第八章 并行前缀和模式——并行算法效率简介

下一个并行模式是前缀和，也称为扫描。并行扫描常用于将看似顺序的操作转换为并行操作。这些操作包括资源分配、工作分配和多项式计算。通常，通常被描述为数学递归的计算可以被并行化为并行扫描操作。并行扫描在大规模并行计算中发挥关键作用的原因很简单:应用程序的任何连续部分都可能极大地限制应用程序的总体性能。许多这样的顺序部分可以通过并行扫描转换为并行计算。并行扫描是一种重要的并行模式的另一个原因是顺序扫描算法是线性算法，具有极高的工作效率，这就强调了并行扫描算法工作效率控制的重要性。对于大数据集，稍微增加算法复杂度就会使并行扫描的运行速度慢于顺序扫描。因此，高效的并行扫描算法也是能够在具有广泛计算资源的并行系统上有效运行的一类重要的并行算法。

8.1背景知识

从数学上讲，广义的扫描操作需要一个二元运算符⊕和n个元素的输入数组[x0，x1，…，xn-1]，并返回以下输出数组：

$$[x_0,\ (x_0 \oplus x_1),\ \ldots,\ (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})]$$

解释一下，如果⊕是一个加法运算，那么对输入数组[3 1 7 0 4 16 3]执行包含扫描操作将返回[3 4 11 11 15 16 22 25]。

广义的扫描操作的应用程序可以这样说明:假设我们有一个40英寸的香肠要供应给8个人。每个人订购不同数量的香肠:3,1,7,0,4,1,6,3英寸。0号人想要3英寸的香肠，1号人想要1英寸，以此类推。香肠既可以按顺序切，也可以并行切。顺序法非常简单。我们首先为0号人剪出3英寸的部分;香肠现在有37英寸长。然后我们为1号人切1英寸的部分;香肠变成36英寸长。我们可以继续切更多的部分，直到我们把3英寸的部分给7号人。到那时，我们已经提供了25英寸的香肠，还有15英寸。

通过广义扫描操作，我们可以根据每个人的订单数量计算出所有切割点的位置;即，给定一个加法操作和一个顺序输入数组[3 1 7 0 4 16 3]，包容性扫描操作返回[3 4 11 11 15 16 22 25]。返回数组中的数字是切割位置。有了这个信息，我们可以同时进行所有的8个切割，从而生成每个人排序的切面。从编号0的人开始，第一个切割点是在3英寸的位置，这样第一个部分将是3英寸长。按照第1号人的命令，第二个切割点是在4英寸的位置，这样第二部分将是1英寸长。最后的切割点将在25英寸的位置，这将产生3英寸长的部分，因为之前的切割点在22英寸的点。7号人最终会得到她点的东西。注意，由于所有的切割点都是从扫描操作中知道的，所以所有的切割都可以并行进行。

总之，考虑广义的扫描操作的一种直观方式是，该操作接受一组人员的订单，并识别所有切割点，然后同时提供服务。订单可以是香肠、面包、营地空间，或者计算机中的连续内存块。只要我们能快速计算出所有的切割点，所有的订单都可以并行送达。

狭义的扫描操作与广义的扫描操作类似，只是前者返回如下输出数组:

$$[0,\ x_0,\ (x_0 \oplus x_1),\ \ldots,\ (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})]$$

第一个输出元素为0，而最后一个输出元素只加到了xn-2。

狭义扫描操作的应用程序与广义扫描操作的应用程序非常相似，但提供的信息略有不同。在香肠示例中，狭义扫描将返回[0 3 4 11 11 15 16 22]，是切段的起始点。编号0的部分从0英寸点开始，编号7的部分从22英寸点开始。起始点信息对于内存分配等应用程序非常有用，在这些应用程序中，分配的内存通过指向起始点的指针返回给请求者。

在广义扫描输出和狭义扫描输出之间可以很容易地进行转换。我们只需要移动所有元素并填充一个元素。要将广义转换为狭义，只需将所有元素向右移动，并为第0个元素填充值0。要将狭义元素转换为广义元素，我们只需要将所有元素移到左侧，并用之前最后一个元素和最后一个输入元素加和，然后填充最后一个元素。不管我们关心的是切点还是段的起点，我们都可以根据需求直接生成一个广义或狭义扫描。

在实践中，并行扫描经常被用作并行算法中的一种基本操作，这些并行算法用于基数排序、快速排序、字符串比较、多项式求值、求解递归、树操作、流压缩和直方图等等。

在我们介绍并行扫描算法及其实现之前，我们首先介绍一种工作效率高的顺序广义扫描算法及其实现，假设所涉及的操作是加法。该算法假设输入元素在x数组中，输出元素将写入y数组中。

```
void sequential_scan(float *x, float *y, int Max_i) {
  int accumulator = x[0];
  y[0] = accumulator;
  for (int i = 1; i < Max_i; i++) {
       accumulator += x[i];
       y[i] = accumulator;
  }
}
```

该算法是高效的，只对每个输入或输出元素执行少量的工作。对于一个相当好的编译器，在处理每个输入x元素时只使用一个加法操作、一个内存加载和一个内存存储。这个工作量几乎是我们能做的最小工作量了。正如我们将看到的，当一个顺序算法是如此的“精简和平均”时，开发一个在数据集变大时能够始终击败顺序算法的并行算法是非常具有挑战性的。

8.2一个简单的并行扫描

我们从一个简单的并行广义扫描算法开始，对每个输出元素执行reduction操作。主要目标是通过计算每个输出元素的相关输入元素的reduction树来快速得到每个元素。可以用多种方式设计每个输出元素的简化树。我们将介绍的第一种方法是基于Kogge-Stone算法，该算法最初是在20世纪70年代发明用于设计快速加法器电路。该算法目前正应用于高速计算机算法硬件的设计中。

如图8.1所示，该算法是一种对输入元素的数组XY进行就地扫描的算法。随后，它迭代地将数组的内容演化为输出元素。在算法开始之前，我们假设XY [i]包含输入元素xi。在迭代n结束时，XY[i]将包含在位置处和位置前最多2n个输入元素的总和;即在迭代1结束时，XY[i]将包含xi-1+xi，在迭代2结束时，XY[i]将包含xi−3+xi−2+xi−1+xi，以此类推。

