CHAPTER

Reduction
And minimizing divergence

10

Chapter Outline

A reduction derives a single value from an array of values. The single value could be the sum, the maximum value, the minimal value, and so on among all elements. The value can also be of various types: integer, single-precision floating-point, double-precision floating-point, half-precision floating-point, characters, and so on. All these types of reductions have the same computation structure. Like a histogram, reduction is an important computation pattern, as it generates a summary from a large amount of data. Parallel reduction is an important parallel pattern that requires parallel threads to coordinate with each other to get correct results. Such coordination must be done carefully to avoid performance bottlenecks, which are commonly found in parallel computing systems. Parallel reduction is therefore a good vehicle for illustrating these performance bottlenecks and introducing techniques for mitigating them.

10.1 Background

Mathematically, a reduction can be defined for a set of items based on a binary operator if the operator has a well-defined identity value. For example, a floating-point addition operator has an identity value of 0.0; that is, an addition of any floating-point value v and value 0.0 results in the value v itself. Thus a reduction

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00023-9

can be defined for a set of floating-point numbers based on the addition operator that produces the sum of all the floating-point numbers in the set. For example, for the set {7.0, 2.1, 5.3, 9.0, 11.2}, the sum reduction would produce 7.0+2.1 +5.3+9.0+11.2 = 34.6.

A reduction can be performed by sequentially going through every element of the array. Fig. 10.1 shows a sequential sum reduction code in C. The code initializes the result variable sum to the identity value 0.0. It then uses a for-loop to iterate through the input array that holds the set of values. During the ith iteration, the code performs an addition operation on the current value of sum and the ith element of the input array. In our example, after iteration 0, the sum variable contains 0.0+7.0 = 7.0. After iteration 1, the sum variable contains 7.0+2.1 = 9.1. Thus after each iteration, another value of the input array would be added (accumulated) to the sum variable. After iteration 5, the sum variable contains 34.6, which is the reduction result.

Reduction can be defined for many other operators. A product reduction can be defined for a floating-point multiplication operator whose identity value is 1.0. A product reduction of a set of floating-point numbers is the product of all these numbers. A minimum (min) reduction can be defined for a minimum comparison operator that returns the smaller value of the two inputs. For real numbers, the identity value for the minimum operator is $+\infty$. A maximum (max) reduction can be defined for a maximum (max) comparison operator that returns the larger value of the two input. For real numbers, the identity value for the maximum operator is $-\infty$.

Fig. 10.2 shows a general form of reduction for an operator, which is defined as a function that takes two inputs and returns a value. When an element is visited during an iteration of the for-loop, the action to take depends on the type of reduction being performed. For example, for a max reduction the Operator function performs a comparison between the two inputs and returns the larger value of the two. For a min reduction the values of the two inputs are compared by the operator function, and the

```
01    sum = 0.0f;
02    for(i = 0; i < N; ++i) {
03        sum += input[i];
04    }
```

FIGURE 10.1

A simple sum reduction sequential code.

```
01    acc = IDENTITY;
02    for(i = 0; i < N; ++i) {
03        acc = Operator(acc, input[i]);
04    }
```

FIGURE 10.2

The general form of a reduction sequential code.

smaller value is returned. The sequential algorithm ends when all the elements have been visited by the for-loop. For a set of N elements the for-loop iterates N iterations and produces the reduction result at the exit of the loop.

10.2 Reduction trees

Parallel reduction algorithms have been studied extensively in the literature. The basic concept of parallel reduction is illustrated in Fig. 10.3, where time progresses downwards in the vertical direction and the activities that threads perform in parallel in each time step are shown in the horizontal direction.

During the first round (time step), four max operations are performed in parallel on the four pairs of the original elements. These four operations produce partial reduction results: the four larger values from the four pairs of original elements. During the second time step, two max operations are performed in parallel on the two pairs of partial reduction results and produce two partial results that are even closer to the final reduction result. These two partial results are the largest value of the first four elements and the largest value of the second four elements in the original input. During the third and final time step, one max operation is performed to generate the final result, the largest value 7 from the original input.

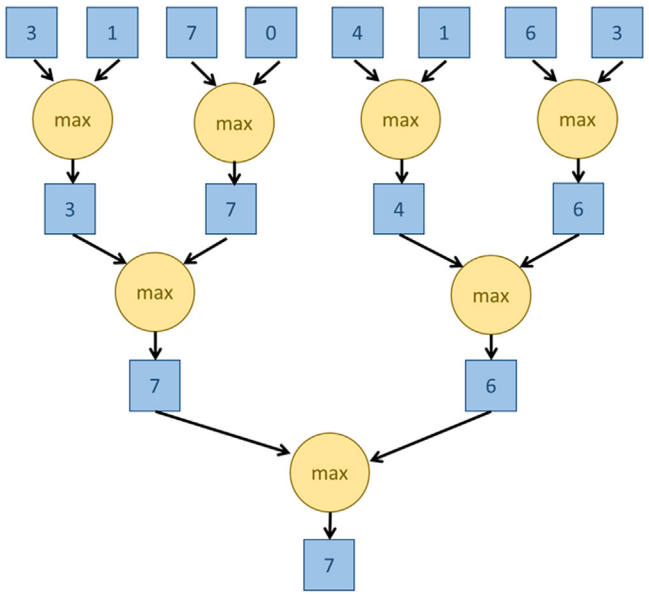


FIGURE 10.3

A parallel max reduction tree.

Note that the order of performing the operations will be changed from a sequential reduction algorithm to a parallel reduction algorithm. For example, for the input at the top of Fig. 10.3, a sequential max reduction like the one in Fig. 10.2 would start first by comparing the identity value ($-\infty$) in acc with the input value 3 and update acc with the winner, which is 3. It will then compare the value of acc (3) with the input value 1 and update acc with the winner, which is 3. This will be followed by comparing the value of acc (3) with the input value 7 and updating acc with the winner, which is 7. However, in the parallel reduction of Fig. 10.3 the input value 7 is first compared with the input value 0 before it is compared against the maximum of 3 and 1.

As we have seen, parallel reduction assumes that the order of applying the operator to the input values does not matter. In a max reduction, no matter what the order is for applying the operator to the input values, the outcome will be the same. This property is guaranteed mathematically if the operator is associative. An operator Θ is associative if $(a \Theta b) \Theta c = a \Theta (b \Theta c)$. For example, integer addition is associative ($(1+2)+3 = 1+(2+3) = 6$), whereas integer subtraction is not associative ($(1-2)-3 \neq 1-(2-3)$). That is, if an operator is associative, one can insert parentheses at arbitrary positions of an expression involving the operator and the results are all the same. With this equivalence relation, one can convert any order of operator application to any other order while preserving the equivalence of results. Strictly speaking, floating-point additions are not associative, owing to potentially rounding results for different ways of introducing parentheses. However, most applications accept floating-point operation results to be the same if they are within a tolerable difference from each other. Such definition allows developers and compiler writers to treat floating-point addition as an associative operator. Interested readers are referred to Appendix A for a detailed treatment.

The conversion from the sequential reduction in Fig. 10.2 to the reduction tree in Fig. 10.3 requires that the operator be associative. We can think of a reduction as a list of operations. The difference of ordering between Fig. 10.2 and Fig. 10.3 is just inserting parenthesis at different positions of the same list. For Fig. 10.2, the parentheses are:

((((((3 max 1) max 7) max 0) max 4) max 1) max 6) max 3

Whereas the parentheses for Fig. 10.3 are:

((3 max 1) max (7 max 0)) max ((4 max 1) max (6 max 3))

We will apply an optimization in Section 10.4 that not only rearranges the order of applying the operator but also rearranges the order of the operands. To rearrange the order of the operands, this optimization further requires the operator to be commutative. An operator is commutative if $a \Theta b = b \Theta a$. That is, the position of the operands can be rearranged in an expression and the results are the same. Note that the max operator is also commutative, as are many other operators such as min, sum, and product. Obviously, not all operators are commutative. For example, addition is commutative ($1+2 = 2+1$), whereas integer subtraction is not ($1-2 \neq 2-1$).

The parallel reduction pattern in Fig. 10.3 is called a reduction tree because it looks like a tree whose leaves are the original input elements and whose root is the final result. The term *reduction tree* is not to be confused with tree data structures, in which the nodes are linked either explicitly with pointers or implicitly with assigned positions. In reduction trees the edges are conceptual, reflecting information flow from operations performed in one time step to those in the next time step.

The parallel operations result in significant improvement over the sequential code in the number of time steps needed to produce the final result. In the example in Fig. 10.3 the for-loop in the sequential code iterates eight times, or takes eight time steps, to visit all of the input elements and produce the final result. On the other hand, with the parallel operations in Fig. 10.3 the parallel reduction tree approach takes only three time steps: four max operations during the first time step, two during the second, and one during the third. This is a decrease of $5/8 = 62.5\%$ in terms of number of time steps, or a speedup of $8/3 = 2.67$. There is, of course, a cost to the parallel approach: One must have enough hardware comparators to perform up to four max operations in the same time step. For N input values, a reduction tree performs ½ N operations during the first round, ¼ N operations in the second round, and so on. Therefore the total number of operations that are performed is defined by the geometric series ½ N+¼ N+$^1/_8$ N+... $^1/_N$ N = N − 1 operations, which is similar to the sequential algorithm.

In terms of time steps, a reduction tree takes $\log_2 N$ steps to complete the reduction process for N input values. Thus it can reduce N = 1024 input values in just ten steps, assuming that there are enough execution resources. During the first step, we need ½ N = 512 execution resources! Note that the number of resources that are needed diminishes quickly as we progress in time steps. During the final time step, we need to have only one execution resource. The level of parallelism at each time step is the same as the number of execution units that are required. It is interesting to calculate the average level of parallelism across all time steps. The average parallelism is the total number of operations that are performed divided by the number of time steps, which is $(N - 1)/\log_2 N$. For N = 1024 the average parallelism across the ten time steps is 102.3, whereas the peak parallelism is 512 (during the first time step). Such variation in the level of parallelism and resource consumption across time steps makes reduction trees a challenging parallel pattern for parallel computing systems.

Fig. 10.5 shows a sum reduction tree example for eight input values. Everything we presented so far about the number of time steps and resources consumed for max reduction trees is also applicable to sum reduction trees. It takes $\log_2 8 = 3$ time steps to complete the reduction using a maximum of four adders. We will use this example to illustrate the design of sum reduction kernels.

Parallel Reduction in Sports and Competitions

Parallel reduction has been used in sports and competitions long before the dawn of computing. Fig. 10.4 shows the schedule of the 2010 World Cup quarterfinals, semifinals, and the final game in South Africa. It should be clear that this is just a rearranged reduction tree. The elimination process of the World Cup is a maximum reduction in which the maximum operator returns the team that "beats" the other team. The tournament "reduction" is done in multiple rounds. The teams are divided into pairs. During the first round, all pairs play in parallel. Winners of the first round advance to the second round, whose winners advance to the third round, and so on. With eight teams entering a tournament, four winners will emerge from the first round (quarterfinals in Fig. 10.4), two from the second round (semifinals in Fig. 10.4), and one final winner (champion) from the third round (final in Fig. 10.4). Each round is a time step of the reduction process.

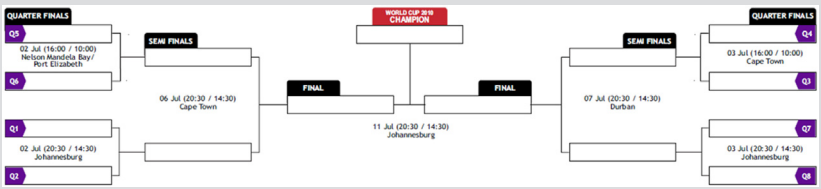


FIGURE 10.4

The 2010 World Cup Finals as a reduction tree.

It should be easy to see that even with 1024 teams, it takes only 10 rounds to determine the final winner. The trick is to have enough resources to hold the 512 games in parallel during the first round, 256 games in the second round, 128 games in the third round, and so on. With enough resources, even with 60,000 teams, we can determine the final winner in just 16 rounds. It is interesting to note that while reduction trees can greatly speed up the reduction process, they also consume quite a bit of resources. In the World Cup example, a game requires a large soccer stadium, officials, and staff as well as hotels and restaurants to accommodate the massive number of fans in the audience. The four quarterfinals in Fig. 10.4 were played in three cities (Nelson Mandela Bay/Port Elizabeth, Cape Town, and Johannesburg) that all together provided enough resources to host the four games. Note that the two games in Johannesburg were played on two different days. Thus sharing resources between two games made the reduction process take more time. We will see similar tradeoffs in computation reduction trees.

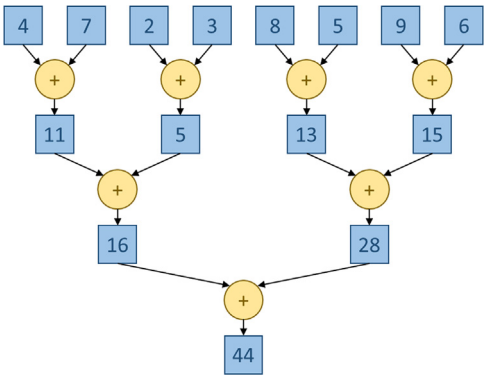


FIGURE 10.5

A parallel sum reduction tree.

10.3 A simple reduction kernel

We are now ready to develop a simple kernel to perform the parallel sum reduction tree shown in Fig. 10.5. Since the reduction tree requires collaboration across all threads, which is not possible across an entire grid, we will start by implementing a kernel that performs a sum reduction tree within a single block. That is, for an input array of N elements we will call this simple kernel and launch a grid with one block of ½N threads. Since a block can have up to 1024 threads, we can process up to 2048 input elements. We will eliminate this limitation in Section 10.8. During the first time step, all ½N threads will participate, and each thread adds two elements to produce ½N partial sums. During the next time step, half of the threads will drop off, and only ¼N threads will continue to participate to produce ¼N partial sums. This process will continue until the last time step, in which only one thread will remain and produce the total sum.

Fig. 10.6 shows the code of the simple sum kernel function, and Fig. 10.7 illustrates the execution of the reduction tree that is implemented by this code. Note that in Fig. 10.7 the time progresses from top to bottom. We assume that the `input` array is in the global memory and that a pointer to the array is passed as an argument when the kernel function is called. Each thread is assigned to a data location that is `2*threadIdx.x` (line 02). That is, the threads are assigned to the even locations in the `input` array: thread 0 to `input[0]`, thread 1 to `input[2]`, thread 2 to `input[4]`, and so on, as shown in the top row of Fig. 10.7. Each thread will be the "owner" of the location to which it is assigned and will be the only thread that writes into that location. The design of the kernel follows the "owner computes" approach, in which every data location is owned by a unique thread and can be updated only by that owner thread.

```
01    __global__ void SimpleSumReductionKernel(float* input, float* output) {
02        unsigned int i = 2*threadIdx.x;
03        for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
04            if (threadIdx.x % stride == 0) {
05                input[i] += input[i + stride];
06            }
07            __syncthreads();
08        }
09        if(threadIdx.x == 0) {
10            *output = input[0];
11        }
12    }
```

FIGURE 10.6

A simple sum reduction kernel.

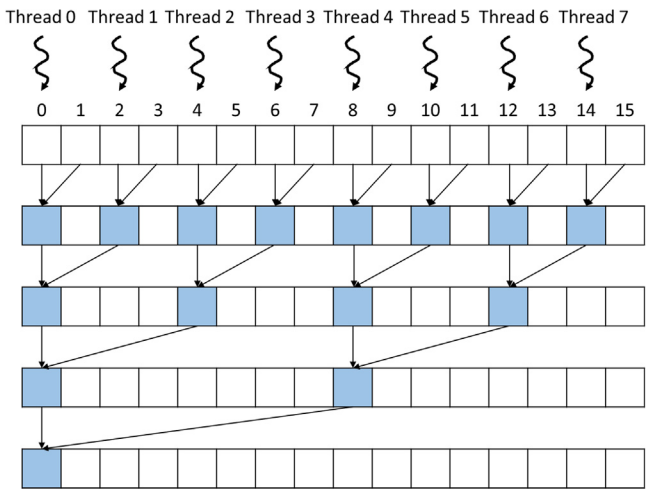


FIGURE 10.7

The assignment of threads ("owners") to the `input` array locations and progress of execution over time for the `SimpleSumReudctionKernel` in Fig. 10.6. The time progresses from top to bottom, and each level corresponds to one iteration of the for-loop.

The top row of Fig. 10.7 shows the assignment of threads to the locations of the input array, and each of the subsequent rows shows the writes to the input array locations at each time step, that is, iteration of the for-loop in Fig. 10.6 (line 03). The locations that the kernel overwrites in each iteration of the for-loop are marked as filled positions of the input array in Fig. 10.7. For example, at the end of the first iteration, locations with even indices are overwritten with the partial sums of pairs of the original elements (0−1, 2−3, 4−5, etc.) in the input array. At the end of the second iteration, the locations whose indices are multiples of 4 are overwritten with the partial sum of four adjacent original elements (0−3, 4−7, etc.) in the input array.

In Fig. 10.6 a stride variable is used for the threads to reach for the appropriate partial sums for accumulation into their owner locations. The stride variable is initialized to 1 (line 03). The value of the stride variable is doubled in each iteration so the stride variable value will be 1, 2, 4, 8, etc., until it becomes greater than `blockIdx.x`, the total number of threads in the block. As shown in Fig. 10.7, each active thread in an iteration uses the stride variable to add into its owned location the input array element that is of distance stride away. For example, in iteration 1, Thread 0 uses stride value 1 to add input[1] into its owned position input[0]. This updates input[0] to the partial sum of the first pair of original values of input[0] and input[1]. In the second iteration, Thread 0 uses stride value 2 to add input[2] to input[0]. At this time, input[2] contains the sum of the original values of input[2] and input[3] and input[0] contains the sum of the original values of input[0] and input[1]. So, after the second iteration, input[0] contains the sum of the original values of the first four elements of the input array. After the last iteration, input[0] contains the sum of all original elements of the input array and thus the result of the sum reduction. This value is written by Thread 0 as the final output (line 10).

We now turn to the if-statement in Fig. 10.6 (line 04). The condition of the if-statement is set up to select the active threads in each iteration. As shown in Fig. 10.7, during iteration *n*, the threads whose thread index (`threadIdx.x`) values are multiples of 2^n are to perform addition. The condition is `threadIdx.x % stride == 0`, which tests whether the thread index value is a multiple of the value of the stride variable. Recall that the value of stride is 1, 2, 4, 8, ... through the iterations, or 2^n for iteration n. Thus the condition indeed tests whether the thread index values are multiples of 2^n. Recall that all threads execute the same kernel code. The threads whose thread index value satisfies the if-condition are the active threads that perform the addition statement (line 05). The threads whose thread index values fail to satisfy the condition are the inactive threads that skip the addition statement. As the iterations progress, fewer and fewer threads remain active. At the last iteration, only thread 0 remains active and produces the sum reduction result.

The `__syncthreads()` statement (line 07 of Fig. 10.6) in the for-loop ensures that all partial sums that were calculated by the iteration have been written into their destination locations in the input array before any one of threads is allowed to begin the next iteration. This way, all threads that enter an iteration will be able to correctly use the partial sums that were produced in the previous iteration. For example, after the first iteration the even elements will be replaced by the pairwise partial sums. The `__syncthreads()` statement ensures that all these partial sums from the first iteration have indeed been written to the even locations of the input array and are ready to be used by the active threads in the second iteration.

10.4 Minimizing control divergence

The kernel code in Fig. 10.6 implements the parallel reduction tree in Fig. 10.7 and produces the expected sum reduction result. Unfortunately, its management of active

and inactive threads in each iteration results in a high degree of control divergence. For example, as shown in Fig. 10.7, only those threads whose `threadIdx.x` values are even will execute the addition statement during the second iteration. As we explained in Chapter 4, Compute Architecture and Scheduling, control divergence can significantly reduce the execution resource utilization efficiency, or the percentage of resources that are used in generating useful results. In this example, all 32 threads in a warp consume execution resources, but only half of them are active, wasting half of the execution resources. The waste of execution resources due to divergence increases over time. During the third iteration, only one-fourth of the threads in a warp are active, wasting three-quarters of the execution resources. During iteration 5, only one out of the 32 threads in a warp are active, wasting $^{31}/_{32}$ of the execution resources.

If the size of the input array is greater than 32, entire warps will become inactive after the fifth iteration. For example, for an input size of 256, 128 threads or four warps would be launched. All four warps would have the same divergence pattern, as we explained in the previous paragraph for iterations 1 through 5. During the sixth iteration, warp 1 and warp 3 would become completely inactive and thus exhibit no control divergence. On the other hand, warp 0 and warp 2 would have only one active thread, exhibiting control divergence and wasting $^{31}/_{32}$ of the execution resource. During the seventh iteration, only warp 0 would be active, exhibiting control divergence and wasting $^{31}/_{32}$ of the execution resource.

In general, the execution resource utilization efficiency for an input array of size N can be calculated as the ratio between the total number of active threads to the total number of execution resources that are consumed. The total number of execution resources that are consumed is proportional to the total number of active warps across all iterations, since every active warp, no matter how few of its threads are active, consumes full execution resources. This number can be calculated as follows:

$$\left(N/64^{*}5 + N/64^{*}1/2 + N/64^{*}1/4 + \ldots + 1\right)^{*}32$$

Here, N/64 is the total number of warps that are launched, since N/2 threads will be launched and every 32 threads form a warp. The N/64 term is multiplied by 5 because all launched warps are active for five iterations. After the fifth iteration the number of warps is reduced by half in each successive iteration. The expression in parentheses gives the total number of active warps across all the iterations. The second term reflects that each active warp consumes full execution resources for all 32 threads regardless of the number of active threads in these warps. For an input array size of 256, the consumed execution resource is (4*5+2+1)*32 = 736.

The number of execution results committed by the active threads is the total number of active threads across all iterations:

$$N/64^{*}(32 + 16 + 8 + 4 + 2 + 1) + N/64^{*}1/2^{*}1 + N/64^{*}1/4^{*}1 + \ldots + 1$$

The terms in the parenthesis give the active threads in the first five iterations for all N/64 warps. Starting at the sixth iteration, the number of active warps is reduced by half in each iteration, and there is only one active thread in each active warp. For an input array size of 256, the total number of committed results

is 4*(32+16+8+4+2+1)+2+1 = 255. This result should be intuitive because the total number of operations that are needed to reduce 256 values is 255.

Putting the previous two results together, we find that the execution resource utilization efficiency for an input array size of 256 is 255/736 = 0.35. This ratio states that the parallel execution resources did not achieve their full potential in speeding up this computation. On average, only about 35% of the resources consumed contributed to the sum reduction result. That is, we used only about 35% of the hardware's potential to speed up the computation.

Based on this analysis, we see that there is widespread control divergence across warps and over time. As the reader might have wondered, there may be a better way to assign threads to the input array locations to reduce control divergence and improve resource utilization efficiency. The problem with the assignment illustrated in Fig. 10.7 is that the partial sum locations become increasingly distant from each other, and thus the active threads that own these locations are also increasingly distant from each other as time progresses. This increasing distance between active threads contributes to the increasing level of control divergence.

