第 4 章　CUDA 程序的优化

4.1　CUDA 程序优化概述

CUDA 程序优化的最终目的，是以最短的时间，在允许的误差范围内完成给定的计算任务。在这里，“最短的时间”是指整个程序的运行时间，更侧重于计算的吞吐量，而不是单个数据的延迟。在开始考虑使用 GPU 和 CPU 协同计算之前，应该先粗略地评估使用 CUDA 是否能达到预想的效果，包括以下几个方面：

1. 精度

目前，GPU 的单精度计算性能要远远超过双精度计算性能，整数乘法、除法、求模等运算的指令吞吐量也较为有限。在科学计算中，由于需要处理的数据量巨大，往往只有在采用双精度或者四精度时才能获得可靠的结果。目前，采用 Tesla 架构的 GPU 还不能很好地满足高精度计算的需求。如果应用程序需要很高的精度，或者需要进行多轮迭代，笔者建议只在关键的步骤中使用双精度，而在其他部分仍然使用单精度浮点以获得指令吞吐量和精度的平衡。如果应用程序对精度有更高的要求，那么现在的架构还不能获得太高的加速比。不过，在 2010 年将会普及的下一代架构中，双精度浮点和整数处理能力将有很大的提升，这种情况会有根本性的改变。

2. 延迟

目前，CUDA 还不能单独为某个处理核心分配任务，因此必须先缓冲一定量的数据，再交给 GPU 进行计算。这样的方式可以获得很高的数据吞吐量，不过单个数据经过缓冲、传输到 GPU 计算、再拷贝回内存的延迟就比直接由 CPU 进行串行处理要长很多。如果对应用实时性要求很高，比如必须在数十微秒内完成对一个输入的处理，那么使用 CUDA 可能会影响系统的整体性能。对于要求实现人机实时交互的系统，应该将延迟控制在数十毫秒的量级，以及时响应用户的输入。通过减小缓冲，可以减小延迟，但缓冲的大小至少应该保证每个内核程序处理的一批数据能够让 GPU 满负荷工作。在大多数情况下，如果应用要求的计算吞吐量大到需要由中高端 GPU 才能实时实现，那么在投入相同成本的前提下，是很难使用 CPU 相近效果的。如果确实对实时性和吞吐量都有很高要求，应该考虑 ASIC、FPGA 或者 DSP 实现，这需要更多的投入，更长的开发时间和硬件开发经验。

3. 计算量

如果计算量太小，那么使用 CUDA 是不划算的。衡量计算量有绝对和相对两种方式。

从绝对量来说，如果待优化的程序使用频率比较低，并且每次调用需要的时间也可以接受，那么使用 CUDA 优化并不会显著改善使用体验。对于一些计算量非常小（整个程序在 CPU 上可以在几十毫秒内完成）的应用来说，使用 CUDA 计算时在 GPU 上的执行时间无法隐藏访存和数据传输的延迟，此时整个应用程序需要的时间反而会比 CPU 更长。此外，虽然 GPU 的单精度浮点处理能力和显存带宽都远远超过了 CPU，但由于 GPU 使用 PCI-E 总线与主机连接，

因此它的输入和输出的吞吐量受到了 IO 带宽的限制。当要计算的问题的计算密集度很低时，执行计算的时间远远比 IO 花费的时间短，那么整个程序的瓶颈就会出现在 PCI-E 带宽上。此时无论如何提高浮点处理能力和显存带宽，都无法提高系统性能。Tesla C1060 带宽与延迟的比较如图 4-1 所示。

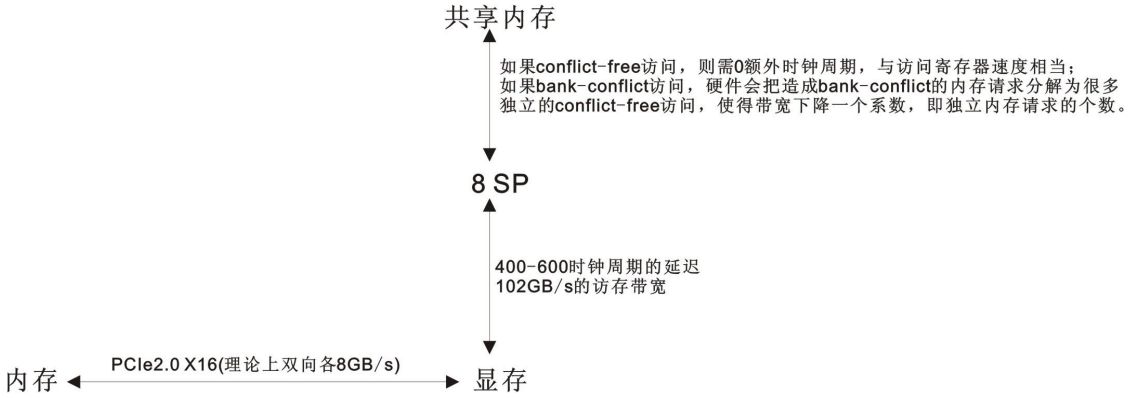


图 4-1　Tesla C1060 带宽与延迟比较

从相对计算量来说，如果可以并行的部分在整个应用中所占的比例不大，那么 GPU 对程序整体性能的提高也不会非常明显。如果整个应用中串行部分占用时间较长，而并行部分较短，那么也需要考虑是否值得使用 GPU 进行并行计算。例如，假设一个程序总的执行时间为 1.0，其中串行部分占 0.8，而并行部分只占 0.2，那么使用 GPU 将并行部分加速 10 倍，总的执行时间也只能从 1.0 降低到 0.82。即使是在 CPU 和 GPU 可以同时并行计算的应用中，执行时间也至少是 CPU 串行计算需要的 0.8。只有在并行计算占用了绝大多数计算时间的应用中，使用 CUDA 加速才能获得很高的加速比。不过，随着 GPU+CPU 并行计算的普及和 GPU 架构的进一步改进，未来可能即使只能获得较小的加速比，也会由 GPU 来执行更多的计算任务。

完成对 GPU 加速效果的粗略评估后，就可以开始着手编写程序了。为了在最短的时间内完成计算，需要考虑算法、并行划分、指令流吞吐量、存储器带宽等多方面因素。总的来说，优秀的 CUDA 程序应该同时具有以下几个特征：

- 在给定的数据规模下，选用算法的计算复杂度不明显高于最优的算法。
- active warp 的数量能够让 SM 满载，并且 active block 数量大于 2，能够有效的隐藏访存延迟。
- 当瓶颈出现在指令流（主要是运算）时，指令流的效率已经经过了充分优化。
- 当瓶颈出现在访存或者 IO 时，程序已经选用了恰当的存储器来储存数据，并使用了适当的存储器访问方式，以获得最大带宽。

按照开发流程的先后顺序，CUDA 程序的编写与优化需要解决以下问题：

（1）确定任务中的串行部分和并行部分，选择合适的算法。首先，需要将问题分为几个步骤，并确定哪些步骤可以用并行算法实现，并确定要使用的算法。

（2）按照算法确定数据和任务的划分方式，将每个需要并行实现的步骤映射为一个满足 CUDA 两层并行模型的内核函数。在这里就要尽量让每个 SM 上拥有至少 6 个活动 warp 和至少 2 个活动线程块。

（3）编写一个能够正确运行的程序，作为优化的起点。程序必须能够稳定运行，不能发生存储器泄漏的情况。为了保证结果正确，在必要的时候必须使用 memory fence、同步、原子操作等功能以及 volatile 关键字。在精度不足或者发生溢出时必须使用双精度浮点或者更长的整数类型。

（4）优化显存访问，避免显存带宽成为瓶颈。在显存带宽得到完全优化前，其他优化不会产生明显结果。显存访问优化中可以使用的技术包括：

- 将可以采用相同的 block 和 grid 维度实现的几个 kernel 合并为一个，减少对显存的访问。
- 除非非常必要，应该尽力避免将线程私有变量分配到 local memory。
- 为满足合并访问，采用 cudaMallocPitch()或者 cudaMalloc3D()分配显存。
- 为满足合并访问，对数据类型进行对齐（使用__align）。
- 为满足合并访问，保证访问的首地址从 16 的整数倍开始，如果可能，尽量让每个线程一次读的数据字长都为 32bit。
- 在数据只会被访问一次，并且满足合并访问的情况下可以考虑使用 zerocopy。
- 在某些情况下，考虑存储器控制器负载不均衡造成分区冲突的影响。
- 使用拥有缓存的常数存储器和纹理存储器提高某些应用的实际带宽。

（5）优化指令流。在编译过程中，编译器对代码会进行一些优化。但是程序员很难直接控制编译器对代码的优化，所以指令流优化不一定能获得立竿见影的效果。但是，仍然有一些准则可以参考，包括：

- 如果只需要少量线程进行操作，一定记得要使用类似“if threadID < N”的方式，避免多个线程同时运行占用更长时间或者产生错误结果。
- 在不会出现不可接受的误差的前提下采用 CUDA 算术指令集中的快速指令。
- 使用#unroll，让编译器能够有效地展开循环。
- 采用原子函数实现更加复杂的算法，并保证结果的正确性。
- 避免多余的同步。
- 如果不产生 bank conflict 的算法不会造成算法效率的下降或者非合并访问，就应该避免 bank conflict。

（6）资源均衡。调整 shared memory 和 register 的使用量。为了使程序能够获得更高的 SM 占用率，应该调整每个线程处理的数据数量、shared memory 和 register 的使用量。这需要在三者间进行调整。当线程处理的子任务间有一些完全相同的部分时，应该只使用少量线程来完成公共部分的计算，再将公用数据通过 shared memory 广播给所有线程。为了获得更高的 SM 占用率，必须控制每个线程的 shared memory 和 register 的使用量。通过调整 block 大小，修改算法和指令，以及动态分配 shared memory，都可以提高 shared 的使用效率。而减小 register 的使用则相对困难，因为 register 的使用量并不是由内核程序中声明的变量多少决定，而是由内核程序中使用寄存器最多的时刻的用量决定的。由于编译器会尽量减小寄存器的用量，因此实际使用的寄存器有可能会小于在程序中声明的量。但是在通常情况下，由于需要暂存中间结果并且一些指令也需要更多的寄存器，一般寄存器用量都大于内核程序中声明的私有变量的总数量。使用以下方法可能可以节约一些寄存器的使用：使用 shared memory 存储变量；使用括号更加明确地表示每个变量的生存周期；用对[u]long 型的处理代替对两个相邻的[u]short 型或者四个相邻的[u]char 型的处理；使用占用寄存器较小的等效指令代替原有指令，如用__sin 函

数代替 sin 函数。不过，由于不能对编译器的优化过程进行控制，即使使用了这些手段也不一定能减小寄存器的用量。值得注意的是，采用--maxrregcount 编译选项只是让编译器将超出限制的私有寄存器分配在 local memory 中，造成较大的访存延迟。

（7）与主机通信优化。由于 PCI-E 带宽相对较小，应该尽量减少 CPU 与 GPU 间传输的数据量，并通过一些手段提高可用带宽。可用的技术包括：

- 使用 cudaMallocHost 分配主机端存储器，可以获得更大的带宽。
- 一次缓存较多的数据，再一并传输，可以获得较高的实际带宽。
- 需要将结果显示到屏幕时，直接使用与图形学 API 互操作功能完成，避免将数据返回。
- 使用流和异步处理隐藏与主机的通信时间。
- 使用 zero-copy 技术和 Write-Combined memory 提高可用带宽。

对于用 CUDA C 语言编写的程序，按照上述流程进行优化是比较适合的。不过在优化中，各种因素往往相互制约，很难同时达到最优。读者需要按照要处理问题的类型、瓶颈出现的部位和原因具体分析。按照预想进行优化也不是总能达到预想中的效果，有时优化手段反而会降低性能。在实践中，仍然需要不断实验各种优化方法，在不断试验与迭代中一步步排除不可行的方案，最后得到一个比较理想的方案。

使用 CUDA C 并不总是能够编译到最优的指令。如果确实必要，可以用 PTX 优化程序中最关键的步骤。

除此以外，还要灵活采用宏和模板，动态分配内存和显存以及动态划分数据等手段提高程序的通用性，并在处理不同规模、不同数据类型的问题时选用不同的优化策略。

在 2010 年将要推出的新处理器中，各种存储器的带宽和延迟会有一定的调整，大部分指令的吞吐量也会有非常显著的提升。随着 GPU 架构的进一步改进和编译器性能的不断提高，下一代 GPU 上的 CUDA 程序优化工作会变得更加简单。

4.2 测量程序运行时间

本节将介绍如何准确地测量 CUDA 程序的运行时间。CUDA 的内核程序运行时间可以在设备端测量，也可以在主机端测量。而 CUDA API 的运行时间则只能从主机端测量。无论是主机端测时还是设备端测时，最好都测量内核函数多次运行的时间，然后再除以运行次数以获得更加准确的结果。使用 CUDA runtime API 时，会在第一次调用 runtime API 函数时启动 CUDA 环境。为了避免将这一部分时间计入，最好在正式测时开始前先进行一次包含数据输入输出的计算，这样也可以使 GPU 从平时的节能模式进入工作状态，使测试结果更加可靠。

4.2.1 设备端测时

设备端测时使用 GPU 中的计时器的时戳计时。实现设备端测时有两种不同的方法，分别是调用 clock()函数和使用 CUDA API 的事件管理功能。

使用 clock()函数计时，在内核函数中要测量的一段代码的开始和结束的位置分别调用一次 clock()函数，并将结果记录下来。由于调用__syncthreads()函数后，一个 block 中的所有 thread 需要的时间是相同的，因此只需要记录每个 block 执行需要的时间就行了，而不需要记录每个

thread 的时间。clock()函数的返回值的单位是 GPU 的时钟周期，需要除以 GPU 的运行频率才能得到以秒为单位的时间。这里测得的时间是一个 block 在 GPU 中上下文保持的时间，而不是实际执行需要的时间；每个 block 实际执行的时间一般要短于测得的结果。下面是一个使用 clock 函数测时的例子。

设备端代码：

```
#ifndef _CLOCK_KERNEL_H_
#define _CLOCK_KERNEL_H_

// 这段代码测量进行归约运算时每个 block 使用的时钟周期数，并将结果存储在显存中
__global__ static void timedReduction(const float * input, float * output, clock_t * timer)
{
    // __shared__ float shared[2 * blockDim.x];
    extern __shared__ float shared[];

    const int tid = threadIdx.x;
    const int bid = blockIdx.x;

     //记录测时开始时的时戳
    if (tid == 0) timer[bid] = clock();

    // Copy input.
    shared[tid] = input[tid];
    shared[tid + blockDim.x] = input[tid + blockDim.x];

    // Perform reduction to find minimum.
    for(int d = blockDim.x; d > 0; d /= 2)
    {
        __syncthreads();

        if (tid < d)
        {
            float f0 = shared[tid];
            float f1 = shared[tid + d];

            if (f1 < f0) {
                shared[tid] = f1;
            }
        }
    }

    // Write result.
    if (tid == 0) output[bid] = shared[0];

    __syncthreads();
```

```
//记录测时结束时的时戳
    if (tid == 0) timer[bid+gridDim.x] = clock();
}

#endif // _CLOCK_KERNEL_H_
```