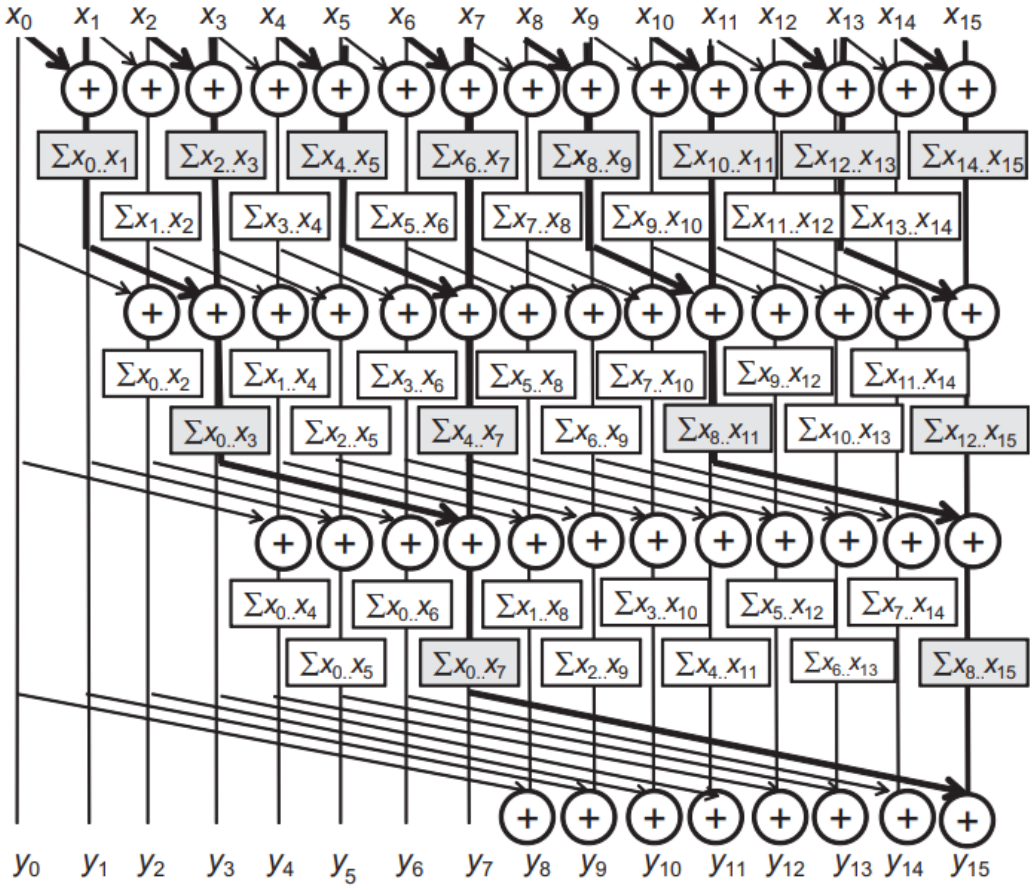

FIGURE 8.1

A parallel inclusive scan algorithm based on Kogge–Stone adder design.

图8.1给出了16个元素的算法步骤。每条竖线表示XY数组的一个元素，XY[0]位于最左边。垂直方向显示迭代的进度，从顶部开始。对于广义扫描，根据定义，y0 = x0；因此，XY[0]包含了它的最终答案。在第一次迭代中，除XY[0]之外的每个位置都将接收其当前内容与其左邻居内容之和，如图8.1中第一行加法运算符所示。XY[i]包含xi-1+xi，如图8.1中第一行加法运算符下的标记框所示。例如，第一次迭代后，XY[3]包含x2+x3，表现为∑x2…x3，XY[1]等于x0+x1，这是该位置的最终答案。因此，在以后的迭代中不应该对XY[1]做进一步的更改。

在第二次迭代中，除了XY[0]和XY[1]之外的每个位置都接收其当前内容和距离两个元素的位置内容的和，如第二行加法运算符下面的标记框所示。XY[i]现在包含xi-3+xi-2+xi-1+xi。例如，第一次迭代后，XY[3]包含x0+x1+x2+x3，表现为∑x0…x3。在第二次迭代之后，XY[2]和XY[3]包含了它们最终的答案，在随后的迭代中不需要更改。

鼓励读者完成余下的迭代。我们现在进行图8.1所示算法的并行实现。我们分配每个线程来演化一个XY元素的内容。我们将编写一个kernel来扫描输入的一个小到可以用一个块来处理的section。section的大小定义为编译时常量SECTION_SIZE。我们假设kernel启动将使用SECTION_SIZE作为块大小，这样线程的数量就等于section元素的数量。每个线程将负责计算一个输出元素。

如果数组只包含section中的元素，则计算所有结果。稍后，我们将对大输入数组的这些分section扫描结果进行最后的调整。我们还假设输入值最初在全局内存数组X中，它的地址作为参数传递给kernel。我们将让区块中的所有线程共同将X数组元素加载到共享内存数组XY中。这种加载是通过让每个线程计算其全局数据索引i = blockIdx.x*blockDim.x + threadIdx.x 来做为输出向量元素所负责的位置。每个线程将该位置的输入元素加载到共享内存中。在kernel的末尾，每个线程将其结果写入指定的输出数组Y中。

```
__global__ void Kogge_Stone_scan_kernel(float *X, float *Y,
  int InputSize) {

  __shared__ float XY[SECTION_SIZE];
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) {
    XY[threadIdx.x] = X[i];
  }
  // the code below performs iterative scan on XY
    ...

  Y[i] = XY[threadIdx.x];
}
```

我们现在关注的是图8.1中每个XY元素for循环的迭代计算的实现：

```
  for (unsigned int stride = 1; stride < blockDim.x; stride
    *= 2) {
    __syncthreads();
    if (threadIdx.x >= stride) XY[threadIdx.x] +=
XY[threadIdx.x-stride];
  }
```

该循环遍历reduction树，查找分配给线程的XY数组位置。我们使用syncthreads()来确保所有线程在任何线程开始下一个迭代之前已经完成了它们在reduce树中添加的上一个迭代。这与在第5章的讨论中对剩余syncthreads()的使用相同。当stride值超过线程的threadIdx.x时值，指定的线程的XY位置可以理解为已经积累了所有需要的输入值。

for循环的执行行为与图8.1中的示例一致。XY小索引位置的动作比大索引位置的动作早结束。当stride值很小时，这种行为会在第一个warp造成一定程度的控制异化。相邻的线程将趋向于执行相同的迭代次数。对于较大大小的块，异化的影响应该是相当温和的。详细的分析留作练习。最终的内核如图8.2所示。

```
__global__ void Kogge-Stone_scan_kernel(float *X, float *Y,
  int InputSize) {

  __shared__ float XY[SECTION_SIZE];

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) {
    XY[threadIdx.x] = X[i];
  }

  // the code below performs iterative scan on XY
  for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();
    if (threadIdx.x >= stride)XY[threadIdx.x] += XY[threadIdx.x-stride];
  }

  Y[i] = XY[threadIdx.x];

}
```

FIGURE 8.2

A Kogge–Stone kernel for inclusive scan.