There is indeed a better assignment strategy that significantly reduces control divergence. The idea is that we should arrange the threads and their owned positions so that they can remain close to each other as time progresses. That is, we would like to have the stride value decrease, rather than increase, over time. The revised assignment strategy is shown in Fig. 10.8 for an input array of 16 elements. Here, we assign the threads to the first half of the locations. During the first iteration, each thread reaches halfway across the input array and adds an input element

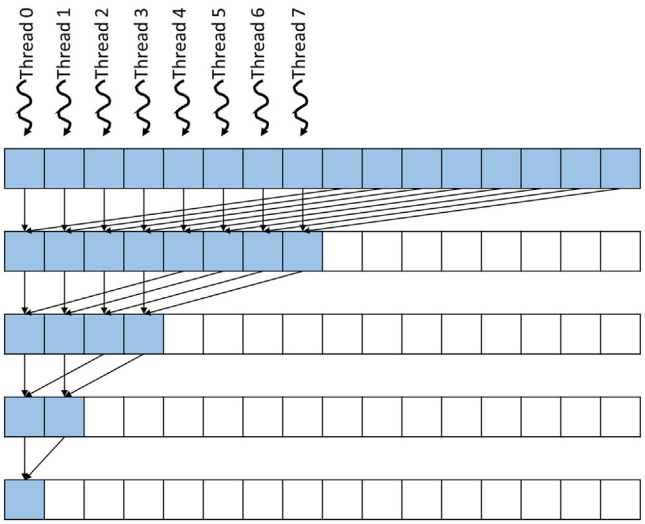


FIGURE 10.8

A better assignment of threads to input array locations for reduced control divergence.

to its owner location. In our example, thread 0 adds input[8] to its owned position input[0], thread 1 adds input[9] to its owned position input[1], and so on. During each subsequent iteration, half of the active threads drop off, and all remaining active threads add an input element whose position is the number of active threads away from its owner position. In our example, during the third iteration there are two remaining active threads: Thread 0 adds input[2] into its owned position input [0], and thread 1 adds input[3] into its owned position input[1]. Note that if we compare the operation and operand orders of Fig. 10.8 to Fig. 10.7, there is effectively a reordering of the operands in the list rather than just inserting parentheses in different ways. For the result to always remain the same with such reordering, the operation must be commutative as well as being associative.

Fig. 10.9 shows a kernel with some subtle but critical changes to the simple kernel in Fig. 10.6. The owner position variable i is set to `threadIdx.x` rather than `2*threadIdx.x` (line 02). Thus the owner positions of all threads are now adjacent to each other, as illustrated in Fig. 10.8. The stride value is initialized as `blockDim.x` and is reduced by half until it reaches 1 (line 03). In each iteration, only the threads whose indices are smaller than the stride value remain active (line 04). Thus all active threads are of consecutive thread indices, as shown in Fig. 10.8. Instead of adding neighbor elements in the first round, it adds elements that are half a section away from each other, and the section size is always twice the number of remaining active threads. All pairs that are added during the first round are `blockDim.x` away from each other. After the first iteration, all the pairwise sums are stored in the first half of the input array, as shown in Fig. 10.8. The loop divides the stride by 2 before entering the next iteration. Thus for the second iteration the stride variable value is half of the `blockDim.x` value. That is, the remaining active threads add elements that are a quarter of a section away from each other during the second iteration.

The kernel in Fig. 10.9 still has an if-statement (line 04) in the loop. The number of threads that execute an addition operation (line 06) in each iteration is the same as in Fig. 10.6. Then why should there be a difference in control divergence between the two kernels? The answer lies in the positions of threads that perform the addition operation relative to those that do not. Let us consider the example

```
01    __global__ void ConvergentSumReductionKernel(float* input, float* output) {
02        unsigned int i = threadIdx.x;
03        for (unsigned int stride = blockDim.x; stride >= 1; stride /= 2) {
04            if (threadIdx.x < stride) {
05                input[i] += input[i + stride];
06            }
07            __syncthreads();
08        }
09        if(threadIdx.x == 0) {
10            *output = input[0];
11        }
12    }
```

FIGURE 10.9

A kernel with less control divergence and improved execution resource utilization efficiency.

of an input array of 256 elements. During the first iteration, all threads are active, so there is no control divergence. During the second iteration, threads 0 through 63 execute the add statement (active), while threads 64 through 127 do not (inactive). The pairwise sums are stored in elements 0 through 63 during the second iteration. Since the warps consist of 32 threads with consecutive threadIdx.x values, all threads in warp 0 through warp 1 execute the add statement, whereas all threads in warp 2 through warp 3 become inactive. Since all threads in each warp take the same path of execution, there is no control divergence!

However, the kernel in Fig. 10.9 does not completely eliminate the divergence caused by the if-statement. The reader should verify that for the 256-element example, starting with the fourth iteration, the number of threads that execute the addition operation will fall below 32. That is, the final five iterations will have only 16, 8, 4, 2, and 1 thread(s) performing the addition. This means that the kernel execution will still have divergence in these iterations. However, the number of iterations of the loop that has divergence is reduced from ten to five. We can calculate the total number of execution resources consumed as follows:

$$\left(N/64^{*}1 + N/64^{*}1/_2 + \ldots + 1 + 5^{*}1\right)^{*}32$$

The part in parentheses reflects the fact that in each subsequent iteration, half of the warps become entirely inactive and no longer consume execution resources. This series continues until there is only one full warp of active threads. The last term (5*1) reflects the fact that for the final five iterations, there is only one active warp, and all its 32 threads consume execution resources even though only a fraction of the threads are active. Thus the sum in the parentheses gives the total number of warp executions through all iterations, which, when multiplied by 32, gives the total amount of execution resources that are consumed.

For our 256-element example the execution resources that are consumed are (4+2+1+5*1)*32 = 384, which is almost half of 736, the resources that were consumed by the kernel in Fig. 10.6. Since the number of active threads in each iteration did not change from Fig. 10.7 to Fig. 10.8, the efficiency of the new kernel in Fig. 10.9 is 255/384 = 66%, which is almost double the efficiency of the kernel in Fig. 10.6. Note also that since the warps are scheduled to take turns executing in a streaming multiprocessor of limited execution resources, the total execution time will also improve with the reduced resource consumption.

The difference between the kernels in Fig. 10.6 and Fig. 10.9 is small but can have a significant performance impact. It requires someone with clear understanding of the execution of threads on the single-instruction, multiple-data hardware of the device to be able to confidently make such adjustments.

10.5 Minimizing memory divergence

The simple kernel in Fig. 10.6 has another performance issue: memory divergence. As we explained in Chapter 5, Memory Architecture and Data Locality, it

is important to achieve memory coalescing within each warp. That is, adjacent threads in a warp should access adjacent locations when they access global memory. Unfortunately, in Fig. 10.7, adjacent threads do not access adjacent locations. In each iteration, each thread performs two global memory reads and one global memory write. The first read is from its owned location, the second read is from the location that is of stride distance away from its owned location, and the write is to its owned location. Since the locations owned by adjacent thread are not adjacent locations, the accesses that are made by adjacent threads will not be fully coalesced. During each iteration the memory locations that are collectively accessed by a warp are of stride distance away from each other.

For example, as shown in Fig. 10.7, when all threads in a warp perform their first read during the first iteration, the locations are two elements away from each other. As a result, two global memory requests are triggered, and half the data returned will not be used by the threads. The same behavior occurs for the second read and the write. During the second iteration, every other thread drops out, and the locations that are collectively accessed by the warp are four elements away from each other. Two global memory requests are again performed, and only one-fourth of the data returned will be used by the threads. This will continue until there is only one active thread for each warp that remains active. Only when there is one active thread in the warp will the warp perform one global memory request. Thus the total number of global memory requests is as follows:

$$\left(N/64^{*}5^{*}2 + N/64^{*}1 + N/64^{*}1/2 + N/64^{*}1/4 + \ldots + 1\right)^{*}3$$

The first term (N/64*5*2) corresponds to the first five iterations, in which all N/64 warps have two or more active threads, so each warp performs two global memory requests. The remaining terms account for the final iterations, in which each warp has only one active thread and performs one global memory request and half of the warps drop out in each subsequent iteration. The multiplication by 3 accounts for the two reads and one write by each active thread during each iteration. In the 256-element example the total number of global memory requests performed by the kernel is (4*5*2+4+2+1)*3 = 141.

For the kernel in Fig. 10.9 the adjacent threads in each warp always access adjacent locations in the global memory, so the accesses are always coalesced. As a result, each warp triggers only one global memory request on any read or write. As the iterations progress, entire warps drop out, so no global memory access will be performed by any thread in these inactive warps. Half of the warps drop out in each iteration until there is only one warp for the final five iterations. Therefore the total number of global memory requests performed by the kernel is as follows:

$$\left(\left(N/64 + N/64^{*}1/2 + N/64^{*}1/4 + \ldots + 1\right) + 5\right)^{*}3$$

For the 256-element example the total number of global memory requests performed is ((4+2+1)+5)*3 = 36. The improved kernel results in 141/36 = 3.9× fewer global memory requests. Since DRAM bandwidth is a limited resource, the execution time is likely to be significantly longer for the simple kernel in Fig. 10.6.

For a 2048-element example the total number of global memory requests that are performed by the kernel in Fig. 10.6 is (32*5*2+32+16+8+4+2+1)*3 = 1149, whereas the number of global memory requests that are performed by the kernel in Fig. 10.9 is (32+16+8+4+2+1+5)*3 = 204. The ratio is 5.6, even more than in the 256-element example. This is because of the inefficient execution pattern of the kernel in Fig. 10.6, in which there are more active warps during the initial five iterations of the execution and each active warp triggers twice the number of global memory requests as the convergent kernel in Fig. 10.9.

In conclusion, the convergent kernel offers more efficiency in using both execution resources and DRAM bandwidth. The advantage comes from both reduced control divergence and improved memory coalescing.

10.6 Minimizing global memory accesses

The convergent kernel in Fig. 10.9 can be further improved by using shared memory. Note that in each iteration, threads write their partial sum result values out to the global memory, and these values are reread by the same threads and other threads in the next iteration. Since the shared memory has much shorter latency and higher bandwidth than the global memory, we can further improve the execution speed by keeping the partial sum results in the shared memory. This idea is illustrated in Fig. 10.10.

The strategy for using the shared memory is implemented in the kernel shown in Fig. 10.11. The idea is to use each thread to load and add two of the original

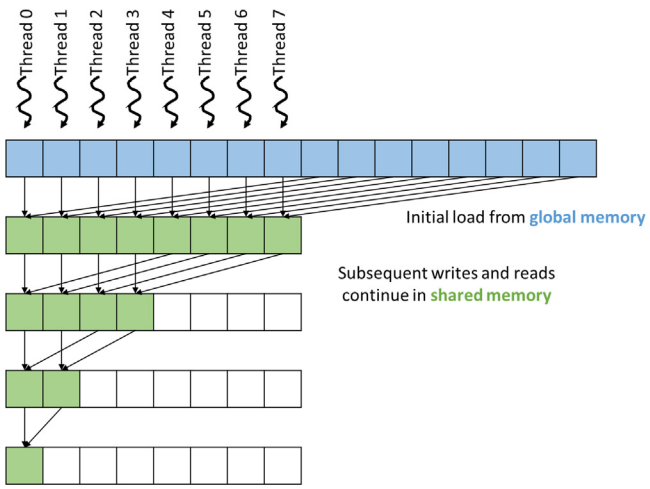


FIGURE 10.10

Using shared memory to reduce accesses to the global memory.

```
01    __global__ void SharedMemorySumReductionKernel(float* input) {
02        __shared__ float input_s[BLOCK_DIM];
03        unsigned int t = threadIdx.x;
04        input_s[t] = input[t] + input[t + BLOCK_DIM];
05        for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2) {
06            __syncthreads();
07            if (threadIdx.x < stride) {
08                input_s[t] += input_s[t + stride];
09            }
10        }
11        if (threadIdx.x == 0) {
12            *output = input_s[0];
13        }
14    }
```

FIGURE 10.11

A kernel that uses shared memory to reduce global memory accesses.

elements before writing the partial sum into the shared memory (line 04). Since the first iteration is already done when accessing the global memory locations outside the loop, the for-loop starts with `blockDim.x/2` (line 04) instead of `blockDim.x`. The `__syncthreads()` is moved to the beginning of the loop to ensure that we synchronize between the shared memory accesses and the first iteration of the loop. The threads proceed with the remaining iterations by reading and writing the shared memory (line 08). Finally, at the end of the kernel, thread 0 writes the sum into output, maintaining the same behavior as in the previous kernels (lines 11−13).

Using the kernel in Fig. 10.11, the number of global memory accesses are reduced to the initial loading of the original contents of the input array and the final write to input[0]. Thus for an N-element reduction the number of global memory accesses is just N+1. Note also that both global memory reads in Fig. 10.11 (line 04) are coalesced. So with coalescing, there will be only (N/32) +1 global memory requests. For the 256-element example the total number of global memory requests that are triggered will be reduced from 36 for the kernel in Fig. 10.9 to 8+1 = 9 for the shared memory kernel in Fig. 10.10, a 4× improvement. Another benefit of using shared memory, besides reducing the number of global memory accesses, is that the input array is not modified. This property is useful if the original values of the array are needed for some other computation in another part of the program.

10.7 Hierarchical reduction for arbitrary input length

All the kernels that we have studied so far assume that they will be launched with one thread block. The main reason for this assumption is that `__syncthreads()` is used as a barrier synchronization among all the active threads. Recall that `__syncthreads()` can be used only among threads in the same block. This limits

the level of parallelism to 1024 threads on current hardware. For large input arrays that contain millions or even billions of elements, we can benefit from launching more threads to further accelerate the reduction process. Since we do not have a good way to perform barrier synchronization among threads in different blocks, we will need to allow threads in different blocks to execute independently.

Fig. 10.12 illustrates the concept of hierarchical, segmented multiblock reduction using atomic operations, and Fig. 10.13 shows the corresponding kernel implementation. The idea is to partition the input array into segments so that each segment is of appropriate size for a block. All blocks then independently execute a reduction tree and accumulate their results to the final output using an atomic add operation.

The partitioning is done by assigning a different value to the `segment` variable according to the thread's block index (line 03). The size of each segment is

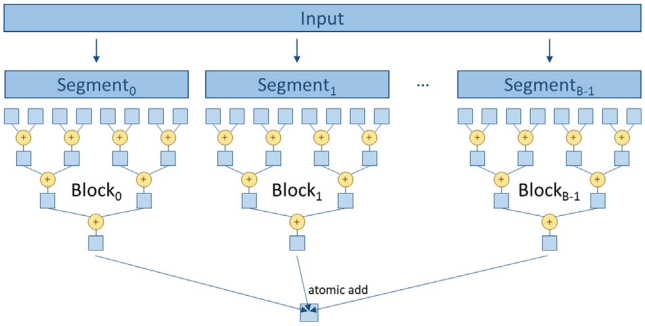


FIGURE 10.12

Segmented multiblock reduction using atomic operations.

```
01    __global__ SegmentedSumReductionKernel(float* input, float* output) {
02        __shared__ float input_s[BLOCK_DIM];
03        unsigned int segment = 2*blockDim.x*blockIdx.x;
04        unsigned int i = segment + threadIdx.x;
05        unsigned int t = threadIdx.x;
06        input_s[t] = input[i] + input[i + BLOCK_DIM];
07        for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2){
08            __syncthreads();
09            if (t < stride) {
10                input_s[t] += input_s[t + stride];
11            }
12        }
13        if (t == 0) {
14              atomicAdd(output, input_s[0]);
15          }
16    }
```

FIGURE 10.13

A segmented multiblock sum reduction kernel using atomic operations.

2*blockDim.x. That is, each block processes 2*blockDim.x elements. Thus when we multiply the size of each segment by blockIdx.x of a block, we have the starting location of the segment to be processed by the block. For example, if we have 1024 threads in a block, the segment size would be 2*1024 = 2048. The starting locations of segments would be 0 for block 0, 2048 (2048*1) for block 1, 4096 (2048*2) for block 2, and so on.

Once we know the starting location for each block, all threads in a block can simply work on the assigned segment as if it is the entire input data. Within a block, we assign the owned location to each thread by adding the threadIdx.x to the segment starting location for the block to which the thread belongs (line 04). The local variable i holds the owned location of the thread in the global input array, whereas t holds the owned location of the thread in the shared input_s array. Line 06 is adapted to use i instead of t when accessing the global input array. The for-loop from Fig. 10.11 is used without change. This is because each block has its own private input_s in the shared memory, so it can be accessed with t = threadIdx.x as if the segment is the entire input.

Once the reduction tree for-loop is complete, the partial sum for the segment is in input_s[0]. The if-statement in line 16 of Fig. 10.13 selects thread 0 to contribute the value in input_s[0] to output, as illustrated in the bottom part of Fig. 10.12. This is done with an atomic add, as shown in line 14 of Fig. 10.13. Once all blocks of the grid have completed execution, the kernel will return, and the total sum is in the memory location pointed to by output.

10.8 Thread coarsening for reduced overhead

The reduction kernels that we have worked with so far all try to maximize parallelism by using as many threads as possible. That is, for a reduction of N elements, N/2 threads are launched. With a thread block size of 1024 threads, the resulting number of thread blocks is N/2048. However, in processors with limited execution resources the hardware may have only enough resources to execute a portion of the thread blocks in parallel. In this case, the hardware will serialize the surplus thread blocks, executing a new thread block whenever an old one has completed.

To parallelize reduction, we have actually paid a heavy price to distribute the work across multiple thread blocks. As we saw in earlier sections, hardware underutilization increases with each successive stage of the reduction tree because of more warps becoming idle and the final warp experiencing more control divergence. The phase in which the hardware is underutilized occurs for every thread block that we launch. It is an inevitable price to pay if the thread blocks are to actually run in parallel. However, if the hardware is to serialize these thread blocks, we are better off serializing them ourselves in a more efficient manner. As we discussed in Chapter 6, Performance Considerations, thread granularity coarsening, or

thread coarsening for brevity, is a category of optimizations that serialize some of the work into fewer threads to reduce parallelization overhead. We start by showing an implementation of parallel reduction with thread coarsening applied by assigning more elements to each thread block. We then further elaborate on how this implementation reduces hardware underutilization.

Fig. 10.14 illustrates how thread coarsening can be applied to the example in Fig. 10.10. In Fig. 10.10, each thread block received 16 elements, which is two elements per thread. Each thread independently adds the two elements for which it is responsible; then the threads collaborate to execute a reduction tree. In Fig. 10.14 we coarsen the thread block by a factor of 2. Hence each thread block receives twice the number of elements, that is, 32 elements, which is four elements per thread. In this case, each thread independently adds four elements before the threads collaborate to execute a reduction tree. The three steps to add the four elements are illustrated by the first three rows of arrows in Fig. 10.14. Note that all threads are active during these three steps. Moreover, since the threads independently add the four elements for which they are responsible, they do not need to synchronize, and they do not need to store their partial sums to shared memory until after all four elements have been added. The remaining steps in performing the reduction tree are the same as those in Fig. 10.10.

Fig. 10.15 shows the kernel code for implementing reduction with thread coarsening for the multiblock segmented kernel. Compared to Fig. 10.13, the kernel has two main differences. The first difference is that when the beginning of the block's segment is identified, we multiply by `COARSE_FACTOR` to reflect the fact that the size of the block's segment is `COARSE_FACTOR` times larger (line 03). The second difference is that when adding the elements for which the thread is responsible, rather than just adding two elements (line 06 in Fig. 10.13), we use a coarsening loop to iterate over the elements and add them based on

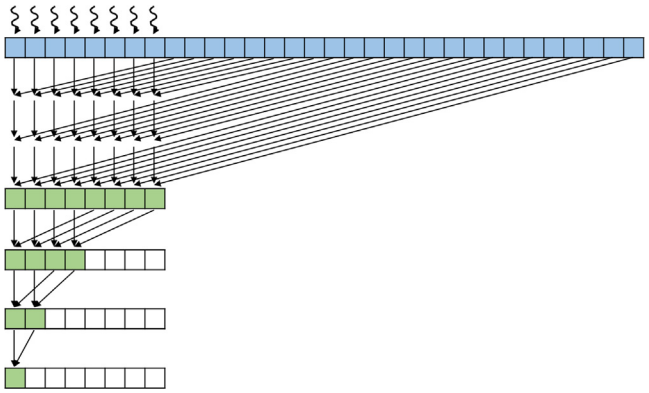


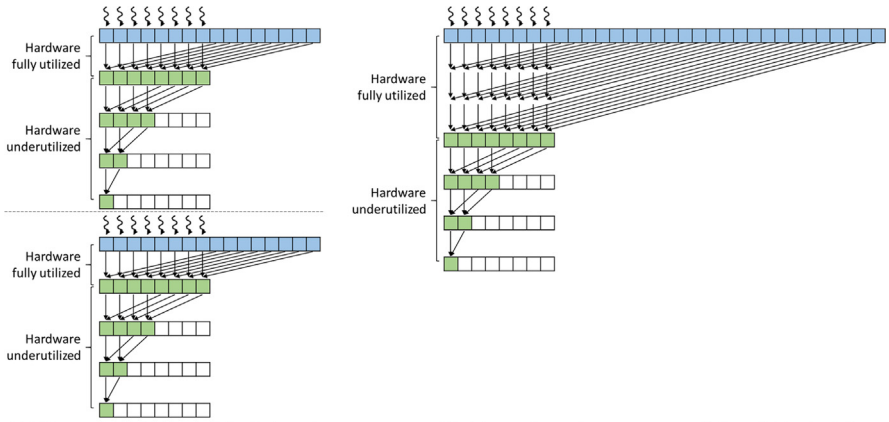
FIGURE 10.14

Thread coarsening in reduction.

```
01    __global__ CoarsenedSumReductionKernel(float* input, float* output) {
02        __shared__ float input_s[BLOCK_DIM];
03        unsigned int segment = COARSE_FACTOR*2*blockDim.x*blockIdx.x;
04        unsigned int i = segment + threadIdx.x;
05        unsigned int t = threadIdx.x;
06        float sum = input[i];
07        for(unsigned int tile = 1; tile < COARSE_FACTOR*2; ++tile) {
08            sum += input[i + tile*BLOCK_DIM];
09        }
10        input_s[t] = sum;
11        for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2){
12            __syncthreads();
13            if (t < stride) {
14                input_s[t] += input_s[t + stride];
15            }
16        }
17        if (t == 0) {
18              atomicAdd(output, input_s[0]);
19          }
20    }
```

FIGURE 10.15

Sum reduction kernel with thread coarsening.



(A) Execution of two original thread blocks serialized by the hardware

(B) Execution of one coarsened thread block doing the work of two original thread blocks

FIGURE 10.16

Comparing parallel reduction with and without thread coarsening.

`COARSE_FACTOR` (lines 06–09 in Fig. 10.15). Note that all threads are active throughout this coarsening loop, the partial sum is accumulated to the local variable `sum`, and no calls to `__syncthreads()` are made in the loop because the threads act independently.

Fig. 10.16 compares the execution of two original thread blocks without coarsening serialized by the hardware, shown in Fig. 10.16A with one coarsened thread block performing the work of two thread blocks, shown in Fig. 10.16B. In

Fig. 10.16A the first thread block performs one step in which each thread adds the two elements for which it is responsible. All threads are active during this step, so the hardware is fully utilized. The remaining three steps execute the reduction tree in which half the threads drop out each step, underutilizing the hardware. Moreover, each step requires a barrier synchronization as well as accesses to shared memory. When the first thread block is done, the hardware then schedules the second thread block, which follows the same steps but on a different segment of the data. Overall, the two blocks collectively take a total of eight steps, of which two steps fully utilize the hardware and six steps underutilize the hardware and require barrier synchronization and shared memory access.