下面是主机端代码，主机端代码根据设备端代码返回时戳的计算时间。

```
#include <stdio.h>
#include <stdlib.h>

#include <cutil_inline.h>

#include "clock_kernel.cu"

// 本程序用于演示如何精确地测量内核执行时间
// Block 之间是并行、乱序执行的，本例测量每一个 block 的执行时间

#define NUM_BLOCKS      64
#define NUM_THREADS     256

int main(int argc, char** argv)

{
    // 使用参数中指定的设备，或者使用浮点处理能力最高的设备

    if ( cutCheckCmdLineFlag(argc, (const char **)argv, "device"))
        cutilDeviceInit(argc, argv);
    else
        cudaSetDevice( cutGetMaxGflopsDeviceId() );

    float * dinput = NULL;
    float * doutput = NULL;
    clock_t * dtimer = NULL;

    clock_t timer[NUM_BLOCKS * 2];
    float input[NUM_THREADS * 2];

    for (int i = 0; i < NUM_THREADS * 2; i++)
    {
        input[i] = (float)i;
    }

    cutilSafeCall(cudaMalloc((void**)&dinput, sizeof(float) * NUM_THREADS * 2));
    cutilSafeCall(cudaMalloc((void**)&doutput, sizeof(float) * NUM_BLOCKS));
    cutilSafeCall(cudaMalloc((void**)&dtimer, sizeof(clock_t) * NUM_BLOCKS * 2));
```

```
    cutilSafeCall(cudaMemcpy(dinput, input, sizeof(float) * NUM_THREADS * 2, cudaMemcpyHostToDevice));

    timedReduction<<<NUM_BLOCKS, NUM_THREADS, sizeof(float) * 2 * NUM_THREADS>>>(dinput,
doutput, dtimer);

    //cutilSafeCall(cudaMemcpy(output, doutput, sizeof(float) * NUM_BLOCKS, cudaMemcpyDeviceToHost));
    cutilSafeCall(cudaMemcpy(timer, dtimer, sizeof(clock_t) * NUM_BLOCKS * 2, cudaMemcpyDeviceToHost));

    cutilSafeCall(cudaFree(dinput));
    cutilSafeCall(cudaFree(doutput));
    cutilSafeCall(cudaFree(dtimer));

    // This test always passes.
    printf( "Test PASSED\n");

    // 计算第一个 block 开始时到最后一个 block 结束之间的时戳数
    clock_t minStart = timer[0];
    clock_t maxEnd = timer[NUM_BLOCKS];

    for (int i = 1; i < NUM_BLOCKS; i++)
    {
        minStart = timer[i] < minStart ? timer[i] : minStart;
        maxEnd = timer[NUM_BLOCKS+i] > maxEnd ? timer[NUM_BLOCKS+i] : maxEnd;
    }

    printf("time = %d\n", maxEnd - minStart);

    cudaThreadExit();

    cutilExit(argc, argv);
}
```

注意：改变 block 和 thread 的数量，会影响 GPU 执行的效率。例如，在 G80（16 个 SM）上执行这段代码时，结果如下：

Block 数量	1	8	16	32	64
时钟周期数	3096	3232	3364	4615	9981

可以发现，当 block 数量少于 SM 数量时，由于一部分 SM 闲置，因此运行时间没有什么变化。当 block 数量达到 16 时，每个 SM 只分到一个 block，依然不能很好地隐藏访存延迟，因此 block 数量从 16 增加到 32 时执行时间没有翻倍。当 block 数量达到 64 时，执行时间才随着 block 数量的增加而线性增加。

使用 CUDA API 的事件管理功能计时则相对简单，下面是一段示意代码：

```
cudaEvent_t start, stop;
```

```
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );
kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y, NUM_REPS); cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

注意 cudaEventElapsedTime()函数返回的时间已经以毫秒为单位，精度为 0.5 微秒。

4.2.2 主机端测时

与普通程序测时一样，CUDA 的主机端测时也采用 CPU 的计时器测时。通常取得 CPU 中计时器的值的方法是调用汇编中的相应指令，或者操作系统提供的 API。此外，一些函数库，如 C 标准库中的 time 库的 clock_t()函数也可以用来测时。不过，clock_t()函数的精度很低，建议在两次调用 clock_t()时，让待测程序运行至少数十次，运行时间达到数秒，再取平均求得每次运行时间。

使用 CPU 测时，一定要牢记 CUDA API 的函数都是异步的。这就是说，在一个 CUDA API 函数在 GPU 上执行完成之前，CPU 线程就已经得到了它的返回值。内核函数和带有 asyn 后缀的存储器拷贝函数都是异步的。

要从主机端准确的测量一个或者一系列 CUDA 调用需要的时间，就要先调用 cudaThreadSynchronize() 函数，同步 CPU 线程与 GPU 之后，才能结束 CPU 测时。cudaThreadSynchronize() 函数的功能是阻塞 CPU 线程，直到 cudaThreadSynchronize() 函数之前所有的 CUDA 调用都已经完成。

与 cudaThreadSynchronize() 函数类似的函数有 cudaStreamSynchronize() 和 cudaEventSynchronize()。它们的作用是阻塞所有 Stream/CUDA Events，直到这条函数前的所有 CUDA 调用都已完成。注意，同一串流中的各个流可能会交替执行，因此即使使用了 cudaStreamSynchronize()函数，也很难测得准确的执行时间。

不过，一串流中的第一个流（ID 为 0 的流）的行为总是同步的，因此使用这些函数对 0 号流进行测时，得到的结果是可靠的。

4.3 任务划分

4.3.1 任务划分原则

首先，需要将要处理的任务划分为几个连续的步骤，并将其划分为 CPU 端程序和 GPU 端程序。划分时需要考虑的原则有：

列出每个步骤的所有可以选择的算法，并比较不同算法在效率和计算复杂度上的差异。

能够并行实现的算法并不一定比串行算法快。在问题规模较小时，计算复杂度阶数更高的算法也有可能比计算复杂度阶数较低的算法耗费更短的时间。根据问题规模，选择适当算法，将任务中耗费时间最多的大规模数据并行、高计算密集度步骤映射到 GPU 上。

在 CPU 上可以并行实现的算法不一定适用于 GPU。CPU 程序主要考虑的是指令间并行和粗粒度的软件线程并行，在每个 CPU 线程内还是串行的。由于 CPU 线程粒度往往太大，因此尽量不要将 CPU 线程直接映射为 GPU 线程。每个 GPU 线程完成的任务更加类似于 CPU 的多轮循环中的一轮。但也不是所有的循环都能映射为一个内核程序，因为有的循环中每一轮运算都依赖上一轮的结果，而 GPU 的线程之间是并行的。此时，需要采取其他方式对任务进行分解。

在两次主机－设备通信之间进行尽量多的计算。由于主机与设备间的数据传输带宽远低于显存带宽，因此最好在两次通信之间让 GPU 进行尽可能多的运算。如果在两次大规模数据并行运算之间存在少量的串行运算，有时即使是在 GPU 上以较低的效率进行这些串行运算也比增加两次主机－设备通信要划算。在 GPU 进行运算的同时，如果可能，也可以让 CPU 进行一些计算，比如准备下一次计算需要的数据。

应该考虑使用流运算隐藏主机－设备通信时间，以及通过 pinned memory、zero-copy、write-combined memory 等手段提高实际传输带宽。在集群中使用 CUDA，还需要考虑节点之间的任务分配与通信问题。

对每个并行步骤进行划分，从不同角度分析有不同的划分方式。

从对显存的访问方向来说，可以按照输入划分或者按照输出划分。如果每个 block 中输入和输出的数据的比例和位置是固定的，并且能够比较容易地满足合并访问要求，那么这种划分方式既是按照输入划分的，也是按照输出划分的。这种情况是最理想的，通过 shared memory 和指针类型转换，大多数输入输出都能够很好地满足合并访问条件。

但如果 block 的输入和输出的数据不相同，或者输入和输出无法同时满足合并访问要求，就必须设法使可用带宽最大化，只按照输入或者只按照输出划分。

按照输入划分的情况有：

- 输入参数很多而输出结果很少，如规约、直方图。
- 输入满足合并访问条件，但是输出位置随机，或者输出时需要进行显存原子操作，在流体力学、分子动力学仿真中可能遇到这种情况。

按照输出划分的情况有：

- 输入参数很少而输出结果很多，如随机数发生器。Block 内每个线程的输入与其他线程共用，比如卷积、滤波中，每个线程的输入与周围线程的输入有公共部分，此时应该先按照合并访问的形式将一块数据读入 shared memory，再由每个线程计算一定数量的输出，可以参考 SDK 中与滤波和卷积有关的几个例子。
- 输入数据在存储器中的位置是随机的，而输出数据时可以满足合并访问条件的情况，大多数使用纹理的应用，以及一些需要查表的运算都属于这种情况。

从显存访问的形式来说，在一个 block 内可以进行一维的带状划分、二维的棋盘划分或者三维的域划分。如果要处理的任务不需要线程间通信，并且对显存的访问都能满足合并访问，那么采用棋盘划分还是带状划分对性能影响并不大。不过，应该尽量使每个 block 中的线程数量是 32 的整数倍，并根据任务的具体情况确定每个维度上的大小，以减少计算访存地址时的

整数除法和求模运算。

如果需要使用纹理的特殊功能进行图像处理，使用二维棋盘划分是比较自然的。

如果问题在一个或者几个维度方向上有局部性，可以利用 shared memory 提高性能或者必须在某几个维度内进行线程间通信，那么 block 的维度应该与需要通信的维度一致。比如本章 4.7.1 节的矩阵乘法例子中，既可以进行一维带状划分，也可以按照二维棋盘划分，但二维划分的算法利用了 shared memory，有效减小了访存次数，并且满足合并访问条件。

对一个 block 的任务进行划分后，再按照 block 的维度和尺寸要求对 grid 进行划分。此时需要考虑的问题是：

- 考虑分区冲突问题，使每个 block 的访存要求均匀分布在显存的各个分区中，例如 4.7.3 节中介绍的矩阵转置，在解决分支冲突问题后，性能有了几倍的提升。
- Block 间负载可以存在一定程度的不均衡，按照 block 为单位分支性能损失也很小。比如，对网络中的数据进行分析时，可以由一个 grid 处理其中缓冲中的多个包，再由每个 SM 处理长度和内容都不同的包。

4.3.2 grid 和 block 维度设计

按照 CUDA 的执行模型，grid 中的各个 block 会被分配到 GPU 的各个 SM 中执行。下面的一些建议能够帮助读者确定合适的 Grid 与 block 尺寸。在设计时，应该优先考虑 block 的尺寸，而 grid 的尺寸一般来说越大越好。

由 3.2.2.3 小节可知，在 Tesla 架构 GPU 的每个 SM 中，至少要有 6 个 active warp 才能有效地隐藏流水线延迟。此外，如果所有的 active warp 都来自同一 block，当这个 block 中的线程进行存储器访问或者同步时，执行单元就会闲置。基于以上原因，最好让每个 SM 上拥有至少 2 个 active block。

一个 SM 上的 active warp 和 active block 数量计算方法如下：

（1）确定每个 SM 使用的资源数量。

使用 nvcc 的--keep 编译选项，或者在.cu 编译规则（cuda build rule）中选择保留中间文件（keep preprocessed files），得到.cubin 文件。用写字板打开.cubin 文件，在每个内核函数的开始部分，可以看到以下几行：

```
lmem = 0
smem = 256
reg    = 8
```

其中，lmem 和 reg 分别代表内核函数中每个线程使用的 local memory 数量和寄存器数量，smem 代表每个 block 使用的 shared memory 数量。以上数据告诉我们：这个内核函数的每个线程使用了 0Byte local memory，8 个寄存器文件（每个寄存器文件的大小是 32bit）；每个 block 使用了 256Byte 的 shared memory。

（2）根据硬件确定 SM 上的可用资源。

可以用 SDK 中的 deviceQuery 获得每个 SM 中的资源。要注意的是，在程序编译时，要使目标代码和目标硬件版本与实际使用的硬件一致（使用-arch、-gencode 和-code 编译选项）。在 G80 和 GT200 架构上，这些限制如表 4-1 所示。

表 4-1　每 SM 上的可用资源个数

架构	G80/G92	GT200
每 SM 上 warp 总数上限	24 个	32 个
每 SM 上 block 总数上限	8 个	8 个
每 SM 上寄存器数量	32bit ×8192 个	32bit ×16384 个
每 SM 上 shared memory 数量	16KB	16KB

此外，每个 block 中的线程数量不能超过 512 个。

（3）计算每个 block 使用的资源，并确定 active block 和 active warp 数量。

假设每个 block 中有 64 个线程，每个 block 使用 256 Byte shared memory，8 个寄存器文件，那么就有：

- 每个 block 使用的 shared memory 是：256Byte。
- 每个 block 使用的寄存器文件数量：8×64 = 512。
- 每个 block 中的 warp 数量：64/32 = 2。

然后，根据每个 block 使用的资源，就可以计算出由每个因素限制的最大 active block 数量。这里，假设在 G80/G92 GPU 中运行这个内核程序：

- 由 shared memory 数量限制的 active block 数量：16384/256 = 64。
- 由寄存器数量限制的 active block 数量：8192/512 = 16。
- 由 warp 数量限制的 active block 数量：　24/2 = 12。
- 每个 SM 中的最大 active block 数量：8。

注意，在计算每个因素限制的 active block 数量时如果发现有除不尽的情况，应该只取结果的整数部分。取上述计算结果中的最小值，可以知道每个 SM 的 active block 数量为 8。

NVIDIA 在 CUDA SDK 中提供的 CUDA occupancy calculator 也可以完成上面的计算。CUDA occupancy calculator 是一个 Excel 文件，存储在 SDK 的 tools 目录下。只要在这个 Excel 表格中输入目标硬件的架构，以及每个 block 中的线程数量、每个 block 使用的 shared memory 数量和每个 thread 使用的寄存器数量，就可以自动计算出 SM 的资源占用率，并以图表的形式显示，十分方便。

block 的尺寸与数据划分紧密相关，在上一节已经进行了一些探讨。较小的 block 使用的资源较少，一般在一个 SM 中能够有更多的 active block；而较大的 block 中有更多的线程可以进行通信，可以获得更高的指令流效率。为了有效利用执行单元，应该让每个 block 中的线程数量是 32 的整数倍，最好让线程数量保持在 64～256 之间。此时，SM 中通常还有足够多的资源来执行至少两个 active block。

block 的维度和每个维度上的尺寸的主要作用是避免做整数除法和求模运算，对执行单元效率没有什么显著影响。在使用中，读者可以按照问题的具体情况自行确定。如果问题的规模对划分方式并不敏感，应该让 blockDim.x 为 16 或者 16 的整数倍，提高访问 global memory 和 shared memory 的效率。

确定 block 的尺寸和维度以后，就可以按照问题的规模确定 grid 中的 block 数量。通常，使用下面的方法来计算 grid 在某个维度上的 block 数量（以 x 轴为例）：

grid 在 x 轴上的 block 数量=（问题在 x 轴上的尺寸+每个 block 在 x 轴上的尺寸-1）/每个 block 在 x 轴上的尺寸

加上（每个 block 在 x 轴上的尺寸 - 1）是因为整数除法只会取结果的整数部分，这样可能使问题的边界得不到处理，引起错误。按照这种方法计算 block 数量，实际的线程数量就会比需要的线程数量更多。因此，在内核程序中，也要对边界部分进行判断，让边界外的线程不参与计算。使用与线性存储器绑定的纹理时，访问边界外时的返回值是 0，可以用来简化边界处理。

理想情况下，grid 中的 block 数量应该至少要比 GPU 的 SM 数量×每个 SM 中 active block 数量大几倍。为了让程序在未来的 GPU 上运行也能获得高效率，应该让 block 的数量尽可能得大。

4.4 存储器访问优化

存储器带宽是计算机性能的瓶颈之一。通常，处理器的计算能力要远远超过内存访问的带宽。例如，GTX280 的单精度浮点处理能力可以达到 1TFLOPS 左右，而内存的访问带宽只有 141GB/s，相差了几倍。在前面的章节中，我们已经讲解了各种存储器的访问方法。本章会对内存访问进行更详细的分析，对程序设计和优化中会遇到的内存问题都做一些详细的讨论。

CUDA 内核程序中不需要考虑 IO 和事务处理，每个 warp 中的指令只有两种：访存和运算。只有当访存结束，数据被准备好以后，运算指令才能被执行。因此，在设计高性能计算程序时，从一开始就必须对存储器访问进行规划，并在开发过程中时刻注意存储器访问对程序性能的影响。支持 CUDA 的显卡在有限的面积上实现了可观的显存带宽，但为此也付出了一定的代价：只有在满足合并访问，并有足够的运算指令隐藏访存延时，才能获得较高的实际可用带宽，否则就会形成“存储器墙”。为了缓解显存带宽与运算性能之间的不平衡，NVIDIA 的 GPU 中采用了一系列寄存器、片内存储器和缓存。这些措施有效地减小了对片外的显存带宽的依赖，但提高了程序设计的难度。

在以运算为主的 CUDA 程序中，应该避免让存储器访问和通信成为性能瓶颈；而在以存储器访问为主的应用中，应该尽可能增大程序的可用带宽。灵活地利用各种存储器特性，实现最大的可用带宽，是 CUDA 程序优化的重要任务之一。

4.4.1 主机－设备通信优化

目前大多数的显卡都通过 PCI-E 总线与主机端连接。一条 PCI-E 2.0×16 通道的理论带宽是双向每向 8GB/s，远小于显存和 GPU 片内存储器带宽。如果需要与主机相互传输的数据比较多，那么 PCI-E 总线带宽就很容易成为阻碍整个程序性能提高的瓶颈。严格地说，PCI-E 带宽不足的问题应该属于“IO 墙”而不是“存储器墙”，但由于 CUDA API 中一系列用于解决主机－设备通信问题的手段与存储器有关，因此将其放在本节介绍。

4.4.1.1 Pinned memory

首先通过下面的代码来讲解访问 pinned memory。