我们可以很容易地将广义扫描kernel转换为狭义扫描kernel。回想一下，狭义扫描就是将广义扫描所有元素向右移动一个位置，并将0元素填充为值0，如图8.3所示。唯一真正的区别是图片顶部元素的对齐方式。图8.3所有的标签框都被更新以反映新的对齐方式。所有的迭代操作保持不变。

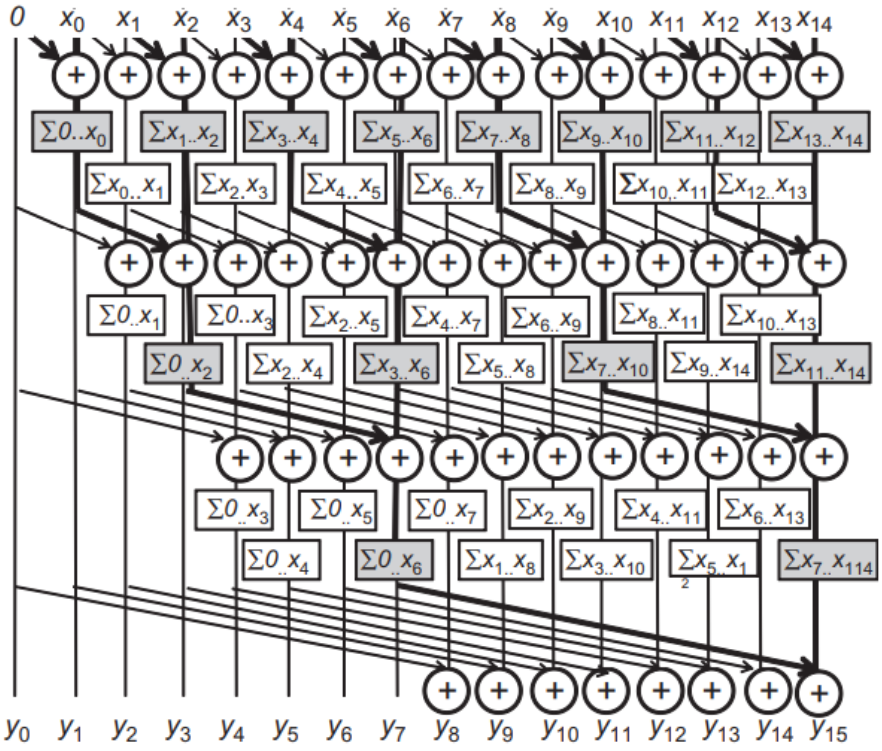

FIGURE 8.3

A parallel exclusive scan algorithm based on Kogge–Stone adder design.

现在我们可以很容易地将图8.2中的kernel转换为狭义扫描内核。我们只需要加载0到XY[0]和X[i−1]到XY[threadIdx.x]，如下代码所示：

```
if (i < InputSize && threadIdx.x != 0) {
  XY[threadIdx.x] = X[i-1];
} else {
  XY[threadIdx.x] = 0;
}
```

请注意，在范围之外的其关联的输入元素XY位置现在也用0填充，这不会造成危害，但会略微简化代码。 我们将剩下的步骤作为练习来完成独占扫描内核。

8.3速度和工作效率

现在，我们在图8.2中分析kernel的速度和工作效率。 所有线程将迭代log2N个步骤，其中N为SECTION_SIZE。在每次迭代中，不活动线程的数量等于stride大小。 因此，算法的完成工作量(for循环的一次迭代，由图8.1中的加法运算表示)计算为

$$\sum(\mathrm{N}-\mathrm{stride}),\ \mathrm{for\ strides}\ 1,2,4,\ldots \mathrm{N}/2(\log_2 \mathrm{N}\ \mathrm{terms})$$

每个想的第一部不同的stride互相独立； 它的总和为N * log2N。第二部分是一个熟悉的几何级数，总和为(N-1)。因此，完成的工作总量为

$$N*\log_2 N - (N-1)$$

回想一下，顺序扫描算法执行的for循环迭代次数为N−1。即使对于中等大小的区域，图8.2中的kernel也要比顺序算法执行更多的工作。在512个元素的情况下，kernel执行的工作大约是顺序代码的8倍。随着N的增大，这个比值会增大。

至于执行速度，顺序代码的for循环执行N次迭代。对于kernel代码，每个线程的for循环最多执行log2N次迭代，这定义了执行kernel所需的最小步骤数。如果执行资源没有限制，kernel代码相对于顺序代码的加速大约为N/log2N。当N = 512时，加速大约是512/9= 56.9。

在一个真实的CUDA GPU设备中，Kogge-Stone的kernel所做的工作量比理论的N*log2N - (N−1)还要多，因为我们使用的是N个线程。虽然许多线程停止参与for循环的执行，但它们仍然消耗执行资源，直到整个线程块完成执行为止。实际上，Kogge-Stone消耗的执行资源更接近于N*log2N。

时间单位的概念将被用作执行时间的指标，以比较不同的扫描算法。顺序扫描大约需要N个时间单位来处理N个输入元素。例如，顺序扫描大约需要1024个时间单位来处理1024个输入元素。对于CUDA设备中的P个执行单元(流处理器streaming processors)，我们可以期望Kogge-Stone kernel执行(N*log2N)/P个时间单位。举例来说，如果我们使用1024个线程和32个执行单元来处理1024个输入元素，那么kernel可能需要(1024*10)/32 = 320个时间单元。在本例中，预期时间为1024/320= 3.2。

Kogge-Stone kernel对顺序代码所做的额外工作在两方面存在问题。首先，使用硬件执行并行kernel的效率要低得多。为了达到收支平衡，并行机器需要的执行单元至少是顺序机器的8倍。如果我们在一个并行机器上执行kernel，其执行资源是顺序机器的四倍，那么执行并行kernel的并行机器的速度只有执行顺序代码的顺序机器的一半。第二，额外的工作消耗了额外的电力。这个额外的需求使得kernel不太适合电力受限的环境，比如移动应用程序。

如果有足够的硬件资源，Kogge-Stone的kernel的优势在于其令人满意的执行速度。 Kogge-Stone kernel通常用于计算元素数量较少的部分(例如32或64)的扫描结果。其执行的控制异化量非常有限。 在新一代的GPU体系结构中，可以使用warp内的混洗指令(shuffle instruction)有效地执行其计算。 我们将在本章后面看到，Kogge-Stone kernel是现代高速并行扫描算法的重要组成部分。

8.4更有效的并行扫描

虽然图8.2中的Kogge-Stone kernel在概念上比较简单，但是在一些实际应用中工作效率比较低。仅对图8.1和8.3进行观察就可以看出，共享几个中间结果可能带来的精简。但是，我们需要有策略地计算要共享的中间结果，然后将它们分发到不同的线程，以便在多个线程之间实现更多的共享。

我们知道，生成一组值的和值的最快并行方式是reduction树。有了足够的执行单元，reduction树就可以以log2N个时间单位生成N个值的和。该树还可以生成一些子和，这些子和可用于计算一些扫描输出值。这一观察形成了Brent–Kung加法器设计的基础，它也可以用于并行扫描算法。

在图8.4中，我们通过四个步骤得到所有16个元素的总和。我们使用最少的操作来生成和。在第一步中，只有XY[i]的奇数元素将被更新为XY[i-1]+XY[i]。第二步，仅更新索引形式为4*n−1的XY元素;这些元素是图8.4中的3,7,11,15。第三步，仅更新索引形式为8*n−1的XY元素;这些元素是7和15。最后，在第四步中，只更新了XY[15]。操作总数为8 + 4 + 2 + 1 = 15。一般来说，对于一个包含N个元素的扫描section，我们在reduction阶段会做(N/2) + (N/4) +…+ 2 + 1 = N−1操作。

