# 第九章 并行直方图计算模式

到目前为止，我们介绍的并行计算模式都将每个输出元素的计算任务分配给单个线程。因此，这些模式服从于owner-computes规则，其中每个线程都可以写入它们指定的输出元素，而不考虑其他线程的干扰。本章介绍了并行直方图计算模式，这是一种经常遇到的应用程序计算模式，其中每个输出元素都可能被所有线程更新。因此，在线程更新输出元素时，必须注意在线程之间进行协调，避免对最终结果的任何干扰破坏。在实际应用中，还有许多重要的并行计算模式无法轻易避免输出干扰，并行直方图计算模式是存在输出干扰的模式之一。我们将首先研究一种使用原子操作（atomic operations）将每个元素更新序列化到的基本方法。这种方法很简单，但效率很低，经常导致执行速度慢的令人失望。然后，我们将介绍一些广泛使用的优化技术，最显著的是私有化，以显著提高执行速度，同时保持正确性。这些技术的成本和效益取决于底层硬件以及输入数据的特征。因此，对于开发人员来说，理解这些技术的关键思想是非常重要的，以便对它们在不同环境下的适用性进行合理的推理。

## 9.1背景知识

直方图以连续的数字间隔显示数据项的频率。在直方图最常见的形式中，值间隔沿横轴绘制，每个间隔中数据项的频率表示为从横轴向上上升的矩形(或条形)的高度。例如，直方图可以用来显示短语“programming massively parallel processors.”中字母出现的频率。为了简单起见，我们假设输入短语都是小写的。通过检查，我们看到有4个字母“a”，0个字母“b”，1个字母“c”，以此类推。我们将每个值区间定义为由四种字母组成的连续范围。因此，第一个值区间是“a”到“d”，第二个值区间是“e”到“h”，以此类推。如图9.1所示，根据我们对值区间的定义，显示了短语“programming massive parallel processors”中字母出现频率的直方图。

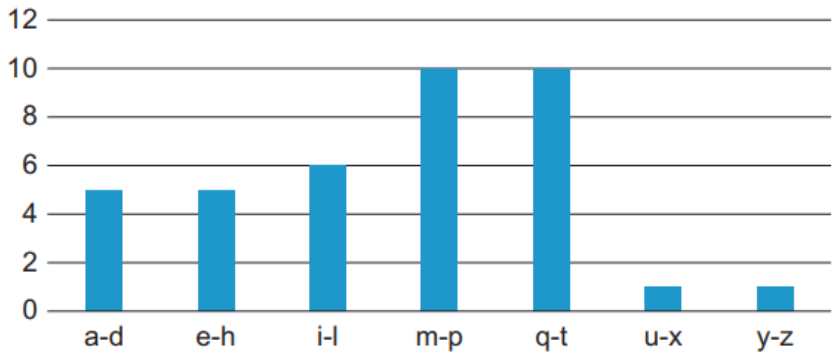


**FIGURE 9.1**

A histogram representation of “programming massively parallel processors.”

直方图提供了数据集的有用摘要。在我们的示例中，我们可以看到所表示的短语由字母组成，这些字母集中在字母表的中间段的字母中，后面段的字母则非常少。这样的直方图的形状有时被称为一个数据集的特性，是一个能够确定数据级是否有明显特征的快速方法。例如，信用卡账户的采购类别的直方图的形状可以用来检测非法交易。当柱状图的形状明显偏离标准时，系统就会发出疑似存在问题的信号。

许多其他应用领域也依赖直方图来进行数据分析。其中一个领域就是计算机视觉。不同类型的物体图像的柱状图往往呈现出不同的形状，比如人脸相比于汽车。通过将图像划分为子区域并分析这些子区域的直方图，就能快速识别出可能包含目标对象的图像或区域。图像子区域直方图的计算过程是计算机视觉中特征提取的基础，其中特征是指图像中感兴趣的模式。在实践中，每当需要分析大量数据以提取有趣

的东西(即“大数据”)时，直方图很可能被用作基础计算。信用卡欺诈检测和计算机视觉显然满足这一描述。其他有此类需求的应用领域包括语音识别、网站购买建议和科学数据分析，如天体物理学中的天体运动关联等等。

直方图可以很容易地按顺序计算，如图9.2所示，为简单起见，该函数只需要识别小写字母。C代码假设输入的数据集是由char组成的数组data[]，直方图将生成到int数组histo[](第一行)。数据项的长度在函数参数length中指定。for循环(第2行到第4行)依次遍历数组，将特定的字母索引标识到index变量中，并增加与该间隔对应的histo[index/4]元素。字母表索引的计算依赖于这样一个事实:输入字符串基于标准ASCII码表示，其中字母表字符“a”到“z”按照字母表顺序被编码为连续的值。

```
1.  sequential_Histogram(char *data, int length, int *histo) {
2.      for (int i = 0; i < length; i++) {
3.          int alphabet_position = data[i] - 'a';
4.          if (alphabet_position >= 0 && alphabet_position < 26) {
5.              histo[alphabet_position/4]++
6.          }
7.      }
8.  }
```

**FIGURE 9.2**

A simple C function for calculating histogram for an input text string.