By contrast, in Fig. 10.16B the same amount of data is processed by only a single thread block that is coarsened by a factor of 2. This thread block initially takes three steps in which each thread adds the four elements for which it is responsible. All threads are active during all three steps, so the hardware is fully utilized, and no barrier synchronizations or accesses to shared memory are performed. The remaining three steps execute the reduction tree in which half the threads drop out each step, underutilizing the hardware, and barrier synchronization and accesses to shared memory are needed. Overall, only six steps are needed (instead of eight), of which three steps (instead of two) fully utilize the hardware and three steps (instead of six) underutilize the hardware and require barrier synchronization and shared memory access. Therefore thread coarsening effectively reduces the overhead from hardware underutilization, synchronization, and access to shared memory.

Theoretically, we can increase the coarsening factor well beyond two. However, one must keep in mind that as we coarsen threads, less work will be done in parallel. Therefore increasing the coarsening factor will reduce the amount of data parallelism that is being exploited by the hardware. If we increase the coarsening factor too much, such that we launch fewer thread blocks than the hardware is capable of executing, we will no longer be able to take full advantage of the parallel hardware execution resources. The best coarsening factor ensures that there are enough thread blocks to fully utilize the hardware, which usually depends on the total size of the input as well as the characteristics of the specific device.

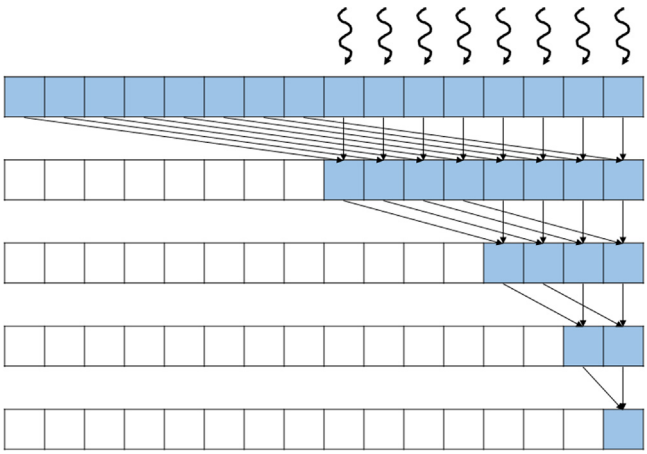
10.9 Summary

The parallel reduction pattern is important, as it plays a key row in many data-processing applications. Although the sequential code is simple, it should be clear to the reader that several techniques, such as thread index assignment for reduced divergence, using shared memory for reduced global memory accesses, segmented reduction with atomic operations, and thread coarsening, are needed to achieve high performance for large inputs. The reduction computation is also an important

foundation for the prefix-sum pattern that is an important algorithm component for parallelizing many applications and will be the topic of Chapter 11, Prefix Sum (Scan).

Exercises

1. For the simple reduction kernel in Fig. 10.6, if the number of elements is 1024 and the warp size is 32, how many warps in the block will have divergence during the fifth iteration?
2. For the improved reduction kernel in Fig. 10.9, if the number of elements is 1024 and the warp size is 32, how many warps will have divergence during the fifth iteration?
3. Modify the kernel in Fig. 10.9 to use the access pattern illustrated below.



4. Modify the kernel in Fig. 10.15 to perform a max reduction instead of a sum reduction.
5. Modify the kernel in Fig. 10.15 to work for an arbitrary length input that is not necessarily a multiple of `COARSE_FACTOR*2*blockDim.x`. Add an extra parameter `N` to the kernel that represents the length of the input.
6. Assume that parallel reduction is to be applied on the following input array:

6	2	7	4	5	8	3	1

Show how the contents of the array change after each iteration if:

a. The unoptimized kernel in Fig. 10.6 is used.

Initial array:

6	2	7	4	5	8	3	1

b. The kernel optimized for coalescing and divergence in Fig. 10.9 is used.

Initial array:

6	2	7	4	5	8	3	1

CHAPTER

11 Prefix sum (scan)

An introduction to work efficiency in parallel algorithms

With special contributions from Li-Wen Chang, Juan Gómez-Luna and John Owens

Chapter Outline

Our next parallel pattern is prefix sum, which is also commonly known as scan. Parallel scan is frequently used to parallelize seemingly sequential operations, such as resource allocation, work assignment, and polynomial evaluation. In general, if a computation is naturally described as a mathematical recursion in which each item in a series is defined in terms of the previous item, 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. For this reason, 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, and stream compaction. The radix sort example will be presented in Chapter 13, Sorting.

Another reason why parallel scan is an important parallel pattern is that it is a typical example of where the work performed by some parallel algorithms can have higher complexity than the work performed by a sequential algorithm, leading to a tradeoff that needs to be carefully made between algorithm complexity and parallelization. As we will show, a slight increase in algorithm complexity can make parallel

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00022-7

scan run more slowly than sequential scan for large datasets. Such consideration is becoming even more important in the age of "big data," in which massive datasets challenge tradition algorithms that have high computational complexity.

11.1 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 following output array:

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

For example, if $\oplus$ is addition, an inclusive scan operation on the input array [3 1 7 0 4 1 6 3] would return [2, 3+1, 3+1+7, 3+1+7+0, ..., 3+1+7+0+4+1+6+3] =[3 4 11 11 15 16 22 25]. The name "inclusive" scan comes from the fact that each output element *includes* the effect of the corresponding input element.

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 eight people. Each person has ordered a different amount in terms of inches: 3, 1, 7, 0, 4, 1, 6, and 3. That is, Person 0 wants 3 inches of sausage, Person 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 0. The sausage is now 37 inches long. We then cut a one-inch section for Person 1. The sausage becomes 36 inches long. We can continue to cut more sections until we serve the 3-inch section to Person 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 ordered by each person. 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 the 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 0. The second cut point is at the 4-inch location; therefore the second section will be 1-inch long, as ordered by Person 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 the 22-inch point. This gives Person 7 what she ordered. Note that since all the cut points are known from the scan operation, all cuts can be done in parallel or in any arbitrary sequence.

In summary, an intuitive way of thinking about inclusive scan is that the operation takes a request 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, campground 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 scan operation with a slightly different arrangement of the output array:

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

That is, each output element *excludes* the effect of the corresponding input element. The first output element is i, the identity value for operator $\oplus$, while the last output element only reflects the contribution of up to x_{n-2}. An identity value for a binary operator is defined as a value that, when used as an input operand, causes the operation to generate an output value that is the same as the other input operand's value. In the case of the addition operator, the identity value is 0, as any number added with zero will result in itself.

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, an 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 0 starts at the 0-inch point. For another example, the section for Person 7 starts at the 22-inch point. The beginning point information is important in applications such as memory allocation, in which the allocated memory is returned to the requester via a pointer to its beginning point.

Note that it is 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 identity value for the 0th element. When converting from exclusive to inclusive, one needs to shift all elements to the left and fill in the last element with the previous last element $\oplus$ the last input element. It is just a matter of convenience that we can directly generate an inclusive or exclusive scan depending on whether we care about the cut points or the beginning points for the sections. Therefore we will present parallel algorithms and implementations only for inclusive scan.

Before we present parallel scan algorithms and their implementations, we would like to show a sequential inclusive scan algorithm and its implementation. We will assume that the operator involved is addition. The code in Fig. 11.1 assumes that the input elements are in the `x` array and the output elements are to be written into the `y` array.

The code initializes the output element y[0] with the value of input element x[0] (line 02). In each iteration of the loop (lines 03–05), the loop body adds one more input element to the previous output element (which stores the accumulation of all the previous input elements) to generate one more output element.

It should be clear that the work done by the sequential implementation of inclusive scan in Fig. 11.1 is linearly proportional to the number of input elements; that is, the computational complexity of the sequential algorithm is $O(N)$.

In Sections 11.2–11.5, we will present alternative algorithms for performing parallel *segmented scan*, in which every thread block will perform a scan on a segment, that is, a section, of elements in the input array in parallel. We will then present in Sections 11.6 and 11.7 methods that combine the segmented scan results into the scan output for the entire input array.

```
01    void sequential_scan(float *x, float *y, unsigned int N) {
02        y[0] = x[0];
03        for(unsigned int i = 1; i < N; ++i) {
04            y[i] = y[i - 1] + x[i];
05        }
06    }
```

FIGURE 11.1

A simple sequential implementation of inclusive scan based on addition.

11.2 Parallel scan with the Kogge-Stone algorithm

We start with a simple parallel inclusive scan algorithm by performing a reduction operation for each output element. One might be tempted to use each thread to perform sequential reduction as shown in Fig. 10.2 for one output element. After all, this allows the calculations for all output elements to be performed in parallel. Unfortunately, this approach will be unlikely to improve the execution time over the sequential scan code in Fig. 11.1. This is because the calculation of y_{n-1} will take n steps, the same number of steps taken by the sequential scan code, and each step (iteration) in the reduction involves the same amount of work as each iteration of the sequential scan. Since the completion time of a parallel program is limited by the thread that takes the longest time, this approach is unlikely be any faster than sequential scan. In fact, with limited computing resources, the execution time of this naïve parallel scan algorithm can be much longer than that of the sequential algorithm. The computation cost, or the total number of operations performed, would unfortunately be much higher for the proposed approach. Since the number of reduction steps for output element i would be i, the total number of steps performed by all threads would be

$$\sum_{i=0}^{n-1} i \quad = \frac{n \cdot (n-1)}{2}$$

That is, the proposed approach has a computation complexity of $O(N^2)$, which is higher than the complexity of the sequential scan, which is $O(N)$, while offering no speedup. The higher computational complexity means that considerably more execution resources need to be provisioned. This is obviously a bad idea.

A better approach is to adapt the parallel reduction tree in Chapter 10, Reduction and Minimizing Divergence, to calculate each output element with a reduction tree of the relevant input elements. There are multiple ways to design the reduction tree for each output element. Since the reduction tree for element i involves i add operations, this approach would still increase the computational complexity to $O(N^2)$ unless we find a way to share the partial sums across the reduction trees of different output elements. We present such a sharing approach that is based on the Kogge-Stone algorithm, which was originally invented for designing fast adder circuits in the 1970s (Kogge & Stone, 1973). This algorithm is still being used in the design of high-speed computer arithmetic hardware.

The algorithm, illustrated in Fig. 11.2, is an in-place scan algorithm that operates on an array XY that originally contains input elements. It iteratively evolves the contents of the array into output elements. Before the algorithm begins, we assume that XY[i] contains input element x_i. After k iterations, XY[i] will contain the sum of up to 2^k input elements at and before the location. For example, after one iteration, 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.

Fig. 11.2 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 Fig. 11.2. 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 Fig. 11.2. For example, after the first iteration, XY[3] contains x_2+x_3, shown as

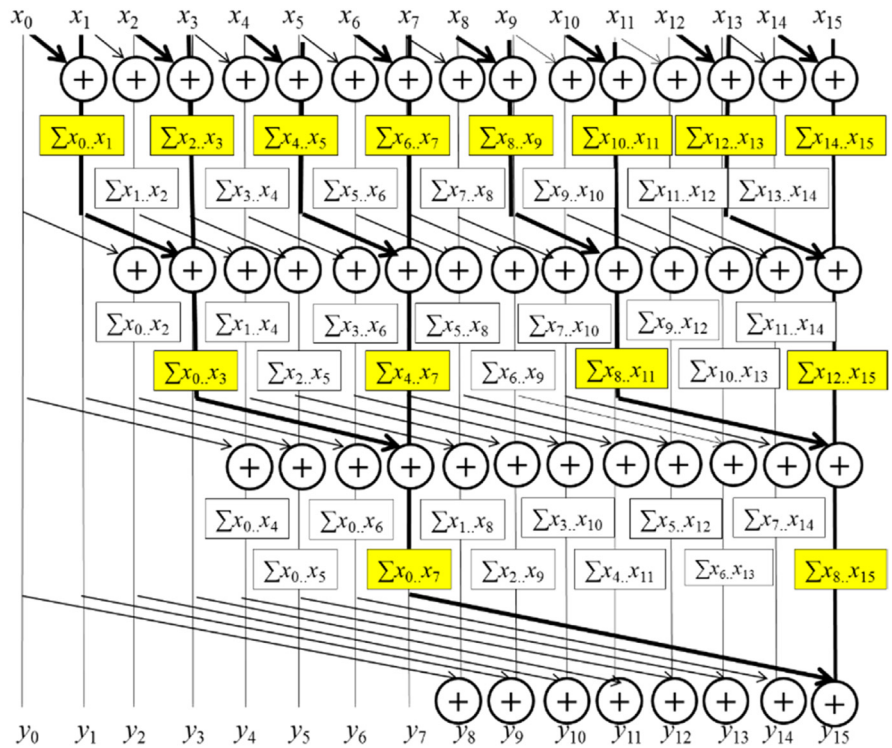


FIGURE 11.2

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

$\sum x_2 \ldots x_3$. 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 labeled boxes below the second row of addition operators. As a result, XY[i] becomes $x_{i-3}+x_{i-2}+x_{i-1}+x_i$. For example, after the second iteration, XY[3] becomes $x_0+x_1+x_2+x_3$, shown as $\sum x_0 \ldots x_3$. Note that after the second iteration, XY[2] and XY[3] have reached 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.

Fig. 11.3 shows the parallel implementation of the algorithm illustrated in Fig. 11.2. We implement a kernel that performs local scans on different segments (sections) of the input that are each small enough for a single block to handle. Later, we will make final adjustments to consolidate these sectional scan results for large input arrays. The size of a section is defined as a compile-time constant `SECTION_SIZE`. We assume that the kernel function will be called using `SECTION_SIZE` as the block size, so there will be the same number of threads and section elements. We assign each thread to evolve the contents of one XY element.

The implementation shown in Fig. 11.3 assumes that input values were originally in a global memory array X, whose address is passed into the kernel as an argument (line 01). We will have all the threads in the block collaboratively load the X array elements into a shared memory array XY (line 02). This is done by having each thread calculate its global data index `i=blockIdx.x*blockDim.x+threadIdx.x` (line 03) for the output vector element position it is responsible for. Each thread loads the input element at

```
01    __global__ void Kogge_Stone_scan_kernel(float *X, float *Y, unsigned int N){
02        __shared__ float XY[SECTION_SIZE];
03      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04      if(i < N) {
05          XY[threadIdx.x] = X[i];
06      } else {
07          XY[threadIdx.x] = 0.0f;
08      }
09      for(unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
10            __syncthreads();
11          float temp;
12          if(threadIdx.x >= stride)
13              temp = XY[threadIdx.x] + XY[threadIdx.x-stride];
14            __syncthreads();
15          if(threadIdx.x >= stride)
16              XY[threadIdx.x] = temp;
17      }
18      if(i < N) {
19          Y[i] = XY[threadIdx.x];
20      }
21  }
```

FIGURE 11.3

A Kogge-Stone kernel for inclusive (segmented) scan.

that position into the shared memory at the beginning of the kernel (lines 04–08). At the end of the kernel, each thread will write its result into the assigned output array Y (lines 18–20).

We now focus on the implementation of the iterative calculations for each XY element in Fig. 11.3 as a for-loop (lines 09–17). The loop iterates through the reduction tree for the XY array position that is assigned to a thread. 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 and the thread no longer needs to be active (lines 12 and 15). Note that we use a barrier synchronization (line 10) to make sure that all threads have finished their previous iteration before any of them starts the next iteration. This is the same use of `__syncthreads()` as in the reduction discussion in Chapter 10, Reduction and Minimizing Divergence.

There is, however, a very important difference compared to reduction in updating of the XY elements (lines 12–16) in each iteration of the for-loop. Note that each active thread first stores the partial sum for its position into a temp variable (in a register). After all threads have completed a second barrier synchronization (line 14), all of them store their partial sum values to their XY positions (line 16). The need for the extra temp and `__syncthreads()` has to do with a write-after-read data dependence hazard in these updates. Each active thread adds the XY value at its own position `(XY[threadIdx.x])` and that at a position of another thread `(XY[threadIdx.x-stride])`. If a thread *i* writes to its output position before another thread *i+stride* has had the chance to read the old value at that position, the new value can corrupt the addition performed by the other thread. The corruption may or may not occur, depending on the execution timing of the threads involved, which is referred to as a race condition. Note that this race condition is different from the one we saw in Chapter 9, Parallel Histogram, with the histogram pattern. The race condition in Chapter 9, Parallel Histogram, was a read-modify-write race condition that can be solved with atomic operations. For the write-after-read race condition that we see here, a different solution is required.

The race condition can be easily observed in Fig. 11.2. Let us examine the activities of thread 4 (x_4) and thread 6 (x_6) during iteration 2, which is represented as the addition operations in the second row from the top. Note that thread 6 needs to add the old value of XY[4] (x_3+x_4) to the old value XY[6] (x_5+x_6) to generate the new value of XY[6] ($x_3+x_4+x_5+x_6$). However, if thread 4 stores its addition result for the iteration ($x_1+x_2+x_3+x_4$) into XY[4] too early, thread 6 could end up using the new value as its input and store ($x_1+x_2+x_3+x_4+x_5+x_6$) into XY[6]. Since x_1+x_2 will be added again to XY[6] by thread 6 in the third iteration, the final answer in XY[6] will become ($2x_1+2x_2+x_3+x_4+x_5+x_6$), which is obviously incorrect. On the other hand, if thread 6 happens to read the old value in XY[4] before thread 4 overwrites it during iteration 2, the results will be correct. That is, the execution result of the code may or may not be correct, depending on the timing of thread execution,

and the execution results can vary from run to run. Such lack of reproducibility can make debugging a nightmare.

The race condition is overcome with the temporary variable used in line 13 and the `__syncthreads( )` barrier in line 14. In line 13, all active threads first perform addition and write into their private temp variables. Therefore none of the old values in XY locations will be overwritten. The barrier `__syncthread( )` in line 14 ensures that all active threads have completed their read of the old XY values before any of them can move forward and perform a write. Thus it is safe for the statement in line 16 to overwrite the XY locations.

The reason why an updated XY position may be used by another active thread is that the Kogge-Stone approach reuses the partial sums across reduction trees to reduce the computational complexity. We will study this point further in Section 11.3. The reader might wonder why the reduction tree kernels in Chapter 10, Reduction and Minimizing Divergence, did not need to use temporary variables and an extra `__syncthreads()`. The answer is that there is no race condition caused by a write-after-read hazard in these reduction kernels. This is because the elements written to by the active threads in an iteration are not read by any of the other active threads during the same iteration. This should be apparent by inspecting Fig. 10.7 and 10.8. For example, in Fig. 10.8, each active thread takes its inputs from its own position `(input [threadIdx.x])` and a position that is of stride distance to the right `(input [threadIdx.x+stride])`. None of the stride distance positions are updated by any active threads during any given iteration. Therefore all active threads will always be able to read the old value of their respective `input[threadIdx.x]`. Since the execution within a thread is always sequential, each thread will always be able to read the old value in `input[threadIdx.x]` before writing the new value into the position. The reader should verify that the same property holds in Fig. 10.7.

If we want to avoid having a second barrier synchronization on every iteration, another way to overcome the race condition is to use separate arrays for input and output. If separate arrays are used, the location that is being written to is different from the location that is being read from, so there is no longer any potential write-after-read race condition. This approach would require having two shared memory buffers instead of one. In the beginning, we load from the global memory to the first buffer. In the first iteration we read from the first buffer and write to the second buffer. After the iteration is over, the second buffer has the most up-to-date results, and the results in the first buffer are no longer needed. Hence in the second iteration we read from the second buffer and write to the first buffer. Following the same reasoning, in the third iteration we read from the first buffer and write to the second buffer. We continue alternating input/output buffers until the iterations complete. This optimization is called *double-buffering*. Double-buffering is commonly used in parallel programming as a way to overcome write-after-read race conditions. We leave the implementation of this optimization as an exercise for the reader.

Furthermore, as is shown in Fig. 11.2, the actions on the smaller positions of XY end earlier than those on 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 arise only in the first warp. The detailed analysis is left as an exercise for the reader.

Although we have shown only an inclusive scan kernel, 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 the identity value. This is illustrated in Fig. 11.4. 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.

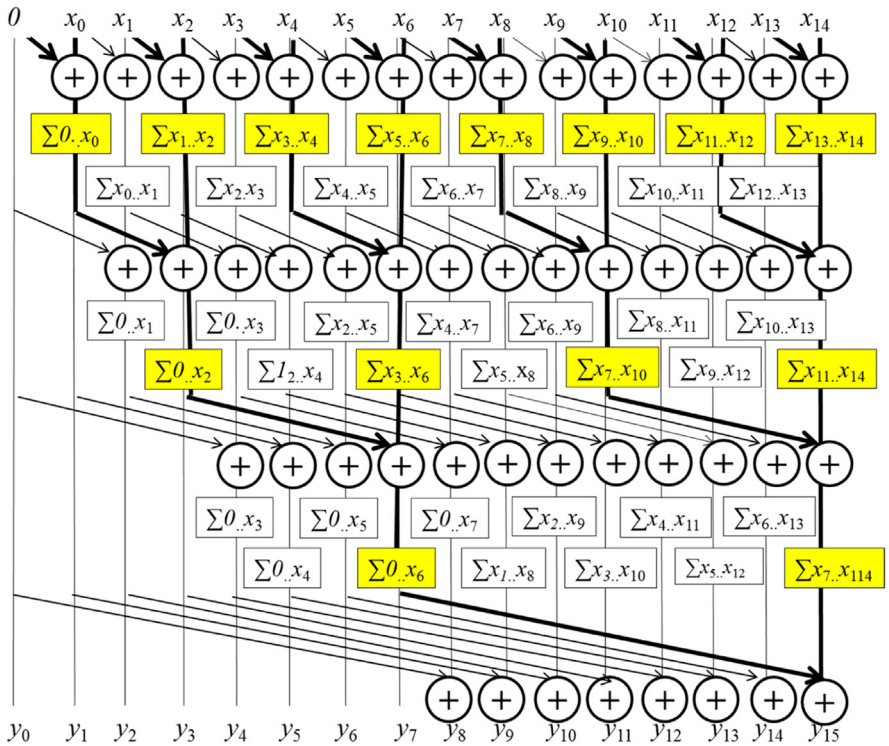


FIGURE 11.4

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

We can now easily convert the kernel in Fig. 11.3 into an exclusive scan kernel. The only modification we need to make is to load 0 into XY[0] and `X[i-1]` into `XY[threadIdx.x]`, as shown in the following code:

```
if (i < N && threadIdx.x != 0) {
    XY[threadIdx.x] = X[i-1];
} else {
    XY[threadIdx.x] = 0.0f;
}
```

By substituting these four lines of code for lines 04–08 of Fig. 11.3, we convert the inclusive scan kernel into an exclusive scan kernel. We leave the work to finish the exclusive scan kernel as an exercise for the reader.

11.3 Speed and work efficiency consideration

One important consideration in analyzing parallel algorithms is *work efficiency*. The work efficiency of an algorithm refers to the extent to which the work that is performed by the algorithm is close to the minimum amount of work needed for the computation. For example, the minimum number of additions required by the scan operation is $N-1$ additions, or $O(N)$, which is the number of additions that the sequential algorithm performs. However, as we saw in the beginning of Section 11.2, the naïve parallel algorithm performs $N^*(N-1)/2$ additions, or $O(N^2)$, which is substantially larger than the sequential algorithm. For this reason, the naïve parallel algorithm is not work efficient.

