Chapter 8

Parallel Patterns: Prefix Sum

An introduction to work efficiency in parallel algorithms

With special contributions by Li-Wen Chang and Juan Gómez-Luna

Keywords: scan, prefix sum, reduction, work efficiency, linear recurrence, Kogge-Stone, Brent-Kung, hierarchical algorithms, adjacent synchronization, stream-based scan

CHAPTER OUTLINE

Our next parallel pattern is prefix sum, which is also commonly known as scan. Parallel scan is frequently used to convert seemingly sequential operations, such as resource allocation, work assignment, and polynomial evaluation into parallel operations. In general, if a computation is naturally described as a mathematical recursion, it can likely be parallelized as a parallel scan operation. Parallel scan plays a key role in massively parallel computing for a simple reason: any sequential section of an application can drastically limit the overall performance of the application. Many such sequential sections can be converted into parallel computation with parallel scan. Another reason why parallel scan is an important parallel pattern is that sequential scan algorithms are linear algorithms and are extremely work-efficient, which makes it also very important to control the work efficiency of parallel scan algorithms. As we will show, a slight increase in algorithm complexity can make parallel scan run slower than sequential scan for large data sets. Therefore, work-efficient parallel

scan also represents an important class of parallel algorithms that can run effectively on parallel systems with a wide range of available computing resources.

8.1. Background

Mathematically, an *inclusive scan* operation takes a binary associative operator $\oplus$, and an input array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the output array

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

For example, If $\oplus$ is addition, then an inclusive scan operation on the input array [3 1 7 0 4 1 6 3], would return [3 4 11 11 15 16 22, 25].

We can illustrate the applications for inclusive scan operations using an example of cutting sausage for a group of people. Assume that we have a 40-inch sausage to be served to 8 people. Each person has ordered a different amount in terms of inches: 3, 1, 7, 0, 4, 1, 6, 3. That is, person number 0 wants 3 inches of sausage, person number 1 wants 1 inch, and so on. We can cut the sausage either sequentially or in parallel. The sequential way is very straightforward. We first cut a 3-inch section for person number 0. The sausage is now 37 inches long. We then cut a 1-inch section for person number 1. The sausage becomes 36 inches long. We can continue to cut more sections until we serve the 3-inch section to person number 7. At that point, we have served a total of 25 inches of sausage, with 15 inches remaining.

With an inclusive scan operation, we can calculate the locations of all the cutting points based on the amount each person orders. That is, given an addition operation and an order input array [3 1 7 0 4 1 6 3], the inclusive scan operation returns [3 4 11 11 15 16 22 25]. The numbers in the return array are cutting locations. With this information, one can simultaneously make all the eight cuts that will generate the sections that each person ordered. The first cut point is at the 3-inch location so the first section will be 3 inches, as ordered by person number 0. The second cut point is at the 4-inch location, therefore the second section will be 1-inch long, as ordered by person number 1. The final cut point will be at the 25-inch location, which will produce a 3-inch long section since the previous cut point is at 22-inch point. This gives person number 7 what she ordered. Note that since all the cut points are known from the scan operation, all cuts can be done in parallel.

In summary, an intuitive way of thinking about inclusive scan is that the operation takes an order from a group of people and identifies all the cut points that allow the orders to be served all at once. The order could be for sausage, bread, camp ground space, or a contiguous chunk of memory in a computer. As long as we can quickly calculate all the cut points, all orders can be served in parallel.

An exclusive scan operation is similar to an inclusive operation with the exception that it returns the output array

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

That is, the first output element is 0 while the last output element only reflects the contribution of up to x_{n-2}.

The applications of an exclusive scan operation are pretty much the same as those for inclusive scan. The inclusive scan provides slightly different information. In the sausage example, exclusive scan would return [0 3 4 11 11 15 16 22], which are the beginning points of the cut sections. For example, the section for person number 0 starts at the 0-inch point. For another example, the section for person number 7 starts at the 22-inch point. The beginning point information is important in applications such as memory allocation, where the allocated memory is returned to the requester via a pointer to its beginning point.

Note that it is fairly easy to convert between the inclusive scan output and the exclusive scan output. One simply needs to shift all elements and fill in an element. When converting from inclusive to exclusive, one can simply shift all elements to the right and fill in value 0 for the 0^{th} element. When converting from exclusive to inclusive, we need to shift all elements to the left and fill in the last element with the previous last element plus the last input element. It is just a matter of convenience that we can directly generate an inclusive or exclusive scan, whether we care about the cut points or the beginning points for the sections.

In practice, parallel scan is often used as a primitive operation in parallel algorithms that perform radix sort, quick sort, string comparison, polynomial evaluation, solving recurrences, tree operations, stream compaction and histograms.

Before we present parallel scan algorithms and their implementations, we would like to first show a work-efficient sequential inclusive scan algorithm and its implementation. We will assume that the operation involved is addition. The algorithm assumes that the input elements are in the x array and the output elements are to be written into the y array.

```
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;
  }
}
```

The algorithm is work-efficient, only a small amount of work is done for each input or output element. With a reasonably good compiler, only one addition, one memory load, and one memory store are used in processing each input x element. This is pretty much the minimal that we will ever be able to do. As we will see, when the sequential algorithm of a

computation is so "lean and mean," it is extremely challenging to develop a parallel algorithm that will consistently beat the sequential algorithm when the data set size becomes large.

8.2. A Simple Parallel Scan

We start with a simple parallel inclusive scan algorithm by doing a reduction operation for each output element. The main idea is to create each element quickly by calculating a reduction tree of the relevant input elements for each output element. There are multiple ways to design the reduction tree for each output element. We will first present one that is based on the Kogge-Stone algorithm that was originally invented for designing fast adder circuits in the 1970s [KS1973]. This algorithm is still being used in the design of high-speed computer arithmetic hardware today.

The algorithm, shown is Figure 8.1, is an in-place scan algorithm that operates on an array XY that originally contains input elements. It then iteratively evolves the contents of the array into output elements. Before the algorithm begins, we assume of XY[i] contains input element x_i. At the end of iteration n, XY[i] will contain the sum of the up to 2^n input elements at and before the location. That is, at the end of iteration 1, XY[i] will contain $x_{i-1}+x_i$ and at the end of iteration 2, XY[i] will contain $x_{i-3}+x_{i-2}+x_{i-1}+x_i$, and so on.