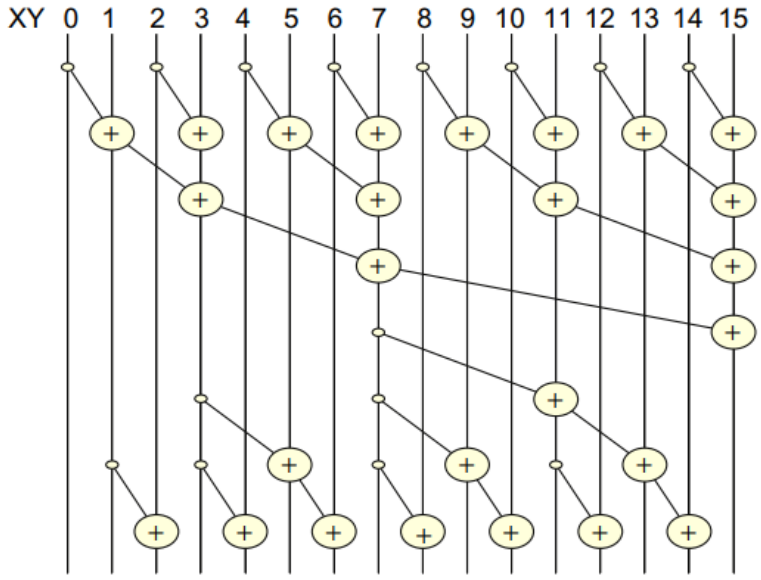

FIGURE 8.4

A parallel inclusive scan algorithm based on the Brent–Kung adder design.

该算法的第二部分是使用一个反向树，以便将section和分配到能够尽快使用这些值的位置，如图8.4的下半部分所示。在还原阶段的最后，我们有很多可用的部分和。图8.5中的第一行显示了在顶部简化树之后XY中的所有部分和。一个重要的发现是XY[0]， XY[7]，和X[15]包含了它们最终的答案。因此，所有剩下的XY元素可以在不超过4个位置处得到它们需要的部分和。

举例来说，XY[14]可以从XY[7]， XY[11]，和XY[13]得到它需要的所有部分和。为了组织我们的后半部分加法运算，我们将首先展示所有需要从4个位置、2个位置和1个位置进行部分和的运算。通过检查，XY[7]包含了右半边许多位置需要的临界值。一个令人满意的方法是将XY[7]加到XY[11]上，得到XY[11]。更重要的是，XY[7]也成为XY[12]， XY[13]，和XY[14]的部分和。没有其他部分和有如此多的用途。因此，在图8.4中的四个位置水平只需要增加一个XY[11] = XY[7] + XY[11]。更新后的部分和显示在图8.5的第二行。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x_0	$x_0.x_1$	x_2	$x_0.x_3$	x_4	$x_4.x_5$	x_6	$x_0.x_7$	x_8	$x_8..x_9$	x_{10}	$x_8.x_{11}$	x_{12}	$x_{12}..x_{13}$	x_{14}	$x_0.x_{15}$
											$x_0..x_{11}$				
					$x_0..x_5$				$x_0..x_9$				$x_0..x_{13}$		

FIGURE 8.5

Partial sums available in each XY element after the reduction tree phase.

我们现在通过使用距离两个位置的部分和来识别所有的加法。XY[2]只需要它在XY[1]中的部分和。XY[4]同样需要它旁边的部分和。第一个XY元素需要部分和距离为2的是XY[5]。一旦我们计算出XY[5] = XY[3] + XY[5]， XY[5]就包含了最终答案。同样的分析表明，XY[6]和XY[8]在XY[5]和XY[7]中用相邻的部分和即可计算。

下一个间隔两个位置加起来是XY[9] = XY[7] + XY[9]，这样XY[9]就完整了。XY[10]可以等待下一轮来追加XY[9]。XY[12]只需要XY[11]，它包含了四个位置相加后的最终答案。最后的相隔两位的位加法是XY[13] = XY[11] + XY[13]。第三行显示了XY[5]， XY[9]，和XY[13]中所有更新的部分和。很明显，现在，每个位置要么是完成的，要么可以添加他们的左邻居来完成。图8.4中的最后一行加法，完成了XY[2]、XY[4]、XY[6]、XY[8]、XY[10]和XY[12]所有不完整位置的内容。

我们可以使用以下循环来实现并行扫描的加法reduction树阶段:

```
for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
  __synchthreads();
  if ((threadIdx.x + 1)%(2*stride) == 0) {
     XY[threadIdx.x] += XY[threadIdx.x - stride];
}
}
```

该循环与图5.2中的简化高度相似。 唯一的区别是，我们希望线程索引的形式为2n -1而不是2n的线程在每次迭代中执行加法运算。 这个目标是在每次迭代中选择要执行加法的线程时，将1加到threadIdx.x的原因。 但是，这种简化方式涉及控制异化问题。 如第5章中所述，一种更好的技术是随着循环的进行使用数量更少的连续线程来执行加法操作：

```
for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
  __syncthreads();
  int index = (threadIdx.x+1) * 2* stride -1;
  if (index < SECTION_SIZE) {
     XY[index] += XY[index - stride];
  }
}
```

通过在for循环的每次迭代中使用更复杂的索引计算，kernel可以减少warp内的控制异化。 图8.4显示了一个块中的16个线程。 在第一次迭代中，步幅等于1。块中的前八个连续线程将满足if条件。 为这些线程计算的索引值为1、3、5、7、9、11、13和15。这些线程将执行图8.4中的第一行加法。 在第二次迭代中，跨度等于2。只有块中的前四个线程将满足if条件。 为这些线程计算的索引值为3、7、11、15。这些线程将执行图8.4中的第二行加法。 由于每次迭代中始终使用连续的线程，因此直到活动线程的数量下降到warp大小以下才出现控制异化问题。

reduction树的实现稍微复杂一些。我们观察到，stride值从SECTION_SIZE/4减小到1。在每次迭代中，我们需要将XY元素的值从步幅值- 1的倍数“推入”到距离它一大步的位置。例如，在图8.4中，步幅值从4减小到1。在图8.4的第一次迭代中，我们的目标是将XY[7]的值加至XY[11]，其中7 = 2*4–1。注意，这个迭代只需要一个线程(线程0)。在第二次迭代中，我们打算将XY[3]、XY[7]和XY[11]的值加到XY[5]、XY[9]和XY[13]。这个过程可以通过以下循环来实现:

```
for (int stride = SECTION_SIZE/4; stride > 0; stride /= 2) {
  __syncthreads();
  int index = (threadIdx.x+1)*stride*2 - 1;
  if(index + stride < SECTION_SIZE) {
    XY[index + stride] += XY[index];
  }
}
```