We now analyze the work efficiency of the Kogge-Stone kernel in Fig. 11.3, focusing on the work of a single thread block. All threads 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 Fig. 8.1) for the algorithm as

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

The first part of each term is independent of stride, and 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

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

The good news is that the computational complexity of the Kogge-Stone approach is $O(N^*\log_2(N))$, better than the $O(N^2)$ complexity of a naïve approach that performs complete reduction trees for all output elements. The bad news is that the Kogge-Stone algorithm is still not as work efficient as the sequential

algorithm. Even for modest-sized sections, the kernel in Fig. 11.3 does much more work than the sequential algorithm. In the case of 512 elements, the kernel does approximately eight times more work than the sequential code. The ratio will increase as N becomes larger.

Although the Kogge-Stone algorithm performs more computations than the sequential algorithm, it does so in fewer steps because of parallel execution. 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 reduction in the number of steps of the kernel code over the sequential code would be approximately $N/\log_2(N)$. For N=512, the reduction in the number of steps 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 $N^*\log2(N) - (N - 1)$. This is because we are using N threads. While many of the threads stop participating in the execution of the for-loop, some of them still consume execution resources until the entire warp completes execution. Realistically, the amount of execution resources consumed by the Kogge-Stone is closer to $N^*\log2(N)$.

We will use the concept of computation steps as an approximate indicator for comparing between scan algorithms. The sequential scan should take approximately N steps to process N input elements. For example, the sequential scan should take approximately 1024 steps to process 1024 input elements. With P execution units in the CUDA device, we can expect the Kogge-Stone kernel to execute for $(N^*\log2(N))/P$ steps. If P is equal to N, that is, if we have enough execution units to process all input element in parallel, then we need $\log2(N)$ steps, as we saw earlier. However, P could be smaller than N. 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 steps. In this case, we expect to achieve a 1024/320=3.2 × reduction in the number of steps.

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. If the hardware does not have enough resources (i.e., if P is small), the parallel algorithm could end up needing more steps than the sequential algorithm. Hence the parallel algorithm will be slower. Second, all the extra work consumes 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 resources. It is typically used to calculate the scan result for a section with a modest number of elements, such as 512 or 1024. This, of course, assumes that GPUs can provide sufficient hardware resources and use the additional parallelism to tolerate latencies. As we have seen, its execution has a 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.

11.4 Parallel scan with the Brent-Kung algorithm

While the Kogge-Stone kernel in Fig. 11.3 is conceptually simple, its work efficiency is quite low for some practical applications. Just by inspecting Figs. 11.2 and 11.4, we can see that there are potential opportunities for further sharing of some intermediate results. However, to allow more sharing across multiple threads, we need to strategically calculate the intermediate results and distribute them to different threads, which may require additional computation steps.

As we know, the fastest parallel way to produce sum values for a set of values is a reduction tree. With sufficient execution units, a reduction tree can generate the sum for N values in $\log_2(N)$ time units. The tree can also generate several sub-sums that can be used in the calculation of some of the scan output values. This observation was used as a basis for the Kogge-Stone adder design and also forms the basis of the Brent-Kung adder design (Brent & Kung, 1979). The Brent-Kung adder design can also be used to implement a parallel scan algorithm with better work efficiency.

Fig. 11.5 illustrates the steps for a parallel inclusive scan algorithm based on the Brent-Kung adder design. In the top half of Fig. 11.5, we produce the sum of all 16 elements in four 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, and 15 in Fig. 11.5, 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 $(N/2)+(N/4)+\ldots+2+1=N-1$ operations for this reduction phase.

The second part of the algorithm is to use a reverse tree to distribute the partial sums to the positions that can use them to complete the result of those positions. The

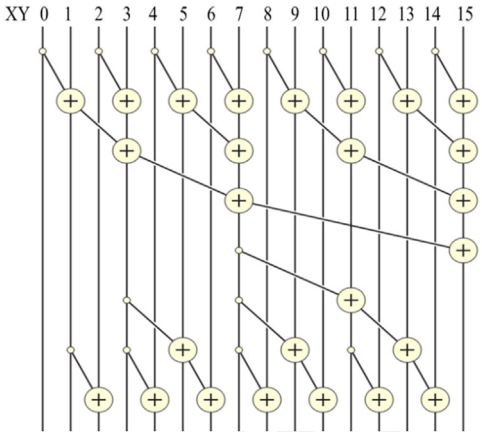


FIGURE 11.5

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

distribution of partial sums is illustrated in the bottom half of Fig. 11.5. To understand the design of the reverse tree, we should first analyze the needs for additional values to complete the scan output at each position of XY. It should be apparent from an inspection of Fig. 11.5 that the additions in the reduction tree always accumulate input elements in a continuous range. Therefore we know that the values that have been accumulated into each position of XY can always be expressed as a range of input elements $x_i...x_j$, where x_i is the starting position and x_j is the ending position (inclusive).

Fig. 11.6 shows the state of each position (column), including both the values already accumulated into the position and the need for additional input element values at each level (row) of the reverse tree. The state of each position initially and after each level of additions in the reverse tree is expressed as the input elements, in the form $x_i...x_j$, that have already been accounted for in the position. For example, $x_8...x_{11}$ in row Initial and column 11 indicates that the values of x_8, x_9, x_{10}, and x_{11} have been accumulated into XY[11] before the reverse tree begins (right after the reduction phase shown in the bottom portion of Fig. 11.5). At the end of the reduction tree phase, we have quite a few positions that are complete with the final scan values. In our example, XY[0], XY[1], XY[3], XY[7], and XY[15] are all complete with their final answers.

The need for additional input element values is indicated by the shade of each cell in Fig. 11.6: White means that the position needs to accumulate partial sums from three other positions, light gray means 2, dark gray means 1, and black means 0. For example, initially, XY[14] is marked white because it has only the value of x_{14} at the end of the reduction tree phase and needs to accumulate the partial sums from XY[7] ($x_0...x_7$), XY[11] ($x_8...x_{11}$), and XY[13] ($x_{12}...x_{13}$) to complete its final scan value ($x_0...x_{14}$). The reader should verify that because of the structure of the reduction tree, the XY positions for an input of size N elements will never need the accumulation from more than $\log_2(N) - 1$ partial sums from other XY positions. Furthermore, these partial sum positions will always be 1, 2, 4, ... (powers of 2) away from each other. In our example, XY[14] needs $\log_2(16) - 1=3$ partial sums from positions that are 1 (between XY[14] and XY[13]), 2 (between XY[13] and XY[11]), and 4 (between XY[11] and XY[7]).

To organize our second half of the addition operations, we will first show all the operations that need partial sums from four positions away, then two positions

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

FIGURE 11.6

Progression of values in XY after each level of additions in the reverse tree.

away, then one position away. During the first level of the reverse tree, we add XY[7] to XY[11], which brings XY[11] to the final answer. In Fig. 11.6, position 11 is the only one that advances to its final answer. During the second level, we complete XY[5], XY[9], and XY[13], which can be completed with the partial sums that are two positions away: XY[3], XY[7], and XY[11], respectively. Finally, during the third level, we complete all even positions XY[2], XY[4], XY[6], XY[8], XY[10], and XY[12] by accumulating the partial sums that are one position away (immediate left neighbor of each position).

We are now ready to implement the Brent-Kung approach to scan. 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 similar to the reduction in Fig. 10.6. There are only two differences. The first difference is that we accumulate the sum value toward the highest position, which is `XY[blockDim.x-1]`, rather than XY[0]. This is because the final result of the highest position is the total sum. For this reason, each active thread reaches for a partial sum to its left by subtracting the stride value from its index. The second difference is that we want the active threads to have a thread index of the form $2^n - 1$ rather than 2^n. This is why we add 1 to the threadIdx.x before the modulo (%) operation when we select the threads for performing addition in each iteration.

One drawback of this style of reduction is that it has significant control divergence problems. As we saw in Chapter 10, Reduction and Minimizing Divergence, a better way is to use a decreasing number of contiguous threads to perform the additions as the loop advances. Unfortunately, the technique we used to reduce divergence in Fig. 10.8 cannot be used in the scan reduction tree phase, since it does not generate the needed partial sum values in the intermediate XY positions. Therefore we resort to a more sophisticated thread index to data index mapping that maps a continuous section of threads to a series of data positions that are of stride distance apart. The following code does so by mapping a continuous section of threads to the XY positions whose indices are of the form $k*2^n - 1$:

```
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 such a complex index calculation in each iteration of the for-loop, a contiguous set of threads starting from thread 0 will be used in every iteration to avoid control divergence within warps. In our small example in Fig. 11.5, there are 16 threads in the block. In the first iteration, stride is equal to 1. The first eight consecutive threads in the block will satisfy the if-condition. The XY index values calculated for these threads will be 1, 3, 5, 7, 9, 11, 13, and 15. These threads will perform the first row of additions in Fig. 11.5. In the second iteration, stride is equal to 2. Only the first four threads in the block will satisfy the if-condition. The index values calculated for these threads will be 3, 7, 11, and 15. These threads will perform the second row of additions in Fig. 11.5. Note that since each iteration will always be using consecutive threads, the control divergence problem does not arise until the number of active threads drops below the warp size.

The reverse tree is a little more complex to implement. We see that the stride value decreases from `SECTION_SIZE`/4 to 1. In each iteration we need to "push" the value of XY elements from positions that are multiples of twice the stride value minus 1 to the right by stride positions. For example, in Fig. 11.5 the stride value decreases from 4 (2^2) down to 1. In the first iteration we would like to push (add) the value of XY[7] into XY[11], where 7 is $2*2^2 - 1$ and distance (stride) is 2^2. Note that only thread 0 will be used for this iteration, since the index calculated for other threads will be too large to satisfy the if-condition. In the second iteration we push the values of XY[3], XY[7], and XY[11] to XY[5], XY[9], and XY[13], respectively. The indices 3, 7, and 11 are $1*2*2^1 - 1$, $2*2*2^1 - 1$, and $3*2*2^1 - 1$, respectively. The destination positions are 2^1 positions away from the source positions. Finally, in the third iteration we push the values at all the odd positions to their even position right-hand neighbor (stride=2°).

On the basis of the discussions above, the reverse tree 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 `XY[index+stride] +=XY[index]` statement reflects the push from the threads' mapped location into a higher position that is stride away.

The final kernel code for a Brent-Kung parallel scan is shown in Fig. 11.7. 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. Therefore we could simply launch a kernel with `SECTION_SIZE`/2 threads in a block. Since we

```
01    __global__ void Brent_Kung_scan_kernel(float *X, float *Y, unsigned int N) {
02        __shared__ float XY[SECTION_SIZE];
03      unsigned int i = 2*blockIdx.x*blockDim.x + threadIdx.x;
04      if(i < N) XY[threadIdx.x] = X[i];
05      if(i + blockDim.x < N) XY[threadIdx.x + blockDim.x] = X[i + blockDim.x];
06      for(unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
07            __syncthreads();
08          unsigned int index = (threadIdx.x + 1)*2*stride - 1;
09          if(index < SECTION_SIZE) {
10              XY[index] += XY[index - stride];
11          }
12      }
13      for (int stride = SECTION_SIZE/4; stride > 0; stride /= 2) {
14            __syncthreads();
15          unsigned int index = (threadIdx.x + 1)*stride*2 - 1;
16          if(index + stride < SECTION_SIZE) {
17              XY[index + stride] += XY[index];
18          }
19      }
20      __syncthreads();
21      if (i < N) Y[i] = XY[threadIdx.x];
22      if (i + blockDim.x < N) Y[i + blockDim.x] = XY[threadIdx.x + blockDim.x];
23  }
```

FIGURE 11.7

A Brent-Kung kernel for inclusive (segmented) scan.

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 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 Harris et al., 2007, for an interesting natively exclusive scan kernel that is based on a different way of designing the reverse tree phase of the scan kernel.

We now turn our attention to the analysis of the number of operations in the reverse 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 $(2 - 1) + (4 - 1) + \ldots + (N/4 - 1) + (N/2 - 1)$, which is $N - 1 - \log_2(N)$. Therefore the total number of operations in the parallel scan, including both the reduction tree ($N - 1$ operations) and the reverse tree ($N - 1 - \log_2(N)$ operations) phases, is $2^*N - 2 - \log_2(N)$. Note that the total number of operations is now $O(N)$, as opposed to $O(N^*\log_2 N)$ for the Kogge-Stone algorithm.

The advantage of the Brent-Kung algorithm over the Kogge-Stone algorithm is quite clear. 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 the Kogge-Stone algorithm, 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, some of the inactive threads may still consume execution resources in a CUDA device because they are bound to other active threads by SIMD. This makes the advantage in work efficiency of Brent-Kung over Kogge-Stone less drastic in a CUDA device.

The main disadvantage of Brent-Kung over Kogge-Stone is its potentially longer execution time despite its higher work efficiency. With infinite execution resources, Brent-Kung should take about twice as long as Kogge-Stone, owing to the need for additional steps to perform the reverse tree phase. However, the speed comparison can be quite different when we have limited execution resources. Using the example in Section 11.3, if we process 1024 input elements with 32 execution units, the Brent-Kung kernel is expected to take approximately (2*1024 − 2 − 10)/32=63.6 steps. The reader should verify that with control divergence there will be about five more steps when the total number of active threads drops below 32 in each phase. This results in a speedup of 1024/73.6=14 over sequential execution. This is in comparison with the 320 time units and speedup of 3.2 for Kogge-Stone. The comparison would be to Kogge-Stone's advantage, of course, when there are many more execution resources and/or when the latencies are much longer.

11.5 Coarsening for even more work efficiency

The overhead of parallelizing scan across multiple threads is like reduction in that it includes the hardware underutilization and synchronization overhead of the tree execution pattern. However, scan has an additional parallelization overhead, and that is reduced work efficiency. As we have seen, parallel scan is less work efficient than sequential scan. This lower work efficiency is an acceptable price to pay to parallelize if the threads were actually to run in parallel. However, if the hardware were to serialize them, we would be better off serializing them ourselves via thread coarsening to improve work efficiency.

We can design a parallel segmented scan algorithm that achieves even better work efficiency by adding a phase of fully independent sequential scans on subsections of the input. Each thread block receives a section of the input that is larger than the original section by the coarsening factor. At the beginning of the algorithm we partition the block's section of the input into multiple contiguous subsections: one for each thread. The number of subsections is the same as the number of threads in the thread block.

The coarsened scan is divided into three phases as shown in Fig. 11.8. During the first phase, we have each thread perform a sequential scan on its contiguous subsection. For example, in Fig. 11.8 we assume that there are four threads in a block. We partition the 16-element input section into four subsections with four elements each. 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), and so on.

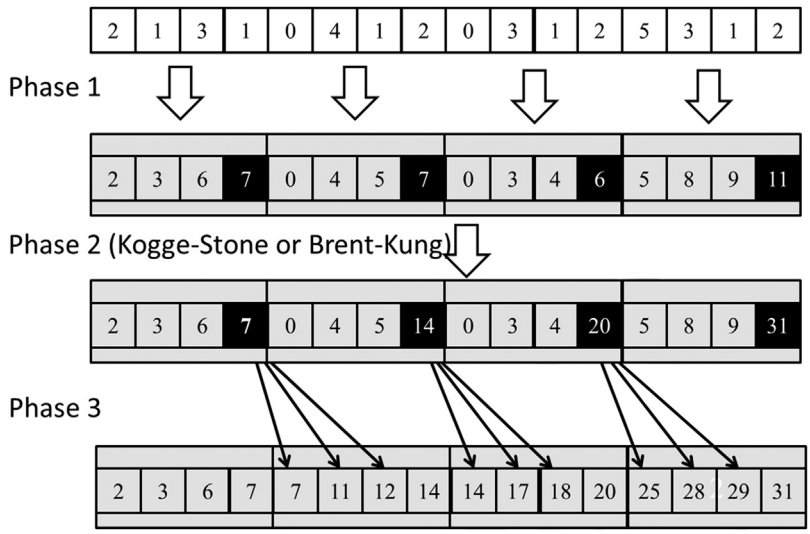


FIGURE 11.8

A three-phase parallel scan for higher work efficiency.

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 would be accessing input element 4, and so on. Therefore we improve memory coalescing by using shared memory to absorb the memory accesses that cannot be coalesced, as was mentioned in Chapter 6, Performance Considerations. That is, we transfer data between shared memory and global memory in a coalesced manner and perform the memory accesses with the unfavorable access pattern in shared memory. At the beginning of the first 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 Fig. 11.8 we have all threads collaborate and load four elements in a coalesced manner: Thread 0 loads element 0, thread 1 loads element 1, and so on. All threads then move to load the next four elements: Thread 0 loads element 4, thread 1 loads element 5, and so on.

Once all input elements are in the shared memory, the threads access their own subsection from the shared memory and perform a sequential scan on it. This is shown as Phase 1 in Fig. 11.8. Note that at the end of Phase 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.

During the second phase, all threads in each block collaborate and perform a scan operation on a logical array that consists of the last element of each section.

This can be done with a Kogge-Stone or Brent-Kung algorithm, since there are only a modest number of elements (number of threads in a block). Note that the thread-to-element mappings need to be slightly modified from those in Figs. 11.3 and 11.7, since the elements that need to be scanned are of stride (four elements in Fig. 11.8) distance away from each other.

In the third phase, each thread adds the new value of the last element of its predecessor's section to its elements. The last elements of each subsection do not need to be updated during this phase. For example, in Fig. 11.8, thread 1 adds the value 7 to elements (0, 4, 5) in its section to produce (7, 11, 12). The last element of the section is already the correct value 14 and does not need to be updated.

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 a block but rather by the size of the shared memory; all elements of the section need to fit into the shared memory.

The major advantage of thread coarsening for scan is its efficient 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 $N - T$ 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 $(N - T + T^*\log_2 T + N - T)/P$ steps. For example, if we use 64 threads and 32 execution units to process 1024 elements, the algorithm should take approximately $(1024 - 64 + 64^*6 + 1024 - 64)/32 = 72$ steps. We leave the implementation of the coarsened scan kernel as an exercise for the reader.

11.6 Segmented 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 kernels that we have presented so far perform local block-wide scans on sections of the input, but we still need a way to consolidate the results from different sections. To do so, we can use a hierarchical scan approach, as illustrated in Fig. 11.9.

For a large dataset we first partition the input into sections so that each of them can fit into the shared memory of a streaming multiprocessor and be processed by a single block. Assume that we call one of the kernels in Figs. 11.3 and 11.7 on a large input dataset. At the end of the grid execution, the Y array will contain the scan results for individual sections, called *scan blocks* in Fig. 11.9. Each element 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 call another kernel that adds the sum of all elements in preceding scan blocks to each element of a scan block.

Fig. 11.10 shows a small example of the hierarchical scan approach of Fig. 11.9. In this example there are 16 input elements that are divided into four

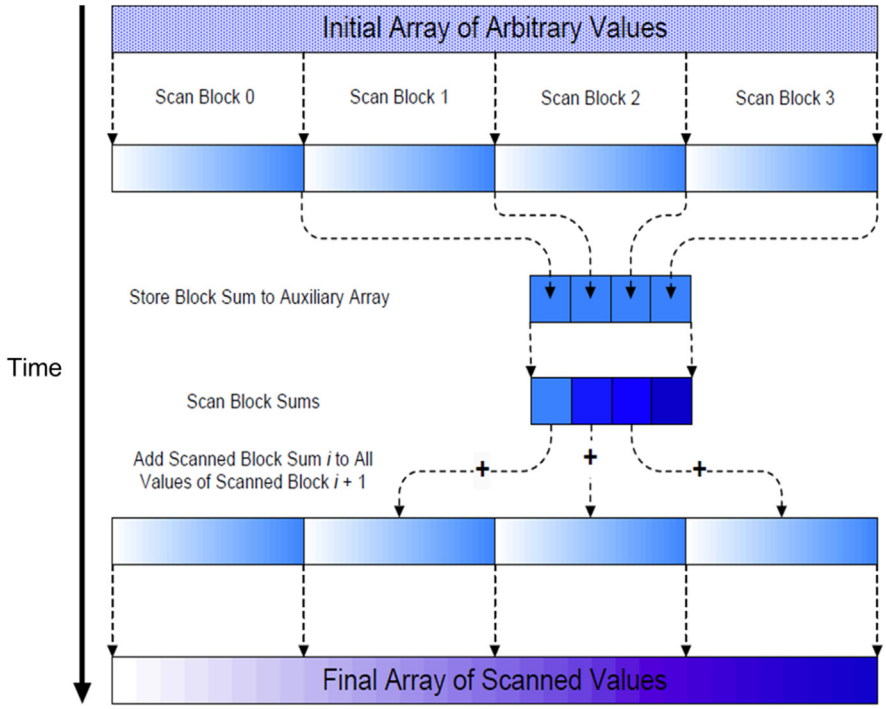


FIGURE 11.9

A hierarchical scan for arbitrary length inputs.

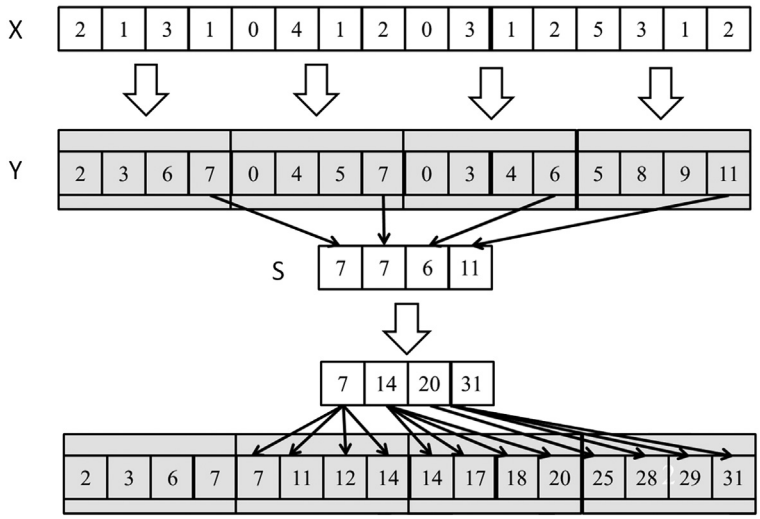


FIGURE 11.10

An example of hierarchical scan.

scan blocks. We can use the Kogge-Stone kernel, the Brent-Kung kernel, or a coarsened kernel to process the individual scan blocks. The kernel treats the four scan blocks as independent input datasets. 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, which is 0, 4, 5, 7. Note that these results do not contain the contributions from any of the elements in scan block 0. To produce the final result for this scan block, the sum of all elements in scan block 0, that is, 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, which is 0, 3, 4, 6. To produce the final results for this scan block, the sum of all elements in both scan block 0 and scan block 1, that is, 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 Fig. 11.10. This brings us to the second step of the segmented scan algorithm in Fig. 11.9, 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 Fig. 11.10, where the last scan output elements of all scan blocks are collected into a new array S. While the second step of Fig. 11.10 is logically the same as the second step of Fig. 11.8, the main difference is that Fig. 11.10 involves threads from different thread blocks. As a result, the last element of each section needs to be collected (written) into a global memory array so that they can be visible across thread blocks.

Gathering the last result of each scan block 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 the array 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 the accumulated sum from the beginning location X[0] to the end location of each scan block. That is, the value in S[0]=7 is the accumulated sum from X[0] to the end of scan block 0, which is X[3]. The value in S[1]=14 is the accumulated sum from X[0] to the end of scan block 1, which is X[7]. Therefore the output values in the S array give the scan results at "strategic" locations of the original scan problem. In other words, in Fig. 11.10 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], respectively. 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 segmented scan algorithm in Fig. 11.10. The second-level scan output values are added to the values of their corresponding scan blocks. For example, in Fig. 11.10, 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 principle behind the segmented scan algorithm is quite similar to the principle behind 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 also based on innovative hardware adder designs.

We can implement the segmented 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 N/`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 one of the three parallel scan kernels configured with a single thread block, which takes S as input and writes S as output without producing any partial sums.

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:

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

In other words, the threads in a block add the sum of all previous scan blocks 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.

11.7 Single-pass scan for memory access efficiency

In the segmented scan mentioned in Section 11.6, the partially scanned results (scan blocks) are stored into global memory before launching the global scan

kernel, and then reloaded back from the global memory by the third kernel. The time for performing these extra memory stores and loads is not overlapped with the computation in the subsequent kernels and can significantly affect the speed of the segmented scan algorithms. To avoid such a negative impact, multiple techniques have been proposed (Dotsenko et al., 2008; Merrill & Garland, 2016; Yan et al., 2013). 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 20, Programming a Heterogeneous Computing Cluster), or domino-style scan algorithm, refers to a segmented scan algorithm in which partial sum data is passed in one direction through the global memory between adjacent thread blocks in the same grid. Stream-based scan builds on a key observation that the global scan step (the middle part of Fig. 11.9) can be performed in a domino fashion and does not truly require a grid-wide synchronization. For example, in Fig. 11.10, scan block 0 can pass its partial sum value 7 to scan block 1 and 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 by adding 7 to all partial scan values in its scan block. This process continues for all thread blocks.

To implement a domino-style scan algorithm, one can write a single kernel to perform all three steps of the segmented scan algorithm in Fig. 11.9. Thread block i first performs a scan on its scan block, using one of the three parallel algorithms we presented in Sections 11.2 through 11.5. 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 adds the value to its local sum and passes the cumulative sum value to its right neighbor block, i+1. It then moves on to add the sum value received from block $i-1$ to all partial scan values to produce all the output values of the scan block.

During the first phase of the kernel, all blocks can execute in parallel. They will be serialized during the data-passing 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 parallelism among blocks during the third phase.

To make this domino-style scan work, adjacent (block) synchronization is needed (Yan et al., 2013). 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, as in a producer-consumer chain. On the producer side (scan block $i-1$), a flag is set to a particular value after the partial sum has been stored to memory, while on the consumer side (scan block i), the flag is checked to see whether it is that particular value before the passed partial sum is loaded. As has been 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 by 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 executed by only one leader thread in each block (e.g., thread with index 0). The rest of threads will wait at `__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 shared memory variable `previous_sum`. It adds the `previous_sum` 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 `scan_value[bid+1]` value arrives to the global memory before the flag is set with `atomicAdd()`.

Although it may appear that the atomic operations on the flags array and the accesses to the `scan_value` array incur global memory traffic, these operations are mostly performed in the second-level caches of recent GPU architectures (Chapter 9, Parallel Histogram). Any such 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 segmented scan algorithm in Section 11.5, the stores to and loads from the S array in the global memory are in a separate kernel and cannot be overlapped with phase 1 or phase 3.

There is one subtle issue with domino-style algorithms. In GPUs, thread blocks may not always be scheduled linearly according to their blockIdx values, which means that 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 order assumed by the adjacent synchronization code and cause performance degradation or even a deadlock. 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.

There are multiple techniques to resolve this issue (Gupta et al., 2012; Yan et al., 2013). Here, we discuss only one particular method, dynamic block index assignment, and leave the rest as a reference for readers. Dynamic block index assignment decouples the assignment of the thread block index from the built-in `blockIdx.x`. In the single-pass scan, the value of the bid variable of each block is no longer tied to the value of `blockIdx.x`. Instead, it is determined by using the following code at the beginning of the kernel:

```
__shared__ unsigned int bid_s;
if (threadIdx.x == 0) {
    bid_s = atomicAdd(blockCounter, 1);
}
__syncthreads();
unsigned int bid = bid_s;
```

The leader thread atomically increments a global counter variable pointed to by `blockCounter`. 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 into a shared memory variable `bid_s` so that it is accessible by all threads of the block after `__syncthreads()`. This guarantees that all scan blocks are scheduled linearly and prevents a potential deadlock. In other words, if a block obtains a bid value of i, then it is guaranteed that a block with value $i - 1$ has been scheduled because it has executed the atomic operation.

11.8 Summary

In this chapter we studied parallel scan, also known as prefix sum, 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 computation based on mathematical recurrence into parallel computation, which helps to reduce sequential bottlenecks in many applications. We show that a simple sequential scan algorithm performs only $N - 1$, or $O(N)$, additions for an input of N elements.

We first introduced a parallel Kogge-Stone segmented scan algorithm that is fast and conceptually simple but not work-efficient. The algorithm performs $O(N^*\log_2 N)$ operations, which is more than its sequential counterpart. As the size of the dataset increases, the number of execution units that are needed for a parallel algorithm to break even with the simple sequential algorithm also increases. Therefore Kogge-Stone scan algorithms are typically used to process modest-sized scan blocks in processors with abundant execution resources.

We then presented a parallel Brent-Kung segmented scan algorithm that is conceptually more complicated. Using a reduction tree phase and a reverse tree phase, the algorithm performs only $2^*N - 3$, or $O(N)$, additions no matter how large the input datasets 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 a data-scalable algorithm. Although the Brent-Kung algorithm has better work efficiency than the Kogge-Stone algorithm, it requires more steps to complete. Therefore in a system with enough execution resources, the Kogge-Stone algorithm is expected to have better performance despite being less work efficient.

We also applied thread coarsening to mitigate the hardware underutilization and synchronization overhead of parallel scan and to improve its work efficiency. Thread coarsening was applied by having each thread in the block perform a work-efficient sequential scan on its own subsection of input elements before the threads collaborate to perform the less work-efficient block-wide parallel scan to produce the entire block's section.

We presented a hierarchical scan approach to extend the parallel scan algorithms to handle input sets of arbitrary sizes. Unfortunately, a straightforward, three-kernel implementation of the segmented scan algorithm incurs redundant global memory accesses whose latencies are not overlapped with computation. For this reason we also presented a domino-style hierarchical scan algorithm to enable a single-pass, single-kernel implementation and improve the global memory access efficiency of the hierarchical scan algorithm. However, this approach requires a carefully designed adjacent block synchronization mechanism using atomic operations, thread memory fence, and barrier synchronization. Special care also must be taken to prevent deadlocks by using dynamic block index assignment.

There are further optimization opportunities for even higher performance implementations using, for example, warp-level shuffle operations. In general, implementing and optimizing parallel scan algorithms on GPUs are complex processes, and the average user is more likely to use a parallel scan library for GPUs such as Thrust (Bell and Hoberock, 2012) than to implement their own scan kernels from scratch. Nevertheless, parallel scan is an important parallel pattern, and it offers an interesting and relevant case study of the tradeoffs that go into optimizing parallel patterns.

Exercises

1. 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.
2. Modify the Kogge-Stone parallel scan kernel in Fig. 11.3 to use double-buffering instead of a second call to `__syncthreads()` to overcome the write-after-read race condition.

3. Analyze the Kogge-Stone parallel scan kernel in Fig. 11.3. Show that control divergence occurs only 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.
4. For the Kogge-Stone scan kernel based on reduction trees, assume that we have 2048 elements. Which of the following gives the closest approximation of how many add operations will be performed?
5. Consider the following array: [4 6 7 1 2 8 5 2]. Perform a parallel inclusive prefix scan on the array, using the Brent-Kung algorithm. Report the intermediate states of the array after each step.
6. 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?
7. Use the algorithm in Fig. 11.4 to complete an exclusive scan kernel.
8. Complete the host code and all three kernels for the segmented parallel scan algorithm in Fig. 11.9.

References

Bell, N., Hoberock, J., 2012. "Thrust: A productivity-oriented library for CUDA." GPU computing gems Jade edition. Morgan Kaufmann, 359–371.

Brent, R. P., & Kung, H. T., 1979. A regular layout for parallel adders, Technical Report, Computer Science Department, Carnegie-Mellon University.

Dotsenko, Y., Govindaraju, N.K., Sloan, P.P., Boyd, C., Manferdelli, J., 2008. Fast scan algorithms on graphics processors. Proc. 22nd Annu. Int. Conf. Supercomputing, 205–213.

Gupta, K., Stuart, J.A., Owens, J.D., 2012. A study of persistent threads style GPU programming for GPGPU workloads. Innovative Parallel Comput. (InPar) 1–14. IEEE.

Harris, M., Sengupta, S., & Owens, J. D. (2007). Parallel prefix sum with CUDA. GPU Gems 3. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf.

Kogge, P., Stone, H., 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Computers C-22, 783–791.

Merrill, D., & Garland, M. (2016, March). Single-pass parallel prefix scan with decoupled look-back. Technical Report NVR2016-001, NVIDIA Research.

Yan, S., Long, G., Zhang, Y., 2013. StreamScan: Fast scan algorithms for GPUs without global barrier synchronization, PPoPP. ACM SIGPLAN Not. 48 (8), 229–238.

CHAPTER

Merge
An introduction to dynamic input data identification

12

With special contributions from Li-Wen Chang and Jie Lv

Chapter Outline

Our next parallel pattern is an ordered merge operation, which takes two sorted lists and generates a combined sorted list. Ordered merge operations can be used as a building block of sorting algorithms, as we will see in Chapter 13, Sorting. Ordered merge operations also form the basis of modern map-reduce frameworks. This chapter presents a parallel ordered merge algorithm in which the input data for each thread is dynamically determined. The dynamic nature of the data access makes it challenging to exploit locality and tiling techniques for improved memory access efficiency and performance. The principles behind dynamic input data identification are also relevant to many other important computations, such as set intersection and set union. We present increasingly sophisticated buffer management schemes for achieving increasing levels of memory access efficiency for order merged and other operations that determine their input data dynamically.

12.1 Background

An ordered merge function takes two sorted lists A and B and merges them into a single sorted list C. For this chapter we assume that the sorted lists are stored in

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00025-2

arrays. We further assume that each element in such an array has a key. An order relation denoted by $\leq$ is defined on the keys. For example, the keys may be simply integer values, and $\leq$ may be defined as the conventional *less than or equal to* relation between these integer values. In the simplest case, the elements consist of just keys.

Suppose that we have two elements e_1 and e_2 whose keys are k_1 and k_2, respectively. In a sorted list based on the relation $\leq$, if e_1 appears before e_2, then $k_1 \leq k_2$. A merge function based on an ordering relation R takes two sorted input arrays A and B having m and n elements, respectively, where m and n do not have be to equal. Both array A and array B are sorted on the basis of the ordering relation R. The function produces an output sorted array C having m + n elements. Array C consists of all the input elements from arrays A and B and is sorted by the ordering relation R.

Fig. 12.1 shows the operation of a simple merge function based on the conventional numerical ordering relation. Array A has five elements (m=5), and array B has four elements (n=4). The merge function generates array C with all its 9 elements (m + n) from A and B. These elements must be sorted. The arrows in Fig. 12.1 show how elements of A and B should be placed into C to complete the merge operation. Whenever the numerical values are equal between an element of A and an element of B, the element of A should appear first in the output list C. This requirement ensures the stability of the ordered merge operation.

In general, an ordering operation is stable if elements with equal key values are placed in the same order in the output as the order in which they appear in the input. The example in Fig. 12.1 demonstrates stability both with and across the input lists of the merge operation. For example, the two elements whose values are 10 are copied from B into C while maintaining their original order. This illustrates stability within an input list of the merge operation. For another example the A element whose value is 7 goes into C before the B element of the same value. This illustrates stability across input lists of the merge operation. The stability property allows the ordering operation to preserve previous orderings that are not captured by the key that is used in the current ordering operation. For example, the lists A and B might have been previously sorted according to a different key before being sorted by the current key to be used for merging.

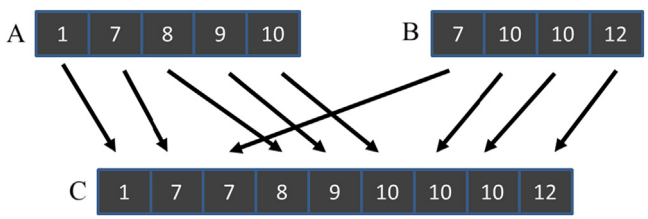


FIGURE 12.1

Example of a merge operation.

Maintaining stability in the merge operation allows the merge operation to preserve the work that was done in the previous steps.

The merge operation is the core of merge sort, an important parallelizable sort algorithm. As we will see in Chapter 13, Sorting, a parallel merge sort function divides the input list into multiple sections and distributes them to parallel threads. The threads sort the individual section(s) and then cooperatively merge the sorted sections. Such a divide-and-concur approach allows efficient parallelization of sorting.

In modern map-reduce distributed computing frameworks, such as Hadoop, the computation is distributed to a massive number of compute nodes. The reduce process assembles the result of these compute nodes into the final result. Many applications require that the results be sorted according to an ordering relation. These results are typically assembled by using the merge operation in a reduction tree pattern. As a result, efficient merge operations are critical to the efficiency of these frameworks.

12.2 A sequential merge algorithm

The merge operation can be implemented with a straightforward sequential algorithm. Fig. 12.2 shows a sequential merge function.

The sequential function in Fig. 12.2 consists of two main parts. The first part consists of a while-loop (line 05) that visits the A and B list elements in order. The loop starts with the first elements: A[0] and B[0]. Every iteration fills one

```
01    void merge_sequential(int *A, int m, int *B, int n, int *C) {
02        int i = 0; // Index into A
03        int j = 0; // Index into B
04        int k = 0; // Index into C
05        while ((i < m) && (j < n)) { // Handle start of A[] and B[]
06            if (A[i] <= B[j]) {
07                C[k++] = A[i++];
08            } else {
09                C[k++] = B[j++];
10            }
11        }
12        if (i == m) { // Done with A[], handle remaining B[]
13            while(j < n) {
14                C[k++] = B[j++];
15            }
16        } else { // Done with B[], handle remaining A[]
17            while(i < m) {
18                C[k++] = A[i++];
19            }
20        }
21    }
```

FIGURE 12.2

A sequential merge function.

position in the output array C; either one element of A or one element of B will be selected for the position (lines 06−10). The loop uses i and j to identify the A and B elements that are currently under consideration; i and j are both 0 when the execution first enters the loop. The loop further uses k to identify the current position to be filled in the output list array C. In each iteration, if element A[i] is less than or equal to B[j], the value of A[i] is assigned to C[k]. In this case, the execution increments both i and k before going to the next iteration. Otherwise, the value of B[j] is assigned to C[k]. In this case, the execution increments both j and k before going to the next iteration.

The execution exits the while-loop when it reaches either the end of array A or the end of array B. The execution moves on to the second part, which is on the right Fig. 12.2. If array A is the one that has been completely visited, as indicated by the fact that i is equal to m, then the code copies the remaining elements of array B to the remaining positions of array C (lines 13−15). Otherwise, array B is the one that was completely visited, so the code copies the remaining elements of A to the remaining positions of C (lines 17−19). Note that the if-else construct is unnecessary for correctness. We can simply have the two while-loops (lines 13−15 and 17−19) follow the first while-loop. Only one of the two while-loops will be entered, depending on whether A or B was exhausted by the first while-loop. However, we include the if-else construct to make the code more intuitive for the reader.

We can illustrate the operation of the sequential merge function using the simple example from Fig. 12.1. During the first three (0−2) iterations of the while-loop, A[0], A[1], and B[0] are assigned to C[0], C[1], and C[2], respectively. The execution continues until the end of iteration 5. At this point, list A is completely visited, and the execution exits the while loop. A total of six C positions have been filled by A[0] through A[4] and B[0]. The loop in the true branch of the if-construct is used to copy the remaining B elements, that is, B[1] through B[3], into the remaining C positions.

The sequential merge function visits every input element from both A and B once and writes into each C position once. Its algorithm complexity is O(m + n), and its execution time is linearly proportional to the total number of elements to be merged.

12.3 A parallelization approach

Siebert and Traff (2012) proposed an approach to parallelizing the merge operation. In their approach, each thread first determines the range of output positions (output range) that it is going to produce and uses that output range as the input to a *co-rank function* to identify the corresponding input ranges that will be merged to produce the output range. Once the input and output ranges have been determined, each thread can independently access its two input subarrays and one

output subarray. Such independence allows each thread to perform the sequential merge function on their subarrays to do the merge in parallel. It should be clear that the key to the proposed parallelization approach is the co-rank function. We will now formulate the co-rank function.

Let A and B be two input arrays with m and n elements, respectively. We assume that both input arrays are sorted according to an ordering relation. The index of each array starts from 0. Let C be the sorted output array that is generated by merging A and B. Obviously, C has m + n elements. We can make the following observation:

Observation 1: For any k such that $0 \leq k < m + n$, there is either (case 1) an i such that $0 \leq i < m$ and C[k] receives its value from A[i] or (case 2) a j such that $0 \leq j < n$ and C[k] receives its value from B[j] in the merge process.

Fig. 12.3 shows the two cases of observation 1. In the first case, the C element in question comes from array A. For example, in Fig. 12.3A, C[4] (value 9) receives its values from A[3]. In this case, k=4 and i=3. We can see that the prefix subarray C[0]−C[3] of C[4] (the subarray of four elements that precedes C[4]) is the result of merging the prefix subarray A[0]−A[2] of A[3] (the subarray of three elements that precedes A[3]) and the prefix subarray B[0] of B[1] (the

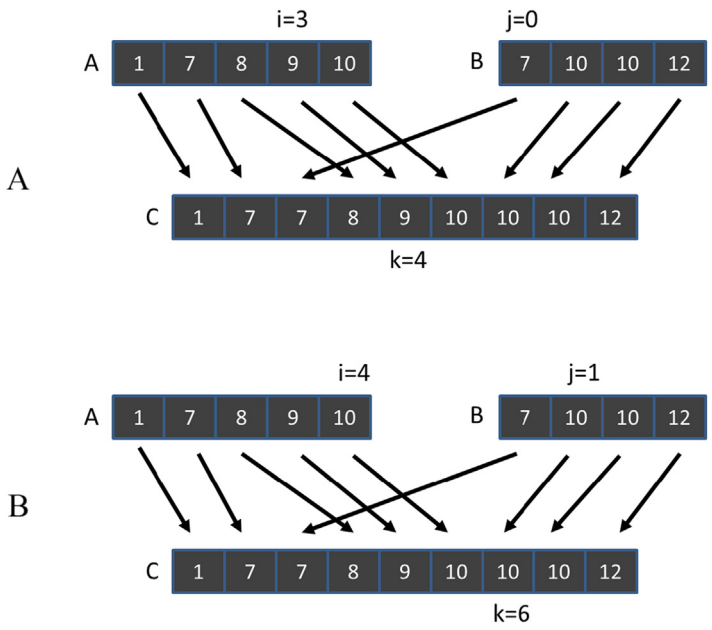


FIGURE 12.3

Examples of observation 1.

subarray of 4 − 3=1 element that precedes B[1]). The general formula is that subarray C[0]−C[k − 1] (k elements) is the result of merging A[0]−A[i − 1] (i elements) and B[0]−B[k − i − 1] (k − i elements).

In the second case, the C element in question comes from array B. For example, in Fig. 12.3B, C[6] receives its value from B[1]. In this case, k=6 and j=1. The prefix subarray C[0]−C[5] of C[6] (the subarray of six elements that precedes C[6]) is the result of merging the prefix subarray A[0]−A[4] (the subarray of five elements that precedes A[5]) and B[0] (the subarray of 1 element that precedes B[1]). The general formula for this case is that subarray C[0]−C[k − 1] (k elements) is the result of merging A[0]−A[k − j − 1] (k − j elements) and B [0]−B[j − 1] (j elements).

In the first case, we find i and derive j as k − i. In the second case, we find j and derive i as k - j. We can take advantage of the symmetry and summarize the two cases into one observation:

Observation 2: For any k such that $0 \leq k < m + n$, we can find i and j such that k=i + j, $0 \leq i < m$ and $0 \leq j < n$ and the subarray C[0]−C[k − 1] is the result of merging subarray A[0]−A[i − 1] and subarray B[0]−B[j − 1].

Siebert and Traff (2012) also proved that i and j, which define the prefix subarrays of A and B that are needed to produce the prefix subarray of C of length k, are unique. For an element C[k] the index k is referred to as its rank. The unique indices i and j are referred to as its co-ranks. For example, in Fig. 12.3A, the rank and co-rank of C[4] are 4, 3, and 1. For another example the rank and co-rank of C[6] are 6, 5, and 1.

The concept of co-rank gives us a path to parallelizing the merge function. We can divide the work among threads by dividing the output array into subarrays and assigning the generation of one subarray to each thread. Once the assignment has been done, the rank of output elements to be generated by each thread is known. Each thread then uses the co-rank function to determine the two input subarrays that it needs to merge into its output subarray.

Note that the main difference between the parallelization of the merge function and the parallelization of all our previous patterns is that the range of input data to be used by each thread cannot be determined with a simple index calculation. The range of input elements to be used by each thread depends on the actual input values. This makes the parallelized merge operation an interesting and challenging parallel computation pattern.

12.4 Co-rank function implementation

We define the co-rank function as a function that takes the rank (k) of an element in an output array C and information about the two input arrays A and B and

returns the co-rank value (i) for the corresponding element in the input array A. The co-rank function has the following signature:

```
int co_rank(int k, int * A, int m, int * B, int n)
```

where k is the rank of the C element in question, A is a pointer to the input A array, m is the size of the A array, B is a pointer to the input B array, n is the size of the input B array, and the return value is i, the co-rank of k in A.The caller can then derive the j, the co-rank value of k in B, as k − i.

Before we study the implementation details of the co-rank function, it is beneficial to first learn about the ways in which a parallel merge function will use it. Such use of the co-rank function is illustrated in Fig. 12.4, where we use two threads to perform the merge operation. We assume that thread 0 generates C[0]−C[3] and thread 1 generates C[4]−C[8].

Intuitively, each thread calls the co-rank function to derive the beginning positions of the subarrays of A and B that will be merged into the C subarray that is assigned to the thread. For example, thread 1 calls the co-rank function with parameters (4, A, 5, B, 4). The goal of the co-rank function for thread 1 is to identify for its rank value k1=4 the co-rank values i1=3 and j1=1. That is, the subarray starting at C[4] is to be generated by merging the subarrays starting at A[3] and B[1]. Intuitively, we are looking for a total of four elements from A and B that will fill the first four elements of the output array prior to where thread 1 will merge its elements. By visual inspection we see that the choice of i1=3 and j1=1 meets our need. Thread 0 will take A[0]−A[2] and B[0], leaving out A[3] (value 9) and B[1] (value 10), which is where thread 1 will start merging.

If we changed the value of i1 to 2, we need to set the j1 value to 2 so that we can still have a total of four elements prior to thread 1. However, this means that we would include B[1] whose value is 10 in thread 0's elements. This value is larger than A[2] (value 8) that would be included in thread 1's elements. Such a change would make the resulting C array not properly sorted. On the other hand,

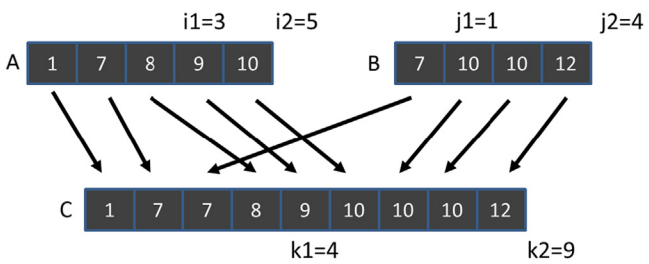


FIGURE 12.4

Example of co-rank function execution.

if we changed the value of i1 to 4, we need to set the j1 value to 0 to keep the total number of elements at 4. However, this would mean that we include A[3] (value 9) in thread 0's elements, which is larger than B[0] (value 7), which would be incorrectly included in thread 1's elements. These two examples point to a search algorithm can quickly identify the value.

In addition to identifying where its input segments start, thread 1 also needs to identify where they end. For this reason, thread 1 also calls the co-rank function with parameters (9, A, 5, B, 4). From Fig. 12.4 we see that the co-rank function should produce co-rank values i2=5 and j2=4. That is, since C[9] is beyond the last element of the C array, all elements of the A and B arrays should have been exhausted if one were trying to generate a C subarray starting at C[9]. In general, the input subarrays to be used by thread t are defined by the co-rank values for thread t and thread t + 1: $A[i_t]-A[i_{t+1}]$ and $B[j_t]-B[j_{t+1}]$.

The co-rank function is essentially a search operation. Since both input arrays are sorted, we can use a binary search or even a higher radix search to achieve a computational complexity of O(log N) for the search. Fig. 12.5 shows a co-rank function based on binary search. The co-rank function uses two pairs of marker variables to delineate the range of A array indices and the range of B array indices being considered for the co-rank values. Variables i and j are the candidate co-rank return values that are being considered in the current binary search iteration. Variables i_low and j_low are the smallest possible co-rank values that could be generated by the function. Line 02 initializes i to its largest possible