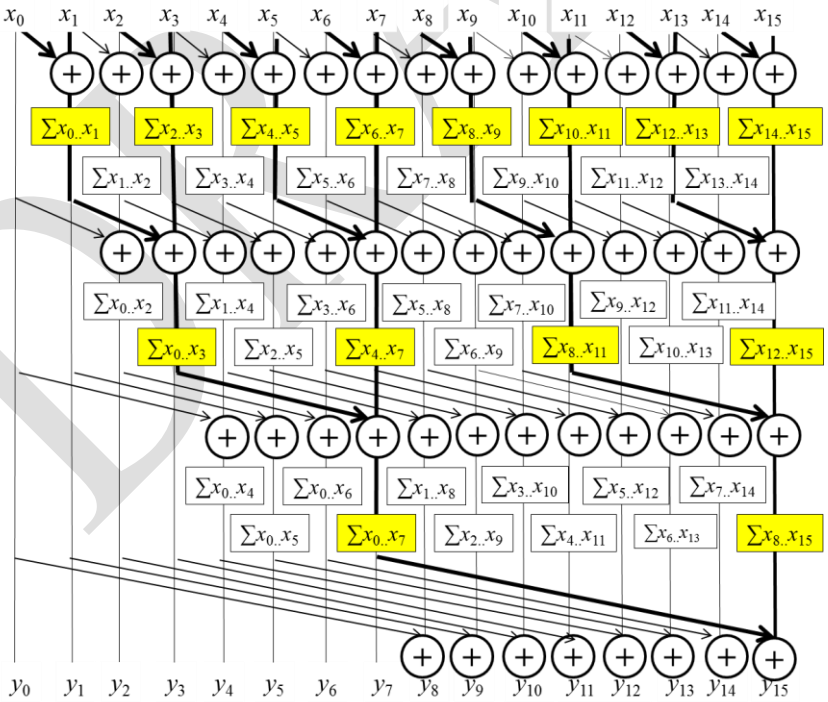

Figure 8.1: *A parallel inclusive scan algorithm based on Kogge-Stone adder design.*

Figure 8.1 illustrates the algorithm with a 16-element input example. Each vertical line represents an element of the XY array, with XY[0] in the leftmost position. The vertical direction shows the progress of iterations, starting from the top of the figure. For inclusive scan, by definition, y_0 is x_0 so XY[0] contains its final answer. In the first iteration, each position other than XY[0] receives the sum of its current content and that of its left neighbor. This is illustrated by the first row of addition operators in Figure 8.1. As a result, XY[i] contains $x_{i-1}+x_i$. This is reflected in the labeling boxes under the first row of addition operators in Figure 8.1. For example, after the first iteration, XY[3] contains x_2+x_3, shown as $\sum x_2..x_3$. Note that after the first iteration, XY[1] is equal to x_0+x_1, which is the final answer for this position. So, there should be no further changes to XY[1] in subsequent iterations.

In the second iteration, each position other than XY[0] and XY[1] receives the sum of its current content and that of the position that is two elements away. This is illustrated in the labeling boxes below the second row of addition operators. As a result, XY[i] now contains $x_{i-3}+x_{i-2}+x_{i-1}+x_i$. For example, after the first iteration, XY[3] contains $x_0+x_1+x_2+x_3$, shown as $\sum x_0..x_3$. Note that after the second iteration, XY[2] and XY[3] contain their final answers and will not need to be changed in subsequent iterations.

The reader is encouraged to work through the rest of the iterations. We now work on the parallel implementation of the algorithm illustrated in Figure 8.1. We assign each thread to evolve the contents of one XY element. We will write a kernel that performs scan on **one section** of the input that is small enough for a block to handle. The size of a section is defined as a compile-time constant SECTION_SIZE. We assume that the kernel launch will use SECTION_SIZE as the block size so there will be the same number of threads and section elements. Each thread will be responsible for calculating one output element.

All results will be calculated as if the array only has the elements in the section. **Later, we will make final adjustments to these sectional scan results for large input arrays.** We also assume that input values were originally in a global memory array X, whose address is passed into the kernel as an argument. We will have all the threads in the block to collaboratively load the X array elements into a shared memory array XY. This is done by having each thread to calculate its global data index `i = blockIdx.x*blockDim.x + threadIdx.x` for the output vector element position it is responsible for. Each thread loads the input element at that position into the shared memory at the beginning of the kernel. At the end of the kernel, each thread will write its result into the assigned output array Y.

```
__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];

  }
```

We now focus on the implementation of the iterative calculations for each XY element in Figure 8.1 as a for-loop:

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

The loop iterates through the reduction tree for the XY array position that is assigned to a thread. Note that we use a barrier synchronization to make sure that all threads have finished their previous iteration of additions in the reduction tree before any of them starts the next iteration. This is the same use of __syncthreads() as in the reduction discussion in Chapter 5. When the stride value becomes greater than a thread's threadIdx.x value, it means that the thread's assigned XY position has already accumulated all the required input values.