虽然可能不知道每个字母的确切编码值，但可以假定字母的编码值是“a”的编码值加上该字母与“a”之间的字母位置差。在输入中，每个字符都存储在其编码值中。因此，表达式data[i] - " a "(第3行)计算出“a”的位置为0的字母位置。如果位置值大于或等于0且小于26，则数据字符实际上是小写字母(第4行)。请记住，我们定义了间隔，使每个间隔包含四个字母。因此，该字母的间隔索引是其字母位置值除以4。我们使用间隔索引来增加histo[]对应的数组元素(第4行)。

图9.2中的C代码非常简单和有效。数组元素在for循环中按顺序被访问，因此无论何时从系统DRAM中提取CPU高速缓存行，它们都可以很好地使用。histo[]数组是太小了，它非常适合放在CPU的一级(L1)缓存中，来确保非常快的更新到histo[]元素。对于大多数现代CPU，这段代码的执行速度是内存限制的，也就是说，受限于data[]元素从DRAM进入CPU缓存的速度。

## 9.2原子操作的使用

并行直方图计算的一个简单策略是将输入数组划分为几个部分，并让每个线程处理其中一个部分。如果我们使用P个线程，每个线程大约会做原来工作的1/P。在我们的讨论中，我们将把这种方法称为“方法一”。使用这种策略，我们应该差不多能够做到接近p倍的加速。图9.3使用我们的文本示例演示了这种方法。为了使示例与图片相匹配，我们将输入减少到短语的前24个字符。我们假设P = 4，每个线程处理一段6个字符。我们在图9.3中展示了这四个线程的工作量。

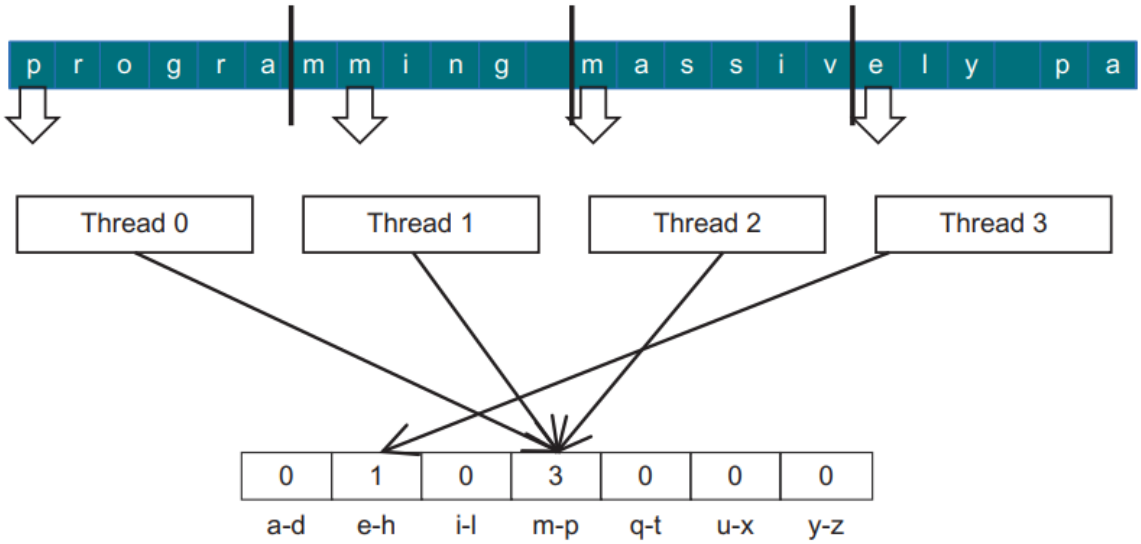


**FIGURE 9.3**

Strategy I for parallelizing histogram computation.