```
#if CUDART_VERSION >= 2020
cutilSafeCall( cudaHostAlloc( (void**)&h_idata, memSize, (wc) ? cudaHostAllocWriteCombined : 0 ) );
cutilSafeCall( cudaHostAlloc( (void**)&h_odata, memSize, (wc) ? cudaHostAllocWriteCombined : 0 ) );
#else
        cutilSafeCall( cudaMallocHost( (void**)&h_idata, memSize ) );
        cutilSafeCall( cudaMallocHost( (void**)&h_odata, memSize ) );
#endif
…
cutilSafeCall( cudaMemcpyAsync( h_odata, d_idata, memSize,
                                cudaMemcpyDeviceToHost, 0) );
```

pinnend memory 的实质是强制让操作系统在物理内存中完成内存申请和释放的工作，这一部分内存不用参与页交换，因此速度比普通的可分页内存快。但是声明这些物理内存只会被分配给对应的 GPU 设备使用，占用了操作系统的可用内存。这可能会影响到 CPU 运行需要的物理内存，所以在考虑整个系统的优化时，需要合理规划 CPU 和 GPU 各自使用的内存，使整个系统达到最优。

4.4.1.2　异步执行

异步执行是指 GPU 进行的操作（内核启动或者异步存储器拷贝函数）从主机端启动后，在 GPU 真正完成这些操作之前，在主机端就可以得到这些函数的返回值，CPU 线程就可以继续进行下一步操作。也就是说，CPU 端的 API 函数和内核启动的结束，和 GPU 端真正完成这些操作是异步的。通过异步函数，CPU 可以在 GPU 端进行运算或者数据传输的同时进行其他操作，更加有效地利用系统中的计算资源。

内核启动和显存内的数据拷贝（Device to Device）总是异步的，而内存和显存间的数据拷贝函数则有异步和同步两个版本。例如，下面的代码中进行了一次同步的数据拷贝，只有确实完成 CPU 和 GPU 间数据传输后，cudaMemcpy()函数才会返回，此时 CPU 才能开始执行后面的 cpuFunction()函数。

```
cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
cpuFunction();
```

而调用异步的 cudaMemcpyAsync()函数时，CPU 线程通过存储器管理 API 函数启动了一次数据传输，在 GPU 传输的同时，cudaMemcpyAsync()函数就已经返回了。此时，cpuFunction()函数实质上是与 GPU 和 CPU 间的数据传输同时进行的。

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice);
cpuFunction();
```

对主机端来说，内核启动是异步的。在下面的代码中，cpuFunction()也会在 kernel 函数在 GPU 上运行结束之前就开始执行：

```
kernel<<<blocks,threads>>>(a_d);
cpuFunction();
```

如果主机端需要使用内核函数计算得到的结果，应该加入同步保证操作的顺序一致性：

```
kernel<<<blocks,threads>>>(a_d);
cudaThreadSynchronize();
```

```
cpuFunction();
```

属于同一个流中的内核启动总是同步的。在下面的代码中，没有显式地为 kernel1 和 kernel2 指定所属的流，此时它们都属于默认的 stream0。因此，kernel2 总是会在 kernel1 执行完成之后才能开始执行。

```
kernel1<<<blocks1, threads1>>> (a_d1);
kernel2<<<blocks2, threads2>>> (a_d2);
```

如果几次内核启动分别属于不同的流，它们的执行可能是乱序的，如果它们都要向同一块存储器写数据，就有可能因为竞写而导致错误。因此在使用 stream 时，stream 之间的数据相关性要尽量小。下面的代码中存在两个流，当 stream0 的异步数据传输在 GPU 上执行完毕时，stream1 的异步数据传输和 stream0 的 kernel 执行可以在 GPU 上同时执行：

```
cudaStreamCeate(&stream0);
cudaStreamCeate(&stream1);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 1);
kernel<<<blocks1, threads1, stream0>>> (a_d1);
kernel<<<blocks1, threads1, stream1>>> (a_d1);
```

在上面的几个例子里，我们介绍了 GPU 操作异步执行的基本概念和使用方法，下面介绍如何使用流和异步提高程序性能。

【方法一】使用流和异步使 GPU 和 CPU 同时进行运算：

```
cudaStreamCeate(&stream1);
cudaMemcpy (DeviceData, HostData, size, cudaMemcpyHostToDevice, stream1);
kernel<<<blocks, threads, 0, stream1>>>(a_d);
cpuFunctionA(CPUresult);
cudaThreadSynchronize();
cudaMemcpy、(GPUResult, DeviceData, size, cudaMemcpyHostToDevice, stream1);
cpuFunctionB(GPUResult, CPUresult);
```

在这个例子中，假设 cpuFunctionA()与 GPU 的操作没有什么冲突，而 cpuFunctionB()则需要用到 cpuFunctionA()和 GPU 运算的结果，那么可以利用内核启动的异步执行让 CPU 和 GPU 同时运算，再使用同步保证 GPU 已经完成操作。这时，cpuFunctionB()函数就可以安全地消费 CPU 和 GPU 同时生产的数据。

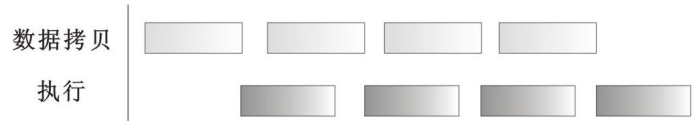
【方法二】利用不同流之间的异步执行，使流之间的传输和运算能够同时执行，更好地利用 GPU 资源：

```
for (i=0; i<nStreams; i++) {
  cudaStreamCeate(&(stream[i]));
}
size=N*sizeof(float)/nStreams;
for (i=0; i<nStreams; i++) {
offset = i*N/nStreams;
cudaMemcpyAsync(a_d+offset, a_h+offset, size, dir, stream[i]);
}
for (i=0; i<nStreams; i++) {
offset = i*N/nStreams;
```

```
kernel<<<N/(nThreads*nStreams), nThreads, 0, stream[i]>>>(a_d+offset);
}
```

在上面的例子中，我们创建了一系列流，每个流都进行了异步传输和内核执行。此时，一个流的传输和另一个流的执行可以同时进行，因此 GPU 就能够持续地进行计算，而不必等待数据，如图 4-2 所示。

（a）kernel 串行执行



（b）kernel 并行执行

图 4-2　kernel 执行方式比较

从图 4-2 可知，不使用异步执行时的程序的运行时间为（执行时间+传输时间）；而使用流和异步后，时间降为（执行时间+传输时间/流的数量）。此时，CPU 和 GPU 间的数据传输时间被有效地隐藏了，程序的性能得到了改善。

4.4.2　全局存储器访问优化

对全局内存的访问是否满足合并访问条件是对 CUDA 程序性能影响最明显的因素之一。在计算能力 1.0/1.1 硬件上，是否满足合并访问条件在很多情况下会使 CUDA 程序的速度产生高达一个数量级的差异！在 3.3.2 节中已经介绍过，当来自一个 Half-warp 的 16 个线程对全局内存进行装载或者存储访问时，如果能满足一定的访问条件，只需要进行一次传输就可以处理这些线程的访存请求。合并访问条件要求同一 half-warp 中的线程要按照一定字长访问经过对齐的段。图 4-3 展示了一个 half-warp 中的所有线程进行 32bit 字（如 float）的合并访问的情况，图中 global memory 中每一行是一个 64Byte 对齐的段（16 个 float），颜色相同的两行则表示一个按照 128Byte 对齐的段，图形下方则表示对全局存储器进行访问的 half-warp 中的线程。

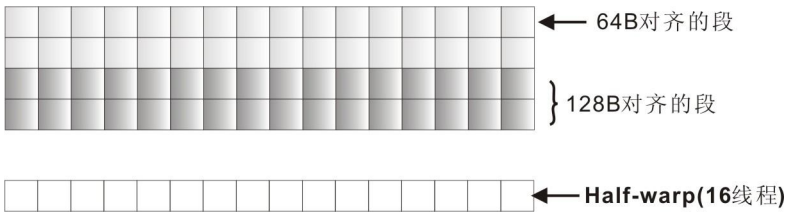


图 4-3　half-warp 与全局存储器中的段

不同计算能力的 GPU 对合并访问条件有不同的要求，其中计算能力 1.2 以上的设备的合并访问要求要宽松一些。下面是各种设备中合并访问条件的具体要求。

- 在计算能力 1.0、1.1 的设备上，一个 half-warp 中的第 k 个线程必须访问段里的第 k 个字，并且 half-warp 访问的段的地址必须对齐到每个线程访问的字长的 16 倍，比如每个线程访问 32 bit 字时，half-warp 访问的段的首地址就必须是 64Byte 的整数倍；如果满足合并访问条件的一个 half-warp 中有一些线程不访存（也不能访问显存的其他位置），此时仍然会进行一次合并访问，只是不访存的线程不会得到访存结果。计算能力 1.0 和 1.1 的设备只支持对字长为 32bit、64bit 和 128bit 的数据的合并访问。如果不满足合并访问条件，那么 half-warp 的访问请求会被解释为 16 次串行的传输。
- 在 1.2 及更高计算能力的设备上，合并访问要求大大放宽了。计算能力 1.2、1.3 的设备支持对字长为 8bit（对应段长 32Byte）、16bit（对应段长 64Byte）、32bit、64bit 和 128bit（三者都对应 128Byte 段长）的数据进行合并访问。注意这里的段长与 1.0/1.1 设备的段对齐长度概念不一样。只要 half-warp 中的线程访问的数据在同一段中，就可以满足合并访问条件，而线程的顺序和线程访问的地址则不再需要像计算能力 1.0/1.1 硬件中那样按顺序依次对应。如果访问的数据首地址没有按段长对齐，一个 half-warp 的访问请求也只会被解释为两次满足合并访问的传输，只是多访问的数据会被丢弃。

下面的协议详细描述了计算能力 1.2/1.3 硬件的一个 half-warp 是如何完成一次合并访问的。

- 首先，找到由最低线程号活动线程（前 half-warp 中的线程 0，或者后 half-warp 中的线程 16）请求访问的地址所在段。对于 8bit 数据来说段长为 32Byte，对于 16bit 数据来说段长为 64Byte，对于 32、64、128bit 数据来说段长为 128Byte。
- 然后，找到所请求访问的地址也在这个段内的活动线程。如果所有线程访问的数据都处于段的前半部分或者后半部分，那么还可以减少一次传输的数据大小。例如，如果一个段的大小为 128Byte，但只有上半部分或下半部分被使用了，那么实际传输的数据大小就可以进一步减小到 64Byte。同理，对 64Byte 的段的合并传输，在只有前半或者后半被使用的情况下也可以减小到 32Byte。
- 进行传输，此时执行访存指令的线程将处于不活动状态，执行资源被释放供 SM 中处于就绪态的其他 warp 使用。
- 重复上述过程，直到 half-warp 所有线程均访问结束。

下面将通过一些小例子来解释对显存的合并访问。

4.4.2.1 一个简单的访问模式

第一个例子很简单，由第 k 个线程访问段中的第 k 个字，但并不是所有线程都进行存储器访问，如图 4-4 所示。

如图中粗线矩形框标注部分所示，这种访问模式会导致一次 64Byte 合并传输。注意，尽管有一个字不被请求，但段里所有的数据都会被装载，只是不用的数据不会被写入寄存器。如果线程 ID 与访问的数据地址不是按照顺序一一对应，而存在交叉访问，那么在 1.2 及以上的设备仍会按一个 64Byte 传输进行处理，但在 1.0、1.1 上则会产生 16 次串行传输。

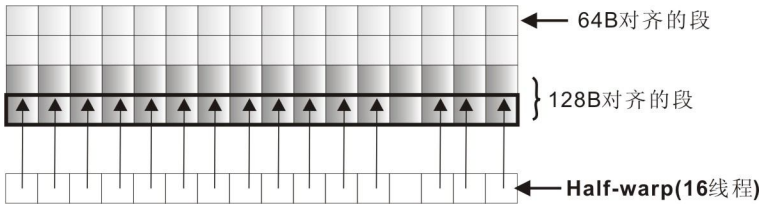


图 4-4　一种简单的合并访问模式

4.4.2.2　一个连续的但未对齐的访问模式

如果 half-warp 内的线程访问地址是连续的，但并没有与段对齐，此时在 1.0、1.1 设备上对一个元素的访问都会导致一次传输（half-warp 共 16 个传输），而在 1.2 及以上的设备则根据 half-warp 请求的所有地址是否位于一个 128Byte 段内而定。如果所有线程都访问处于同一段的数据，就只会进行一次 128Byte 合并访问，如图 4-5 所示。

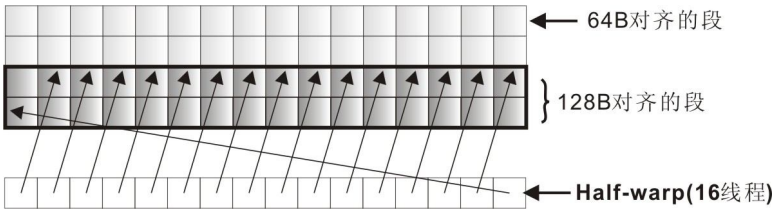


图 4-5　128Byte 段内的非对齐访问

如果 half-warp 访存的地址连续，但横跨两个 128 Byte 段，此时会产生两次传输。如图 4-6 所示，产生一个 64 Byte 传输和一个 32 Byte 传输。

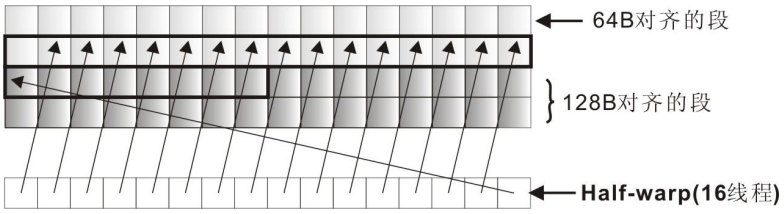


图 4-6　两个 128Byte 段内的非对齐访问

通过运行时 API（如 cudaMalloc()）分配的存储器，已经能保证其首地址至少会按 256Byte 进行对齐。因此，选择合适的线程块大小（例如 16 的倍数），能使 half-warp 的访问请求按段长对齐。使用__align__(8)和__align__(16)限定符来定义结构体，可以使对结构体构成的数组进行访问时能够对齐到段。