The execution behavior of the for-loop is consistent with the example shown in Figure 8.1. The actions of the smaller positions of XY end earlier than the larger positions (see the if-statement condition). This will cause some level of control divergence in the first warp when stride values are small. Note that adjacent threads will tend to execute the same number of iterations. The effect of divergence should be quite modest for large block sizes since divergence will only arise in the first warp. The detailed analysis is left as an exercise. The final kernel is shown in Figure 8.2.

```
    __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

We can easily convert an inclusive scan kernel to an exclusive scan kernel. Recall that an exclusive scan is equivalent to an inclusive scan with all elements shifted to the right by one

position and element 0 filled with value 0. This is illustrated in Figure 8.3. Note that the only real difference is the alignment of elements on top of the picture. All labeling boxes are updated to reflect the new alignment. All iterative operations remain the same.

We can now easily convert the kernel in Figure 8.2 into an exclusive scan kernel. The only modification we need to do is to load 0 into XY[0] and X[i-1] into XY[threadIdx.x], as shown in the code below:

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

Note that the XY positions whose associated input elements are outside the range are now also filled with 0. This causes no harm and yet it simplifies the code slightly. We leave the work to finish the exclusive scan kernel as an exercise.

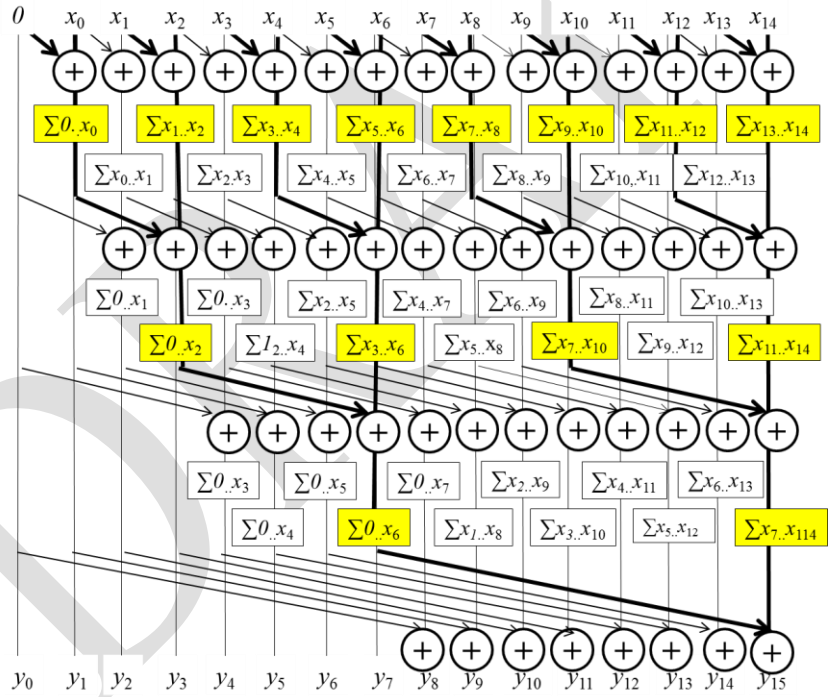

Figure 8.3 A parallel exclusive scan algorithm based on Kogge-Stone adder design.

8.3. Speed and Work Efficiency

We now analyze the speed and work efficiency of the kernel in Figure 8.2. All threads will iterate up to $\log_2 N$ steps, where N is the SECTION_SIZE. In each iteration, the number of inactive threads is equal to the stride size. Therefore, we can calculate the amount of work

done (one iteration of the for-loop, represented by the add operation in Figure 8.1) for the algorithm as

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

The first part of each term is independent of stride, its summation adds up to $N*\log_2 N$. The second part is a familiar geometric series and sums up to (N-1). So the total amount of work done is

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

Recall that the number of for-loop iterations executed for a sequential scan algorithm is N-1. Note that even for modest sized sections, the kernel in Figure 8.2 does much more work than the sequential algorithm. In the case of 512 elements, the kernel does approximately 8 times more work than the sequential code. The ratio will increase as N becomes larger.

As for execution speed, the for-loop of the sequential code executes N iterations. As for the kernel code, the for-loop of each thread executes up to $\log_2 N$ iterations, which defines the minimal number of steps needed for executing the kernel. With unlimited execution resources, the speedup of the kernel code over the sequential code would be approximately $N/\log_2 N$. For N=512, the speedup would be about 512/9 = 56.9.

In a real CUDA GPU device, the amount of work done by the Kogge-Stone kernel is more than the theoretical $N*\log_2 N - (N-1)$. This is because we are using N threads. While many of the threads stop participating in the execution of the for-loop, they still consume execution resources until the entire thread block completes execution. Realistically, the amount of execution resources consumed by the Kogge-Stone is closer to $N*\log_2 N$.

We will use the concept of time units as an approximate indicator of execution time for comparing between scan algorithms. The sequential scan should take approximately N time units to process N input elements. For example, the sequential scan should take approximately 1024 time units to process 1024 input elements. With P execution units (streaming processors) in the CUDA device, we can expect the Kogge-Stone kernel to execute for $(N*\log_2 N)/P$ time units. For example, if we use 1024 threads and 32 execution units to process 1024 input elements, the kernel will likely take (1024*10)/32 = 320 time units. In this case, we expect to achieve a speedup of 1024/320 = 3.2.

The additional work done by the Kogge-Stone kernel over the sequential code is problematic in two ways. First, the use of hardware for executing the parallel kernel is much less efficient. We see that one needs to have at least 8 times more execution units in a parallel machine than the sequential machine just to break even. For example, if we execute the kernel on a parallel machine with four times the execution resources as a sequential machine, the parallel machine executing the parallel kernel can end up with only half the speed of the sequential machine executing the sequential code. Second, all the extra work consume additional

energy. This makes the kernel less appropriate for power-constrained environments such as mobile applications.

The strength of the Kogge-Stone kernel is that it can achieve very good execution speed when there is enough hardware resource. It is typically used to calculate the scan result for a section with modest number of elements, such as 32 or 64. As we have seen, its execution has very limited amount of control divergence. In newer GPU architecture generations, its computation can be efficiently performed with shuffle instructions within warps. We will see later in this chapter that it is an important component of the modern high-speed parallel scan algorithms.

8.4. A More Work-Efficient Parallel Scan

While the Kogge-Stone kernel in Figure 8.2 is conceptually simple, its work efficiency is quite low for some practical applications. Just by inspecting Figures 8.1 and 8.3, we can see that there are potential opportunities for sharing some intermediate results to streamline the operations performed. However, in order to allow more sharing across multiple threads, we need to strategically calculate the intermediate results to be shared and then quickly distribute them to different threads.

As we know, the fastest parallel way to produce sum values for a set of values is reduction tree. With sufficient execution units, a reduction tree can generate the sum for N values in $\log_2 N$ time units. Furthermore, the tree can also generate a number of sub-sums that can be used in the calculation of some of the scan output values. This observation forms the basis of the Brent-Kung adder design [BK1979], which can also be used in a parallel scan algorithm.

In Figure 8.4, we produce the sum of all 16 elements in 4 steps. We use the minimal number of operations needed to generate the sum. During the first step, only the odd element of XY[i] will be updated to XY[i-1]+XY[i]. During the second step, only the XY elements whose indices are of the form of 4*n-1, which are 3, 7, 11, 15 in Figure 8.4, will be updated. During the third step, only the XY elements whose indices are of the form 8*n-1, which are 7 and 15, will be updated. Finally, during the fourth step, only XY[15] is updated. The total number of operations performed is 8+4+2+1 = 15. In general, for a scan section of N elements, we would do (N/2)+(N/4)+…+2+1= N-1 operations for this reduction phase.

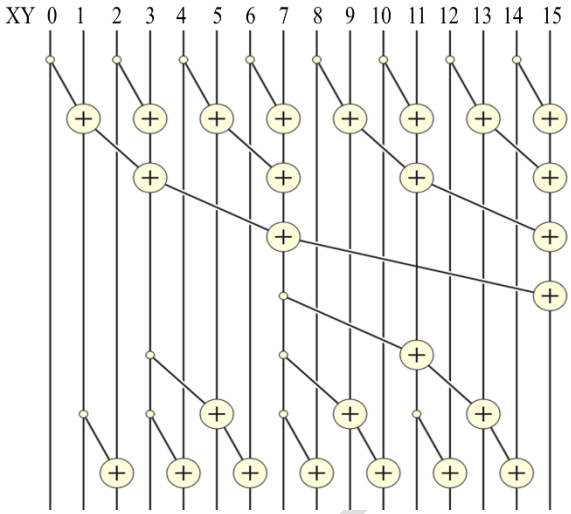

Figure 8.4: *A parallel inclusive scan algorithm based on Brent-Kung adder design.*

The second part of the algorithm is to use a reverse tree to distribute the partial sums to the positions that can use these values as quickly as possible. This is illustrated in the bottom half of Figure 8.4. At the end of the reduction phase, we have quite a few usable partial sums. For our example, the first row of Figure 8.5 shows all the partial sums in XY right after the top reduction tree. An important observation is that XY[0], XY[7] and X[15] contain their final answers. Therefore, all remaining XY elements can obtain the partial sums they need from no farther than 4 positions away.

For example, XY[14] can obtain all the partial sums it needs from XY[7], XY[11], and XY[13]. To organize our second half of the addition operations, we will first show all the operations that need partial sums from 4 positions away, then 2 positions away, then 1 position way. By inspection, XY[7] contains a critical value needed by many positions in the right half. A good way is to add XY[7] to XY[11], which brings XY[11] to the final answer. More importantly, XY[7] also becomes a good partial sum for XY[12], XY[13], and XY[14]. No other partial sums have so many uses. Therefore, there is only one addition XY[11] = XY[7]+XY[11] that needs to occur in the 4-position level in Figure 8.4. We show the updated partial sum in the second row of Figure 8.5.

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*

We now identify all additions using partial sums that are 2 positions away. We see that XY[2] only needs the partial sum that is next to it in XY[1]. Same with XY[4]; it needs the partial sum next to it to be complete. The first XY element that can need a partial sum two positions away is XY[5]. Once we calculate XY[5] = XY[3]+XY[5], XY[5] contains the final answer. Same analysis show that XY[6] and XY[8] can become complete with the partial sums next to them in XY[5] and XY[7].

The next 2-position addition is XY[9] = XY[7]+XY[9], which makes XY[9] complete. XY[10] can wait for the next round to catch XY[9]. XY[12] only needs the XY[11], which contains its final answer after the 4-position addition. The final 2-position addition is XY[13] = XY[11]+XY[13]. The third row shows all the updated partial sums in XY[5], XY[9], and XY[13]. It is clear that now every position is either complete or can be completed when added by their left neighbor. This leads to the final row of additions in Figure 8.4, which completes the contents for all the incomplete positions XY[2], XY[4], XY[6], XY[8], XY[10], and XY[12].

We could implement the reduction tree phase of the parallel scan using the following loop:

```
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];
  }
}
```

Note that this loop is very similar to the reduction in Figure 5.2. The only difference is that we want the threads that have thread index in the form of 2^n-1, rather than 2^n to perform addition in each iteration. This is why we added 1 to the threadIdx.x when we select the threads for performing addition in each iteration. However, this style of reduction is known to have control divergence problems. As we have seen in Chapter 5, a better way is to use a decreasing number of contiguous threads to perform the additions as the loop advances:

```
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];
  }
}
```

By using a more complex index calculation in each iteration of the for-loop, the execution of the kernel now has much fewer control divergence within warps. In our example in Figure 8.4, there are 16 threads in the block. In the first iteration, stride is equal to 1. The first 8 consecutive threads in the block will satisfy the if-condition. The index values calculated for these threads will be 1, 3, 5, 7, 9, 11, 13, 15. These threads will perform the first row of additions in Figure 8.4. In the second iteration, stride is equal to 2. Only the first 4 threads

in the block will satisfy the if-condition. The index values calculated for these threads will be 3, 7, 11, 15. These threads will perform the second row of additions in Figure 8.4. Note that since each iteration will always be using consecutive threads in each iteration, the control divergence problem does not arise until the number of active threads drops below the warp size.

The distribution tree is a little more complex to implement. We make an observation that the stride value decreases from SECTION_SIZE/4 to 1. In each iteration, we need to "push" the value of XY element from a position that is a multiple of stride value minus 1 to a position that is stride away. For example, in Figure 8.4, the stride value decreases from 4 down to 1. In the first iteration in Figure 8.4, we would like to push the value of XY[7] to XY[11], where 7 is 2*4-1. Note that only thread 0 will be used for this iteration. In the second iteration, we would like to push the values of XY[3], XY[7], and XY[11] to XY[5], XY[9], and XY[13]. This can be implemented with the following loop:

```
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];
  }
}
```

The calculation of index is similar to that in the reduction tree phase. The final kernel code for a Brent-Kung parallel scan is shown in Figure 8.6. The reader should notice that we never need to have more than SECTION_SIZE/2 threads for either the reduction phase or the distribution phase. So, we could simply launch a kernel with SECTION_SIZE/2 threads in a block. Since we can have up to 1024 threads in a block, each scan section can have up to 2048 elements. However, we will need to have each thread to load two X elements at the beginning and store two Y elements at the end.

As was in the case of the Kogge-Stone scan kernel, one can easily adapt the Brent-Kung inclusive parallel scan kernel into an exclusive scan kernel with a minor adjustment to the statement that loads X elements into XY. Interested readers should also read [Harris2007] for an interesting natively exclusive scan kernel that is based on a different way of designing the distribution tree phase of the scan kernel.

```
__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*

We now turn our attention to the analysis of the number of operations in the distribution tree stage. The number of operations is (16/8)-1+(16/4)-1+(16/2)-1. In general, for N input elements, the total number of operations would be (2-1)+(4-1)+…+(N/4-1)+(N/2-1) which is N-1-$\log_2$(N). The total number of operations in the parallel scan, including both the reduction tree (N-1 operations) and the inverse reduction tree phases (N-1-$\log_2$(N) operations), is 2*N-2-$\log_2$(N). Note that the upper bound of the number of operation is now proportional to N, rather than N*$\log_2$(N).

The advantage of the Brent-Kung algorithm is quite clear in the comparison. As the input section becomes bigger, the Brent-Kung algorithm never performs more than 2 times the number of operations performed by the sequential algorithm. In an energy-constrained execution environment, the Brent-Kung algorithm strikes a good balance between parallelism and efficiency.