每个线程遍历其分配的部分，并为每个字符增加到对应的间隔计数器。图9.3显示了四个线程在第一次迭代中所采取的行动。注意，线程0、1和2都需要更新相同的计数器(m-p)，这是一种称为输出干扰的冲突。我们必须理解数据竞争发生的条件和原子操作的概念，以便在他/她的并行代码中安全地处理这样的输出干扰。

histo[]数组中间隔计数器的增量是对内存位置的更新或者说是读-改-写（read-modify-write）操作。该操作包括读取内存位置(读)、向读取内容添加一个值(修改)以及将新值写回内存位置(写)。读-改-写是一种通用操作，用于跨并发线程安全协调协作活动。

例如，当我们向一家航空公司预订航班时，我们打开座位地图并查找可用的座位(读)，我们选择一个座位来预订(修改)，并在座位地图中将座位状态更改为不可用(写)。可能会发生一个不好的潜在场景，如下:

- 两名乘客同时打开同一航班的座位地图。
- 两个顾客都选了同一个座位，比如9C。
- 两个客户都将座位图中9C座位的状态更改为不可用

在这一通操作之后，两个客户逻辑上得出结论，他们现在是座位9C的独家所有者。我们可以想象，当他们登上飞机，发现其中一人不能坐预定的座位时，他们会有一个不愉快的情况!信不信由你，由于机票预订软件的缺陷，这种不愉快的情况确实在现实生活中发生过。

另一个例子是，一些商店允许顾客不用排队等候服务。他们要求每位顾客从一个取号机器取一个号码。服务台一个接着一个显示下一个要服务的数字。当服务台空闲时，服务台要求客户提供与该号码匹配的票证，验证该票证，并将显示的号码更新为下一个更高的号码。理想情况下，所有顾客将按照他们进入商店的顺序得到服务。一个不好的结果是两个顾客同时在两个取号机器取号，并且都收到相同号码的票。一旦服务台呼叫那个号码，两个客户都会觉得他们才是应该得到服务的人。

在这两个示例中，不好结果都是由一种称为竞争条件的现象造成的，即两个或多个同步更新操作的结果取决于所涉及的操作的相对时间。有些结果是正确的，有些是不正确的。图9.4说明了在我们的文本直方图示例中，当两个线程试图更新相同的histo[]元素时的竞争状态。图9.4中的每一行表示一段时间内的活动，时间从上到下进行。

| Time | Thread 1 | Thread 2 |
|---|---|---|
| 1 | (0) Old ← histo[x] | |
| 2 | (1) New ← Old + 1 | |
| 3 | (1) histo[x] ← New | |
| 4 | | (1) Old ← histo[x] |
| 5 | | (2) New ← Old + 1 |
| 6 | | (2) histo[x] ← New |

(A)

| Time | Thread 1 | Thread 2 |
|---|---|---|
| 1 | (0) Old ← histo[x] | |
| 2 | (1) New ← Old + 1 | |
| 3 | | (0) Old ← histo[x] |
| 4 | (1) histo[x] ← New | |
| 5 | | (1) New ← Old + 1 |
| 6 | | (1) histo[x] ← New |

(B)

**FIGURE 9.4**

Race condition in updating a *histo[]* array element.

图9.4(A)描述了一个场景，线程1在时间段1到3完成了它的读-改-写序列的所有三个部分，然后线程2在时间段4开始它的操作序列。每个操作前面圆括号中的值显示了被写入的目的地的值，假设histo[x]的值最初为0。通过这种交错，histo[x]之后的值是2，正如人们所期望的那样。也就是说，两个线程都成功地增加了histo[x]元素。元素值开始时是0，在操作完成后是2。

在图9.4(B)中，两个线程的读-改-写序列重叠。注意，线程1在时间段4将新值写入histo[x]。当线程2在时间段3读取histo[x]时，它的值仍然是0。结果，它计算并最终写入到histo[x]的新值是1，而不是2。问题是线程2在线程1完成更新之前过早地读取了histo[x]。最终结果是histo[x]事后的值是1，这是不正确的。线程1的更新丢失了。

在并行执行期间，线程可以以任何相对于其他线程的顺序运行。在我们的例子中，线程2可以很容易地在线程1之前开始它的更新序列。9.5显示了两个这样的场景。在图9.5(A)中，线程2在线程1开始更新之前完成了它的更新。在图9.5(B)中，线程1在线程2完成更新之前开始更新。很明显，9.5(A)中的序列为histo[x]提供了正确的结果，而9.5(B)中的序列则产生了错误的结果。

histo[x]的最终值取决于相关操作的相对时间，这一事实表明存在竞争情况。我们可以通过防止线程1和线程2的操作序列交叉来消除这种变化。也就是说，我们希望允许图9.4(A)和9.5(A)所示的时间顺序，同时排除图9.4(B)和9.5(B)所示的可能性。这样的时间约束可以通过使用原子操作来强制执行。

| Time | Thread 1 | Thread 2 |
|---|---|---|
| 1 | | (0) Old ← histo[x] |
| 2 | | (1) New ← Old + 1 |
| 3 | | (1) histo[x] ← New |
| 4 | (1) Old ← histo[x] | |
| 5 | (2) New ← Old + 1 | |
| 6 | (2) histo[x] ← New | |