该index的计算与reduction树阶段相似。Brent-Kung并行扫描的最终kernel代码如图8.6所示。读者应该注意到，对于reduction阶段或distribute阶段，没有必要使用超过SECTION_SIZE/2的线程。因此，我们可以简单地在一个块中启动一个带有SECTION_SIZE/2线程的内核。因为块中最多可以有1024个线程，所以每个扫描部分最多可以有2048个元素。然而，每个线程必须在开始时加载两个X元素，并在结束时存储两个Y元素。

```
__global__ void Brent_Kung_scan_kernel(float *X, float *Y,
  int InputSize) {

  __shared__ float XY[SECTION_SIZE];
  int i = 2*blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) XY[threadIdx.x] = X[i];
  if (i+blockDim.x < InputSize) XY[threadIdx.x+blockDim.x] = X[i+blockDim.x];

  for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
    __syncthreads();
    int index = (threadIdx.x+1) * 2* stride -1;
    if (index < SECTION_SIZE) {
      XY[index] += XY[index - stride];
    }
  }

  for (int stride = SECTION_SIZE/4; stride > 0; stride /= 2) {
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index + stride < SECTION_SIZE) {
      XY[index + stride] += XY[index];
    }
  }

  __syncthreads();
  if (i < InputSize) Y[i] = XY[threadIdx.x];
  if (i+blockDim.x < InputSize) Y[i+blockDim.x] = XY[threadIdx.x+blockDim.x];
}
```

FIGURE 8.6

A Brent–Kung kernel for inclusive scan.

与Kogge-Stone扫描内kernel的情况一样，可以很容易地将Brent-Kung包含的并行扫描kernel调整为独占扫描kernel，只需要对将X元素加载到XY中的语句进行小幅调整。[Harris 2007]提出了一种有趣的本地专有扫描kernel，它重新设计扫描kernel distribution树阶段的另一种方法。

现在，我们将注意力转向对distribution树阶段中的操作数的分析。 运算次数为(2-1) + (4×1) + (16/2×1) 。通常，对于N个输入元素，操作总数为(2×1)+(4×1)+ ... +(N / 4×1)+(N / 2×1)，即N -1-log2(N)。此表达式导致并行扫描中的操作总数为2N-2-log2(N)，包括reduction树(N-1个操作)和逆reduction树阶段(N-1-log2(N)个操作))。现在，操作数与N成正比，而不是N * log2(N)。
在比较中，Brent-Kung算法的优势非常明显。 随着输入部分的增加，Brent-Kung算法执行的操作数永远不会超过顺序算法执行的操作数的两倍。 在能量受限的执行环境中，Brent-Kung算法在并行性和效率之间取得了很好的平衡。

尽管Brent-Kung算法比Kogge-Stone算法具有更高的理论工作效率，但其在CUDA内核实现中的优势更加有限。 回想一下，Brent-Kung算法正在使用N / 2个线程。 主要区别是活动线程数在reduction树上的下降比Kogge-Stone算法快得多。 但是，不活动的线程继续消耗CUDA设备中的执行资源。 因此，Brent–Kung内核消耗的资源数量实际上接近（N / 2）*（2 * log2（N）-1）。这一发现使Brent-Kung算法的工作效率类似于CUDA设备中的Kogge-Stone。在8.4节中，如果我们用32个执行单元处理1024个输入元素，则Brent–Kung内核预计将占用大约512 *（2 * 10-1）/ 32 = 304个时间单位。 加速比为1024/304 = 3.4。

8.5一种更加高效的并行扫描

通过在输入的各个子部分上添加完全独立的扫描阶段，我们可以设计一种比Brent-Kung算法更高的工作效率的并行扫描算法。 在算法开始时，我们将输入部分划分为多个子部分。 子部分的数量与线程块中的线程数相同，每个线程一个。 在第一阶段，每个线程对其子部分执行扫描。 在图8.7中，我们假设一个块包含四个线程。 我们将输入部分分为四个小节。 在第一阶段，线程0将对其部分（2、1、3、1）执行扫描并生成（2、3、6、7）。线程1将对其部分（0，4，1，2）执行扫描，并生成（0，4，5，7），依此类推。

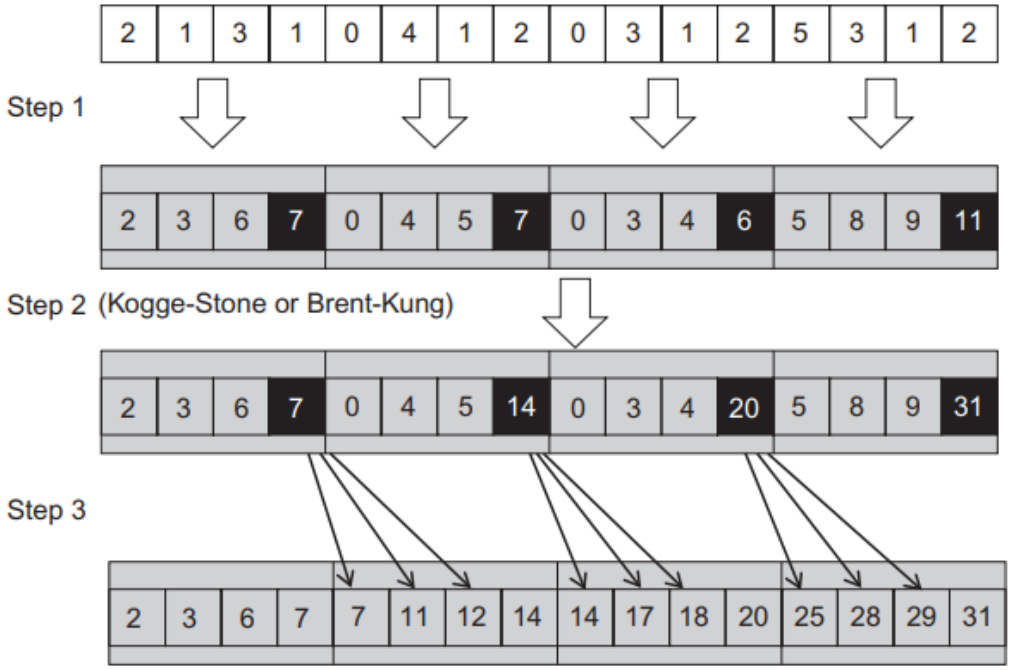

FIGURE 8.7

Three-phase parallel scan for higher work efficiency and speed.

