockIdx.x` and

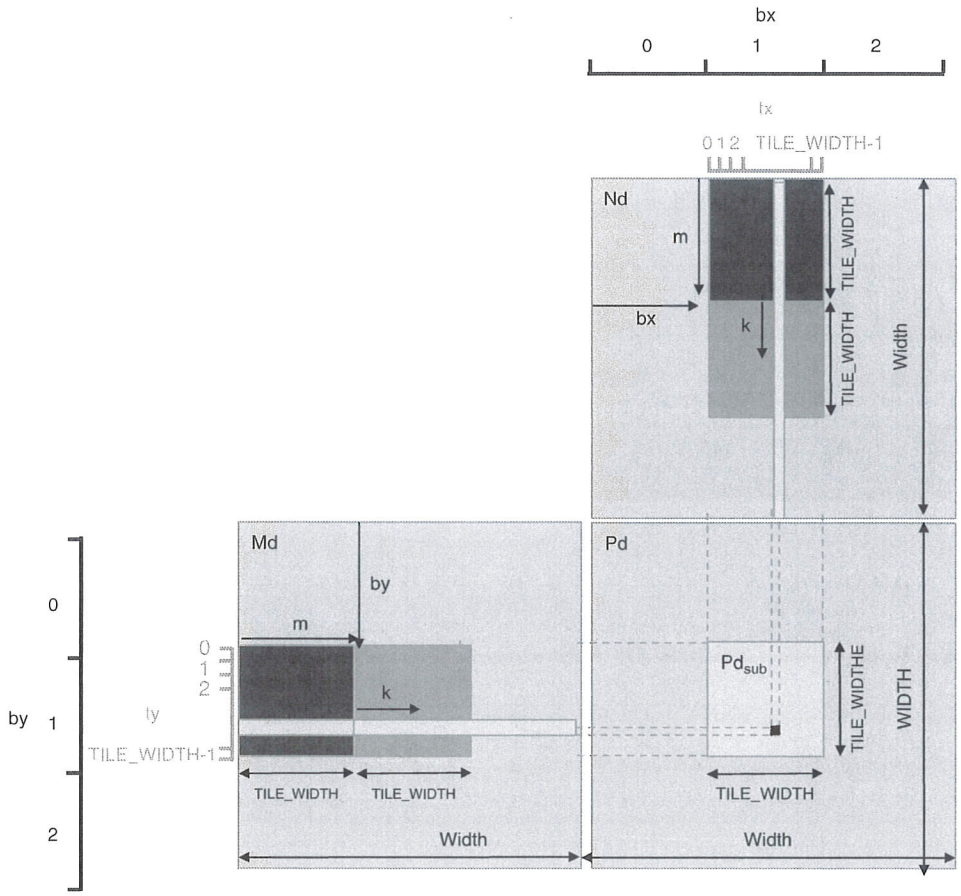


FIGURE 5.8

Calculation of the matrix indices in tiled multiplication.

`tx = threadIdx.x` processes the column (`bx*TILE_WIDTH + tx`) of **Nd**, as shown at the top of Figure 5.8. Line 6 stores this number into the `Col` variable of each thread. These variables will be used when the threads load **Md** and **Nd** elements into the shared memory.

In each phase, Line 9 loads the appropriate **Md** element into the shared memory. Because we already know the row index of **Md** and column index of **Nd** elements to be processed by the thread, we will focus on the column index of **Md** and row index of **Nd**. As shown in Figure 5.8, each block has `TILE_WIDTH`^{2} threads that will collaborate to load `TILE_WIDTH`^{2} **Md** elements into the shared memory. Thus, all we need to do is to assign each thread to load one **Md** element. This is conveniently done using the

substantial 大量，实质。

Q：对比直接使用for循环 /__global__有什么不同。
各有优劣，使用__global__能提高灵活性和并行粒度，但也有kernel的启动开销
(而且很多有前后等待的问题，eg. 先搬数据，后计算)
直接用循环更直观。

Tip：Figure 5.7中循环也是用到CUDA并行的，并不是因为代码的循环，它就线性了。

#Tip: 在Figure 5.7中使用了两个循环完成对整个矩阵的加载，但是没有发挥GPU/CUDA的能力。应该把5.7中的两个循环抽象出来作为__global__函数执行。

blockIdx and threadIdx. Note that the beginning index of the section of **Md** elements to be loaded is m*TILE_WIDTH; therefore, an easy approach is to have every thread load an element from that point identified by the threadIdx value. This is precisely what we have in Line 9, where each thread loads Md[Row*Width + (m*TILE_WIDTH + tx)]. Because the value of Row is a linear function of ty, each of the TILE_WIDTH^2 threads will load a unique **Md** element into the shared memory. Together, these threads will load the orange square subset of **Md** shown in Figure 5.8. The reader should use the small example in Figures 5.5 and 5.6 to verify that the address calculation works correctly.

Line 11 calls the __syncthreads() barrier synchronization function to make sure that all threads in the same block have completed loading the **Md** and **Nd** tiles into **Mds** and **Nds**. Once the tiles of **Md** and **Nd** are loaded in **Mds** and **Nds**, the loop in Line 12 performs the phases of the dot product based on these elements. The progression of the loop for thread (tx, ty) is shown in Figure 5.8, with the direction of the **Md** and **Nd** data usage marked with k, the loop variable in Line 12. Note that the data will be accessed from **Mds** and **Nds**, the shared memory arrays holding these **Md** and **Nd** elements. Line 14 calls the __syncthreads() barrier synchronization function to make sure that all threads of the block have completed using the **Mds** and **Nds** contents before any of them loops back to the next iteration and loads the next tile of **Md** and **Nd**.