(A)

| Time | Thread 1 | Thread 2 |
|---|---|---|
| 1 | | (0) Old ← histo[x] |
| 2 | | (1) New ← Old + 1 |
| 3 | (0) Old ← histo[x] | |
| 4 | | (1) histo[x] ← New |
| 5 | (1) New ← Old + 1 | |
| 6 | (1) histo[x] ← New | |

(B)

**FIGURE 9.5**

Race condition scenarios where Thread 2 runs ahead of Thread 1.

对内存位置的原子操作指的是对该内存位置执行读-改-写顺序的操作，其方式是该位置的读-改-写序列与其他的不能重叠。也就是说，操作的读、修改和写部分构成了一个不可分割的单元，因此被称为原子操作。在实践中，原子操作是通过硬件支持来实现的，它可以锁定在同一位置上操作的其他线程，直到当前操作完成。在我们的示例中，这种支持消除了图9.4(B)和图9.5(B)中所描述的情况的可能性，因为在之前的线程完成更新之前，后面的线程不能开始它的更新序列。

重要的是要记住，原子操作并不强制特定的线程执行顺序。在我们的例子中，图9.4(A)和9.5(B)所示的顺序都是原子操作所允许的。线程1可以在线程2之前或之后运行。执行的规则是，如果两个线程中的任何一个开始对相同的内存位置进行原子操作，则在前导线程完成其原子操作之前，尾部线程不能对该内存位置执行任何操作。这有效地序列化了在内存位置上执行的原子操作。

原子操作通常根据在内存位置上执行的操作来命名。在我们的文本柱状图示例中，我们向内存位置添加一个值，因此原子操作称为原子添加，其他类型的原子操作包括减法、增量、减量、最小、最大、逻辑和或等。

CUDA程序可以通过函数调用在内存位置上执行原子添加操作:

```
int atomicAdd(int* address, int val);
```

> **内部函数**
>
> 现代处理器经常提供一些特殊的指令，这些指令要么执行关键功能(比如原子操作)，要么可以显著增强性能(比如向量指令)。这些指令通常作为内部函数暴露给程序员，或者简单地说明。从程序员的角度来看，这些是库函数。但是，编译器会以一种特殊的方式处理它们;每一个这样的调用都被翻译成相应的特殊指令。在最终的代码中通常没有函数调用，只有与用户代码一致的特殊指令。所有主要的现代编译器，如Gnu C Compiler (gcc)、Intel C Compiler和LLVM C Compiler都支持内部函数。

这个函数是一个内部函数，它将被编译成一个原子操作的硬件指令，该指令在全局或共享内存中读取地址参数所指向的32位内容，将val添加到旧内容中，并将结果存储回相同地址的内存中。该函数返回地址的旧值。

图9.6所示为CUDA kernel基于方法一进行并行直方图计算。第1行计算每个线程的全局线程索引。第2行将缓buffer中的数据总量除以线程总数，以确定每个线程要处理的字符数。使用第2章数据并行计算中介绍的上限公式来确保输入缓冲区的所有内容都得到了处理。注意，最后几个线程可能会处理只被部分填充的部分。例如，如果我们在输入缓冲区中有1000个字符和256个线程，我们将为前250个线程中的每个线程分配(1000−1)/256 + 1 = 4个元素。最后6个线程将处理空部分。

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
1.  int i = threadIdx.x + blockIdx.x * blockDim.x;
2.  int section_size = (size-1)  / (blockDim.x * gridDim.x) +1;
3.  int start = i*section_size;