```
01    int co_rank(int k, int* A, int m, int* B, int n) {
02        int i = k < m ? k : m; // i = min(k,m)
03        int j = k - i;
04        int i_low = 0 > (k-n) ? 0 : k-n; // i_low = max(0,k-n)
05        int j_low = 0 > (k-m) ? 0 : k-m; // i_low = max(0,k-m)
06        int delta;
07        bool active = true;
08        while(active) {
09            if (i > 0 && j < n && A[i-1] > B[j]) {
10                delta = ((i - i_low +1) >> 1) ; // ceil(i-i_low)/2)
11                j_low = j;
12                j = j + delta;
13                i = i - delta;
14            } else if (j > 0 && i < m && B[j-1] >= A[i]) {
15                delta = ((j - j_low +1) >> 1) ;
16                i_low = i;
17                i = i + delta;
18                j = j - delta;
19            } else {
20                active = false;
21            }
22        }
23        return i;
24    }
```

FIGURE 12.5

A co-rank function based on binary search.

value. If the k value is greater than m, line 02 initializes i to m, since the co-rank i value cannot be larger than the size of the A array. Otherwise, line 02 initializes i to k, since i cannot be larger than k. The co-rank j value is initialized as k − i (line 03). Throughout the execution the co-rank function maintains this important invariant relation. The sum of the i and j variables is always equal to the value of the input variable k (the rank value).

The initialization of the i_low and j_low variables (lines 4 and 5) requires a little more explanation. These variables allow us to limit the scope of the search and make it faster. Functionally, we could set both values to zero and let the rest of the execution elevate them to more accurate values. This makes sense when the k value is smaller than m and n. However, when k is larger than n, we know that the i value cannot be less than k − n. The reason is that the greatest number of C[k] prefix subarray elements that can come from the B array is n. Therefore a minimum of k − n elements must come from A. Therefore the i value can never be smaller than k − n; we may as well set i_low to k − n. Following the same argument, the j_low value cannot be less than k − m, which is the least number of elements of B that must be used in the merge process and thus the lower bound of the final co-rank j value.

We will use the example in Fig. 12.6 to illustrate the operation of the co-rank function in Fig. 12.5. The example assumes that three threads are used to merge arrays A and B into C. Each thread is responsible for generating an output subarray of three elements. We will first trace through the binary search steps of the co-rank function for thread 1, which is responsible for generating C[3]−C[5]. The reader should be able to determine that thread 1 calls the co-rank function with parameters (3, A, 5, B, 4).

As is shown in Fig. 12.5, line 2 of the co-rank function initializes i to 3, which is the k value, since k is smaller than m (value 5) in this example. Also, i_low is set 0. The i and i_low values define the section of A array that is currently being searched to determine the final co-rank i value. Thus only 0, 1, 2, and 3 are being considered for the co-rank i value. Similarly, the j and j_low values are set to 0 and 0.

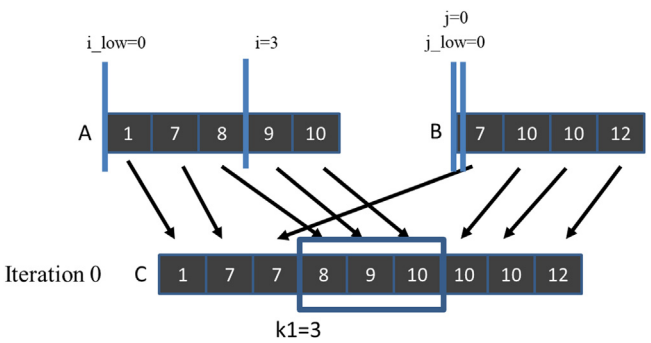


FIGURE 12.6

Iteration 0 of the co-rank function operation example for thread 1.

The main body of the co-rank function is a while-loop (line 08) that iteratively zooms into the final co-rank i and j values. The goal is to find a pair of i and j values that result in A[i-1] ≤ B[j] and B[j-1] < A[i]. The intuition is that we choose the i and j values so none of the values in the A subarray used for generating the previous output subarray (referred to as the previous A subarray) should be greater than any elements in the B subarray used for generating the current output subarray (referred to as the current B subarray). Note that the largest A element in the previous subarray could be equal to the smallest element in the current B subarray, since the A elements take precedence in placement into the output array whenever a tie occurs between an A element and a B element because of the stability requirement.

In Fig. 12.5 the first if-construct in the while-loop (line 09) tests whether the current i value is too high. If so, it will adjust the marker values so that it reduces the search range for i by about half toward the smaller end. This is done by reducing the i value by about half the difference between i and i_low. In Fig. 12.7, for iteration 0 of the while-loop, the if-construct finds that the i value (3) is too high, since A[i − 1], whose value is 8, is greater than B[j], whose value is 7. The next few statements proceed to reduce the search range for i by reducing its value by delta=(3 − 0 + 1) ≫ 1=2 (lines 10 and 13) while keeping the i_low value unchanged. Therefore the i_low and i values for the next iteration will be 0 and 1.

The code also makes the search range for j to be comparable to that of i by shifting it to above the current j location. This adjustment maintains the property that the sum of i and j should be equal to k. The adjustment is done by assigning the current j value to j_low (line 11) and adding the delta value to j (line 12). In our example the j_low and j values for the next iteration will be 0 and 2.

During iteration 1 of the while-loop, illustrated in Fig. 12.7, the i and j values are 1 and 2. The if-construct (line 9) finds the i value to be acceptable since A[i − 1] is A[0] whose value is 1, while B[j] is B[2] whose value is 10, so A[i − 1] is less than B[j]. Thus the condition of the first if-construct fails, and the body of the if-construct is skipped. However, the j value is found to be too high during this iteration, since B[j − 1] is B[1] (line 14), whose value is 10, while A[i] is A

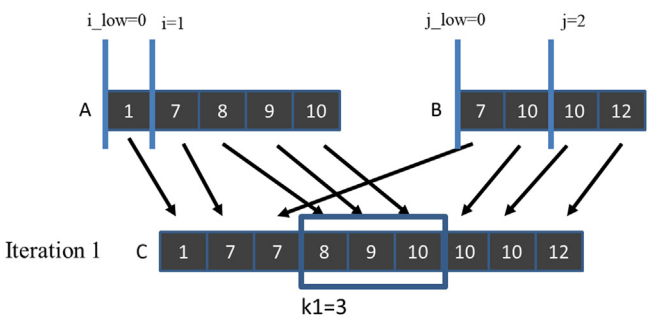


FIGURE 12.7

Iteration 1 of the co-rank function operation example for thread 1.

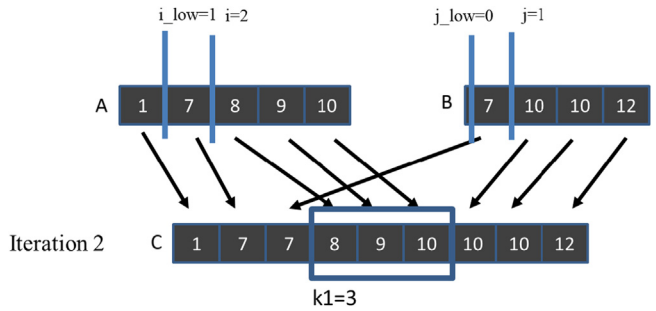


FIGURE 12.8

Iteration 2 of the co-rank function operation example for thread 0.

[1], whose value is 7. Therefore the second if-construct will adjust the markers for the next iteration so that the search range for j will be reduced by about half toward the lower values. This is done by subtracting delta=(j − j_low + 1) ≫ 1=1 from j (lines 15 and 18). As a result, the j_low and j values for the next iteration will be 0 and 1. It also makes the next search range for i the same size as that for j but shifts it up by delta locations. This is done by assigning the current i value to i_low (line 16) and adding the delta value to i (line 17). Therefore the i_low and i values for the next iteration will be 1 and 2, respectively.

During iteration 2, illustrated in Fig. 12.8, the i and j values are 2 and 1. Both if-constructs (lines 9 and 14) will find both i and j values acceptable. For the first if-construct, A[i − 1] is A[1] (value 7) and B[j] is B[1] (value 10), so the condition A[i − 1] ≤ B[j] is satisfied. For the second if-construct, B[j − 1] is B[0] (value 7) and A[i] is A[2] (value 8), so the condition B[j − 1] < A[i] is also satisfied. The co-rank function sets a flag to exit the while-loop (lines 20 and 08) and returns the final i value 2 as the co-rank i value (line 23). The caller thread can derive the final co-rank j value as k − i=3 − 2=1. An inspection of Fig. 12.8 confirms that co-rank values 2 and 1 indeed identify the correct A and B input subarrays for thread 1.

The reader should repeat the same process for thread 2 as an exercise. Also, note that if the input streams are much longer, the delta values will be reduced by half in each step, so the algorithm is of $\log_2(N)$ complexity, where N is the maximum of the two input array sizes.

12.5 A basic parallel merge kernel

For the rest of this chapter we assume that the input A and B arrays reside in the global memory. We further assume that a kernel is launched to merge the two input arrays to produce an output array C that is also in the global memory. Fig. 12.9 shows a basic kernel that is a straightforward implementation of the parallel merge function described in Section 12.3.

```
01  __global__ void merge_basic_kernel(int* A, int m, int* B, int n, int* C) {
02    int tid = blockIdx.x*blockDim.x + threadIdx.x;
03    int elementsPerThread = ceil((m+n)/(blockDim.x*gridDim.x));
04    int k_curr = tid*elementsPerThread; // start output index
05    int k_next = min((tid+1)*elementsPerThread, m+n); // end output index
06    int i_curr = co_rank(k_curr, A, m, B, n);
07    int i_next = co_rank(k_next, A, m, B, n);
08    int j_curr = k_curr - i_curr;
09    int j_next = k_next - i_next;
10    merge_sequential(&A[i_curr], i_next-i_curr, &B[j_curr], j_next-j_curr, &C[k_curr]);
11  }
```

FIGURE 12.9

A basic merge kernel.

As we can see, the kernel is simple. It first divides the work among threads by calculating the starting point of the output subarray to be produced by the current thread (k_curr) and that of the next thread (k_next). Keep in mind that the total number of output elements may not be a multiple of the number of threads. Each thread then makes two calls to the co-rank function. The first call uses k_curr as the rank parameter, which is the first (lowest-indexed) element of the output subarray that the current thread is to generate. The returned co-rank value, i_curr, gives the lowest-indexed input A array element that belongs to the input subarray to be used by the thread. This co-rank value can also be used to get j_curr for the B input subarray. The i_curr and j_curr values mark the beginning of the input subarrays for the thread. Therefore &A[i_curr] and &B[j_curr] are the pointers to the beginning of the input subarrays to be used by the current thread.

The second call uses k_next as the rank parameter to get the co-rank values for the next thread. These co-rank values mark the positions of the lowest-indexed input array elements to be used by the next thread. Therefore i_next − i_curr and j_next − j_curr give the sizes of the subarrays of A and B to be used by the current thread. The pointer to the beginning of the output subarray to be produced by the current thread is &C[k_curr]. The final step of the kernel is to call the merge_sequential function (from Fig. 12.2) with these parameters.

The execution of the basic merge kernel can be illustrated with the example in Fig. 12.8. The k_curr values for the three threads (threads 0, 1, and 2) will be 0, 3, and 6. We will focus on the execution of thread 1 whose k_curr value will be 3. The i_curr and j_curr values determined from the first co-rank function call are 2 and 1. The k_next value for thread 1 will be 6. The second call to the co-rank function helps determine the i_next and j_next values of 5 and 1. Thread 1 then calls the merge function with parameters (&A[2], 3, &B[1], 0, &C[3]). Note that the 0 value for parameter n indicates that none of the three elements of the output subarray for thread 1 should come from array B. This is indeed the case in Fig. 12.8: output elements C[3] − C[5] all come from A[2] − A[4].

While the basic merge kernel is quite simple and elegant, it falls short in memory access efficiency. First, it is clear that when executing the merge_sequential function, adjacent threads in a warp are not accessing adjacent memory locations when they read and write the input and output subarray elements. For the example in Fig. 12.8, during the first iteration of the merge_sequential function execution, the three adjacent threads would read A[0], A[2], and B[0]. They will then write to C[0], C[3], and C[6]. Thus their memory accesses are not coalesced, resulting in poor utilization of memory bandwidth.

Second, the threads also need to access A and B elements from the global memory when they execute the co-rank function. Since the co-rank function does a binary search, the access patterns are somewhat irregular and will be unlikely to be coalesced. As a result, these accesses can further reduce the efficiency of utilizing the memory bandwidth. It would be helpful if we can avoid these uncoalesced accesses to the global memory by the co-rank function.

12.6 A tiled merge kernel to improve coalescing

In Chapter 6, Performance Considerations, we mentioned three main strategies for improving memory coalescing in kernels: (1) rearranging the mapping of threads to data, (2) rearranging the data itself, and (3) transferring the data between the global memory and the shared memory in a coalesced manner and performing the irregular accesses in the shared memory. For the merge pattern we will use the third strategy, which leverages shared memory to improve coalescing. Using shared memory also has the advantage of capturing the small amount of data reuse across the co-rank functions and the sequential merge phase.

The key observation is that the input A and B subarrays to be used by the adjacent threads are adjacent to each other in memory. Essentially, all threads in a block will collectively use larger, block-level subarrays of A and B to generate a larger, block-level subarray of C. We can call the co-rank function for the entire block to get the starting and ending locations for the block-level A and B subarrays. Using these block-level co-rank values, all threads in the block can cooperatively load the elements of the block-level A and B subarrays into the shared memory in a coalesced pattern.

Fig. 12.10 shows the block-level design of a tiled merge kernel. In this example, we assume that three blocks will be used for the merge operation. At the bottom of the figure, we show that C is partitioned into three block-level subarrays. We delineate these partitions with gray vertical bars. On the basis of the partition, each block calls the co-rank functions to partition the input array into subarrays to be used for each block. We also delineate the input partitions with gray vertical bars. Note that the input partitions can vary significantly in size according to the actual data element values in the input arrays. For example, in Fig. 12.8 the input A subarray is significantly larger than the input B subarray for thread 0. On the

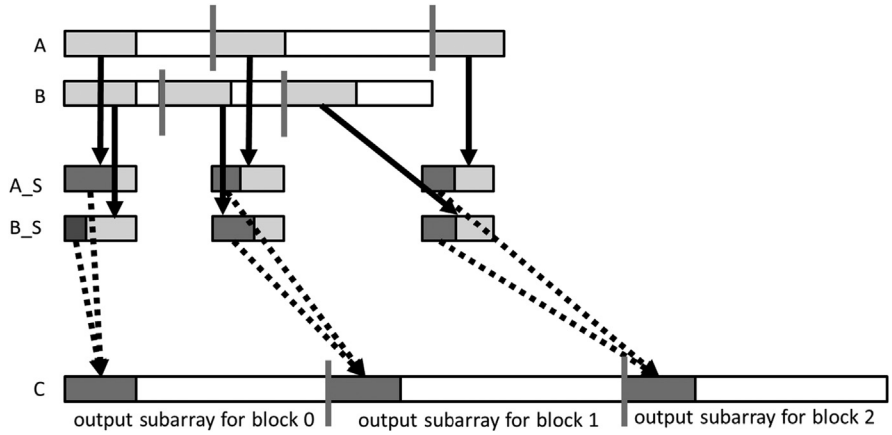


FIGURE 12.10

Design of a tiled merge kernel.

other hand, the input A subarray is significantly smaller than the input B subarray for thread 1. Obviously, the combined size of the two input subarrays must always be equal to the size of the output subarray for each thread.

We will declare two shared memory arrays A_S and B_S for each block. Owing to the limited shared memory size, A_S and B_S may not be able to cover the entire input subarrays for the block. Therefore we will take an iterative approach. Assume that the A_S and B_S arrays can each hold x elements, while each output subarray contains y elements. Each thread block will perform its operation in y/x iterations. During each iteration, all threads in a block will cooperatively load x elements from the block's input A subarray and x elements from its input B subarray.

The first iteration of each thread is illustrated in Fig. 12.10. We show that for each block, a light gray section of the input A subarray is loaded into A_S, and a light gray section of the input B subarray is loaded into B_S. With x A elements and x B elements in the shared memory, the thread block has enough input elements to generate at least x output array elements. All threads are guaranteed to have all the input subarray elements they need for the iteration. One might ask why loading a total of 2x input elements can guarantee the generation of only x output elements. The reason is that in the worst case, all elements of the current output section may all come from one of the input sections. This uncertainty of input usage makes the tiling design for the merge kernel much more challenging than the previous patterns. One can be more accurate in loading the input tiles by first calling the co-rank function for the current and next output sections. In this case, we pay an additional binary search operation to save on redundant data loading. We leave this alternative implementation as an exercise. We will also increase the efficiency of memory bandwidth utilization with a circular buffer design in Section 12.7.

Fig. 12.10 also shows that threads in each block will use a portion of the A_S and a portion of the B_S in each iteration, shown as dark gray sections, to generate a section of x elements in their output C subarray. This process is illustrated with the dotted arrows going from the A_S and B_S dark gray sections to the C dark gray sections. Note that each thread block may well use a different portion of its A_S versus B_S sections. Some blocks may use more elements from A_S, and others may use more from B_S. The actual portions that are used by each block depend on the input data element values.

Fig. 12.11 shows the first part of a tiled merge kernel. A comparison against Fig. 12.9 shows remarkable similarity. This part is essentially the block-level version of the setup code for the thread-level basic merge kernel. Only one thread in the block needs to calculate the co-rank values for the rank values of the beginning output index of the current block and that of the beginning output index of the next block. The values are placed into the shared memory so that they can be visible to all threads in the block. Having only one thread to call the co-rank functions reduces the number of global memory accesses by the co-rank functions and should improve the efficiency of the global memory accesses. A barrier synchronization is used to ensure that all threads wait until the block-level co-rank values are available in the shared memory A_S[0] and A_S[1] locations before they proceed to use the values.

Recall that since the input subarrays may be too large to fit into the shared memory, the kernel takes an iterative approach. The kernel receives a tile_size argument that specifies the number of A elements and B elements to be accommodated in the shared memory. For example, a tile_size value of 1024 means that 1024 A array elements and 1024 B array elements are to be accommodated in the

```
01 __global__ void merge_tiled_kernel(int* A,int m, int* B, int n, int* C, int tile_size) {
   /* shared memory allocation */
02    extern __shared__ int shareAB[];
03   int * A_S = &shareAB[0];                      // shareA is first half of shareAB
04   int * B_S = &shareAB[tile_size];              // shareB is second half of shareAB
05   int C_curr = blockIdx.x * ceil((m+n)/gridDim.x); // start point of block's C subarray
06   int C_next = min((blockIdx.x+1) * ceil((m+n)/gridDim.x), (m+n)); // ending point

07   if (threadIdx.x ==0){
08     A_S[0] = co_rank(C_curr, A, m, B, n); // Make block-level co-rank values visible
09     A_S[1] = co_rank(C_next, A, m, B, n); // to other threads in the block
10   }
11   __syncthreads();
12   int A_curr  = A_S[0];
13   int A_next  = A_S[1];
14   int B_curr = C_curr - A_curr;
15   int B_next   = C_next - A_next;
16   __syncthreads();
```

FIGURE 12.11

Part 1: Identifying block-level output and input subarrays.

shared memory. This means that each block will dedicate (1024 + 1024) × 4=8192 bytes of shared memory to hold the A and B array elements.

As a simple example, assume that we would like to merge an A array of 33,000 elements (m=33,000) with a B array of 31,000 elements (n=31,000). The total number of output C elements is 64,000. Further assume that we will use 16 blocks (gridDim.x=16) and 128 threads in each block (blockDim.x=128). Each block will generate 64,000/16=4000 output C array elements.

If we assume that the tile_size value is 1024, the while-loop in Fig. 12.12 will need to take four iterations for each block to complete the generation of its 4000 output elements. During iteration 0 of the while-loop, the threads in each block will cooperatively load 1024 elements of A and 1024 elements of B into the shared memory. Since there are 128 threads in a block, they can collectively load 128 elements in each iteration of the for-loop (line 26). So the first for-loop in Fig. 12.12 will iterate 8 times for all threads in a block to complete the loading of the 1024 A elements. The second for-loop will also iterate 8 times to complete the loading the 1024 B elements. Note that threads use their threadIdx.x values to select the element to load, so consecutive threads load consecutive elements. The memory accesses are coalesced. We will come back later and explain the if-conditions and how the index expressions for loading the A and B elements are formulated.

Once the input tiles are in the shared memory, individual threads can divide up the input tiles and merge their portions in parallel. This is done by assigning a section of the output to each thread and running the co-rank function to determine the sections of shared memory data that should be used for generating that output section. The code in Fig. 12.13 completes this step. Keep in mind that this is a continuation of the while-loop that started in Fig. 12.12. During each iteration of the while-loop, threads in a block will generate a total of tile_size C elements, using the data that we loaded into shared memory. (The exception is the last

```
17    int counter = 0;                                          //iteration counter
18    int C_length = C_next - C_curr;
19    int A_length = A_next - A_curr;
20    int B_length = B_next - B_curr;
21    int total_iteration = ceil((C_length)/tile_size);         //total iteration
22    int C_completed = 0;
23    int A_consumed = 0;
24    int B_consumed = 0;
25    while(counter < total_iteration){
        /* loading tile-size A and B elements into shared memory */
26        for(int i=0; i<tile_size; i+=blockDim.x){
27            if( i + threadIdx.x < A_length - A_consumed) {
28                A_S[i + threadIdx.x] = A[A_curr + A_consumed + i + threadIdx.x ];
29            }
30        }
31        for(int i=0; i<tile_size; i+=blockDim.x) {
32            if(i + threadIdx.x  < B_length - B_consumed) {
33                B_S[i + threadIdx.x] = B[B_curr + B_consumed + i + threadIdx.x];
34            }
35        }
36        __syncthreads();
```

FIGURE 12.12

Part 2: Loading A and B elements into the shared memory.