值得注意的是，如果每个线程通过访问全局内存中的输入直接执行扫描，则它们的访问将不会合并。 例如，在第一次迭代中，线程0将访问输入元素0，线程1输入元素4，依此类推。 因此，我们使用第4章``内存和数据局部性"中介绍的Corner turning技术来改善内存合并。 在该阶段的开始，所有线程都会协作将输入迭代地加载到共享内存中。 在每次迭代中，相邻线程加载相邻元素以启用内存合并。 在图8.7中，所有线程必须协作并以合并的方式加载四个元素：线程0加载元素0，线程1加载元素1，依此类推。 所有线程都移动以加载接下来的四个元素：线程0加载元素4，线程1加载元素5，依此类推。

一旦所有输入元素都在共享内存中，线程就从共享内存访问它们自己的子部分，如图8.7所示为第1步。 在第1步结束时，每个节的最后一个元素（第二行中以黑色突出显示）包含此部分中所有输入元素的总和。 部分0的最后一个元素包含值7，该值是该部分中输入元素（2、1、3、1）的总和。

在第二阶段，每个块中的所有线程进行协作，并对由所有部分的最后一个元素组成的逻辑阵列执行扫描操作。 可以使用Kogge-Stone或Brent-Kung算法执行此过程，因为只涉及少量的元素（一个块中的线程数）。 在第3步中，每个线程将其前一个部分的最后一个元素的新值添加到其元素中。 在此阶段中，无需更新每个小节的最后一个元素。 在图8.7中，线程1将值7加到元素（0，4，5）的部分中，便产生（7，11，12）。请注意，该部分的最后一个元素已经是正确的值14，不需要更新。

使用这种三阶段方法，我们可以使用比section中元素数量少得多的线程数。 该section的最大大小不再受块中线程数的限制，而是受到共享内存的大小的限制。 该部分中的所有元素都必须放于共享内存中。 此限制将在分层方法中消除，将在本章其余部分中讨论。

三阶段方法的主要优点是有效利用执行资源。 假设我们对阶段2使用Kogge–Stone算法。对于N个元素的输入列表，如果我们使用T线程，则阶段1的工作量为N-1，阶段2的工作量为T*log2T ，阶段3的工作量为 N–T。如果我们使用P个执行单位，则执行可以预期采用（N-1 + T * log2T + NT）/ P个时间单位。

举例来说，如果我们使用64个线程和32个执行单元来处理1024个元素，则该算法应采用大约（1024-1 + 64 * 6 + 1024-64）/ 32 = 74个时间单位。 该数字导致加速比为1024/74 = 13.8。

8.6任意长度输入的分层并行扫描

对于许多应用程序，扫描操作可以处理数百万甚至数十亿元的元素。到目前为止介绍的三种内核都假定整个输入可以加载到共享内存中。显然，我们不能期望这些大型扫描应用程序的所有输入元素都适合共享内存，这就是为什么我们说这些kernel处理输入的一部分。而且，只使用一个线程块来处理这些大数据集将会失去并行性的机会。幸运的是，分层方法可以扩展我们目前生成的扫描kernel，以处理任意大小的输入。这种方法如图8.8所示。

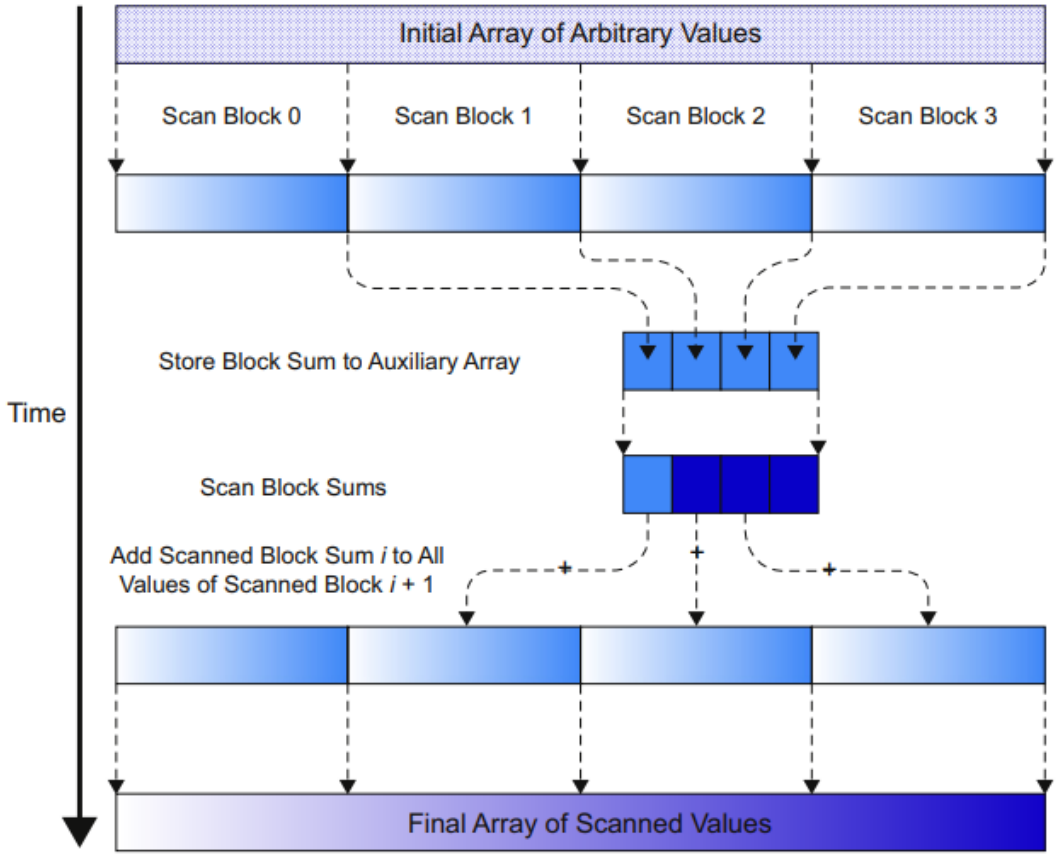

FIGURE 8.8

A hierarchical scan for arbitrary length inputs.

对于大型数据集，我们首先将输入划分为多个部分，以便每个部分都能放入共享内存中，并由单个区块处理。对于当前的CUDA设备，图8.8中的Brent-Kung kernel可以通过在每个区块中使用1024个线程来处理每个部分中最多2048个元素。为了说明，如果输入数据包含2,000,000个元素，我们可以使用ceil(2,000000 /2048.0) = 977个线程块。网格的x维中最多有65,536个线程块，这种方法可以处理输入集中最多134,217,728个元素。如果输入大于这个数字，可以使用额外的层次结构级别来处理真正任意数量的输入元素。但是，在本章中，我们将把讨论限制在两层层次结构中，它最多可以处理134,217,728个元素。

假设我们在一个大型输入数据集上启动第8.2、8.4和8.5节中的三种kernel之一。在网格执行结束时，Y数组将包含单个部分的扫描结果，称为扫描块，如图8.8所示。扫描块中的每个结果值只包含同一扫描块中所有前面元素的累加值。这些扫描块需要组合成最终的结果;也就是说，我们需要编写并启动另一个内核，它将前面扫描块中的所有元素的总和加到扫描块的每个元素上。