4.4.2.3　访问时段不对齐的后果

下面的例子可以说明访问段不对齐时产生的后果。

```
__global__ void offsetCopy(float* odata, float* idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
```

```
    odata[xid] = idata[xid];
}
```

在本例中，数据从输入数组 idata 拷贝到输出数组，两个数组均位于全局存储器中。在主机端代码中，kernel 在一个 for()循环里执行，首地址偏移量 offset 从 1 到 32 以步长为 1 变化（图 4-5 和图 4-6 分别对应 offset 分别为 1 和 17）。在 GeForce GTX 280（1.3 设备）与 Quadro FX 5600（1.0 设备）上运行这个程序，取不同 offset 时的有效带宽如图 4-7 所示。

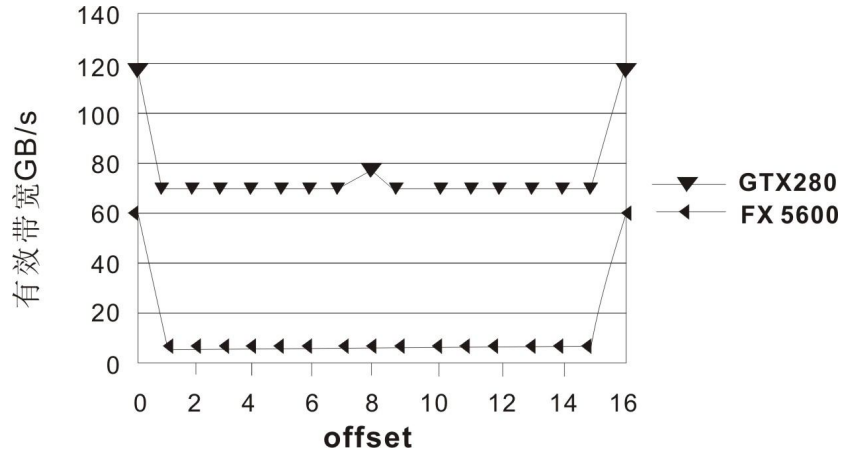


图 4-7　offsetKernel 性能

在 Quadro FX 5600 中，当 offset 为 0 或 16 时满足合并访问条件，一个 half-warp 只产生一次传输，有效带宽可以达到近 60GB/s；而当 Offset 为其他值时，half-warp 就会产生 16 次传输，此时带宽只有 6.6GB/s，产生了几乎 8 倍的性能下降。这是因为每个线程都会产生一次 32Byte 数据传输（最小的传输大小），但其中只有 4Byte 有用，因此此时性能只有完全对齐情况下的 4/32=1/8。这两个数字反映了有效带宽（4Byte）与实际带宽（32Byte）的区别。正因为合并访问有可能对性能造成如此巨大的影响，因此合并访问优化在 CUDA 程序编写中是非常重要的一步。

在 GTX 280 上，刚才的程序的运行情况要好一些。对 offset 的所有取值，half-warp 的访问请求都只会被解释为一个或两个传输。当只进行一次传输时，有效带宽可以达到 120GB/s；如果发生两次传输，那么有效带宽为 66GB/s。传输的数据数量取决于 offset 和 warp ID 的奇偶性。如果 offset 为 0 或 16，那么每个 half-warp 的访存请求都只会导致一次 64Byte 传输（见图 4-4）；如果 offset 为 1～7，或 9～15，warp ID 为偶数的 warp 的访存请求会被整合为一次 128Byte 的传输（见图 4-5），而 warp ID 为奇数的 warp 的访存请求则会被整合为两次传输：一次 64Byte 传输和一次 32Byte 传输（见图 4-6）；当 offset 为 8 时，ID 为偶数的 warp 会产生一次 128Byte 传输，而 ID 为奇数的 warp 则会产生两次 32Byte 传输。这里，之所以会产生两次 32Byte 传输而不是一次 64Byte 和一次 32Byte 传输，是因为这样访存可以获得更高的带宽（图 4-7 中 offset=8）。

4.4.2.4　间隔访问（strided access）

上面的 offsetCopy 例子中，1.2 及以上设备对合并访问的松散约束使得带宽达到了理想带

宽的 1/2。但 1.2 及以上设备在 half-warp 连续线程访问有一定间隔的显存地址时，性能会严重下降。这种情况在处理多维数据或矩阵时会频繁出现。例如一个 half-warp 按列而不是按行去访问矩阵元素时（矩阵按行存储），线程就会间隔地访问存储器中的数据。

下面这个示例展示了间隔访问对有效带宽的影响，线程间相隔 stride 个元素将数据从 idata 拷贝到 odata。

```
__global__ void strideCopy(float *odata, float *idata, int stride)
{
    int xid = (blockIdx.x * blockDim.x + threadIdx.x) * stride;
    odata[xid] = idata[xid];
}
```

图 4-8 展示了上述代码可能造成的结果，half-warp 中的线程按 stride=2 进行访存，在 GTX 280 上被整合为一个 128Byte 传输。

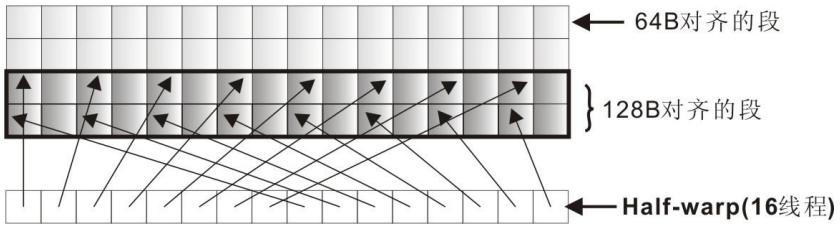


图 4-8　half-warp 访存（stride = 2）

尽管 stride=2 时可以被整合为一次传输，注意传输的数据中有一半会被丢弃，这就表示浪费了带宽。随着 stride 的增加，有效带宽就会下降，直到 half-warp 内的每个线程产生一次传输，如图 4-9 所示。

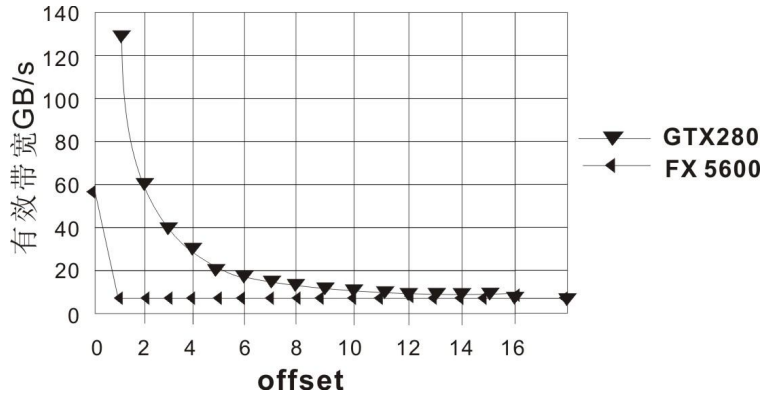


图 4-9　strideCopy 性能

注意，在 Quadro FX 5600 上（1.0 设备），只要 stride 值不为 1，half-warp 就会产生 16 次传输。

如图 4-9 所示，应该避免间隔访问显存的情况。我们可以借助 shared memory 来实现这一点。

4.4.3 共享存储器访问优化

共享存储器位于 GPU 片内，速度比 local/global memory 快很多。在不发生 bank conflict 的情况下，shared memory 的延迟几乎只有 local 或 global memory 的 1/100，访问速度与寄存器相当。

4.4.3.1 共享存储器与 bank conflict

为了能够在并行访问时获得高带宽，共享存储器被划分为大小相等，能被同时访问的存储器模块，称为 bank。由于不同的存储器模块可以互不干扰地同时工作，因此对位于 n 个 bank 上的 n 个地址的访问能够同时进行，此时的有效带宽是只有一个 bank 时的 n 倍。

但如果 half-warp 请求访问的多个地址位于同一个 bank 中，就出现了 bank conflict。由于存储器模块在一个时刻无法响应多个请求，因此这些请求就必须被串行地完成。硬件会将造成 bank conflict 的一组访存请求划分为几次不存在 conflict 的独立请求，此时的有效带宽会降低几倍，降低的倍数就是拆分得到的不存在 conflict 的请求个数。但这也有一种例外情况，当一个 half-warp 中的所有线程都请求访问同一个地址时，会产生一次广播，此时反而只需要一次就可以响应所有线程的请求。

为了减少 bank conflict，必须先了解 shared memory 地址如何被映射到各个 memory bank，以及如何通过优化调度访存请求来避免 bank conflict。对共享存储器的访问应避免 bank conflict 造成序列化访问。

bank 的组织方式是：每个 bank 的宽度固定为 32bit，相邻的 32bit 字被组织在相邻的 bank 中，每个 bank 在每个时钟周期可以提供 32bit 的带宽。

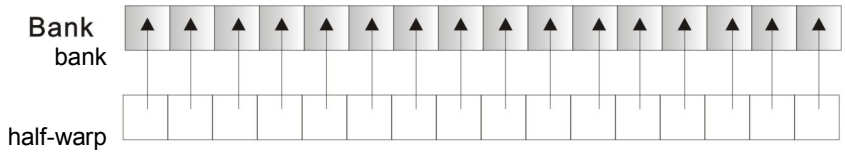
对于计算能力 1.x 设备，每个 warp 大小都是 32 个线程，而一个 SM 中的 shared memory 被划分为 16 个 bank（按 0～15 编号）。一个 warp 中的线程对共享存储器的访问请求会被划分为两个 half-warp 的访问请求，只有处于同一 half-warp 内的线程才可能发生 bank conflict，而一个 warp 中位于前 half-warp 的线程与位于后 half-warp 的线程间则不会发生 bank conflict。

下面这个例子展示了 shared memory 的一般使用方法，线程从数组中读取 32bit 字（索引由 tid 及间隔 s 确定）：

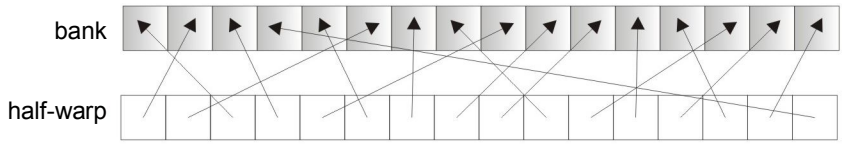
```
__shared__ float shared[32];
float data = shared[BaseIndex + s * tid];
```

在这段代码中，如果 s*n 是 m（m 是 bank 数）的倍数，或者 n 是 m/d （其中 d 是 m 和 s 的最大公约数）的倍数，就会发生 bank conflict。因此，只要 half-warp 小于或者等于 m/d，就不会发生 bank conflict。对于计算能力 1.x 的硬件，当 d=1 或者 s 为奇数（因为 m 是 2 的幂）时，就可以避免 bank conflict 的发生。在 SDK 的很多例子中，都使用了宽度为 17 或者 threadDim.x+1 的行来避免 bank conflict。

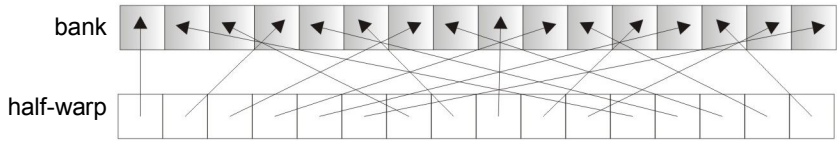
图 4-10 是不存在 bank conflict 的 shared memory 访问的例子；图 4-11 是存在 bank conflict 的 shared memory 访问的例子。



（a）线性地址（stride=32bit）

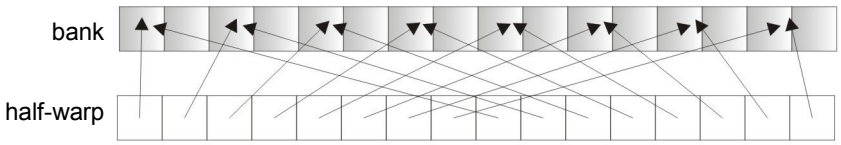


（b）地址随机排列

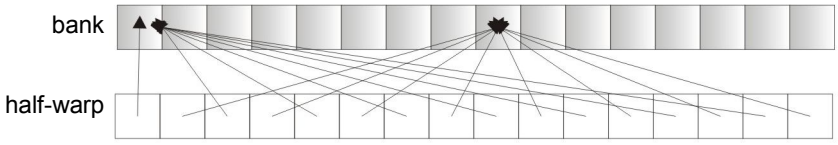


（c）线性地址（stride = 3×32bit）

图 4-10　没有 bank conflict 的共享存储器访问示例



（a）线性地址（stride = 2×32bit）造成 2-way bank conflict



（b）线性地址（stride = 8×32bit）造成 8-way bank conflict

图 4-11　产生 bank conflict 的共享存储器访问示例

如果每个线程访问的数据大小不是 32bit 时，也会发生 bank conflict。例如以下对 char 数组的访问会造成 4-way bank conflict：

```
__shared__ char shared[32];
char data = shared[BaseIndex + tid];
```

因为此时 shared[0]、shared[1]、shared[2]、shared[3]属于同一个 bank。对同样的数组，按下面的形式进行访问，则可以避免 bank conflict 问题：

```
char data = shared[BaseIndex + 4 * tid];
```

又如，对 double 数组进行访问时存在 2-way 冲突，此时访存请求会被编译为两个独立的 32bit 请求：

```
__shared__ double shared[32];
```

```
double data = shared[BaseIndex + tid];
```

遇到这种情况时，可以采用下面的办法来避免 bank conflict。但这种做法并不总是能提升性能，在下一代架构中反而会降低性能。

```
__shared__ int shared_lo[32];
__shared__ int shared_hi[32];

double dataIn;
shared_lo[BaseIndex + tid] = __double2loint(dataIn);
shared_hi[BaseIndex + tid] = __double2hiint(dataIn);

double dataOut = __hiloint2double(shared_hi[BaseIndex + tid], shared_lo[BaseIndex + tid]);
```

对一个结构体赋值会被编译为几次访存请求，例如下面的代码：

```
__shared__ struct type shared[32];
struct type data = shared[BaseIndex + tid];
```

如果 type 定义为（1），那么对 type 的访问会被编译为三次独立的存储器读访问，因为每个结构体的同一成员之间有 3 个 32bit 字的间隔，所以不存在 bank conflict。如果 type 定义为（2），那么对 type 的访问会被编译为两个独立的存储器访问，此时每个结构体成员都有 2 个 32bit 字的间隔，线程 ID 相隔 8 的线程间就会发生 bank conflict。如果 type 定义为（3），那么对 type 的访问会被编译为两个独立的存储器访问，因为每个结构体成员都是通过 5byte 的间隔来访问，所以总是会存在 bank conflict。

```
struct type {
   float x, y, z;
};
```

（1）

```
struct type {
   float x, y;
};
```

（2）

```
struct type {
   float f;
   char   c;
};
```

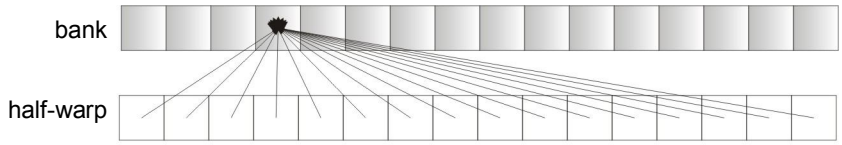
（3）

shared memory 采用了广播机制。在响应一个对同一个地址的读请求时，一个 32bit 字可以被读取并同时广播给不同的线程。当 half-warp 有多个线程读取同一个 32bit 字地址中的数据时，可以减少 bank conflict 的数量。而如果 half-warp 中的线程全都读取同一地址中的数据时，此时完全不会发生 bank conflict。不过，如果 half-warp 内有多个线程要对同一地址进行写操作，此时则会产生不确定的结果，发生这种情况时应该使用对 shared memory 的原子操作。