```
37        int c_curr  = threadIdx.x   *  (tile size/blockDim.x);
38        int c_next = (threadIdx.x+1) * (tile_size/blockDim.x);
39        c_curr  = (c_curr <= C_length - C_completed) ? c_curr : C_length - C_completed;
40        c_next = (c_next <= C_length - C_completed) ? c_next : C_length - C_completed;
          /* find co-rank for c_curr and c_next */
41        int a_curr = co_rank(c_curr, A_S, min(tile_size, A_length-A_consumed),
                                          B_S, min(tile size, B length-B consumed));
42        int b_curr = c_curr - a_curr;
43        int a_next = co_rank(c_next, A_S, min(tile size, A length-A consumed),
                                          B_S, min(tile size, B length-B consumed));
44        int b_next = c_next - a_next;

          /* All threads call the sequential merge function */
45        merge_sequential (A_S+a_curr, a_next-a_curr, B_S+b_curr, b_next-b_curr,
               C+C_curr+C_completed+c_curr);
          /* Update the number of A and B elements that have been consumed thus far */
46        counter ++;
47        C_completed += tile_size;
48        A_consumed += co_rank(tile_size,  A_S, tile_size, B_S, tile_size);
49        B_consumed = C_completed - A_consumed;
50        __syncthreads();
51     }
52  }
```

FIGURE 12.13

Part 3: All threads merge their individual subarrays in parallel.

iteration, which will be addressed later.) The co-rank function is run on the data in shared memory for individual threads. Each thread first calculates the starting position of its output range and that of the next thread and then uses these starting positions as the inputs to the co-rank function to identify its input ranges. Each thread will then call the sequential merge function to merge its portions of A and B elements (identified by the co-rank values) from the shared memory into its designated range of C elements.

Let us resume our running example. In each iteration of the while-loop, all threads in a block will be collectively generating 1024 output elements, using the two input tiles of A and B elements in the shared memory. (Once again, we will deal with the last iteration of the while-loop later.) The work is divided among 128 threads, so each thread will be generating eight output elements. While we know that each thread will consume a total of eight input elements in the shared memory, we need to call the co-rank function to find out the exact number of A elements versus B elements that each thread will consume and their start and end locations. For example, one thread may use three A elements and five B elements, while another may use six A elements and two B elements, and so on.

Collectively, the total number of A elements and B elements that are used by all threads in a block for the iteration will add up to 1024 in our example. For example, if all threads in a block use 476 A elements, we know that they also used 1024 − 476=548 B elements. It may even be possible that all threads end up using 1024 A elements and 0 B elements. Keep in mind that a total of 2048 elements are loaded in the shared memory. Therefore in each iteration of the while-loop, only half of the A and B elements that were loaded into the shared memory will be used by all the threads in the block.

We are now ready to examine more details of the kernel function. Recall that we skipped the explanation of the index expressions for loading the A and B

elements from the global memory into the shared memory. For each iteration of the while-loop, the starting point for loading the current tile in the A and B array depends on the total number of A and B elements that have been consumed by all threads of the block during the previous iterations of the while-loop. Assume that we keep track of the total number of A elements that were consumed by all the previous iterations of the while-loop in variable A_consumed. We initialize A_consumed to 0 before entering the while-loop. During iteration 0 of the while-loop, all blocks start their tiles from A[A_curr] since A_consumed is 0 at the beginning of iteration 0. During each subsequent iteration of the while-loop, the tile of A elements will start at A[A_curr + A_consumed].

Fig. 12.14 illustrates the index calculation for iteration 1 of the while-loop. In our running example in Fig. 12.10 we show the A_S elements that are consumed by the block of threads during iteration 0 as the dark gray portion of the tile in A_S. During iteration 1 the tile to be loaded from the global memory for block 0 should start at the location right after the section that contains the A elements consumed in iteration 0. In Fig. 12.14, for each block, the section of A elements that is consumed in iteration 0 is shown as the small white section at the beginning of the A subarray (marked by the vertical bars) assigned to the block. Since the length of the small section is given by the value of A_consumed, the tile to be loaded for iteration 1 of the while-loop starts at A[A_curr + A_consumed]. Similarly, the tile to be loaded for iteration 1 of the while-loop starts at B[B_curr + B_consumed].

Note that in Fig. 12.13, A_consumed (line 48) and C_completed are accumulated through the while-loop iterations. Also, B_consumed is derived from the accumulated A_consumed and C_completed values, so it is also accumulated through the while-loop iterations. Therefore they always reflect the number of A

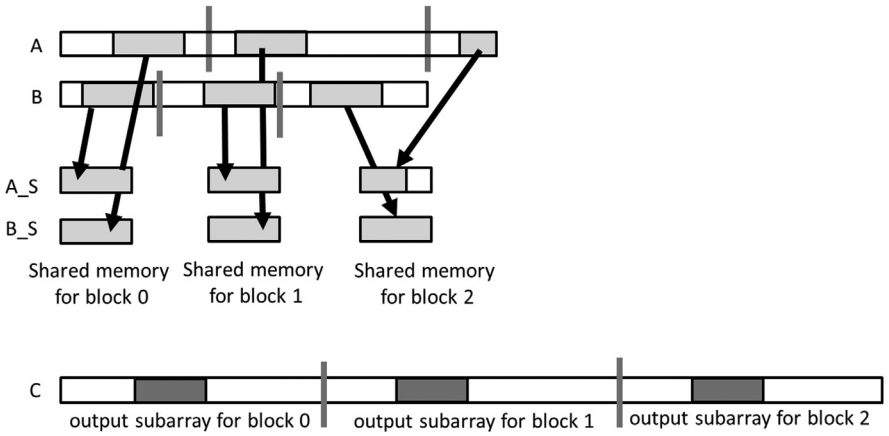


FIGURE 12.14

Iteration 1 of the while-loop in the running example.

and B elements that are consumed by all the iterations so far. At the beginning of each iteration the tiles to be loaded for the iteration always start with A[A_curr + A_consumed] and B[B_curr + B_consumed].

During the last iterations of the while-loop, there may not be enough input A or B elements to fill the input tiles in the shared memory for some of the thread blocks. For example, in Fig. 12.14, for thread block 2, the number of remaining A elements for iteration 1 is less than the tile size. An if-statement should be used to prevent the threads from attempting to load elements that are outside the input subarrays for the block. The first if-statement in Fig. 12.12 (line 27) detects such attempts by checking whether the index of the A_S element that a thread is trying to load exceeds the number of remaining A elements given by the value of the expression A_length − A_consumed. The if-statement ensures that the threads load only the elements that are within the remaining section of the A subarray. The same is done for the B elements (line 32).

With the if-statements and the index expressions, the tile loading process should work well as long as A_consumed and B_consumed give the total number of A and B elements consumed by the thread block in previous iterations of the while-loop. This brings us to the code at the end of the while-loop in Fig. 12.13. These statements update the total number of C elements generated by the while-loop iterations thus far. For all but the last iteration, each iteration generates additional tile_size C elements.

The next two statements update the total number of A and B elements consumed by the threads in the block. For all but the last iteration the number of additional A elements consumed by the thread block is the returned value of

```
co_rank(tile_size, A_S, tile_size, B_S, tile_size)
```

As we mentioned before, the calculation of the number of elements consumed may not be correct at the end of the last iteration of the while-loop. There may not be a full tile of elements left for the final iteration. However, since the while-loop will not iterate any further, the A_consumed, B_consumed, and C_completed values will not be used so the incorrect results will not cause any harm. However, one should remember that if for any reason these values are needed after exiting the while-loop, the three variables will not have the correct values. The values of A_length, B_length, and C_length should be used instead, since all the elements in the designated subarrays to the thread block will have been consumed at the exit of the while-loop.

The tiled kernel achieves substantial reduction in global memory accesses by the co-rank function and makes the global memory accesses coalesce. However, as is, the kernel has a significant deficiency. It makes use of only half of the data that is loaded into the shared memory in each iteration. The unused data in the shared memory is simply reloaded in the next iteration. This wastes half of the memory bandwidth. In the next section we will present a circular buffer scheme

for managing the tiles of data elements in the shared memory, which allows the kernel to fully utilize all the A and B elements that have been loaded into the shared memory. As we will see, this increased efficiency comes with a substantial increase in code complexity.

12.7 A circular buffer merge kernel

The design of the circular buffer merge kernel, which will be referred to as merge_circular_buffer_kernel, is largely the same as that of the merge_tiled_kernel kernel in the previous section. The main difference lies in the management of the A and B elements in the shared memory to enable full utilization of all the elements loaded from the global memory. The overall structure of the merge_tiled_ kernel is shown in Figs. 12.12 through 12.14; it assumes that the tiles of the A and B elements always start at A_S[0] and B_S[0], respectively. After each while-loop iteration the kernel loads the next tile, starting from A_S[0] and B_S [0]. The inefficiency of the merge_tiled_kernel comes from the fact that part of the next tiles of elements are in the shared memory, but we reload the entire tile from the global memory and write over these remaining elements from the previous iteration.

Fig. 12.15 shows the main idea of merge_circular_buffer_kernel. We will continue to use the example from Figs. 12.10 and 12.14. Two additional variables, A_S_start and B_S_start, are added to allow each iteration of the while-loop in

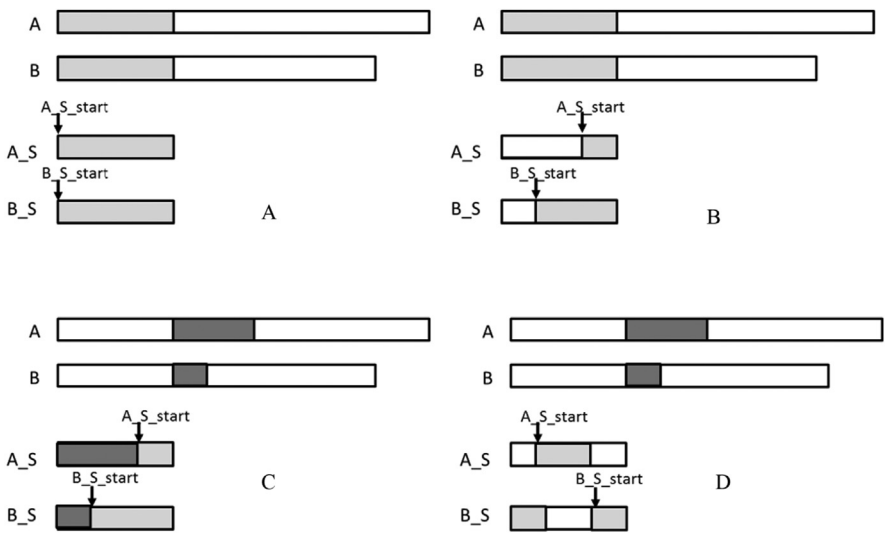


FIGURE 12.15

A circular buffer scheme for managing the shared memory tiles.

Fig. 12.12 to start its A and B tiles at dynamically determined positions inside A_S[0] and B_S[0], respectively. This added tracking allows each iteration of the while-loop to start the tiles with the remaining A and B elements from the previous iteration. Since there is no previous iteration when we first enter the while-loop, these two variables are initialized to 0 before entering the while-loop.

During iteration 0, since the values of A_S_start and B_S_start are both 0, the tiles will start with A_S[0] and B_S[0]. This is illustrated in Fig. 12.15A, where we show the tiles that will be loaded from the global memory (A and B) into the shared memory (A_S and B_S) as light gray sections. Once these tiles have been loaded into the shared memory, merge_circular_buffer_kernel will proceed with the merge operation in the same way as the merge_tile_kernel.

We also need to update the A_S_start and B_S_start variables for use in the next iteration by advancing the value of these variables by the number of A and B elements consumed from the shared memory during the current iteration. Keep in mind that the size of each buffer is limited to tile_size. At some point, we will need to reuse the buffer locations at the beginning part of the A_S and B_S arrays. This is done by checking whether the new A_S_start and B_S_start values exceed the tile_size. If so, we subtract tile_size from them as shown in the following if-statement:

```
A_S_start = (A_S_start + A_S_consumed)%tile_size;
B_S_start = (B_S_start + B_S_consumed)%tile_size;
```

Fig. 12.15B illustrates the update of the A_S_start and B_S_start variables. At the end of iteration 0 a portion of the A tile and a portion of the B tile have been consumed. The consumed portions are shown as white sections in A_S and B_S in Fig. 12.15B. We update the A_S_start and B_S_start values to the position immediately after the consumed sections in the shared memory.

Fig. 12.15C illustrates the operations for filling the A and B tiles at the beginning of iteration 1 of the while-loop. A_S_consumed is a variable that is added to track the number of A elements used in the current iteration. The variable is useful for filling the tile in the next iteration. At the beginning of each iteration we need to load a section of up to A_S_consumed elements to fill up the A tile in the shared memory. Similarly, we need to load a section of up to B_S_consumed elements to fill up the B tile in the shared memory. The two sections that are loaded are shown as dark gray sections in Fig. 12.15C. Note that the tiles effectively "wrap around" in the A_S and B_S arrays, since we are reusing the space of the A and B elements that were consumed during iteration 0.

Fig. 12.15D illustrates the updates to A_S_start and B_S_start at the end of iteration 1. The sections of elements that were consumed during iteration 1 are shown as the white sections. Note that in A_S, the consumed section wraps around to the beginning part of A_S. The value of the A_S_start variable is also wrapped around by the % modulo operator. It should be clear that we will need

to adjust the code for loading and using the tiled elements to support this circular usage of the A_S and B_S arrays.

Part 1 of merge_circular_buffer_kernel is identical to that of merge_tiled_kernel in Fig. 12.11, so we will not present it. Fig. 12.16 shows part 2 of the circular buffer kernel. Refer to Fig. 12.12 for variable declarations that remain the same. New variables A_S_start, B_S_start, A_S_consumed, and B_S_consumed are initialized to 0 before we enter the while-loop.

Note that the exit conditions of the two for-loops have been adjusted. Instead of always loading a full tile, as was the case in the merge kernel in Fig. 12.12, each for-loop in Fig. 12.16 is set up to load only the number of elements that are needed to refill the tiles, given by A_S_consumed. The section of the A elements to be loaded by a thread block in the ith for-loop iteration starts at global memory location A[A_curr + A_consumed + i]. Note that i is incremented by blockDim.x after each iteration. Thus the A element to be loaded by a thread in the ith for-loop iteration is A[A_curr + A_consumed + i + threadIdx.x]. The index for each thread to place its A element into the A_S array is A_S_start + (tile_size − A_S_consumed) + I + threadIdx, since the tile starts at A_S[A_S_start] and there are (tile_size − A_S_consumed) elements remaining in the buffer from the previous iteration of the while-loop. The modulo (%) operation checks whether the index value is greater than or equal to tile_size. If it is, it is wrapped back into the beginning part of the array by subtracting tile_size from the index value. The same analysis applies to the for-loop for loading the B tile and is left as an exercise for the reader.

Using the A_S and B_S arrays as circular buffers also incurs additional complexity in the implementation of the co-rank and merge functions. Part of the additional complexity could be reflected in the thread-level code that calls these functions. However, in general, it is better if one can efficiently handle the complexities inside the library functions to minimize the increased level of complexity in the user code. We show such an approach in Fig. 12.17. Fig. 12.17A shows

```
25  int A_S_start = 0;
26  int B_S_start = 0;
27  int A_S_consumed = tile_size;  //in the first iteration, fill the tile size
28  int B_S_consumed = tile_size;  //in the first iteration, fill the tile_size
29  while(counter < total_iteration) {
      /* loading A_S_consumed elements into A_S */
30      for(int i=0; i<A_S_consumed; i+=blockDim.x) {
31        if(i+threadIdx.x < A_length-A_consumed && (i+threadIdx.x) < A_S_consumed) {
32             A_S[(A_S_start + (tile_size - A_S_consumed)+ i + threadIdx.x)%tile_size] =
                 A[A_curr + A_consumed + i + threadIdx.x ];
33        }
34      }
      /* loading B_S_consumed elements into B_S */
35      for(int i=0; i<B_S_consumed; i+=blockDim.x) {
36        if(i+threadIdx.x < B_length-B_consumed && (i+threadIdx.x) < B_S_consumed) {
37           B_S[(B_S_start + (tile_size - A_S_consumed) +i + threadIdx.x)%tile_size] =
               B[B_curr + B_consumed + i + threadIdx.x];
38        }
39    }
```

FIGURE 12.16

Part 2 of a circular buffer merge kernel.

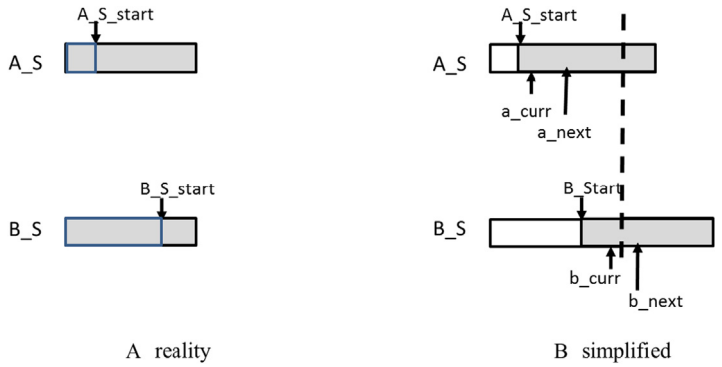


FIGURE 12.17

A simplified model for the co-rank values when using a circular buffer.

the implementation of the circular buffer. A_S_start and B_S_start mark the beginning of the tile in the circular buffer. The tiles wrap around in the A_S and B_S arrays, shown as the light gray section to the left of A_S_start and B_S_start.

Keep in mind that the co-rank values are used for threads to identify the starting position, ending position, and length of the input subarrays that they are to use. When we employ circular buffers, we could provide the co-rank values as the actual indices in the circular buffer. However, this would incur quite a bit of complexity in the merge_circular_buffer_kernel code. For example, the a_next value could be smaller than the a_curr value, since the tile is wrapped around in the A_S array. Thus one would need to test for the case and calculate the length of the section as a_next-a_curr + tile_size. However, in other cases when a_next is larger than a_curr, the length of the section is simply a_next − a_curr.

Fig. 12.17B shows a simplified model for defining, deriving, and using the co-rank values with the circular buffer. In this model, each tile appears to be in a continuous section starting at A_S_start and B_S_start. In the case of the B_S tile in Fig. 12.17A, b_next is wrapped around and would be smaller than b_curr in the circular buffer. However, as is shown in Fig. 12.17B, the simplified model provides the illusion that all elements are in a continuous section of up to tile_size elements; thus a_next is always larger than or equal to a_curr, and b_next is always larger than or equal to b_curr. It is up to the implementation of the co_rank_circular and merge_sequential_circular functions to map this simplified view of the co-rank values into the actual circular buffer indices so that they can carry out their functionalities correctly and efficiently.

The co_rank_circular and merge_sequential_circular functions have the same set of parameters as the original co_rank and merge functions plus three additional parameters: A_S_start, B_S_start, and tile_size. These three additional parameters inform the functions where the current starting point of the buffers are and how big the buffers are. Fig. 12.18 shows the revised thread-level code based

```
40        int c_curr  = threadIdx.x   *  (tile_size/blockDim.x);
41        int c_next = (threadIdx.x+1) * (tile_size/blockDim.x);

42        c_curr  = (c_curr <= C_length-C_completed) ? c_curr : C_length-C_completed;
43        c_next = (c_next <= C_length-C_completed) ? c_next : C_length-C_completed;
        /* find co-rank for c_curr and c_next */
44        int a_curr = co_rank_circular(c_curr,
                        A_S, min(tile_size, A_length-A_consumed),
                        B_S, min(tile_size, B_length-B_consumed),
                        A_S_start, B_S_start, tile_size);
45        int b_curr = c_curr - a_curr;
46        int a_next = co_rank_circular(c_next,
                        A_S, min(tile_size, A_length-A_consumed),
                        B_S, min(tile_size, B_length-B_consumed),
                        A_S_start, B_S_start, tile_size);

47        int b_next = c_next - a_next;
        /* All threads call the circular-buffer version of the sequential merge function */
48        merge_sequetial_circular( A_S, a_next-a_curr,
                        B_S, b_next-b_curr,  C+C_curr+C_completed+c_curr,
                        A_S_start+a_curr, B_S_start+b_curr, tile_size);

        /* Figure out the work has been done */
49        counter ++;
50        A_S_consumed = co_rank_circular(min(tile_size,C_length-C_completed),
                        A_S, min(tile_size, A_length-A_consumed),
                        B_S, min(tile_size, B_length-B_consumed),
                        A_S_start, B_S_start, tile_size);

51        B_S_consumed = min(tile_size, C_length-C_completed) - A_S_consumed;
52        A_consumed += A_S_consumed;
53        C_completed += min(tile_size, C_length-C_completed);
54        B_consumed = C_completed - A_consumed;

55        A_S_start = (A_S_start + A_S_consumed) % tile_size;
56        B_S_start = (B_S_start + B_S_consumed) % tile_size;
57        __syncthreads();
58      }
59  }
```

FIGURE 12.18

Part 3 of a circular buffer merge kernel.

on the simplified model for the co-rank value using circular buffers. The only change to the code is that the co_rank_circular and merge_sequential_circular functions are called instead of the co_rank and merge functions. This demonstrates that a well-designed library interface can reduce the impact on the user code when employing sophisticated data structures.

Fig. 12.19 shows an implementation of the co-rank function that provides the simplified model for the co-rank values while correctly operating on circular buffers. It treats i, j, i_low, and j_low values in exactly the same way as the co-rank function in Fig. 12.5. The only change is that i, i − 1, j, and j − 1 are no longer used directly as indices in accessing the A_S and B_S arrays. They are used as offsets that are to be added to the values of A_S_start and B_S_start to form the index values i_cir, i_m_1_cir, j_cir, and j_m_1_cir. In each case, we need to test whether the actual index values need to be wrapped around to the beginning part of the buffer. Note that we cannot simply use i_cir − 1 to replace i − 1. We need to form the final index value and check for the need to wrap it around. It should be clear that the simplified model also helps to keep the co-rank function code simple: All the manipulations of the i, j, i_low, and j_low values remain the same; they do not need to deal with the circular nature of the buffers.

```
int co_rank_circular(int k, int* A, int m, int* B, int n, int A_S_start, int
B_S_start, int tile_size) {
    int i= k < m ? k : m;  // i = min(k,m)
    int j = k- i;
    int i_low = 0 > (k-n) ? 0 : k-n; // i_low = max(0, k-n)
    int j_low = 0 > (k-m) ? 0 : k-m; // i_low = max(0,k-m)
    int delta;
    bool active = true;
    while(active) {
        int i_cir = (A_S_start+i) % tile_size;
        int i_m_1_cir = (A_S_start+i-1) % tile_size);
        int j_cir = (B_S_start+j) % tile_size);
        int j_m_1_cir = (B_S_start+i-1) % tile_size);
        if (i > 0 && j < n && A[i_m_1_cir] > B[j_cir]) {
            delta = ((i - i_low +1) >> 1) ; // ceil(i-i_low)/2)
            j_low = j;
            i = i - delta;
            j = j + delta;
        } else if (j > 0 && i < m && B[j_m_1_cir] >= A[i_cir]) {
            delta = ((j - j_low +1) >> 1) ;
            i_low = i;
            i = i + delta;
            j = j - delta;
        } else {
            active = false;
        }
    }
    return i;
}
```

FIGURE 12.19

A co_rank_circular function that operates on circular buffers.

Fig. 12.20 shows an implementation of the merge_sequential_circular function. Similarly to the co_rank_circular function, the logic of the code remains essentially unchanged from the original merge function. The only change is in the way in which i and j are used to access the A and B elements. Since the merge_sequential_circular function will be called only by the thread-level code of merge_circular_buffer_kernel, the A and B elements that are accessed will be in the A_S and B_S arrays. In all four places where i or j is used to access the A or B elements, we need to form the i_cir or j_cir and test whether the index value needs to be wrapped around to the beginning part of the array. Otherwise, the code is the same as that of the merge function in Fig. 12.2.

Although we did not list all parts of merge_circular_buffer_kernel, the reader should be able to put it all together on the basis of the parts that we discussed. The use of tiling and circular buffers adds quite a bit of complexity. In particular, each thread uses quite a few more registers to keep track of the starting point and the remaining number of elements in the buffers. All these additional usages can potentially reduce the occupancy, or the number of thread-blocks that can be assigned to each of the streaming multiprocessors when the kernel is executed. However, since the merge operation is memory bandwidth bound, the computational and register resources are likely underutilized. Thus increasing the number of registers that are used and address calculations to conserve memory bandwidth is a reasonable tradeoff.