图8.9显示了图8.8中分层扫描方法的一个示例。一共有16个输入元素被分成4个扫描块。我们可以使用Kogge-Stone kernel、Brent-Kung kernel或三阶段kernel来处理单个扫描块。内核将这四个扫描块作为独立的输入数据集。扫描kernel结束操作后，每个Y元素都在其扫描块中包含扫描结果。为了说明，扫描块1有输入0、4、1、2。扫描内核为这个部分生成扫描结果:0、4、5、7。这些结果不包括来自扫描块0中的任何元素的贡献。为了产生这个扫描块的最终结果，扫描块0中所有元素的总和，即2+1+3+1 = 7，扫描块1的每个结果元素都要加之。

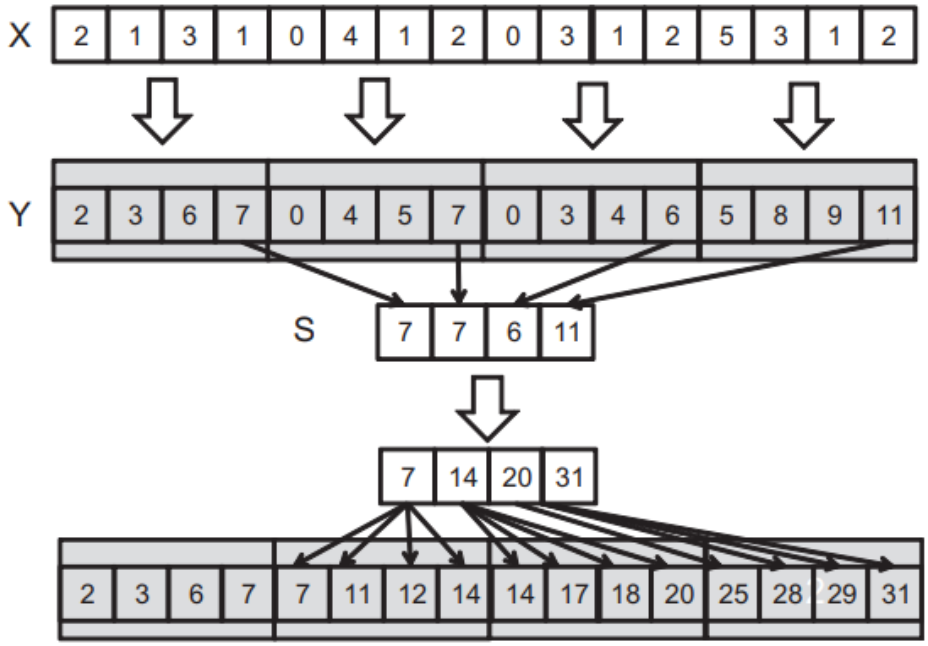

FIGURE 8.9

An example of hierarchical scan.

另一个例子如下：扫描块2中的输入为0、3、1、2。kernel为该扫描块生成扫描结果：0、3、4、6。要生成此扫描块的最终结果， 扫描块0和扫描块1中所有元素的总和，2 + 1 + 3 + 1 + 0 + 4 + 1 + 2 = 14，应添加到扫描块2的每个结果元素中。

每个扫描块的最后一个输出元素是扫描块所有输入元素的总和。 这些值在图8.9中为7、7、6和11。图8.8中分层扫描算法的第二步将来自每个扫描块的最后结果元素收集到一个数组中，并对这些输出元素执行扫描。 此步骤也显示在图8.9中，其中所有的最后扫描输出元素都收集到一个新数组S中。

可以通过更改在扫描kernel末尾代码来执行此过程，以便每个块的最后一个线程通过使用其blockIdx.x作为索引将其结果写入S数组。 然后在S上执行扫描操作以产生输出值7、14、20、31。这些第二级扫描输出值中的每一个都是从开始位置X [0]到每个扫描块结束的累加和。 S [0] = 7中的输出值是从X [0]到扫描块0末尾的累加和，即X [3]。S [1] = 14中的输出值是从X [0]到扫描块1末尾的累加和，即X [7]。

因此，S数组中的输出值会在原始扫描问题的“战略”位置产生扫描结果。 在图8.9中，S [0]， S [1]， S [2]和S [3]中的输出值是位置X [3]， X [7]， X [11]和X [15]的最终扫描结果。 这些结果可用于使每个扫描块中的部分结果达到最终值。 这将我们带到图8.8中的分层扫描算法的最后一步。，把第二级扫描输出值添加到其相应扫描块的值中。

为了说明，在图8.9中，将S[0]的值(值7)加到线程块1的Y[0]、Y[1]、Y[2]、Y[3]上，从而完成这些位置的结果。这些位置的最终结果是7、11、12、14，因为S[0]包含原始输入X[0]到X[3]的值之和。这些最终结果是14、17、18和20。S[1](14)的值会加到Y[8]， Y[9]， Y[10]， Y[11]，从而完成这些位置的结果。S[2](20)的值会加上Y[12] Y[13] Y[14] Y[15]。最后，S[3]的值是原始输入的所有元素的和，这也是Y[15]的最终结果。

熟悉计算机算法的读者应该认识到，分层扫描算法与现代处理器硬件加法器中的超前进位算法十分相似。考虑到目前为止我们研究的两种并行扫描算法都是基于创新的硬件加法器设计，这种相似性应该是可以预料的。

我们可以实现三中kernel的分层扫描。第一个kernel与三层kernel大致相同。(我们也可以简单地使用Kogge - Stone kernel 或 Brent-Kung kernel 。) 我们需要添加一个参数S，它的维度是InputSize/SECTION_SIZE。在kernel的末尾，我们添加了一个条件语句。块中的最后一个线程将扫描块中最后一个XY元素的输出值写入到S的blockIdx.x位置:

```
__syncthreads();
if (threadIdx.x == blockDim.x-1) {
  S[blockIdx.x] = XY[SECTION_SIZE - 1];
}
```