The benefit of the tiled algorithm is substantial. For matrix multiplication, the global memory accesses are reduced by a factor of TILE_WIDTH. If one uses 16×16 tiles, we can reduce the global memory accesses by a factor of 16. This reduction allows the 86.4-GB/s global memory bandwidth to serve a much larger floating-point computation rate than the original algorithm. More specifically, the global memory bandwidth can now support [(86.4/4) × 16] = 345.6 gigaflops, very close to the peak floating-point performance of the G80. This effectively removes global memory bandwidth as the major limiting factor of matrix multiplication performance.

5.4 MEMORY AS A LIMITING FACTOR TO PARALLELISM

Although CUDA registers, shared memory, and constant memory can be extremely effective in reducing the number of accesses to global memory, one must be careful not to exceed the capacity of these memories. Each CUDA device offers a limited amount of CUDA memory, which limits

granularity 颗粒度.

simultaneously 同时.

the number of threads that can simultaneously reside in the streaming multiprocessors for a given application. In general, the more memory locations each thread requires, the fewer the number of threads that can reside in each SM and thus the fewer number of threads that can reside in the entire processor.

In the G80, each SM has 8K (= 8192) registers, which amounts to 128K (= 131,072) registers for the entire processor. This is a very large number, but it only allows each thread to use a very limited number of registers. Recall that each G80 SM can accommodate up to 768 threads. In order to fill this capacity, each thread can use only 8K/768 = 10 registers. If each thread uses 11 registers, the number of threads that can be executed concurrently in each SM will be reduced. Such reduction is done at the block granularity. For example, if each block contains 256 threads, the number of threads will be reduced 256 at a time; thus, the next lower number of threads from 768 would be 512, a 1/3 reduction of threads that can simultaneously reside in each SM. This can greatly reduce the number of warps available for scheduling, thus reducing the processor's ability to find useful work in the presence of long-latency operations.

Shared memory usage can also limit the number of threads assigned to each SM. In the G80, there are 16 kilobytes (kB) of shared memory in each SM. Keep in mind that shared memory is used by blocks, and recall that each SM can accommodate up to 8 blocks. In order to reach this maximum, each block must not use more than 2 kB of shared memory. If each block uses more than 2 kB of memory, the number of blocks that can reside in each SM is such that the total amount of shared memory used by these blocks does not exceed 16 kB; for example, if each block uses 5 kB of shared memory, no more than 3 blocks can be assigned to each SM.

For the matrix multiplication example, shared memory can become a limiting factor. For a tile size of 16×16, each block requires 16 × 16 × 4 = 1 kB of storage for **Mds**. Another 1 kB is needed for **Nds**. Thus, each block uses 2 kB of shared memory. The 16-kB shared memory allows 8 blocks to simultaneous reside in an SM. Because this is the same as the maximum allowed by the threading hardware, shared memory is not a limiting factor for this tile size. In this case, the real limitation is the threading hardware limitation that only 768 threads are allowed in each SM. This limits the number of blocks in each SM to 3. As a result, only 3 × 2 kB = 6 kB of the shared memory will be used. These limits do change from device generation to the next but are properties that can be determined at runtime;

for example, the GT200 series of processors can support up to 1024 threads in each SM. See Appendix B for shared memory and other resource limitations in other types of GPU devices.

5.5 SUMMARY

In summary, CUDA defines registers, shared memory, and constant memory that can be accessed at higher speed and in a more parallel manner than the global memory. Using these memories effectively will likely require redesign of the algorithm. We used matrix multiplication as an example to illustrate tiled algorithms, a popular strategy to enhance the locality of data access and enable effective use of shared memory. We demonstrated that, with 16×16 tiling, global memory accesses are no longer the major limiting factor for matrix multiplication performance.

It is, however, important for CUDA programmers to be aware of the limited sizes of these types of memory. Their capacities are implementation dependent. Once their capacities are exceeded, they become limiting factors for the number of threads that can be simultaneously executing in each SM. The ability to reason about hardware limitations when developing an application is a key aspect of computational thinking. The reader is also referred to Appendix B for a summary of resource limitations of several different devices.

Although we introduced tiled algorithms in the context of CUDA programming, it is an effective strategy for achieving high performance in virtually all types of parallel computing systems. The reason is that an application must exhibit locality in data access in order to make effective use of high-speed memories in these systems. In a multicore CPU system, for example, data locality allows an application to effectively use on-chip data caches to reduce memory access latency and achieve high performance. Readers will find the tiled algorithm useful when they develop parallel applications for other types of parallel computing systems using other programming models.

Our goal for this chapter was to introduce the different types of CUDA memory. We introduced the tiled algorithm as an effective strategy for using shared memory, but we have not discussed the use of constant memory. We will explain the use of constant memory in Chapter 8. Furthermore, we will study a different form of tiling that enables more effective use of registers in Chapter 9.

5.6 EXERCISES

5.1 Consider the matrix addition where each element of the output matrix is the sum of the corresponding elements of the two input matrices. Can one use shared memory to reduce the global memory bandwidth consumption? *Hint:* Analyze the elements accessed by each thread and see if there is any commonality between threads.

5.2 Draw the equivalent of Figure 5.4 for an 8×8 matrix multiplication with 2×2 tiling and 4×4 tiling. Verify that the reduction in global memory bandwidth is indeed proportional to the dimension size of the tiles.

5.3 What type of incorrect execution behavior can happen if one forgets to use `__syncthreads()` function in the kernel of Figure 5.7? Note that there are two calls to `__syncthreads()`, each for a different purpose.

5.4 Assuming that capacity were not an issue for registers or shared memory, give one case that it would be valuable to use shared memory instead of registers to hold values fetched from global memory. Explain your answer.