 // All threads handle blockDim.x * gridDim.x
   // consecutive elements
4.  for (k = 0; k < section_size; k++) {
5.    if (start+k < size) {
6.      int alphabet_position = buffer[start+k] – ‘a’;
7.      if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo[alphabet_position/4]), 1);
      }
    }
}
```

**FIGURE 9.6**

A CUDA kernel for calculation histogram based on Strategy I.

第3行使用第1行计算的全局线程索引来计算每个线程要处理的部分的起始点。在上面的示例中，线程i要处理的部分的起始点是i*4，因为每个部分由4个元素组成。也就是说，线程0的起始点是0，线程8的起始点是32，以此类推。

从第4行开始的for循环与我们在图9.2中看到的非常相似。这是因为每个线程本质上是在其分配的部分上执行顺序直方图计算。但有两个显著的不同。首先，字母位置的计算由if条件保护。这个测试确保只有索引在buffer内的线程才会访问缓冲区。它是为了防止接收到只有部分元素或完全为空元素的线程进行超边界的内存访问。

最后，图9.2中的递增表达式(histo[alphabet_position/4]++)在图9.6的第6行变成了一个atomicAdd()函数调用。第一个参数是要更新的位置的地址&(histo[alphabet_position/4])。要添加到位置的值1是第二个参数。这确保了不同线程对任何histo[]数组元素的同步更新都被正确序列化。

# 9.3块分区与交叉分区

在方法一中，我们将buffer[]的元素划分为连续的元素或块，并将每个块分配给一个线程。这种分区策略通常称为块分区。将数据划分为连续块是一种直观且概念简单的方法。在并行执行通常只涉及少量线程的CPU上，块分区通常是最佳的执行策略，因为每个线程的顺序访问模式可以很好地利用缓存。由于每个CPU缓存通常只支持少量线程，因此不同线程对缓存的使用几乎没有干扰。缓存线中的数据一旦被线程引入，就可以保留下来以供后续访问使用。

正如我们在第5章中了解到的那样，SM中大量同时活动的线程通常会导致对缓存的过多干扰，以至于不能指望缓存行中的数据仍然可用于下一个线程的所有顺序访问方法一。相反，我们需要确保warp中的线程访问连续的位置以启用内存合并。 这意味着我们需要调整buffer[]的策略。

图9.7显示了我们的文本直方图示例中用于内存合并的理想访问模式。在第一次迭代中，四个线程访问字符0到3(“prog”)，如图11.7(A)所示。通过内存合并，所有的元素只需要访问一次DRAM就可以获取。在第二次迭代中，四个线程在一次合并内存访问中访问字符“ramm”。显然，这是一个玩具例子。实际上，会有更多的线程。线程在每次迭代中处理的字符数量与性能之间存在微妙的关系。为了充分利用缓存和SMs之间的带宽，每个线程应该在每次迭代中处理4个字符。

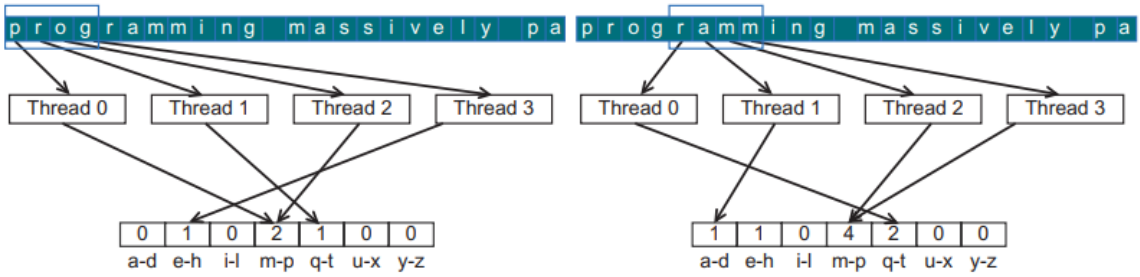


**FIGURE 9.7**

Desirable access pattern to the input buffer for memory coalescing—Strategy II.

既然我们了解了所需的访问模式，我们就可以推导出解决这个问题的分区策略。与块分区策略不同，我们将使用交叉分区策略，其中每个线程将处理由所有线程在一次迭代中处理的元素分隔的元素。在图9.7中，线程0要处理的分区是元素0(“p”)、4(“r”)、8(“i”)、12(“m”)、16(“i”)和20(“y”)。线程1过程元素1 (r)、5(“a”),9(“n”),和13(“a”),17(“v”),21(“_”)。这被称为交错分区的原因应该很清楚：不同线程所处理的分区彼此交错。

图9.8显示了基于方法二的修正后的kernel。它在第2行中通过计算一个跨越值来实现交叉分配，这个跨越值是kernel调用期间启动的线程总数(blockDim.x*gridDim.x)。在while循环的第一次迭代中，每个线程使用其全局线程索引索引输入缓冲区:线程0访问元素0，线程1访问元素1，线程2访问元素2，等等。因此，所有线程共同处理第一批blockDim.x*gridDim个元素。在第二次迭代中，所有线程处理的元素索引都增加了blockDim.x*gridDim。并共同处理blockDim.x*gridDim个的下一批x元素。

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{

1. unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;

// All threads handle blockDim.x * gridDim.x consecutive elements in each iteration
2.  for (unsigned int i = tid;  i < size; i += blockDim.x*gridDim.x ) {
3.      int alphabet_position = buffer[i] – 'a';
4.      if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo[alphabet_position/4]), 1);
    }
}
```