While the Brent-Kung algorithm has a much higher level of theoretical work-efficiency than Kogge-Stone, its advantage in a CUDA kernel implementation is more limited. Recall that the Brent-Kung algorithm is using N/2 threads. The major difference is that the number of active threads drops much faster through the reduction tree than the Kogge-Stone algorithm. However, the inactive threads still consume execution resources in a CUDA device. As a result, the amount of resources consumed by Brent-Kung kernel is actually closer to (N/2)*(2*$\log_2$(N) -1). This makes the work-efficiency of Brent-Kung about the same of that of Kogge-Stone in a CUDA device. Using the example in Section 8.4, if we process 1024 input elements with 32 execution units, the Brent-Kung kernel is expected to take approximately 512*(2*10-1)/32 = 304 time units. This results in a speedup of 1024/304 = 3.4.

8.5. An Even More Work-Efficient Parallel Scan

We can design a parallel scan algorithm that achieves even better work efficiency than Brent-Kung by adding a phase of fully independent scans on sub-sections of the input. At the

beginning of the algorithm, we partition the section of input into sub-sections. The number of sub-sections is the same as the number of threads in a thread block, one for each thread. During the first phase, we have each thread to perform a scan on its sub-section. For example, in Figure 8.7, we assume that there are four threads in a block. We partition the input section into four sub-sections. During the first phase, thread 0 will perform scan on its section (2, 1, 3, 1) and generate (2, 3, 6, 7). Thread 1 will perform scan on its section (0, 4, 1, 2) and generate (0, 4, 5, 7), etc.

Note that if each thread directly performs scan by accessing the input from global memory, their accesses would not be coalesced. For example, during the first iteration, thread 0 would be accessing input element 0, thread 1 input element 4, etc. Therefore, we use the **corner turning** technique presented in Chapter 4, to improve memory coalescing. At the beginning of the phase, all threads collaborate to load the input into the shared memory in an iterative manner. In each iteration, adjacent threads load adjacent elements to enable memory coalescing. For example, in Figure 8.7, we will have all threads to collaborate and load four elements in a coalesced manner: thread 0 to load element 0, thread 1 element 1, etc. All threads then move to load the next four elements: thread 0 to load element 4, thread 1 element 5, etc.

Once all input elements are in the shared memory, the threads access their own sub-section from the shared memory. This is shown as Step 1 in Figure 8.7. Note that at the end of Step 1, the last element of each section (highlighted as black in the second row) contains the sum of all input elements in the section. For example the last element of section 0 contains value 7, the sum of the input elements (2, 1, 3, 1) in the section.

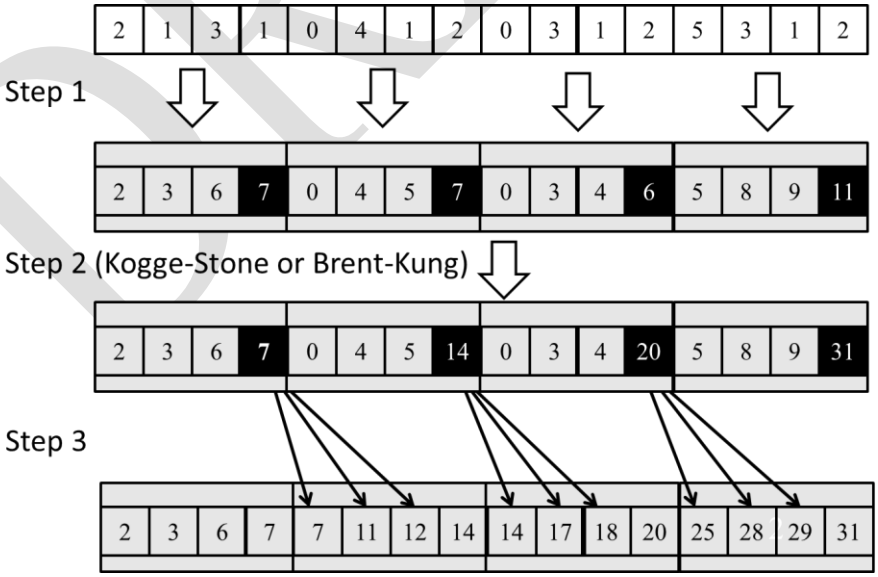

Figure 8.7 A two-phase parallel scan for higher work efficiency and speed

During the second phase, all threads in each block collaborate and perform a scan operation on a logical array that consists of last elements of all sections. This can be done with a Kogge-Stone or Brent-Kung algorithm since there are only a modest number (number of threads in a block) of elements. In Step 3, each thread adds the new value of the last element of its predecessor's section to its elements. The last elements of each sub-section do not need to be updated during this phase. For example, in Figure 8.7, thread 1 adds the value 7 to elements (0, 4, 5) in its section to produce (7, 11, 12). Note that the last element of the section is already the correct value 14 and does not need to be updated.

Note that with this three-phase approach, we can use a much smaller number of threads than the number of elements in a section. The maximal size of the section is no longer limited by the number of threads one can have in the block but rather the size of the shared memory; all elements of the section need to fit into the shared memory. This limitation will be removed in the hierarchical methods to be discussed in the remainder of this chapter.

The major advantage of the three-phase approach is the efficiency of its use of execution resources. Assume that we use Kogge-Stone for phase 2. For an input list of N elements, if we use T threads, the amount of work done by each phase is N-1 for phase 1, $T*\log_2 T$ for phase 2, and N-T for phase 3. If we use P execution units, we can expect that the execution will take $(N-1 + T*\log_2 T + N-T) / P$ time units.

For example, if we use 64 threads and 32 execution units to process 1024 elements, the algorithm should take approximately (1024-1 + 64*6 + 1024-64) / 32 = 74 time units. This results in a speedup of 1024/74 = 13.8.

8.6. Hierarchical Parallel Scan for Arbitrary-Length Inputs

For many applications, the number of elements to be processed by a scan operation can be in the millions or even billions. The three kernels that we presented so far assume that the entire input can be loaded in the shared memory. Obviously, we cannot expect that all input elements of these large scan applications can fit into the shared memory. This is the reason we say that they process a section of the input. Furthermore, it would be a loss of parallelism opportunity if we used only one thread block to process these large data sets. Fortunately, there is a hierarchical approach to extending the scan kernels that we have generated so far to handle inputs of arbitrary size. The approach is illustrated in Figure 8.8.