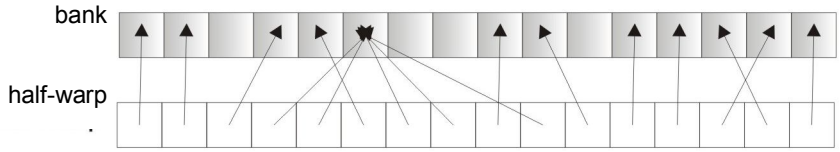
对不同地址的访存请求会被分为若干个处理步，每两个执行单元周期完成一步，每步都只处理一个 conflict-free 的访存请求的子集，直到 half-warp 的所有线程请求均完成。在每一步中都会按下列规则构建子集：

- 从尚未访问的地址所指向的字中，选出一个作为广播字。
- 继续选取访问其他 bank，并且不存在 bank conflict 的线程，再与上一步中广播字对应的线程一起构建一个子集。在每个周期中，选择哪个字作为广播字，以及选择哪些与其他 bank 对应的线程，都是不确定的。

图 4-12 是采用广播机制的内存读操作的例子。



（a）half-warp 从同一 32bit 字地址中读（无冲突）



（b）如果 bank5 的 32bit 字在第一步被选为广播字，则无冲突，否则造成 2-way bank conflict

图 4-12　共享存储器广播模式示例

4.4.3.2　通过 kernel 参数使用共享存储器

共享存储器保存着加载 kernel 时传递过来的参数，以及 kernel 执行配置参数。如果参数列表很长，建议将其中的一部分参数放入 constant memory，并从中取用。

对于参数列表很长的 kernel，将一些参数放入 constant memory 可以节约共享存储器。

4.4.4　使用纹理存储器和常数存储器加速

4.4.4.1　纹理存储器

纹理存储器能够通过缓存利用数据的局部性，提高效率。它的主要用途是用于存放图像和查找表。使用 texture 时的好处有：

- 不用严格遵守合并访问条件，也能获得很高带宽。
- 对于随机访问，如果要访问的数据并不是很多，效率也不会特别差。
- 可以使用线性滤波和自动类型转换等功能调用硬件的不可编程计算资源，而不必占用可编程计算单元。

4.4.4.2　常数存储器

常数存储器主要用于存放指令中的常数。对当前硬件来说，如果一个 half-warp 的线程访问常数存储器中相同的一个数据，可能只需要一个周期就可以获得这个数据。实际使用 constant memory 时速度一般还是低于立即数或者 shared memory，但还是明显高于将数据存放在显存中的情况。如果有因为尺寸太大无法存放在寄存器或者 shared memory 中的查找表一类的常数数组，可以考虑将其放在常数存储器中获得一定的加速。

4.5　指令流优化

SM 处理一条 warp 指令，首先为 warp 里的每个线程读指令操作数，执行指令，最后为

warp 里的每个线程写入计算结果。因此，有效的指令吞吐量不仅取决于名义上的指令吞吐量，还取决于内存延迟和带宽。建议采取以下手段增大指令吞吐量：

- 避免使用低吞吐量指令。
- 对每种类型的存储器进行优化，有效利用带宽。
- 允许线程调度单元尽量用多的数学计算来覆盖访存延迟，这就需要有高的算术密度（或者说，对于每一个内存操作都有大量的算术操作来覆盖），同时每个多处理器有很多的活动线程。

本节中，吞吐量是指每个多处理器在一个时钟周期下执行的操作数目。对于大小为 32 的 warp，一条指令由 32 个操作构成。因此，如果记 T 为每个时钟下的操作数目，那么指令吞吐量就是每 32/T 个时钟周期一条指令。

所有的吞吐量都是针对一个多处理器而言的。所以，要计算整个设备的吞吐量需要乘以设备的多处理器个数。

对程序中需要多次运行的代码进行指令级优化可以有效地提高指令流的吞吐量。本节将对各种指令进行详细讨论。

4.5.1 算术指令

CUDA 的硬件特别适合进行单精度浮点运算，因此应该尽量使用单精度浮点单元进行计算。

如果在代码中可以使用单精度浮点代替双精度浮点，那么我们强烈建议使用 float 型和单精度浮点数学函数。如果代码中使用了双精度，那么将代码编译到不支持双精度的硬件设备上时，例如计算能力为 1.2 或更低的设备，每个双精度的变量将会转成单精度格式（但大小仍是 64bit），并且双精度算术运算也会转为单精度算术运算。

下面是算术运算的吞吐量和使用时需要注意的问题，以及优化方法。

4.5.1.1 单精浮点基本算术运算

单精浮点加、乘、乘加运算的吞吐量是每个时钟周期 8 个操作。

求倒数运算的吞吐量是每个时钟周期 2 个操作。

单精浮点除操作是每个时钟周期 0.88 个操作，但是__fdividef(x, y)提供一个更快速的版本，能达到每个时钟周期 1.6 个操作。

4.5.1.2 单精浮点的平方根和倒数平方根

倒数平方根的吞吐量是每个时钟周期 2 个操作。

单精浮点平方根的计算方法是求倒数平方根的倒数，而不是在倒数平方根后做乘法，这是为了在处理 0 和无穷大时能计算得到正确的结果。因此，它的吞吐量是每个时钟周期 1 个操作。

4.5.1.3 单精浮点的对数运算

__logf(x)的吞吐量是每个时钟周期 2 个操作。

4.5.1.4　正弦和余弦运算

__sinf(x), __cosf(x),__ exp(x)的吞吐量是每个时钟周期 1 个操作。

sinf(x), cosf(x), tanf(x), sincosf(x)和相应的双精指令开销非常昂贵，尤其是 x 绝对值较大时更是如此，此时就要将 x 的绝对值减小，称为归约操作。

归约代码（参见 math_functions.h 中的实现）由两个代码路径组成，快路径和慢路径。快路径适用于参数较小的情况，它本质上是一些乘加操作。慢路径适用于参数较大的情况，包含一系列的计算，以在整个参数范围内求得正确结果。

目前，三角函数中用的是快路径，单精的话参数要小于 48039.0，双精的话要小于 2147483648.0。

慢路径比快路径要求更多的寄存器。为减小寄存器压力，归约操作可能会使用延迟较高的 local memory，速度会进一步降低。目前，在单精的函数中用了 28bytes 的 local memory，在双精函数中用了 44bytes，当然这个数字也不一而论。由于慢路径中冗长的计算和 local memory 的使用，所以使用慢路径的三角函数吞吐量会慢一个数量级。

需要同时计算 sin 和 cos 值时，可以使用 sincos 系列函数节约时间，包括：

- 高速版本的单精度浮点__sincosf()函数。
- 高精度的单精度浮点 sincosf()函数。
- 双精度浮点的 sincos()函数。

4.5.1.5　整数算术运算

整数加法的吞吐量是每个时钟周期 8 个操作。

32bit 整数乘的吞吐量是每个时钟周期 2 个操作，但是__mul24 和__umul24 提供有符号和无符号的 24 位整数乘法，吞吐量可以达到每个时钟周期 8 个操作。在未来的架构中，__[u]mul24 将会比 32it 整数乘更慢，应该为不同版本硬件编译不同的程序。

整数除和模运算开销特别大，应尽量地避免或用位运算代替。例如如果 n 是 2 的幂次方，那么(i/n)与(i>>log2(n))是等价的，(i%n)与(i&(n-1))是等价的，如果 n 在程序中已经固定的就是 2 的幂次方，那么编译器将会自动进行这些转换。

4.5.1.6　比较运算

比较、min、max 操作的吞吐量是每时钟周期 8 个操作。

4.5.1.7　位运算

任何位运算的吞吐量是每时钟周期 8 个操作。

4.5.1.8　类型转换

类型转换的吞吐量是每时钟周期 8 个操作。

有些时候，编译器会插入一些转换指令，也就引入了一些额外的执行周期。像这种情况：

- 操作于 char 或 short 上的函数，这些操作数一般需要转换成 int。

- 双精浮点常量（定义的时候没有任何后缀）作为单精浮点计算的输入。

最后这种情况能通过使用单精浮点常量解决，声明变量的时候加上后缀，例如3.141592653589793f、1.0f、0.5f。

4.5.2 控制流指令

控制流指令（if、switch、do、for、while）可能引起一个 warp 内的线程跳转到不同的分支，这将严重地影响指令吞吐量。一旦发生分支，那么不同的执行路径就必须被串行地执行，导致这个 warp 中指令总数增多。当所有分支的指令都执行结束以后，这些线程才会重回到同一条执行路径上。

在控制流只与线程的 ID 有关时，各 warp 在 block 中的分布是确定的，应该修改控制条件，尽量避免在 warp 内发生分支。例如，当控制条件只取决于(threadIdx/warp size)时， warp 内就不会出现分支，因为此时控制条件是严格按照 warp 对齐的。

有的时候，编译器可能会展开循环或通过谓词执行优化 if、switch 语句，这种情况下，warp 也不会分支。程序员同样能够使用#pragma unroll 指令控制 loop unrolling。

谓词执行是编译器与硬件共同完成的：编译器将几个分支中的指令作为一个代码块处理，并将其中的指令与谓词关联；而硬件在执行时则根据谓词有选择地执行这些语句，从而将控制依赖转换为数据依赖。使用谓词执行时，依赖于控制条件的指令并不会被跳过。在编译阶段，这些依赖于控制的指令都会与一个值为 true 或 false 的谓词相关联，在执行时，尽管分支中的指令仍然会被调度执行，但只有谓词为真的指令才会真正被执行。如果指令在某条分支中的谓词为假，虽然这条指令还是要跟着预测值为真的指令一起执行，但它不用将结果写回，也不用计算访存地址或者读取操作数。

在由分支条件控制的指令数目小于等于某个阈值的情况下，编译器会自动使用谓词指令替换分支指令。如果编译器认为某个条件有可能造成很多存在分支的 warp，那么阈值是 7，否则阈值是 4。

4.5.3 访存指令

访存指令包括任何读写 shared/local/global memory 的指令。对 local memory 的访问只有在寄存器不够用或者编译器无法解析地址时才会发生。

由于寄存器文件的大小、shared memory 每 bank 的宽度，以及对显存进行合并访问时达到最大带宽的访问宽度都是 32bit，因此在读写时将较大的数据（如 float3，double）拆分成每线程 32bit，或者将多个[u]char 或[u]short 合并成每线程 32bit 的形式访问，可以获得更高的性能。

存储操作的吞吐量是每个时钟周期 8 个操作。当访问 local/global memory 时，会有额外的 400～600 个时钟周期的访问延迟。

举个例子，如下赋值操作的吞吐量：

```
__shared__ float shared[32];
__device__ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

对 global memory 进行的读操作的吞吐量是每个时钟周期 8 个操作，在进行 shared memory 读、写的时候也是每个时钟周期 8 个操作，然而读、写 global memory 时还有 400～600 个时钟周期的访存延迟。

如果有足够多的算术指令可以在访存期间被发射、执行，那么这些执行的时间就可以隐藏大部分 global memory 访存延迟。不过，总的来说，对显存的访问还是越少越好。

4.5.4　同步指令

在每个线程都不用等待任何其他线程的情况下，__syncthreads()的吞吐量是每时钟周期 8 个操作。

4.6　CUDA profiler 的使用

CUDA profiler 是 NVIDIA 公司在 CUDA toolkit 中提供的用于分析 CUDA 程序性能的程序。CUDA profiler 位于 CUDA toolkit 安装目录的 cudaprof\bin 目录中。在 cudaprof\doc 目录中提供了 CUDA profiler 相关文档，而 projects 目录中则是对 SDK 中一些自带项目进行分析的工程。

打开 cudaprof.exe，单击菜单中的 File→New，或者点击工具栏中的新建按钮，就会弹出建立分析项目（.cpj）的对话框，在这个对话框里可以选择分析项目的名称和项目存储路径。

设置完以后会自动弹出建立分析会话的对话框。一个分析项目中可以存在多个分析会话。进行分析时，先设置好要分析的 exe 文件、工作目录、分析时的命令行和允许的最大运行时间等，然后保存这些参数，或者直接运行。如果以后要修改分析会话的参数，可以选择菜单 Profiles→Session Settings，或者工具栏中的 Session Settings 按钮更改配置。单击 Profiles→Start，或者工具栏中的 Start 按钮，就可以按照分析会话的配置启动 CUDA 程序，并记录分析结果。分析结果是对 CUDA 程序中每一个 GPU 函数（包括 memcopy 和内核函数）各项的测试结果。

4.6.1　图形分析

除了测试结果列表，CUDA profiler 也提供了直观的图形供程序员参考，这几种图形是：

（1）Summary Plot：绘出所有 kernel 函数和 memory copy 在总 GPU 时间中占用的百分比。

（2）GPU Time height Plot：按照调用次序绘出所有 kernel 函数和 memory copy 占用时间的柱状图。

（3）GPU Time Width Plot：按照调用次序绘出所有 kernel 函数和 memory copy 占用时间，更加直观地反映了整个程序的运行状况。

通过更改菜单中 Options 中的各项设置，可以修改上述图表中显示的项目数量和方式。

4.6.2　图表分析

CUDA profiler 测试结果可以用以下几种图表进行分析：

（1）Profiler output table：该表中会列出会话中设置的要分析的项目的测试结果。

（2）Summary table：该表中列出会话中所有 GPU 函数的主要分析结果，包括调用次数、占用 GPU 时间、占用 GPU 百分比、存储器读写带宽等。

（3）Memory table：列出 memory copy 的调用次数、传输数据大小和传输方向。

4.6.2.1 图表分析之 profiler output table

在 profiler output table 中，可以看到每个 GPU 函数的详细信息，包括：

（1）GPU Timestamp：函数开始时的时戳。

（2）Method：GPU 函数名称。GPU 内核函数会用内核函数的名字，而存储器拷贝则会被命名为“memcpy*”，例如 memcpyDToHasync，说明这是一次从显存到内存的异步传输。

（3）GPU Time。

（4）CPU Time。

（5）Stream Id：流处理中各个流的 Id。

（6）内核函数的分析指标。

- Occupancy：SM 占用率，即每个 SM 上的活动 warp 数量与允许的最大活动 warp 数量之比。
- Profiler 计数器，如表 4-2 所示。

表 4-2 profiler 内核函数分析指标

名称	意义
gld uncoalesced	对 global memory 的非合并装载（读）数量
gld coalesced	对 global memory 的合并装载数量
gld request	global memory 的装载请求数量（只能对计算能力 1.2 及以上 GPU 使用）
gld_32/64/128b	合并宽度为 32Byte、64Byte 或 128Byte 的 global memory 装载访问数量（只能对计算能力 1.2 及以上 GPU 使用）
gst uncoalesced	对 global memory 的非合并存储（写）数量
gst coalesced	对 global memory 的合并存储数量
gst request	global memory 的存储请求数量（只能对计算能力 1.2 及以上 GPU 使用）
gst_32/64/128b	合并宽度为 32Byte、64Byte 或 128Byte 的 global memory 存储访问数量（只能对计算能力 1.2 及以上 GPU 使用）
local load	local memory 装载操作数量
local store	local memory 存储操作数量
tlb hit	指令和常数缓存命中数量
tlb miss	指令和常数缓存不命中数量
sm cta launched	每个 SM 中的活动 block 数量
branch	kernel 中的分支数量
divergent branch	kernel 中不走向同一分支的分支数量
instructions	执行的指令数量

续表

名称	意义
warp serialize	含有由于访问 shared 或者 constant memory 造成的串行操作的 warp 数量
cta launched	Number of threads blocks executed
grid size X	grid 在 x 维度上的尺寸
grid size Y	grid 在 y 维度上的尺寸
block size X	block 在 x 维度上的尺寸
block size Y	block 在 y 维度上的尺寸
block size Z	block 在 z 维度上的尺寸
dyn smem per block	每个 block 中由动态分配方式分配的 shared memory 大小
sta smem per block	每个 block 中由静态分配方式分配的 shared memory 大小
reg per thread	每个线程的寄存器大小

（7）存储器传输的分析指标。

- mem transfer size：传输的数据大小，单位为字节。

4.6.2.2　图表分析之 profiler summary table

在 profiler summary table 中显示的项目包括：

- Method：函数名称。
- #calls：函数调用次数。
- GPU usec：GPU 时间，单位为毫秒。
- CPU usec：CPU 时间，单位为毫秒。
- %GPU time：GPU 时间占总时间的百分比。
- 每个分析结果的总数。
- glob mem read throughput(GB/s)：Global memory 的实际读操作带宽。
- glob mem write throughput(GB/s): Global memory 的实际写操作带宽。
- glob mem overall throughput(GB/s)：Global memory 的实际带宽。
- instruction throughput：每个内核函数的指令吞吐量。

4.7　优化应用举例

4.7.1　矩阵乘法的优化

4.7.1.1　matrixMul 0

下面给出矩阵乘的基本实现。如图 4-13 所示，基本的矩阵乘法使用了带状划分，每个线程负责读取 A 中的一行和 B 中的一列，并计算出 C 中相应位置上的值。于是，整个 kernel 从

全局存储器对 A 矩阵进行了 B.width 次读取、对 B 矩阵进行了 A.height 次读取，才能完成对矩阵 C 的计算。

为讲解方便起见，假设数组在每个维度上的尺寸都是 BLOCK_SIZE 的整数倍。本例中，BLOCK_SIZE 为 16，block 尺寸为（16,16），grid 尺寸为（8,5）。数组 A、B、C 的大小如图 4-13 所示，图中的每个小矩形块（也称 tile 或 blocking）是 16*16 的数组元素块。

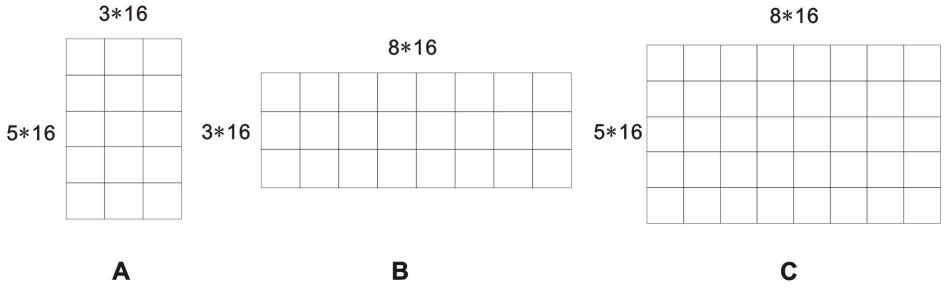


图 4-13　C = A * B

以下是主机端代码，A、B 矩阵均按行存储，且各维度均是 BLOCK_SIZE 的整数倍。