**FIGURE 9.8**

A CUDA kernel for calculating histogram based on Strategy II.

for循环控制每个线程的迭代。当线程的索引超过输入buffer的有效范围(i大于或等于大小)时，该线程已经完成了对其分区的处理，并将退出循环。由于buffer的大小可能不是线程总数的倍数，因此有些线程可能不参与最后一部分的处理。因此，有些线程会比其他线程少执行一次for- loop迭代。

由于合并内存访问，图9.8中的版本可能会比图9.6中的执行速度快好几倍。然而，仍然有很大的改进空间，我们将在本章的其余部分展示。有趣的是，图9.8中的代码实际上更简单，尽管交错分区在概念上比块分区更复杂。这在性能优化中经常是正确的。虽然优化在概念上可能很复杂，但它的实现可能相当简单。

## 9.4原子操作的延迟与通量

在图9.6和9.8的kernel中使用的原子操作通过将任何同一个位置的同步更新序列化来确保更新的正确性。我们都知道，序列化大规模并行程序的任何部分都可能大大增加执行时间，降低程序的执行速度。因此，重要的是这样的序列化操作所占的执行时间要尽可能少。

正如我们在第5章中了解到的，在DRAMs中对数据的访问延迟可能需要数百个时钟周期。在第3章中，我们了解到GPU使用零周期上下文切换来容忍这种延迟。只要我们有许多线程，它们的内存访问延迟可以相互重叠，执行速度就受到内存系统吞吐量的限制。因此，重要的是GPU充分利用DRAM加速、bank和channel来实现非常高的内存访问吞吐量。

在这一点上，读者应该清楚，高内存访问吞吐量的关键是假设许多DRAM访问可以同时进行。不幸的是，当许多原子操作更新相同的内存位置时，这种假设就不成立了。在这种情况下，在前面线程的读-改-写序列完成之前，后面线程的读-改-写序列不能开始。如图9.9所示，对同一内存位置执行原子操作时，在任何单位时间内只有一个操作在进行。每个原子操作的持续时间大约是一次内存读取的延迟时间(原子操作时间的左边部分)加上一次内存写入的延迟时间(原子操作时间的右边部分)。每个读-改-写操作的这些时间段的长度(通常是数百个时钟周期)定义了必须用于服务每个原子操作的最短时间，从而限制了吞吐量或执行原子操作的速度。

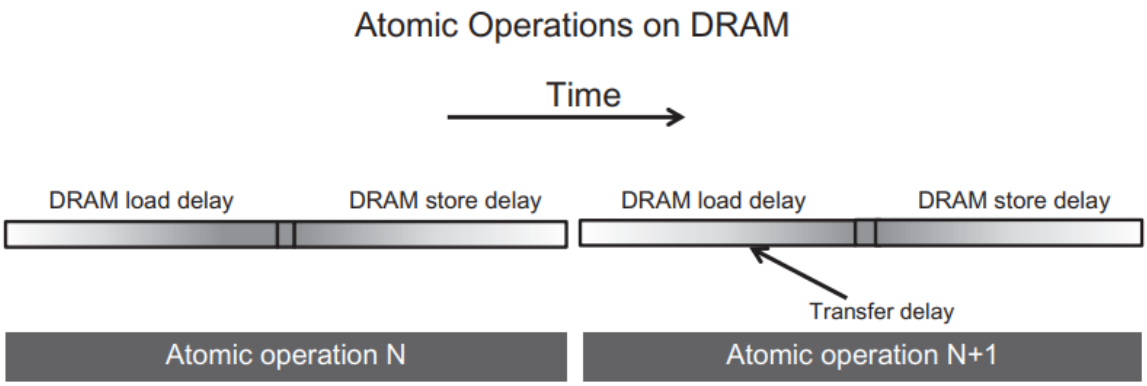


**FIGURE 9.9**

Throughput of atomic operation is determined by the memory access latency.