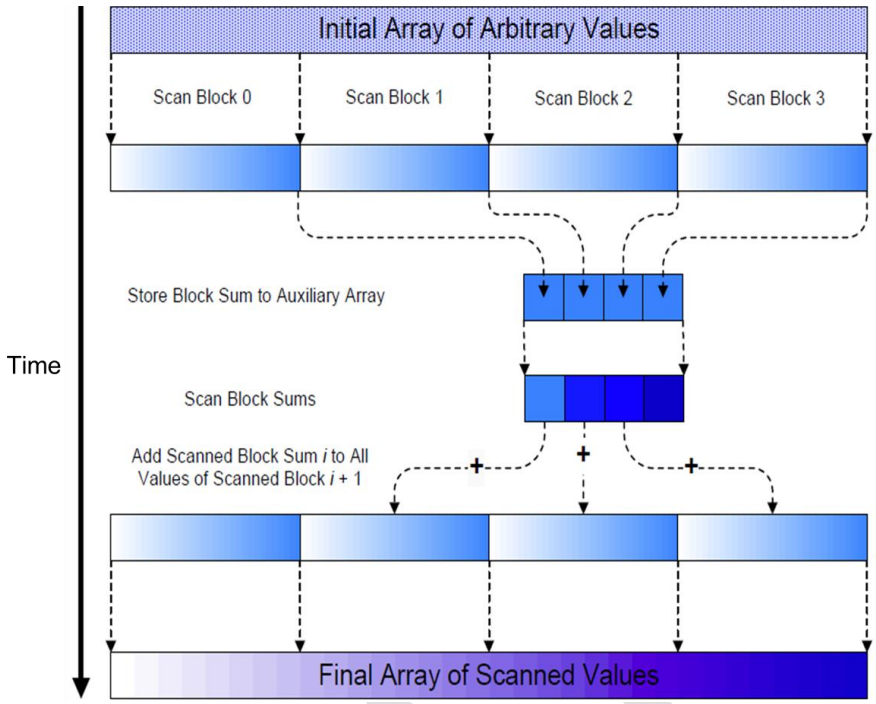

Figure 8.8 *A hierarchical scan for arbitrary length inputs*

For a large data set, we first partition the input into sections so that each of them can fit into the shared memory and processed by a single block. For the current generation of CUDA devices, the Brent-Kung kernel in Figure 8.8 can process up to 2048 elements in each section using 1024 threads in each block. For example, if the input data consist of 2,000,000 elements, we can use ceil(2,000,000/2048.0) = 977 thread blocks. With up to 65,536 thread blocks in the x-dimension of a grid, this approach can process up to 134,217,728 elements in the input set. If the input is even bigger than this, we can use additional levels of hierarchy to handle truly arbitrary number of input elements. However, for this chapter, we will restrict our discussion to a two-level hierarchy that can process up to 134,217,728 elements.

Assume that we launch one of the three kernels in Sections 8.2, 8.4 and 8.5 on a large input data set. At the end of the grid execution, the `Y` array will contain the scan results for individual sections, called *scan blocks* in Figure 8.8. Each result value in a scan block only contains the accumulated values of all preceding elements in the same scan block. These scan blocks need to be combined into the final result. That is, we need to write and launch another kernel that adds the sum of all elements in preceding scan blocks to each element of a scan block.

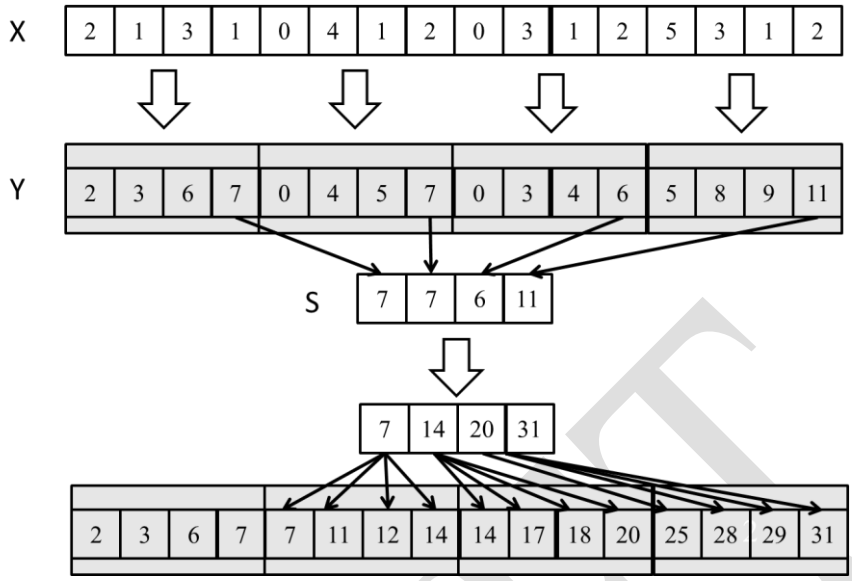

Figure 8.9: *An example of hierarchical scan.*

Figure 8.9 shows a small example of the hierarchical scan approach of Figure 8.8. In this example, there are 16 input elements that are divided into 4 scan blocks. We can use the Kogge-Stone kernel, the Brent-Kung kernel, or the three-phase kernel to process the individual scan blocks. The kernel treats the four scan blocks as independent input data sets. After the scan kernel terminates, each `Y` element contains the scan result within its scan block. For example, scan block 1 has inputs 0, 4, 1, 2. The scan kernel produces the scan result for this section, 0, 4, 5, 7. Note that these results do not contain the contributions from any of the elements in scan block 0. In order to produce the final result for this scan block, the sum of all elements in scan block 0, 2+1+3+1=7, should be added to every result element of scan block 1.

For another example, the inputs in scan block 2 are 0, 3, 1, 2. The kernel produces the scan result for this scan block, 0, 3, 4, 6. In order to produce the final results for this scan block, the sum of all elements in both scan block 0 and scan block 1, 2+1+3+1+0+4+1+2=14, should be added to every result element of scan block 2.

It is important to note that the last output element of each scan block gives the sum of all input elements of the scan block. These values are 7, 7, 6, and 11 in Figure 8.9. This brings us to the second step of the hierarchical scan algorithm in Figure 8.8, which gathers the last result elements from each scan block into an array and performs a scan on these output elements. This step is also illustrated in Figure 8.9, where the last scan output elements of all collected into a new array `S`.