```
#define BLOCK_SIZE 16
void MatMul(const Matrix A, const Matrix N, Matrix C)
{
    //为 A、B 加载做好准备，分配显存空间并初始化
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void**)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,cudaMemcpyHostToDevice);
    略,d_B 的分配

    //为结果矩阵 C 分配空间
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc((void**)&d_C.elements, size);

    //启用 kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // 将结果矩阵 C 从显存中拷贝回内存
    cudaMemcpy(C.elements, Cd.elements, size,  cudaMemcpyDeviceToHost);

    //释放显存空间
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
```

```
    cudaFree(d_C.elements);
}
```

以下是 matrixMul 0 版本的 kernel 实现，每个线程都负责 C 中一个位置的元素值的计算，例如 tid 线程负责 C 中(r, c)位置的值，for()循环完成 A 中第 r 行与 B 中第 r 列对应元素的乘加操作。整个计算过程如图 4-14 所示。

```
//kernel 定义
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // 每个线程负责计算 C 中的一个元素，并将值累加到 Cvalue
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                        * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

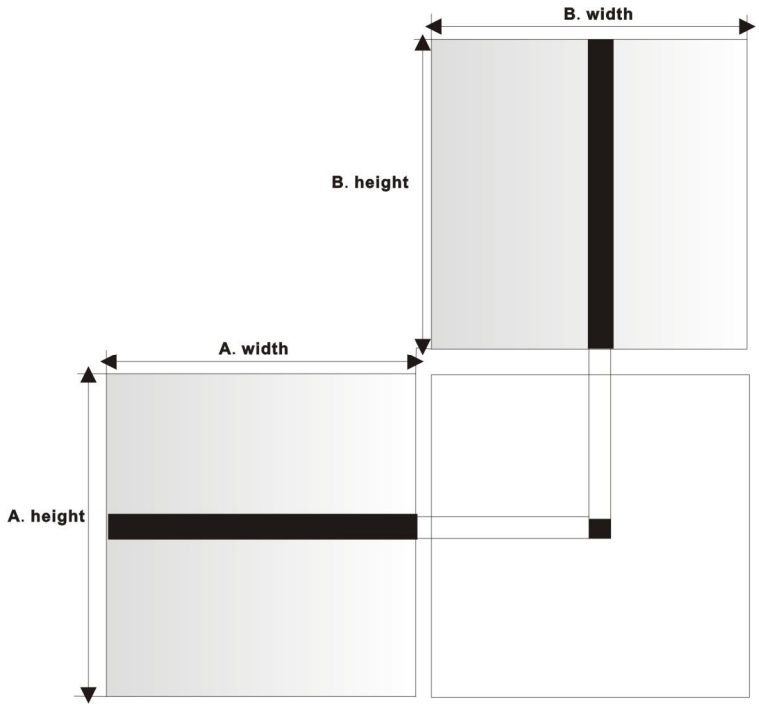


图 4-14　矩阵相乘中 C 中元素的计算

4.7.1.2　matrixMul 1

访存共享存储器的延迟远小于全局存储器，并且使用共享存储器进行线程间通信，还可以通过更灵活的算法获得更好的性能。以下代码对矩阵进行了棋盘划分，使用了共享存储器来

实现矩阵乘法。与上一个版本相比，这一个版本还能避免按列读取 B 中元素，引起的非合并访问造成的性能下降。

//------------------matrixMul_kernel.cu------------------

```
__global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // 该 block 要处理的 A 中第一个子块的起始地址
    int aBegin = wA * BLOCK_SIZE * by;
    // 该 block 要处理的 A 中最后一个子块的起始地址
    int aEnd   = aBegin + wA - 1;
    // for 循环中每次迭代的 A 的步长
    int aStep  = BLOCK_SIZE;

    //该 block 要处理的 B 中第一个子块的起始地址
    int bBegin = BLOCK_SIZE * bx;
    // for 循环中每次迭代的 B 的步长
    int bStep  = BLOCK_SIZE * wB;

    //Csub 用于存储该线程进行乘加操作的结果值
    float Csub = 0;

    //循环 A、B 子块，对相应元素进行乘加
    for (int a = aBegin, b = bBegin; a <= aEnd;  a += aStep, b += bStep) {
        //声明用于存储 A、B 子块的共享存储器数组
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // 完成数据全局存储器到共享存储器的拷贝，每个线程负责一个元素
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];
        __syncthreads();

        // 进行两个子块的乘加，每个线程负责 C 中一个元素值的计算
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);
        __syncthreads();
    }

    //向全局存储器的写回操作，一个线程负责一个元素，该 block 负责一个 C 子块
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

在这个实现中，每个 block 负责数组 C 中的一个方块 Csub 的计算，其中每个线程负责计算 Csub 的一个元素。如图 4-15 所示，Csub 是两个矩形矩阵的乘：A 的 sub-matrix，维度是(A.width, block_size)，与 Csub 有相同的行索引；B 的 sub-matrix，维度是(block_size, A.width)，与 Csub 有相同的列索引。为了适应设备资源，两个矩形矩阵被分为大小为 block_size 的正方形矩阵，Csub 的计算就是这些小正方形矩阵的乘积的和。这些乘积如下计算：首先从 global memory 加载两个相应的正方形矩阵到 shared memory，一个线程负责加载一个元素；接下来每个线程负责计算乘积中的一个元素。每个线程累积每个这样乘积的结果到一个寄存器中，一旦结束就将结果写回 global memory。

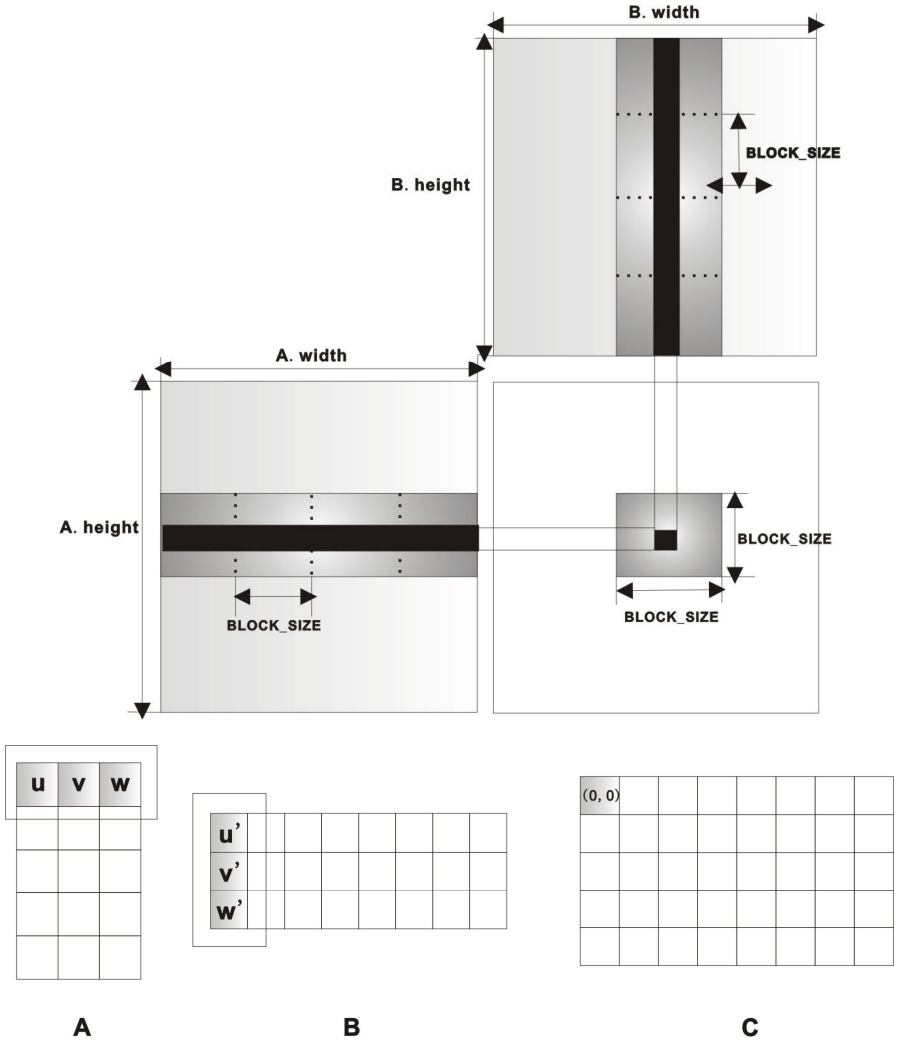


图 4-15　分块计算矩阵乘法

这种棋盘划分方式利用了高速的 shared memory，同时也节约了大量 global memory 带宽，因为 A 只被读 B.width/block_size 次，B 也只被读 A.height/block_size 次。

具体是这样实现的：例如 block(0,0)需要由 A 的一行 tile 和 B 的一列 tile 计算得到，如图 4-15 中矩形框标注的部分。对于每一个 block：

- 第一次循环，从 global memory 中取出 A 的 u 块和 B 的 u′块，分别放入 shared memory 中的 AS[16][16]、BS[16][16]中，然后 block 中的每个 thread 根据 AS、BS、tx、ty 计算出一个 Csub。
- 第二次 for 循环中，从 global memory 中取出 v 和 v′块并放入 AS、BS 中，每个 thread 在原来的 Csub 基础上再进行加和。
- 依此类推。
- 最后一次循环中，w 和 w′被放入 AS、BS，每个 thread 做最后的 Csub 加和，这时的 Csub 就是每个 thread 计算出来的最终结果了，在这里相当于计算出了（0，0）tile。最后，每个 block 中的 16*16 个线程，对应完成 C 中 16*16 个位置上（即 1 个 tile）的输出，输出到显存中。

这个代码实现中不存在 bank conflict 问题。因为在进行两个子块乘加的 for()循环中，每 half-warp 访问的是 AS[]一行的元素，分别分布在 16 个 bank，对 BS[]的访问也分布在 16 个不同的 bank，不存在 bank conflict 问题。

但这个版本仍有许多地方需要改进，例如数组大小必须是 BLOCK_SIZE 的整数倍，每个 tile 会被重复取出多次。访问显存的高延迟，使得我们必须思考如何将“好不容易”取到的数据得到充分使用。限于篇幅，这里仅给出一些提示：

（1）可以使用 cudaMallocPitch()来解决数组各维大小不是 2 的幂次方的问题。它会在 global memory 中做好数组宽度、起始地址的工作，自动以最佳倍数来分配内存。

（2）可以使用 cudaMemcpy2D()进行二维数组的复制，显存上数组 pitch 与内存上数组 pitch 可以不同。

（3）如果是浮点运算，可以借助 Kahan’s Summation Formula 提高计算精度。

4.7.1.3 matrixMul 2

除了共享存储器，我们也应该充分利用 GPU 的寄存器资源。register 是矢量计算单元（一条指令可取 4 个），shared memory 是标量单元（一条指令只可取一个），因此理论上使用寄存器更加快速。事实上，每个 SM 都拥有 8K 或 16K 的寄存器资源，线程少的话每个线程分得的 register 还是很可观的。根据这个思想，下面给出了改进的 multrixMul 2 版本矩阵乘法。

在这个版本的代码中，A、B、C 均按列存储，并且数组各维大小均是 BLOCK_SIZE 整数倍，A、B 矩阵均为 64*64，而 block 的维度是(16,4)。B 被划分为 16*16 的子块，并借助共享存储器进行计算；对 A、C 按列做 vector，这样保证对显存中 C 的 stride-1 访问。

```
__global__ void sgemmNN( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k)
{
    //计算每个线程的起始指针
    A += blockIdx.x * 64 + threadIdx.x + threadIdx.y*16;
    B += threadIdx.x + ( blockIdx.y * 16 + threadIdx.y ) * ldb;
    C += blockIdx.x * 64 + threadIdx.x + (threadIdx.y + blockIdx.y * ldc ) * 16;

    __shared__ float bs[16][17];
    float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
```

```
    const float *Blast = B + k;
    do
    {
#pragma unroll
        //读 B 中 16*16 的块到 bs[][]，从全局存储器到共享存储器
        for( int i = 0; i < 16; i += 4 )
            bs[threadIdx.x][threadIdx.y+i] = B[i*ldb];
        B += 16;
        __syncthreads();
#pragma unroll