第二个kernel只是三个并行扫描kernel中的一个，它将S作为输入，写入S作为输出。

第三个kernel将S和Y数组作为输入，并将其输出写回Y中。假设我们用每个块中SECTION_SIZE个线程启动kernel，每个线程将一个S元素(由blockIdx.x-1选择)添加到一个Y元素:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
Y[i] += S[blockIdx.x-1];
```

块中的线程将前一个扫描块的和添加到它们的扫描块的元素中。作为练习，完成每个kernel和host代码的详细信息留给读者。

8.7单次扫描以提高内存访问效率

在第8.6节中提到的分层扫描中，部分扫描的结果在启动全局扫描kernel之前被存储到全局存储器中，然后由第三个kernel从全局存储器中重新加载。 这些额外的内存存储和加载的延迟与后续kernel中的计算不重叠。 延迟还可能显着影响分层扫描算法的速度。 为了避免这种负面影响，已经提出了多种技术[DGS 2008] [YLZ 2013] [MG 2016]。本章讨论基于流的扫描算法。 鼓励读者阅读参考文献，以了解其他技术。

在CUDA C编程的背景下,基于流的扫描算法(不要与CUDA流混淆,后者将在编程异构计算集群章节中提到)指的是一种分层扫描算法，其中部分总和数据传递向一个方向通过邻近的线程之间的全局内存块。基于流的扫描基于关键观察，即可以以多米诺骨牌方式执行全局扫描步骤（图8.8的中间部分）。例如，在图8.9中，扫描块0可以通过其部分和值7扫描块1，完成其任务。扫描块1从扫描块0接收到局部和值7,与局部和值7相加得到14，将局部和值14传递给扫描块2，完成最后一步。

在基于流的扫描中，可以编写一个kernel来执行图8.8所示的分级扫描算法的所有三个步骤。线程块i首先对其扫描块执行扫描，使用8.2-8.5节中的三种并行算法之一。然后，该块等待它的左邻居块i−1传递和值。一旦接收到block i−1的和，block就生成并将其和值传递给它的右邻居block i+1。然后，该块继续将从块i−1接收到的和值相加，以完成扫描块的所有输出值。

在kernel的第一阶段，所有块都可以并行执行。这些块将在数据流阶段被序列化。然而，一旦每个块从它的前任接收到总和值，它就可以与从它们的前任接收到总和值的所有其他块并行地执行它的最后阶段。只要能快速地通过块传递和值，块之间就有足够的并行性。

为了使这种基于流的扫描起作用，[YLZ 2013]提出了相邻(块)同步。相邻同步是一种自定义同步，允许相邻线程块同步和/或交换数据。在扫描中，数据从扫描块i−1传递到扫描块i，类似于生产-消费链。 在生产者端（Scan Block i-1），在将部分和存储到存储器后，将标志设置为特定值，而在消费者端（Scan Block i），检查标志以确定是否为 加载传递的部分和之前的特定值。 如前所述，加载的值被添加到本地和，然后被传递到下一个块(扫描块i+1)。相邻同步可以使用原子操作来实现。下面的代码段演示了如何使用原子操作来实现相邻同步。

```
__shared__ float previous_sum;
if (threadIdx.x == 0){
  // Wait for previous flag
  while (atomicAdd(&flags[bid], 0) == 0){;}
  // Read previous partial sum
  previous_sum = scan_value[bid];
  // Propagate partial sum
  scan_value[bid + 1] = previous_sum + local_sum;
  // Memory fence
  __threadfence();
  // Set flag
  atomicAdd(&flags[bid + 1], 1);
}
__syncthreads();
```

此代码段仅由每个块中的一个引导线程执行（例如，索引为0的线程）。 其余线程将在最后一行的__syncthreads（）中等待。 在块竞标中，引导线程重复检查全局存储数组flags [bid]，直到将其设置为止。 然后，它通过访问全局存储器阵列scan_value [bid]从其前任加载部分和，并将该值存储到其本地寄存器变量previous_sum中。 它用局部局部和局部和求和，并将结果存储到全局存储阵列scan_value [bid + 1]中。 内存隔离函数__threadfence（）确保在使用atomicAdd（）在设置标志之后，将部分总和完全存储到内存中。 必须将数组scan_value声明为volatile，以防止编译器优化，重新排序或寄存器分配对scan_value元素的访问。

对flags数组的原子操作以及对scan_value数组的访问可能会导致全局内存通信。 但是，这些操作主要在最近的GPU架构的二级缓存中执行（更多详细信息，请参见：并行模式：并行直方图计算）。 全局存储器的任何存储和加载都可能与其他块的阶段1和阶段3计算活动重叠。 同时，在执行第8.5节中的三阶段扫描算法时，全局存储器中S数组元素的存储和加载均位于单独的kernel中，并且不能与阶段1和阶段3重叠。

基于流的算法有一个微妙的问题。 在GPU中，线程块可能并不总是根据其blockIdx值进行线性调度。 可以在扫描块i + 1之后计划并执行扫描块i。在这种情况下，调度程序安排的执行顺序可能会与相邻同步代码所假定的顺序相抵触，并导致性能损失甚至死锁。 例如，调度器可以在调度扫描块i-1之前调度扫描块i至扫描块i + N。如果扫描块i到扫描块i + N占用了所有流式多处理器，则扫描块i-1将无法开始执行，直到其中至少一个完成执行。 但是，它们所有人都在等待来自扫描块i-1的总和值。 这种情况导致系统陷入僵局。

为了解决这个问题，已经提出了多种技术[YLZ 2013] [GSO 2012]。在这里，我们仅讨论一种特定的方法，即动态块索引分配。 其余的留给读者参考。 动态块索引分配基本上将线程块索引的使用与内置的blockIdx.x解耦。 在扫描中，扫描块i不再与blockIdx.x的值绑定。 而是在调度线程块之后使用以下代码来计算它：

```
__shared__ int sbid;
if (threadIdx.x == 0)
  sbid = atomicAdd(DCounter, 1);
__syncthreads();
const int bid = sbid;
```

引导线程以原子方式递增由DCounter指向的全局计数器变量。 全局计数器存储已调度的下一个块的动态块索引。 然后，领导线程将获取的动态块索引值存储在共享内存变量sbid中，以便在__syncthreads（）之后，该块的所有线程都可以访问它。 此过程可确保线性安排所有扫描块，并防止潜在的死锁。

8.8总结

在本章中，我们将扫描作为一种重要的并行计算模式进行了研究。 扫描可以将资源并行分配给需求不一致的各方。 该过程将看似顺序的递归计算转换为并行计算，这有助于减少各种应用程序中的顺序瓶颈。 我们展示了一个简单的顺序扫描算法仅对N个元素的输入执行N个加法运算。

我们首先介绍了一种并行的Kogge–Stone扫描算法，该算法快速且概念上简单但工作效率低。 随着数据集大小的增加，使用简单的顺序算法实现并行算法达到收支平衡所需的执行单元数也会增加。 对于1024个元素的输入，并行算法执行的加法运算比顺序算法多九倍。 该算法还需要至少九倍的执行资源，才能使顺序算法达到收支平衡。 因此，Kogge–Stone扫描算法通常用于中等大小的扫描块中。

然后，我们提出了一种并行的Brent-Kung扫描算法，该算法在概念上比Kogge-Stone算法更复杂。 使用归约树阶段和分布树阶段，该算法仅执行2 * N-3加法，而与输入数据集的大小无关。 随着操作数随输入集大小的增加而线性增加，因此工作高效的算法通常被称为数据可缩放算法。 不幸的是，由于CUDA设备中线程的性质，Brent-Kung内核的资源消耗最终与Kogge-Stone内核的资源消耗非常相似。 事实证明，采用转弯转弯和势垒同步的三相扫描算法可有效解决工作效率问题。

我们还提出了一种分层方法来扩展并行扫描算法，以便管理任意大小的输入集。 不幸的是，分层扫描算法的一种简单的三层实现会导致冗余的全局内存访问，其访问延迟不会与计算重叠。 我们还提到可以使用基于流的分层扫描算法来实现单遍，单kernel实现并提高分层扫描算法的全局内存访问效率。 但是，此算法需要使用原子操作，线程存储围栏和阻碍同步函数精心设计的相邻块同步。 另外，需要特别注意防止使用动态块索引分配的死锁。