This can be done by changing the code at the end of the scan kernel so that the last thread of each block writes its result into an `S` array using its `blockIdx.x` as index. A scan operation is

then performed on S to produce output values 7, 14, 20, 31. Note that each of these second-level scan output values are accumulated sum from the beginning location X[0] to the end of each scan block. That is, Output value in S[0]=7 is the accumulated sum from X[0] to the end of scan block 0, which is X[3]. Output value in S[1]=14 is the accumulated sum from X[0] to the end of scan block 1, which is X[7]. [1]

Therefore, the output values in the S array give the scan results at "strategic" locations of the original scan problem. That is, in Figure 8.9, the output values in S[0], S[1], S[2], and S[3] give the final scan results for the original problem at positions X[3], X[7], X[11], and X[15]. These results can be used to bring the partial results in each scan block to their final values. This brings us to the last step of the hierarchical scan algorithm in Figure 8.8. The second-level scan output values are added to the values of their corresponding scan blocks.

For example, in Figure 8.9, the value of S[0] (value 7) will be added to Y[0], Y[1], Y[2], Y[3] of thread block 1, which completes the results in these positions. The final results in these positions are 7, 11, 12, 14. This is because S[0] contains the sum of the values of the original input X[0] through X[3]. These final results are 14, 17, 18, and 20. The value of S[1] (14) will be added to Y[8], Y[9], Y[10], Y[11], which completes the results in these positions. The value of S[2] (20) will be added to Y[12], Y[13], Y[14], Y[15]. Finally, the value of S[3] is the sum of all elements of the original input, which is also the final result in Y[15].

Readers who are familiar with computer arithmetic algorithms should recognize that the hierarchical scan algorithm is quite similar to the carry look-ahead in hardware adders of modern processors. This should be no surprise considering that the two parallel scan algorithms that we have studied so far are based on innovative hardware adder designs.

We can implement the hierarchical scan with three kernels. The first kernel is largely the same as the three-phase kernel. (We could just as easily use the Kogge-Stone kernel or the Brent-Kung kernel.) We need to add one more parameter S, which has the dimension of InputSize/SECTION_SIZE. At the end of the kernel, we add a conditional statement for the last thread in the block to write the output value of the last XY element in the scan block to the blockIdx.x position of S:

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

The second kernel is simply the one of the three parallel scan kernels, which takes S as input and writes S as output.

[1] While the second step of Figure 8.9 is logically the same as the second step of Figure 8.7. The main difference is that Figure 8.9 involves threads from different thread blocks. As a result, the last element of each section needs to be collected into a global memory array so that they can be visible across thread blocks.

The third kernel takes the S array and Y array as inputs and writes its output back into Y. Assuming that we launch the kernel with SECTION_SIZE threads in each block, each thread adds one of the S elements (selected by the blockIdx.x-1) to one Y element:

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

That is, the threads in a block add the sum of the previous scan block to the elements of their scan block. We leave it as an exercise for the reader to complete the details of each kernel and complete the host code.

8.7. Single-Pass Scan for Memory Access Efficiency

In the hierarchical scan mentioned in Section 8.6, the partial scanned results are stored into global memory before launching the global scan kernel, and then re-loaded back from the global memory by the third kernel. The latencies of these extra memory stores and loads are not overlapped with the computation in the subsequent kernels and can significantly impact the speed of the hierarchical scan algorithms. In order to avoid such negative impact, multiple techniques [DGS2008][YLZ2013][MG2016] have been proposed. In this chapter, a stream-based scan algorithm is discussed. The reader is encouraged to read the references to understand the other techniques.

In the context of CUDA C programming, a stream-based scan algorithm (not to be confused with CUDA Streams to be introduced in Chapter 18) refers to a hierarchical scan algorithm where partial sum data is passed in one direction through the global memory between neighboring thread blocks. Stream-based scan builds on a key observation that the global scan step (middle part of Figure 8.8) can be performed in a domino fashion. For example, in Figure 8.9, Scan Block 0 can pass its partial sum value 7 to Scan Block 1, and then complete its job. Scan Block 1 receives the partial sum value 7 from Scan Block 0, sums up with its local partial sum value 7 to get 14, passes its partial sum value 14 to Scan Block 2, and then completes its final step.

In a stream-based scan, one can write a single kernel to perform all the three steps of the hierarchical scan algorithm in Figure 8.8. Thread block i first performs a scan on its scan block, using one of the three parallel algorithms we presented in Sections 8.2 through 8.5. It then waits for its left neighbor block i-1 to pass the sum value. Once it receives the sum from block i-1, it generates and passes its sum value to its right neighbor block i+1. It then moves on to add the sum value received from block i-1 to finish all the output values of the scan block.

During the first phase of the kernel, all block can execute in parallel. They will be serialized during the data streaming phase. However, as soon as each block receives the sum value from its predecessor, it can perform its final phase in parallel with all other blocks that have

received the sum values from their predecessors. As long as the sum values can be passed through the blocks quickly, there can be ample of parallelism among blocks.

In order to make this stream-based scan work adjacent (block) synchronization has been proposed in [YLZ2013]. Adjacent synchronization is a customized synchronization to allow the adjacent thread blocks to synchronize and/or exchange data. Particularly, in scan, the data are passed from Scan Block i-1 to Scan Block i, like a produce-consumer chain. On the producer side (Scan Block i-1), the flag is set to a particular value after the partial sum is stored to memory, while on the consumer side (Scan Block i), the flag is checked to see if it is that particular value before the passed partial sum is loaded. As mentioned, the loaded value is further added with the local sum and then is passed to the next block (Scan Block i+1). Adjacent synchronization can be implemented using atomic operations. The following code segment illustrates the use of atomic operations to implement adjacent synchronization.

```
__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();
```

This code section is only executed by one leader thread in each block (e.g., thread with index 0). The rest of threads will wait in __syncthreads() in the last line. In block bid, the leader thread checks flags[bid], a global memory array, repeatedly, until it is set. Then, it loads the partial sum from its predecessor by accessing the global memory array scan_value[bid] and stores the value into its local register variable previous_sum. It sums up with its local partial sum local sum, and stores the result into the global memory array scan_value[bid+1]. The memory fence function __threadfence() is required to ensure that the partial sum is completely stored to memory before the flag is set with atomicAdd(). The array scan_value must be declared as volatile, in order to prevent the compiler from optimizing, reordering, or register allocating the accesses to the scan_value elements.

Although it may appear that the atomic operations on the flags array and the accesses to the scan_value array could incur global memory traffic, these operations are mostly performed in the second-level caches of recent GPU architectures (more details in Chapter 9). Any stores and loads to the global memory will likely be overlapped with the phase 1 and phase 3 computational activities of other blocks. On the other hand, when executing the three-kernel scan algorithm in Section 8.5, the stores to and loads from the S array elements in the

global memory are in a separate kernel and cannot be overlapped with the phase 1 and phase 3.