        //每个线程完成 A 中一列与 bs[][]中一列对应元素的乘加，每个 block 完成 A 中列与 bs[][]
        for( int i = 0; i < 16; i++, A += lda )
        {
            c[0] += A[0]*bs[i][0]; c[1] += A[0]*bs[i][1];
            c[2] += A[0]*bs[i][2]; c[3] += A[0]*bs[i][3];
            c[4] += A[0]*bs[i][4]; c[5] += A[0]*bs[i][5];
            c[6] += A[0]*bs[i][6]; c[7] += A[0]*bs[i][7];
            c[8] += A[0]*bs[i][8]; c[9] += A[0]*bs[i][9];
            c[10] += A[0]*bs[i][10];c[11] += A[0]*bs[i][11];
            c[12] += A[0]*bs[i][12];c[13] += A[0]*bs[i][13];
            c[14] += A[0]*bs[i][14];c[15] += A[0]*bs[i][15];
        }
        __syncthreads();
    } while( B < Blast );
    //略，结果写回全局存储器
}
```

下面以 block(0, 0)为例进行分析。每个 block 完成 C 中一个 16*64 矩阵的元素值的计算，其中每个线程完成一个 vector（16 行 1 列）的计算。

block0 中的 64 个线程在 A、B、C 矩阵中的起始地址如图 4-16（a）中的箭头所示。首先，以 64 个线程将 B 中的 16*16 数据块从全局存储器拷贝到共享存储器，注意这里进行的是带转置的拷贝。拷贝后 bs[][]的结果如图 4-16（b）所示。在接下来的 for()循环中，每个线程执行了 A 中一列（16 行 1 列的 vector）和 bs[][]中一列对应元素的乘加操作，那么整个 block 就完成了 Asub(16*64)与 bs[][]的乘加，并将结果累加于 Csub(16*64)中。而 while()内的一轮循环就完成了 Asub(16*64 子矩阵)与 B 中一个 16*16 子矩阵的乘加。请注意，此时 64 个线程的 c[]组成的 16*64 的 Csub 并不是最终结果，还需要以 while(B<Blast)来遍历整个 Bsub(16*64)。在本例中，在矩阵 A、B 都是 64*64 的情况下，while()循环会执行 4 次。接下来的执行过程如图 4-16（d）所示，对 A 中每一个 Asub(16*64)与 B 中相应的(16*16)矩阵块进行计算。循环 4 次以后最后累加得到的即为最终的 Csub，此时将结果传回全局存储器即可。

这与早期矢量处理器上的两个算法——Agarwal、Gustavson 为 IBM 3090 Vector Facility，以及 Anderson et al.为 Cray X1 设计的算法思想相同。A、C vector 存储于矢量 register 或 cache 中，B blocking 存储于另外一个可被 A 中不同元素共享的内存上，这说明了 CUDA 与一些传统超级计算机在架构上存在相似性。

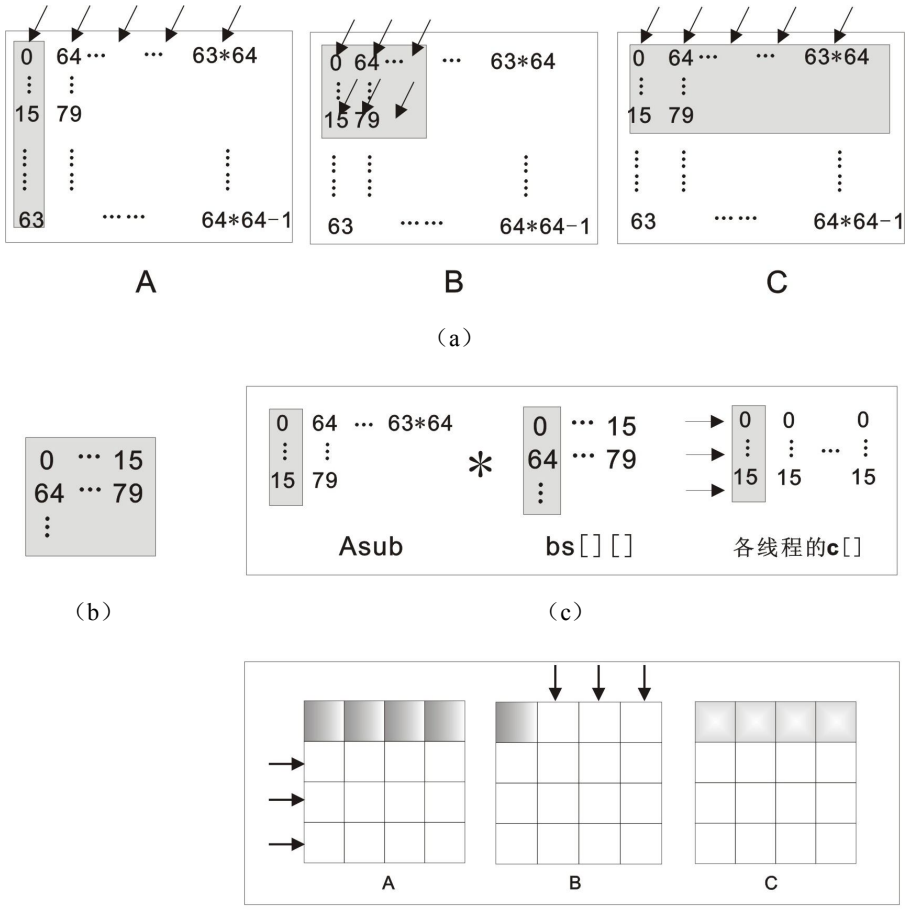


图 4-16　利用寄存器进行矩阵乘法

4.7.2　并行归约的优化

本节将通过 SDK 中的 reduction 示例演示对 CUDA 程序进行逐步优化的过程。reduction 并行归约（或叫并行缩减）提供了一种可以实现并行求和、求乘、求最大（小）值等各种操作的方法，用途十分广泛。本节的 reduction 示例中使用的运算是加，可以实现对数组元素的并行求和操作。下面给出 7 个版本的 kernel 程序，并逐一进行分析。在这一段代码中，假设有 6 个一维 block，每个 block 有 512 个线程，而输入数组中的元素数量为 3072 个，由每个线程对应处理一个数据。

4.7.2.1　reduce0

以下是 reduce0 版本的 kernel 程序。

//----------------reduction_kernel.cu----------------

```
#ifndef _REDUCE_KERNEL_H_
#define _REDUCE_KERNEL_H_
```

```
#include <stdio.h>
#include "sharedmem.cuh"

#ifdef __DEVICE_EMULATION__
#define EMUSYNC __syncthreads()
#else
#define EMUSYNC
#endif

// Macros to append an SM version identifier to a function name
// This allows us to compile a file multiple times for different architecture
// versions
// The second macro is necessary to evaluate the value of the SMVERSION macro
// rather than appending "SMVERSION" itself
#define FUNCVERSION(x, y) x ## _ ## y
#define XFUNCVERSION(x, y) FUNCVERSION(x, y)
#define FUNC(NAME) XFUNCVERSION(NAME, SMVERSION)

/*
    并行归约求和程序，使用了共享存储器
    - 对 n 个输入元素有 log(n)个执行步
    - 使用 n 个线程
    - 输入数组元素个数需是 2 的幂次方
*/
/* 这个版本的 reduction 程序使用取模操作来确定哪些线程是活动线程。取模操作在 GPU 上是很慢的，而
且这样的线程模式下 warp 内 32 个线程不都是活动的，这就造成了低效率。
*/
template <class T>
__global__ void
FUNC(reduce0)(T *g_idata, T *g_odata)
{
    SharedMemory<T> smem;
    T *sdata = smem.getPointer();

    //加载共享存储器
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // 在共享存储器上进行 reduce 操作
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        //取模操作是很慢的
        if ((tid % (2*s)) == 0) {
            sdata[tid] += sdata[tid + s];
        }
```

```
        __syncthreads();
    }

    //将 block 的计算结果写回全局存储器
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

上段代码中,首先由每个线程读入一个数据，完成从全局存储器向共享存储器的数据拷贝，随后进行了一次同步操作，以保证所有数据都可以安全地被其他线程访问。在接下来的 for() 循环中，每一轮都只使用上一轮循环中一半的线程进行 tid 与 tid+s 的求和，即规约的过程。在这里，s 可以理解为跨度，对于每一个 block，有：

- 第一次循环（s=1），只有 tid=0、2、4、6、8…510 线程真正执行计算，即 sdata[tid] 与其后跨度为 1 的元素的加和操作。
- 第二次循环（s=2），只有 tid=0、4、8、12、16…508 线程真正执行计算，即 sdata[tid] 与其后跨度为 2 的元素的加和。
- 依此类推。
- 直到最后一次（s=256），只有 tid=0 线程真正执行计算，即 sdata[tid]与其后跨度为 256 的元素的加和。此时 sdata[0]中的结果即为该 block 输入的 512 个数据之和。

注意到，每次循环中都进行了一次__syncthreads()同步，这是因为下一轮循环中的线程要用到上一轮中其他线程计算的结果。reduce0 的执行过程如图 4-17 所示。

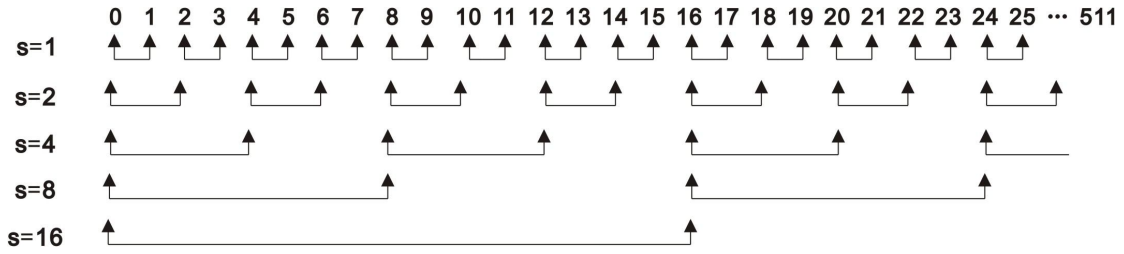


图 4-17　reduce0 加和过程

这一版本代码中，主要存在两个问题：

（1）在判断线程是否需要参与运算时使用了取模运算，由于 GPU 的整数处理单元功能较弱，做整数的模运算和除法运算的开销都特别大，应尽量避免使用或用位运算代替。

（2）每次循环时，只有 tid 是 2*s 倍数的线程才真正参与运算，因此每个 warp 中都只有部分线程执行加和操作（s=1 时 warp 中参与运算的线程最多，但也只有一半），但同时绝大多数 warp 都有线程要执行加和。于是，在分支时，绝大多数的 warp 都有要执行加法的线程，需要执行加法分支，只有少数的 warp 中的 thread 全部为空分支。

4.7.2.2　reduce1

针对 reduce0 中的“取模”问题，我们给出 reduce1 版本。

```
/* 这个版本使用连续线程，但它交叠的取址方式导致严重的分区冲突问题 */
template <class T>
__global__ void
```

```
FUNC(reduce1)(T *g_idata, T *g_odata)
{
    同 reduce0
    for(unsigned int s=1; s < blockDim.x; s *= 2)
    {
        int index = 2 * s * tid;

        if (index < blockDim.x)
        {
            sdata[index] += sdata[index + s];
        }
        __syncthreads();
    }

    //将本 block 的计算结果写回全局存储器
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

下面对 for()循环进行分析。对于每一个 block：

- 第一次循环（s=1），index<blockDim.x，即要求 2*tid<512，那么只有 tid=0、2、4、6…254 线程在做有效计算，sdata[2*tid]与其后跨度为 1 的元素加和。
- 第二次循环（s=2），4*tid<512，那么只有 tid=0、4、8、12…128 在做有效计算，sdata[4*tid]与其后跨度为 2 的元素加和。
- 依此类推。
- 直至最后一次循环（s=256），那么只有 tid=0 在做有效运算，sdata[0]+=sdata[256]。

其实，reduce0 与 reduce1 的计算逻辑是一样的，reduce1 通过引入 index 避免了取模运算。但是在 reduce0、reduce1 两个版本中，随着 s 的不断增加，都会造成越来越剧烈的 bank conflict。例如 T 为 float 类型，那么 s=1 时，0～15 号线程 half-warp 分别访问 sdata[1/3/5/../31]，形成 2 路冲突，图 4-18 中下划线标注；s=2 时，half-warp 会形成 4 路冲突，图中以带阴影的矩形进行标注；s=4 时，half-warp 会形成 8 路冲突，图中以不带阴影的矩形标注；以此类推。

Bank	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	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
					…								…			

图 4-18　reduce1 共享存储器 bank conflict 示意

4.7.2.3　reduce2

针对前两个版本中存在的 bank conflict 及 warp 分支问题，我们进一步分析 reduce2 版本。

```
/*
这个版本的 reduction 使用序列地址，避免了 bank conflict.
*/
template <class T>
__global__ void
FUNC(reduce2)(T *g_idata, T *g_odata)
{
    同 reduce0
    for(unsigned int s=blockDim.x/2; s>0; s>>=1)
    {
        if (tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    //将本 block 的计算结果写回全局存储器
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

reduce2 版本中，每个 thread 操作的元素都是相邻的，因此不会造成 bank conflict。例如对于每一个 block，第一次循环（s=256），tid=0…255 都会进行 sdata[tid]与 sdata[tid+256]元素的加和，所以说相邻 thread 操作相邻元素。此外，每个 warp 中的所有线程要么全部执行加法，要么全部不执行，也避免了分支效率问题。

4.7.2.4 reduce3

下面的 reduce3 版本使得相同的线程数可以完成更多元素的求和操作。上面的第三个版本已经将 shared memory 内的规约算法的效率挖掘殆尽了。剩下的工作就是在编程技巧和 shared memory 外做文章了。注意到上面的算法中，每次循环中需要访问的 shared memory 和参与运算的线程都比上一次要少。

下面的 reduce3 版本在访问 global memory 时就进行了一次加法，这样每个 block 只需要同样的线程，就能完成多一倍的元素的规约求和操作。

```
/*
    这个版本的程序使用了 n/2 个线程--This version uses n/2 threads --
    在从全局内存读的时候执行了第一层次的归约
*/
template <class T>
__global__ void
FUNC(reduce3)(T *g_idata, T *g_odata)
{
    SharedMemory<T> smem;
    T *sdata = smem.getPointer();
```

```
    //执行第一层次的归约（并行读数据），写入共享存储器
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
    __syncthreads();

    //在共享存储器做归约
    for(unsigned int s=blockDim.x/2; s>0; s>>=1)
    {
        if (tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    //将本 block 的计算结果写回全局存储器
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

还是 6 个 block，每个 block 有 512 个线程，第 0 个 block 的 512 个线程现在就负责前 1024 个数据（而不是原来的 512 个），例如对于线程 0 有 sdata[0]=g_idata[0]+g_idata[511]；第 1 个 block 负责 1024～2047 的数据加和；第 5 个 block 负责 5120～6144 的数据加和，而 6144 是之前能处理的数据即 3072 的两倍。

4.7.2.5　reduce4

根据 warp 内线程不需要同步的特点，我们进一步得到 reduce4 版本。这个版本中每次循环都进行了一次__syncthreads()同步。实际上，当只剩最后 32 个线程时，也就只有一个 warp 需要执行加法，其他的 warp 都执行空分支。由于一个 warp 内的运算总是满足顺序一致性的，因此在一个 warp 中也就不需要再进行同步了。

注意到条件编译#ifdef __DEVICE_EMULATION，这是为了减少 CPU 模拟运行时的分支。此外，由于 emu 模式下 CPU 的 warp size 是 1，为了能够得到正确的模拟结果，仍然需要在最后一个 warp 中加入 EMUSYNC。

```
/*
    这个版本的代码展开了最后一个 warp 以避免同步操作
*/
template <class T, unsigned int blockSize>
__global__ void
FUNC(reduce4)(T *g_idata, T *g_odata)
{
    同 reduce3
    for(unsigned int s=blockDim.x/2; s>32; s>>=1)
    {
        if (tid < s)
```

```
        {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
#ifndef __DEVICE_EMULATION__
    if (tid < 32)
#endif
    {
        if (blockSize >=  64) { sdata[tid] += sdata[tid + 32]; EMUSYNC; }
        if (blockSize >=  32) { sdata[tid] += sdata[tid + 16]; EMUSYNC; }
        略
        if (blockSize >=   2) { sdata[tid] += sdata[tid +  1]; EMUSYNC; }
    }
    // 将本 block 的计算结果写回全局存储器
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