例如，假设一个内存系统具有64位双数据率DRAM接口、8通道、1GHz时钟频率，典型的访问延迟为200个周期。内存系统的峰值访问吞吐量是8(字节/传输)*2(传输每个时钟每个通道)*1G(时钟每秒)*8(通道)= 128GB/秒。假设每个数据访问为4字节，系统的峰值访问吞吐量为每秒32G数据元素。

然而，当在特定内存位置上执行原子操作时，可以实现的最高吞吐量是每400个周期执行一个原子操作(200个周期用于读，200个周期用于写)。这意味着基于时间的吞吐量为1/400(原子/时钟)*1G(时钟/秒)= 2.5Matomics/秒。这大大低于大多数用户对GPU内存系统的期望。

在实践中，并非所有原子操作都将在单个内存位置上执行。在我们的文本直方图示例中，直方图有7个间隔。如果输入字符均匀地分布在字母表中，原子操作就均匀地分布在histo[]元素中。这将把吞吐量提高到每秒7* 250万= 1750万原子操作。实际上，分布因子往往比直方图中的间隔数7要低得多，因为字符在字母表中往往有分布偏向。例如，在图9.1中，我们看到示例短语中的字符严重偏向于m-p和q-t区间。为了更新这些间隔而产生的大量争用流量很可能会将可实现的吞吐量降低到远远低于每秒1750万次原子操作。

对于图9.6和9.8的kernel，原子操作的低吞吐量会对执行速度产生显著的负面影响。为了简单起见，假设原子操作的吞吐量是每秒1750万个原子操作。我们可以看到图9.8中的内核执行了大约6个算术运算(–, >=, <, /, +, +)，每个原子操作。因此，kernel的最大算术执行吞吐量将是6*17.5M = 105M算术运算每秒。这只是现代GPU上每秒1,000,000m或更多算术运算的峰值吞吐量的一小部分!这种现象激发了若干类别的优化，以提高并行直方图计算的速度，以及使用原子操作的其他类型的计算的速度。

## 9.5缓存中的原子操作

上一节的一个关键观点是，内存访问的长延迟会导致在竞争激烈的内存位置上执行原子操作的低吞吐量。根据这种观点，提高原子操作吞吐量的一个明显方法是减少对竞争激烈的位置的访问延迟。缓存内存是减少内存访问延迟的主要工具。

最近的GPU允许原子操作在最后一级缓存中执行，它被所有的SMs共享。在原子操作期间，如果在最后一级缓存中找到更新的变量，则在缓存中更新它。如果不能在最后一级缓存中找到它，它将触发缓存丢失并被带到他要更新的那个缓存中去。由于原子操作更新的变量往往会被许多线程大量访问，因此一旦从DRAM引入这些变量，它们往往会留在缓存中。由于对最后一级缓存的访问时间是以几十个周期而不是数百个周期为单位的，因此只要允许在上一级缓存中执行原子操作，其吞吐量至少可以提高一个数量级。这点在从Tesla一代到Fermi一代的原子操作吞吐量的巨大改进中是显而易见的，在Fermi一代中，原子操作首先在最后一级缓存(L2)中得到支持。然而，对于许多应用程序来说，改进的吞吐量仍然不够。

## 9.6私有化

通过在共享内存中放置数据，可以显著减少访问内存的延迟。共享内存是每个SM的私有内存，具有非常短的访问延迟(几个周期)。回想一下，减少的延迟直接转化为原子操作的吞吐量的增加。问题在于，由于共享内存的私有性质，一个线程块中的线程更新对其他线程块中的线程不再可见。程序员必须显式地处理在线程块之间缺乏直方图更新可见性的问题。

一种称为私有化的技术通常用于解决并行计算中的输出干扰问题。其思想是将竞争激烈的输出数据结构复制到私有副本中，以便每个线程(或部分线程)可以更新其私有副本。这样做的好处是可以以更少的争用和更低的延迟访问私有副本。这些私有副本可以极大地增加更新数据结构的吞吐量。缺点是在计算完成后，私有副本需要合并到原始数据结构中。我们必须小心地平衡竞争程度与合并成本。因此，在大规模并行系统中，私有化通常是为部分线程而不是单个线程进行的。

在我们的文本直方图示例中，我们可以为每个线程块创建一个私有直方图。在这个方案下，几百个线程将处理存储在短延迟共享内存中的直方图副本，而不是数以万计的线程处理存储在中等延迟二级缓存或长延迟DRAM中的直方图。更少的竞争线程和更短的访问延迟的综合作用可以导致更新吞吐量的数量级增加。

图9.10显示了一个私有化的直方图kernel。第2行分配了一个共享内存数组histo_s[]，它的维度是在kernel启动时设置的。在第3行for循环中，线程区块中的所有线程同时初始化它们的私有直方图副本。第5行中的barrier同步确保所有私有直方图在任何线程开始更新它们之前已经被正确初始化。

```
__global__ void histogram_privatized_kernel(unsigned char* input, unsigned int* bins,
   unsigned int num_elements, unsigned int num_bins) {

1.  unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;

    // Privatized bins
2.  extern __shared__ unsigned int histo_s[];
3.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx +=blockDim.x) {
4.     histo_s[binIdx] = 0u;
    }
5.  __syncthreads();

     // Histogram
6.  For (unsigned int i = tid; i < num_elements; i += blockDim.x*gridDim.x) {
      int alphabet_position = buffer[i] – “a”;
7.    if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo_s[alphabet_position/4]), 1);
    }
8.  __syncthreads();

   // Commit to global memory
9.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {
10.    atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
    }
}
```

**FIGURE 9.10**

A privatized text histogram kernel.