There is one subtle issue with stream-based algorithms. In GPUs, thread blocks may not *always* be scheduled linearly according to their blockIdx values, which means Scan Block i may be scheduled and performed after Scan Block i+1. In this situation, the execution order arranged by the scheduler might contradict the execution assumed by the adjacent synchronization code, and cause performance loss or even a dead lock. For example, the scheduler may schedule Scan Block i through Scan Block i+N before it schedules Scan Block i-1. If Scan Block i through Scan Block i+N occupy all the streaming multiprocessors, Scan Block i-1 would not be able to start execution until at least one of them finishes execution. However, all of them are waiting for the sum value from Scan Block i-1. This causes the system to deadlock.

To resolve this issue, multiple techniques [YLZ2013][GSO2012] have been proposed. Here we only discuss one particular method, dynamic block index assignment, and leave the rest as reference for readers. Dynamic block index assignment basically decouples the usage of thread block index from the built-in blockIdx.x. In scan, the particular i of the Scan Block i is not tied to the value of blockIdx.x anymore. Instead, it is calculated using the following code after the thread block is scheduled:

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

The leader thread increments atomically a global counter variable pointed by DCounter. The global counter stores the dynamic block index of the next block that is scheduled. The leader thread then stores the acquired dynamic block index value in a shared memory variable sbid, so that it is accessible by all threads of the block after __syncthreads(). This guarantees all Scan Blocks are scheduled linearly, and prevents a potential deadlock.

8.8. Summary

In this chapter, we studied scan as an important parallel computation pattern. Scan is used to enable parallel allocation of resources to parties whose needs are not uniform. It converts seemingly sequential recursive computation into parallel computation, which helps to reduce sequential bottlenecks in many applications. We show that a simple sequential scan algorithm performs only N additions for an input of N elements.

We first introduced a parallel Kogge-Stone scan algorithm that is fast, conceptually simple but not work-efficient. As the data set size increases, the number of execution units needed for a parallel algorithm to break even with the simple sequential algorithm also increases. For an input of 1024 elements, the parallel algorithm performs over 9 times more additions

than the sequential algorithm and requires at least 9 times more execution resources to break even with the sequential algorithm. This is why Kogge-Stone scan algorithms are typically used within modest-sized scan blocks.

We then presented a parallel Brent-Kung scan algorithm that is conceptually more complicated. Using a reduction tree phase and a distribution tree phase, the algorithm performs only 2*N-3 additions no matter how large the input data sets are. Such a work-efficient algorithm whose number of operations grows linearly with the size of the input set is often also referred to as data-scalable algorithms. Unfortunately, due to the nature of threads in a CUDA device, the resource consumption of a Brent-Kung kernel ends up very similar to that of a Kogge-Stone kernel. A three-phase scan algorithm that employees corner turning and barrier synchronization proves to be effective in addressing the work-efficiency problem.

We also presented a hierarchical approach to extending the parallel scan algorithms to handle the input sets of arbitrary sizes. Unfortunately, a straightforward, three-kernel implementation of the hierarchical scan algorithm incurs redundant global memory accesses whose latencies are not overlapped with computation. We show that one can use a stream-based hierarchical scan algorithm to enable a single-pass, single kernel implementation and improve the global memory access efficiency of the hierarchical scan algorithm. This, however, requires a carefully designed adjacent block synchronization using atomic operations, thread memory fence, and barrier synchronization. Special care also has to be given to prevent deadlocks using dynamic block index assignment.

References

[HSO2007] Mark Harris, Shubhabrata Sengupta, John D Owens, *Parallel Prefix Sum with CUDA*, GPU Gems 3, 2007. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf

[DGS2008] Yuri Dotsenko, Naga K. Govindaraju, , Peter-Pike Sloan, Charles Boyd, and John Manferdelli, Fast Scan Algorithms on Graphics Processors. In Proceedings of the 22nd annual International Conference on Supercomputing (pp. 205-213), 2008.

[YLZ2013] Shengen Yan, Guoping Long, and Yunquan Zhang, StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchronization, PPoPP. In ACM SIGPLAN Notices (Vol. 48, No. 8, pp. 229-238), 2013.

[GSO2012] Kshitij Gupta, Jeff A. Stuart and John D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In Innovative Parallel Computing (InPar), 2012 (pp. 1-14). IEEE.

[KS1973] Kogge, P. & Stone, H. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". IEEE Transactions on Computers, 1973, C-22, 783-791

[BK1979] Brent, R.P. and Kung, H.T., “A regular layout for parallel adders,” Technical Report, Computer Science Department, Carnegie-Mellon University, 1979

[MG2016] Merrill, D. and Garland, M. Single-pass Parallel Prefix Scan with Decoupled Look-back. Technical Report NVR2016-001, NVIDIA Research. Mar. 2016.

8.7. Exercises

1. Analyze the parallel scan kernel in Figure 8.2. Show that control divergence only occurs in the first warp of each block for stride values up to half of the warp size. That is, for warp size 32, control divergence will occur to iterations for stride values 1, 2, 4, 8, and 16.

2. For the Brent-Kung scan kernel, assume that we have 2048 elements, how many add operations will be performed in both the reduction tree phase and the inverse reduction tree phase?
   (A) (2048-1)*2
   (B) (1024-1)*2
   (C) 1024*1024
   (D) 10*1024

3. For the Kogge-Stone scan kernel based on reduction trees, assume that we have 2048 elements, which of the following gives the closest approximation on how many add operations will be performed?
   (A) (2048-1)*2
   (B) (1024-1)*2
   (C) 1024*1024
   (D) 10*1024

4. Use the algorithm in Figure 8.3 to complete an exclusive scan kernel.

5. Complete the host code and all the three kernels for the hierarchical parallel scan algorithm in Figure 8.9.

6. Analyze the hierarchical parallel scan algorithm and show that it is work efficient and the total number of additions is no more than 4*N-3.

7. Consider the following array: [4 6 7 1 2 8 5 2]. Perform a parallel inclusive prefix scan on the array using the Kogge-Stone algorithm. Report the intermediate states of the array after each step.

8. Repeat the previous problem using the work-efficient algorithm.

9. Using the two-level hierarchical scan discussed in section 8.5, what is the largest possible dataset that can be handled if computing on a:

   a) GeForce GTX 280?
   b) Tesla C2050?
   c) GeForce GTX 690?