4.7.2.6 reduce5

循环在 GPU 上的执行效率不高，reduce5 版本完全展开了循环。当每个 block 中的 thread 数量是 2 的 n 次方时，下面的代码可以获得最高的效率。

```
/*
    这个版本的代码完全展开了循环。使用了一个模板参数进行代码优化（线程数需是 2 的幂次方）。这
    要求主机端有一个 switch 语句来处理不同大小的线程块。
*/
template <class T, unsigned int blockSize>
__global__ void
FUNC(reduce5)(T *g_idata, T *g_odata)
{
    同 reduce4
    // 在共享存储器进行归约操作
    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }
    同 reduce4
}
```

4.7.2.7 reduce6

最后的版本中在循环读取 global memory，并将读到的数据直接加到 shared memory 中。在这个循环中，绝大多数 warp 都能保持满载，因此可以提高效率。

```
/*
这个版本的代码中，每个线程顺序地加若干个元素，这就减少了整个算法的开销，而整个复杂度还维持在
O(n)（step complexity 为 O(logn)）
```

```
*/
template <class T, unsigned int blockSize>
__global__ void
FUNC(reduce6)(T *g_idata, T *g_odata, unsigned int n)
{
    SharedMemory<T> smem;
    T *sdata = smem.getPointer();

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    //每个线程归约若干个元素，具体个数由活动线程块决定（通过 gridSize）。越多的块导致越大的 gridSize，
每个线程处理的元素越少。
    while (i < n)
    {
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];
        i += gridSize;
    }
    __syncthreads();

    同 reduce5
}
```

4.7.2.8　选择 kernel

在下面的主机端代码中，可以根据不同的数据规模选择不同的 kernel 进行计算，并且使用了模板以适应不同的数据类型。

```
template <class T>
void
FUNC(reduce)(int size, int threads, int blocks,
             int whichKernel, T *d_idata, T *d_odata)
{
    dim3 dimBlock(threads, 1, 1);
    dim3 dimGrid(blocks, 1, 1);
    int smemSize = threads * sizeof(T);

    //选择要加载哪个版本的 reduction 程序
    switch (whichKernel)
    {
    case 0:
        FUNC(reduce0)<<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata);
        break;
    略//case 1、2、3 与 case 0 类同
```

```
    case 4:
        switch (threads)
        {
        case 512:
            FUNC(reduce4)<T, 512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        略//256、128、…、2
        case  1:
            FUNC(reduce4)<T,    1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        }
        break;
    略//case 5、6、default 与 case4 类同
}
extern "C"
void FUNC(reduceInt)(int size, int threads, int blocks,
                     int whichKernel, int *d_idata, int *d_odata)
{
    FUNC(reduce)<int>(size, threads, blocks, whichKernel, d_idata, d_odata);
}
//同理，有 FUNC(reduceFloat)、FUNC(reduceDouble)等
#endif
```

4.7.3 矩阵转置的优化

Transpose 是一个矩阵转置程序。本节将通过三个版本的 CUDA 矩阵转置程序介绍如何在访问 global memory 时避免非合并访问和分区冲突问题（partition conflict，见 3.3.3.3 节），以及如何利用 shared memory 进行线程间通信，实现效率更高的算法。

首先，需要理解如何对矩阵进行棋盘划分实现转置。

如图 4-19 所示的矩阵按照棋盘划分为多个 3*3 的子块。(0,2)块经转置后位于输出矩阵的(2,0)位置，并且对块内元素也要进行相应转置。

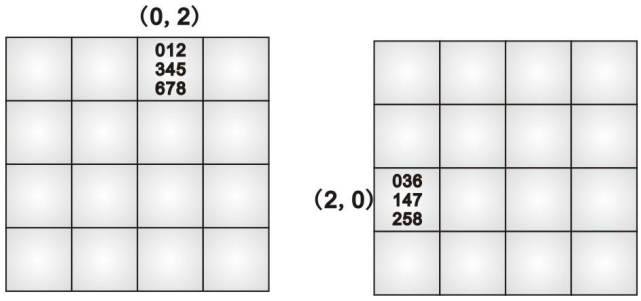


图 4-19　数组的分块转置

下面给出矩阵转置的主机端代码的主要部分。

```
//设置执行参数
dim3 grid(size_x / BLOCK_DIM, size_y / BLOCK_DIM, 1);
```

```
    dim3 threads(BLOCK_DIM, BLOCK_DIM, 1);

    //调用 transpose_naïve 或者 transpose 内核函数
    //transpose_naive<<< grid, threads >>>(d_odata, d_idata, size_x, size_y);
transpose<<< grid, threads >>>(d_odata, d_idata, size_x, size_y);
```

4.7.3.1　transpose_naive

//------------------transpose_naive_kernel.cu------------------

```
#define BLOCK_DIM 16
// naïve transpose 在向显存写入结果时不满足合并访问条件，严重影响了性能
__global__ void transpose_naive(float *odata, float* idata, int width, int height) {
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

    if (xIndex < width && yIndex < height)    {
        unsigned int index_in  = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

对于 256*4096 的数组，其 grid 如图 4-20 所示，图中每个小矩形代表一个 16*16 的 block，每个 block，负责处理输入数组对应位置上的一块数据。程序中 xIndex 和 yIndex 是线程在整个 grid 中的位置，而 index_in 和 index_out 分别是线程从输入矩阵读和向输出矩阵写的数据的地址。

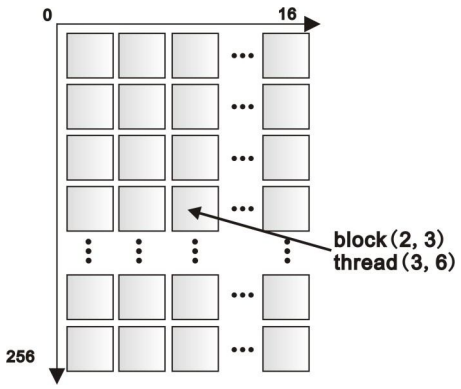


图 4-20　本例 grid 表示

每个 block 中的一个 half warp 按行连续读入数据，在读入数据时尚满足合并访问条件；但在向输出数组写入结果时，每个 half warp 写入的数据之间的间隔很大，不满足合并访问条件。此时，一个 half-warp 的访存请求就会产生 16 次单独的传输，这将大大降低可用的显存带宽。

4.7.3.2 transpose

注意到每个 16*16 分块中，在输入中处于同一列的数据处于输出矩阵同一行中，有可能可以用合并访问方式写入显存中。通过引入 shared memory 实现线程间通信，就可以让一个 half-warp 中的线程按照合并访问方式输入一行数据后，再以合并访问方式输出数据。

本例中，首先用合并访问方式将数据从显存读入共享存储器中，经过同步后，每个线程与和它按对角线对称的线程交换操作的数据，再按照合并访问方式将结果写到显存中。这种划分方式很好地说明了 CUDA 的两层并行：在同一个 block 中实现需要进行数据交换和通信的细粒度并行，而在各个 block 间实现不需要进行数据交换的粗粒度并行。

```
/* 这个 kernel 对全局内存器的所有读写均是合并读写，也避免了共享存储器中的 bank conflict 问题。这比此前的 naïve 版本要快 11 倍。注意，共享存储器大小是(BLOCK_DIM+1)*BLOCK_DIM。注意每一行都加了 1，这样才能保证 half-warp 按列访问数组的时候不发生 bank conflict。*/

__global__ void transpose(float *odata, float *idata, int width, int height) {
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1]; //静态分配 sharedMem

    //把矩阵块读入共享存储器
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if((xIndex < width) && (yIndex < height)) { //保证内存访问不会超过边界
        unsigned int index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }
    __syncthreads();

    //将转置后的矩阵写回全局存储器
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if((xIndex < height) && (yIndex < width)) {
        unsigned int index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

注意到 share memory 中的数组 block 大小被设成了 16×17，而不是 16×16。这样每行中处于同一列的数据就会被存储在不同的 shared memory bank 中，避免了 bank conflict。

transpose 和 transpose_naive 的效率可以相差一个数量级以上。从上面的例子可以看出，使用 shared memory 可以大大提高某些场合下程序的运行效率。

4.7.3.3 transposeDiagonal

在前两个版本的转置实现中，都存在分区冲突问题（见 3.3.3.3 节）。例如，在 GTX 280GPU 中存在 8 个分区，每个分区中存储连续的 256Byte 数据。如果输入数组的每行中有 2048 个 float 元素，那么 0～7 号元素处于 partition 0 中，而 8～15 号元素处于 partition 1 中，并依次类推，

直到 partition 7 中的第 504～511 个元素。而第 512～519 号元素又被存放在 partition 0 中。

前两个版本中，每个 block 从输入数组中读取一个 32×32 的子块，如图 4-21 所示，block 0 和 block1 读入的数据位于 partition 0 中，block2 和 block 3 读入的数据位于 partition 1 中，并依次类推。总的来说，各个 block 对各个 partition 的读访问请求的分布基本均衡。

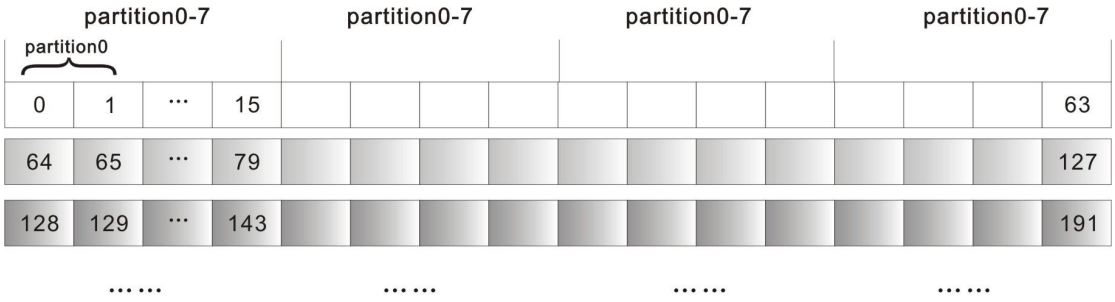


图 4-21　各 block 读取首地址的 partition 分布

但是在写入结果时，刚才的两个例子就发生了严重的分区冲突问题。如图 4-22 所示，block 0～block 63、block 64～block 127 要写入数据的地址都处于第 0 个 partition 中。虽然各个 block 的执行顺序有一定的随机性，但 ID 相近的 block 总的来说还是会集中在同一段时间内执行。那么这一段时间就只有一个 partition 对应的存储器控制器会处理来自所有 SM 的访存请求，使得可用显存带宽降低到理论值的 1/8。

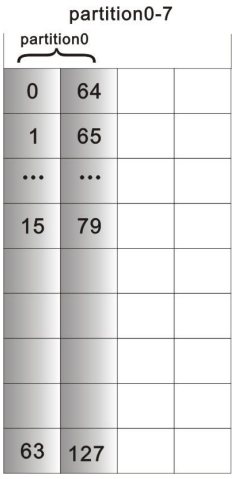


图 4-22　各 block 写回首地址的 partition 分布

那么应该如何设计，实现数据访问在多个 partition 间的负载均衡呢？这里给出了转置的第三个版本。

```
__global__ void transposeDiagonal(float *odata, float *idata, int width,int height, int nreps)
{
    __shared__ float tile[TILE_DIM][TILE_DIM+1];
    int blockIdx_y= blockIdx.x;
    int blockIdx_x= (blockIdx.x+blockIdx.y)%gridDim.x;
    int xIndex = blockIdx_x* TILE_DIM + threadIdx.x;
```

```
        int yIndex = blockIdx_y* TILE_DIM + threadIdx.y;
        int index_in = xIndex + (yIndex)*width;
        xIndex = blockIdx_y* TILE_DIM + threadIdx.x;
        yIndex = blockIdx_x* TILE_DIM + threadIdx.y;
        int index_out = xIndex + (yIndex)*height;

        for (int r=0; r < nreps; r++) {
                for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
                        tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
                }
                __syncthreads();
                for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
                        odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
                }
        }
}
```

在这个版本的代码中，重点是

```
int blockIdx_x= (blockIdx.x+blockIdx.y)%gridDim.x;
```

其中(blockIdx.x+blockIdx.y) 实现输入数据块的增量，取模控制着循环处理一行。图 4-23 和图 4-24 分别是各 block 读和写子块的分区分布。主对角线上的白色矩形是 block(0,0)～block(63,0)负责的子块；淡灰色矩形是 block(0,1)～block(63,1)负责的子块；灰色矩形是 block(0,2)～block(63,2)负责的子块；其他依此类推。

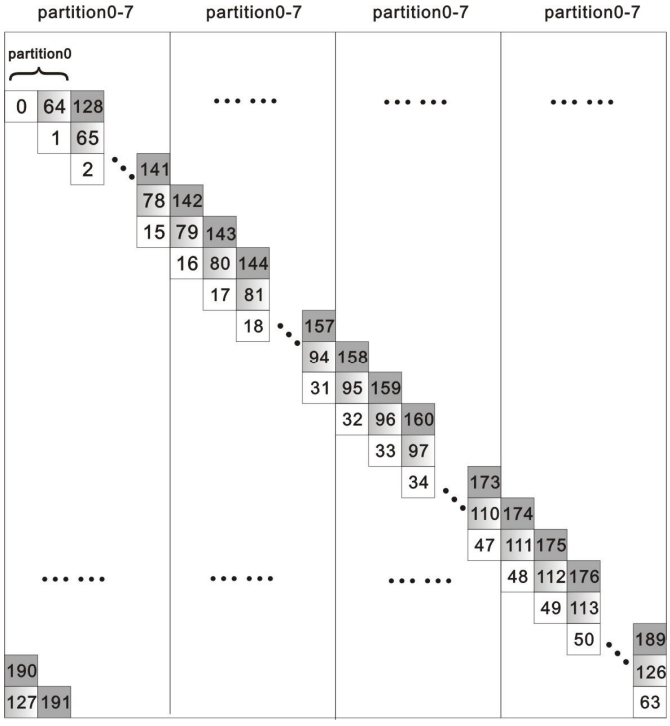


图 4-23　各 block 负责读入的子块的分区分布

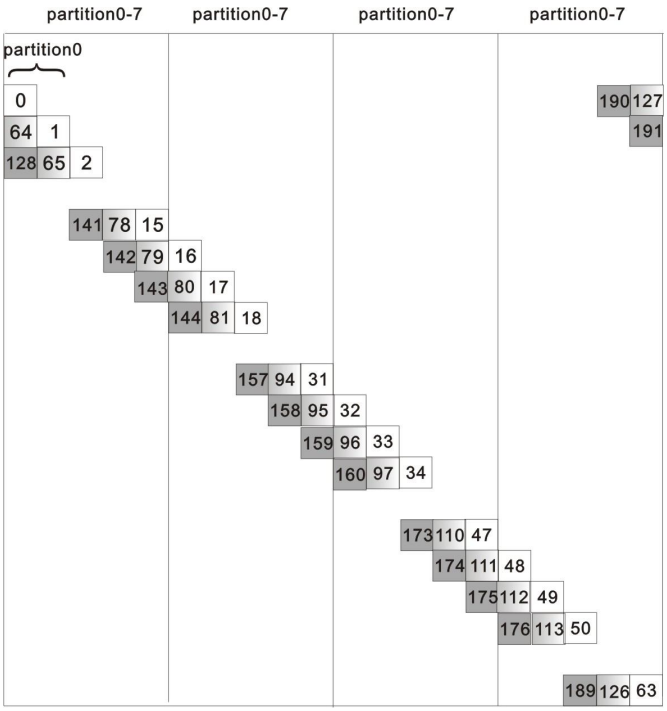


图 4-24　各 block 负责写回的子块的分区分布

从图 4-23 和图 4-24 中可以看出，在这一个版本中，相邻的 block 处理的是一系列子块位于与矩阵对角线平行的一条线上，从而使得从显存读写数据时在各个分区上的负载均匀分布，避免了 partition conflict 问题。