第6-7行for循环与图9.8中的循环相同，只是原子操作是在共享内存histo_s[]上执行的。第8行barrier同步确保线程块中的所有线程在将私有副本合并到原始直方图之前完成它们的更新。

最后，第9-10行for循环同时地将私有直方图值合并到原始版本中。注意，原子添加操作用于更新原始的直方图元素。这是因为多个线程块可以同时更新相同的柱状图元素，并且必须使用原子操作进行正确的序列化。请注意，图9.10中的两个for循环都是为了让kernel能够处理任意数量分区的直方图。

## 9.7聚合

一些数据集在局部区域有大量相同的数据值。例如，在天空的图片中，可以有大量相同值的像素块。如此高浓度的相同值会导致严重的竞争发生，降低并行直方图计算的吞吐量。

对于这样的数据集，一个简单而有效的优化是，如果每个线程更新直方图的相同元素，那么将连续的更新聚合为单个更新。这样的聚合减少了对高度竞争的直方图元素的原子操作数量，从而提高了计算的有效吞吐量。

图9.11显示了聚合的文本直方图kernel。每个线程声明三个额外的寄存器变量curr_index、prev_index和accumulator。累加器跟踪到目前为止已聚合的更新的数量，而prev_index跟踪其更新已聚合的直方图元素的索引。每个线程将prev_index初始化为−1(第6行)，这样就不会有字母输入与之匹配。累加器被初始化为零(第7行)，表示没有聚合任何更新。

```
1.  unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;

    // Privatized bins
2.  extern __shared__ unsigned int histo_s[];
3.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx +=blockDim.x) {
4.      histo_s[binIdx] = 0u;
    }
5.  __syncthreads();

6.  unsigned int prev_index = -1;
7.  unsigned int accumulator = 0;

8.  for(unsigned int i = tid; i < num_elements; i += blockDim.x*gridDim.x) {
9.    int alphabet_position = buffer[i] – “a”;
10.   if (alphabet_position >= 0 && alpha_position < 26) {
11.         unsigned int curr_index = alphabet_position/4;
12.         if (curr_index != prev_index) {
13.           if (accumulator >= 0) atomicAdd(&(histo_s[alphabet_position/4]), accumulator);
14.           accumulator = 1;
15.           prev_index = curr_index;
            }
16.         else {
17.           accumulator++;
            }
        }
    }
18. __syncthreads();

    // Commit to global memory
19.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {
20.      atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
    }
}
```

**FIGURE 9.11**

An aggregated text histogram kernel.

当找到一个字母数据时，线程将要更新的直方图元素的索引(curr_index)与当前正在聚合的直方图元素的索引(prev_index)进行比较。如果索引不同，则对柱状图元素的连续聚合更新已经结束(第12行)。线程使用原子操作将累加器值添加到直方图元素中，该元素的索引由prev_index跟踪。这将有效地清除前一批聚合更新的全部贡献。如果curr_index与prev_index匹配，则线程只需向累加器添加一个(第17行)，将聚合更新的连续扩展1。

需要记住的一件事是，聚合的kernel需要更多的语句和变量。因此，如果竞争率较低，则聚合kernel的执行速度可能低于简单kernel。但是，如果数据分布导致原子操作执行中出现严重的争用，那么聚合将显著提高性能。

## 9.8总结

直方图是分析大数据集的一项重要计算。它还代表了一类重要的并行计算模式，其中每个线程的输出位置都是依赖于数据的，这使得应用owner-computes规则不可行。因此，它是引入原子操作的一种自然工具，原子操作可以确保多个线程对同一内存位置的读-改-写操作的完整性。不幸的是，正如我们在本章中解释的那样，原子操作的吞吐量比简单的内存读或写操作低得多，因为它们的吞吐量大约是内存延迟的两倍。因此，在存在严重竞争的情况下，柱状图计算的计算吞吐量可能低得惊人。私有化作为一种重要的优化技术被引入，它系统地减少争用，并允许使用本地内存(如共享内存)，后者支持低延迟，从而提高吞吐量。事实上，支持区块中的线程之间的快速原子操作是共享内存的基本用例。对于导致严重竞争的数据集，聚合还可以显著提高执行速度。

## 9.9练习

同学们自己看书吧