```
void merge_sequential_circular(int *A, int m, int *B, int n, int *C, int
A_S_start, int B_S_start, int tile_size) {
    int i = 0;  //virtual index into A
    int j = 0;  //virtual index into B
    int k = 0; //virtual index into C
    while ((i < m) && (j < n)) {
        int i_cir = (A_S_start + i) % tile_size;
        int j_cir = (B_S_start + j) % tile_size;
        if (A[i_cir] <= B[j_cir]) {
            C[k++] = A[i_cir]; i++;
        } else {
            C[k++] = B[j_cir]; j++;
        }
    }
    if (i == m) { //done with A[] handle remaining B[]
        for (; j < n; j++) {
            int j_cir = (B_S_start + j) % tile_size;
            C[k++] = B[j_cir];
        }
    } else { //done with B[], handle remaining A[]
        for (; i <m; i++) {
            int i_cir = (A_S_start + i) % tile_size);
            C[k++] = A[i_cir];
        }
    }
}
```

FIGURE 12.20

Implementation of the merge_sequential_circular function.

12.8 Thread coarsening for merge

The price of parallelizing merge across many threads is primarily the fact that each thread has to perform its own binary search operations to identify the co-ranks of its output indices. The number of binary search operations that are performed can be reduced by reducing the number of threads that are launched, which can be done by assigning more output elements per thread. All the kernels that are presented in this chapter already have thread coarsening applied because they are all written to process multiple elements per thread. In a completely uncoarsened kernel, each thread would be responsible for a single output element. However, this would require a binary search operation to be performed for every single element, which would be prohibitively expensive. Hence coarsening is essential for amortizing the cost of the binary search operation across a substantial number of elements.

12.9 Summary

In this chapter we introduced the ordered merge pattern whose parallelization requires each thread to dynamically identify its input position ranges. Because the input ranges are data dependent, we resort to a fast search implementation of the

co-rank function to identify the input range for each thread. The fact that the input ranges are data dependent also creates extra challenges when we use a tiling technique to conserve memory bandwidth and enable memory coalescing. As a result, we introduced the use of circular buffers to allow us to make full use of the data loaded from global memory. We showed that introducing a more complex data structure, such as a circular buffer, can significantly increase the complexity of the code that uses the data structure. Thus we introduce a simplified buffer access model for the code that manipulates and uses the indices to remain largely unchanged. The actual circular nature of the buffers is exposed only when these indices are used to access the elements in the buffer.

Exercises

1. Assume that we need to merge two lists A=(1, 7, 8, 9, 10) and B=(7, 10, 10, 12). What are the co-rank values for C[8]?
2. Complete the calculation of co-rank functions for thread 2 in Fig. 12.6.
3. For the for-loops that load A and B tiles in Fig. 12.12, add a call to the co-rank function so that we can load only the A and B elements that will be consumed in the current generation of the while-loop.
4. Consider a parallel merge of two arrays of size 1,030,400 and 608,000. Assume that each thread merges eight elements and that a thread block size of 1024 is used.
   a. In the basic merge kernel in Fig. 12.9, how many threads perform a binary search on the data in the global memory?
   b. In the tiled merge kernel in Figs. 12.11–12.13, how many threads perform a binary search on the data in the global memory?
   c. In the tiled merge kernel in Figs. 12.11–12.13, how many threads perform a binary search on the data in the shared memory?

References

Siebert, C., Traff, J.L., 2012. Efficient MPI implementation of a parallel, stable merge algorithm. Proceedings of the 19th European conference on recent advances in the message passing interface (EuroMPI'12). Springer-Verlag Berlin, Heidelberg, pp. 204–213.

CHAPTER

13 Sorting

With special contributions from Michael Garland

Chapter Outline

Sorting algorithms place the data elements of a list into a certain order. Sorting is foundational to modern data and information services, since the computational complexity of retrieving information from datasets can be significantly reduced if the dataset is in proper order. For example, sorting is often used to canonicalize the data for fast comparison and reconciliation between data lists. Also, the efficiency of many data-processing algorithms can be improved if the data is in certain order. Because of their importance, efficient sorting algorithms have been the subject of many computer science research efforts. Even with these efficient algorithms, sorting large data lists is still time consuming and can benefit from parallel execution. Parallelizing efficient sorting algorithms is challenging and requires elaborate designs. This chapter presents the parallel designs for two important types of efficient sorting algorithms: radix sort and merge sort. Most of the chapter is dedicated to radix sort; merge sort is discussed briefly on the basis of the parallel merge pattern that was covered in Chapter 12, Merge. Other popular parallel sorting algorithms, such as transposition sort and sampling sort, are also briefly discussed.

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00019-7

13.1 Background

Sorting is one of the earliest applications for computers. A sorting algorithm arranges the elements of a list into a certain order. The order to be enforced by a sorting algorithm depends on the nature of these elements. Examples of popular orders are numerical order for numbers and lexicographical order for text strings. More formally, any sorting algorithm must satisfy the following two conditions:

1. The output is in either nondecreasing or nonincreasing order. For nondecreasing order, each element is no smaller than the previous element according to the desired order. For nonincreasing order, each element is no larger than the previous element according to the desired order.
2. The output is a permutation of the input. That is, the algorithm must retain all of the original input elements while reordering them into the output.

In its simplest form, the elements of a list can be sorted according to the values of each element. For example, the list [5, 2, 7, 1, 3, 2, 8] can be sorted into a nondecreasing order output [1, 2, 2, 3, 5, 7, 8].

A more complex and common use case is that each element consists of a key field and a value field and the list should be sorted on the basis of the key field. For example, assume that each element is a tuple (age, income in thousands of dollars). The list [(30,150), (32,80), (22,45), (29,80)] can be sorted by using the income as the key field into a nonincreasing order [(30,150), (32,80), (29,80), (22,45)].

Sorting algorithms can be classified into stable and unstable algorithms. A stable sort algorithm preserves the original order of appearance when two elements have equal key value. For example, when sorting the list [(30,150), (32,80), (22,45), (29,80)] into a nonincreasing order using income as the key field, a stable sorting algorithm must guarantee that (32, 80) appears before (29,80) because the former appear before the latter in the original input. An unstable sorting algorithm does not offer such a guarantee. Stable algorithms are required if one wishes to use multiple keys to sort a list in a cascaded manner. For example, if each element has a primary key and a secondary key, with stable sorting algorithms, one can first sort the list according to the secondary key and then sort one more time with the primary key. The second sort will preserve the order produced by the first sort.

Sorting algorithms can also be classified into comparison-based and noncomparison-based algorithms. Comparison-based sorting algorithms cannot achieve better than $O(N \cdot \log N)$ complexity when sorting a list of N elements because they must perform a minimal number of comparisons among the elements. In contrast, some of the noncomparison-based algorithms can achieve better than $O(N \cdot \log N)$ complexity, but they may not generalize to arbitrary types of keys. Both comparison-based and noncomparison-based sorting algorithms can be parallelized. In this chapter we present a parallel noncomparison-based sorting

algorithm (radix sort) as well as a parallel comparison-based sorting algorithm (merge sort).

Because of the importance of sorting, the computer science research community has produced a great spectrum of sorting algorithms based on a rich variety of data structures and algorithmic strategies. As a result, introductory computer science classes often use sorting algorithms to illustrate a variety of core algorithm concepts, such as big O notation; divide-and-conquer algorithms; data structures such as heaps and binary trees; randomized algorithms; best-, worst-, and average-case analysis; time-space tradeoffs; and upper and lower bounds. In this chapter we continue this tradition and use two sorting algorithms to illustrate several important parallelization and performance optimization techniques (Satish et al., 2009).

13.2 Radix sort

One of the sorting algorithms that is highly amenable to parallelization is radix sort. Radix sort is a noncomparison-based sorting algorithm that works by distributing the keys that are being sorted into buckets on the basis of a radix value (or base in a positional numeral system). If the keys consist of multiple digits, the distribution of the keys is repeated for each digit until all digits are covered. Each iteration is stable, preserving the order of the keys within each bucket from the previous iteration. In processing keys that are represented as binary numbers, choosing a radix value that is a power of 2 is convenient because it makes iterating over the digits and extracting them easy. Each iteration essentially handles a fixed-size slice of the bits from the key. We will start by using a radix of 2 (i.e., a 1-bit radix) and then extend to larger radix values later in the chapter.

Fig. 13.1 shows an example of how a list of 4-bit integers can be sorted with radix sort using a 1-bit radix. Since the keys are 4 bits long and each iteration processes 1 bit, four iterations are required in total. In the first iteration the least significant bit (LSB) is considered. All the keys in the iteration's input list whose LSB is 0 are placed on the left side of the iteration's output list, forming a bucket for the 0 bits. Similarly, all the keys in the iteration's input list whose LSB is 1 are placed on the right side of the iteration's output list forming a bucket for the 1 bits. Note that within each bucket in the output list, the order of the keys is preserved from that in the input list. In other words, keys that are placed in the same bucket (i.e., that have the same LSB) must appear in the same order in the output list as they did in the input list. We will see why this stability requirement is important when we discuss the next iteration.

In the second iteration in Fig. 13.1, the output list from the first iteration becomes the new input list, and the second LSB of each key is considered. As in the first iteration, the keys are separated into two buckets: a bucket for the keys whose second LSB is 0 and another bucket for the keys whose second LSB is 1.

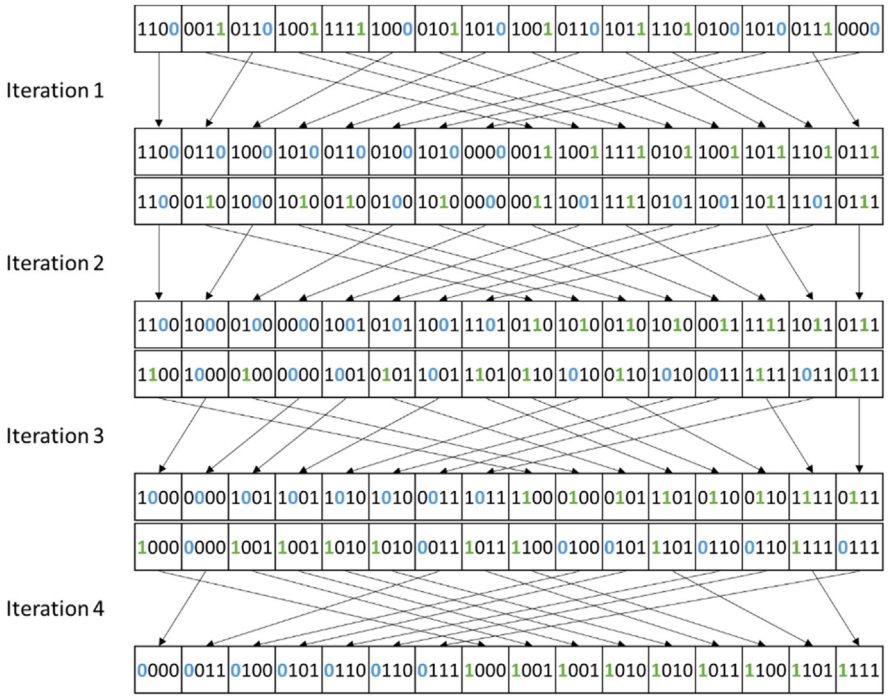


FIGURE 13.1

A radix sort example.

Since the order from the previous iterations is preserved, we observe that the keys in the second iteration's output list are now sorted by the lower two bits. In other words, all the keys whose lower two bits are 00 come first, followed by those whose lower two bits are 01, followed by those whose lower two bits are 10, followed by those whose lower two bits are 11.

In the third iteration in Fig. 13.1 the same process is repeated while considering the third bit in the keys. Again, since the order from previous iterations is preserved, the keys in the output list of the third iteration are sorted by the lower three bits. Finally, in the fourth and last iteration the same process is repeated while considering the fourth or most significant bit. At the end of this iteration the keys in the final output list are sorted by all four bits.

13.3 Parallel radix sort

Each iteration in radix sort depends on the entire result of the previous iteration. Hence the iterations are performed sequentially with respect to each other. The

opportunity for parallelizing radix sort arises within each iteration. For the rest of this chapter we will focus on the parallelization of a single radix sort iteration, with the understanding that the iterations will be executed one after the other. In other words, we will focus on the implementation of a kernel that performs a single radix sort iteration and will assume that the host code calls this kernel once for each iteration.

One straightforward approach to parallelize a radix sort iteration on GPUs is to make each thread responsible for one key in the input list. The thread must identify the position of the key in the output list and then store the key to that position. Fig. 13.2 illustrates this parallelization approach that is applied to the first iteration from Fig. 13.1. Threads in Fig. 13.2 are illustrated as curvy arrows, and thread blocks are illustrated as boxes around the arrows. Each thread is responsible for the key below it in the input list. In this example the 16 keys are processed by a grid with four thread blocks having four threads each. In practice, each thread block may have up to 1024 threads, and the input is much larger, resulting in many more thread blocks. However, we have used a small number of threads per block to simplify the illustration.

With every thread assigned to a key in the input list, the challenge remains for each thread to identify the destination index of its key in the output list. Identifying the destination index of the key depends on whether the key maps to the 0 bucket or the 1 bucket. For keys mapping to the 0 bucket, the destination index can be found as follows:

destination of a **zero** = # **zeros** before
= # **keys** before - # **ones** before
= key index - # **ones** before

The destination index of a key that maps to the 0 bucket (i.e., destination of a 0) is equivalent to the number of keys before the key that also map to the 0 bucket (i.e., # zeros before). Since all keys map to either the 0 bucket or the

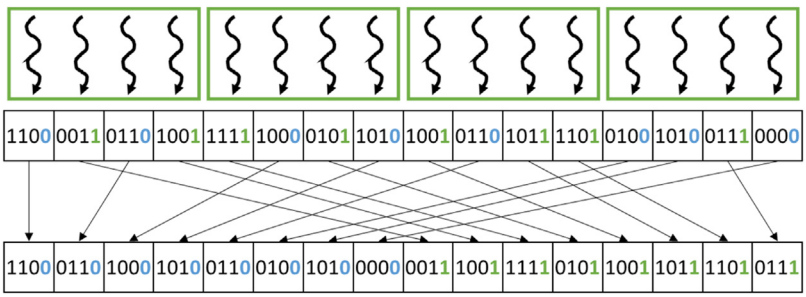


FIGURE 13.2

Parallelizing a radix sort iteration by assigning one input key to each thread.

1 bucket, the number of keys before the key mapping to the 0 bucket is equivalent to the total number of keys before the key (i.e., # keys before) minus the number of keys before the key mapping to the 1 bucket (i.e., # ones before). The total number of keys before the key is just the index of the key in the input list (i.e., the key index), which is trivially available. Hence the only nontrivial part of finding the destination index of a key that maps to the 0 bucket is counting the number of keys before it that map to the 1 bucket. This operation can be done by using an exclusive scan, as we will see shortly.

For keys mapping to the 0 bucket, the destination index can be found as follows:

```
destination of a one  = # zeros in total + # ones before
                      = (# keys in total - # ones in total) + # ones before
                      = input size - # ones in total + # ones before
```

All keys mapping to the 0 bucket must come before the keys mapping to the 1 bucket in the output array. For this reason, the destination index of a key that maps to the 1 bucket (i.e., destination of a 1) is equivalent to the total number of keys mapping to the 0 bucket (i.e., # zeros in total) plus the number of keys before the key that map to the 1 bucket (i.e., # ones before). Since all keys map to either the 0 bucket or the 1 bucket, the total number of keys mapping to the 0 bucket is equivalent to the total number of keys in the input list (i.e., # keys in total) minus the total number of keys mapping to the 1 bucket (i.e., # ones in total). The total number of keys in the input list is just the input size, which is trivially available. Hence the nontrivial part of finding the destination index of a key that maps to the 1 bucket is counting the number of keys before it that map to the 1 bucket, which is the same information that is needed for the 0 bucket case. Again, this operation can be done by using an exclusive scan, as we will see shortly. The total number of keys mapping to the 1 bucket can be found as a byproduct of the exclusive scan.

Fig. 13.3 shows the operations that each thread performs to find its key's destination index in the example in Fig. 13.2. The corresponding kernel code to perform these operations is shown in Fig. 13.4. First, each thread identifies the index of the key for which it is responsible (line 03), performs a boundary check (line 04), and loads the key from the input list (line 06). Next, each thread extracts from the key the bit for the current iteration to identify whether it is a 0 or a 1 (line 07).

Here, the iteration number `iter` tells us the position of the bit in which we are interested. By shifting the key to the right by this amount, we move the bit to the rightmost position. By applying a bitwise-and operation (`&`) between the shifted key and a `1`, we zero out all the bits in the shifted key except the rightmost bit. Hence the value of `bit` will be the value of the bit in which we are interested. In the example in Fig. 13.3, since the example is for iteration 0, the LSB is extracted, as shown in the row labeled bits.

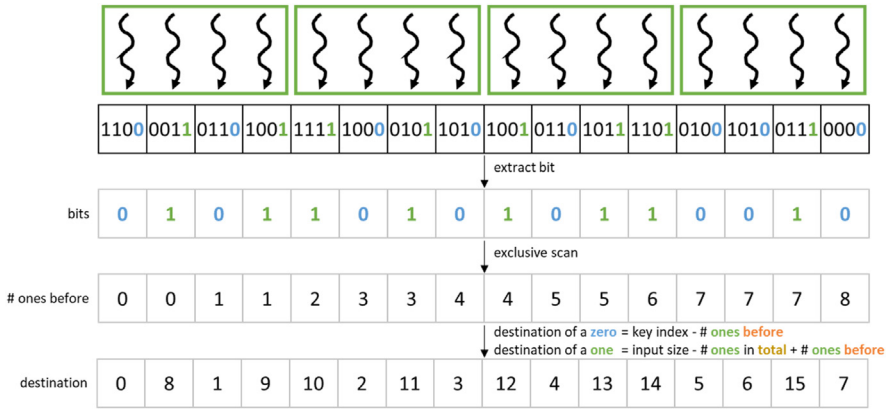


FIGURE 13.3

Finding the destination of each input key.

```
01  __global__ void radix_sort_iter(unsigned int* input, unsigned int* output,
02                       unsigned int* bits, unsigned int N, unsigned int iter) {
03      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04      unsigned int key, bit;
05      if(i < N) {
06          key = input[i];
07          bit = (key >> iter) & 1;
08          bits[i] = bit;
09      }
10      exclusiveScan(bits, N);
11      if(i < N) {
12          unsigned int numOnesBefore = bits[i];
13          unsigned int numOnesTotal = bits[N];
14          unsigned int dst = (bit == 0)?(i - numOnesBefore)
15                                       :(N - numOnesTotal - numOnesBefore);
16          output[dst] = key;
17      }
18  }
```

FIGURE 13.4

Radix sort iteration kernel code.

Once each thread has extracted the bit in which it is interested from the key, it stores the bit to memory (line 08), and the threads collaborate to perform an exclusive scan on the bits (line 10). We discussed how to perform an exclusive scan in Chapter 11, Prefix Sum (Scan). The call to exclusive scan is performed outside the boundary check because threads may need to perform a barrier synchronization in the process, so we need to ensure that all threads are active. To synchronize across all threads in the grid, we assume that we can use sophisticated techniques similar to those used in the single-pass scan discussed in Chapter 11, Prefix Sum (Scan). Alternatively, we could terminate the kernel, call another kernel from the host to perform the scan, and then call a third kernel to perform the operations after the scan. In this case, each iteration would require three grid launches instead of one.

The array resulting from the exclusive scan operation contains, at each position, the sum of the bits before that position. Since these bits are either 0 or 1, the sum of the bits before the position is equivalent to the number of the 1's before the position (i.e., the number of keys that map to the 1 bucket). In the example in Fig. 13.3 the result of the exclusive scan is shown in the row labeled # ones before. Each thread accesses this array to obtain the number of 1's before its position (line 12) and the total number of 1's in the input list (line 13). Each thread can then identify the destination of its key, using the expressions that we derived previously (lines 14−15). Having identified its destination index, the thread can proceed to store the key for which it is responsible at the corresponding location in the output list (line 16). In the example in Fig. 13.3 the destination indices are shown in the row labeled destination. The reader can refer to Fig. 13.2 to verify that the values that are obtained are indeed the right destination indices of each element.

13.4 Optimizing for memory coalescing

The approach we just described is effective at parallelizing a radix sort iteration. However, one major source of inefficiency in this approach is that the writes to the output list exhibit an access pattern that cannot be adequately coalesced. Consider how each thread in Fig. 13.2 writes its key to the output list. In the first thread block, the first thread writes to the 0 bucket, the second thread writes to the 1 bucket, the third thread writes to the 0 bucket, and the fourth thread writes to the 1 bucket. Hence threads with consecutive index values are not necessarily writing to consecutive memory locations, resulting in poor coalescing and requiring multiple memory requests to be issued per warp.

Recall from Chapter 6, Performance Considerations, that there are various approaches to enable better memory coalescing in kernels: (1) rearranging the threads, (2) rearranging the data that the threads access, or (3) performing the uncoalesceable accesses on shared memory and transferring data between shared memory and global memory in a coalesced way. To optimize for coalescing in this chapter, we will use the third approach. Instead of having all threads write their keys to global memory buckets in an uncoalesced manner, we will have each thread block maintain its own local buckets in the shared memory. That is, we will no longer perform a global sort as shown in Fig. 13.4. Rather, the threads in each block will first perform a block-level local sort to separate the keys mapping to the 0 bucket and the keys mapping to the 1 bucket in shared memory. After that, the buckets will be written from shared memory to global memory in a coalesced manner.

Fig. 13.5 shows an example of how memory coalescing can be enhanced for the example in Fig. 13.2. In this example, each thread block first performs a local radix sort on the keys that it owns and stores the output list into the shared

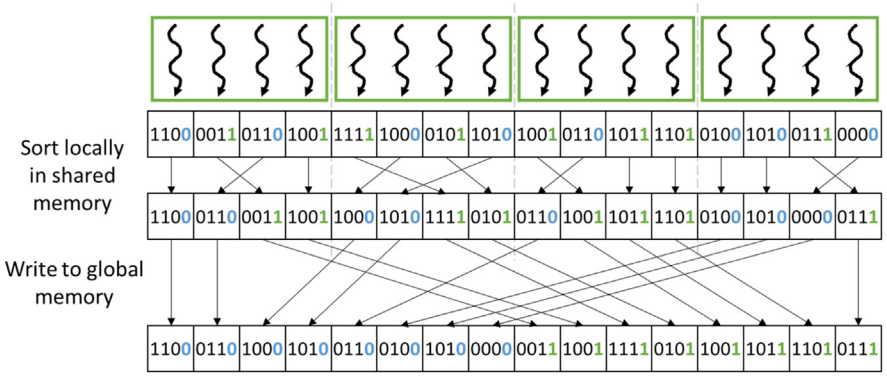


FIGURE 13.5

Optimizing for memory coalescing by sorting locally in shared memory before sorting into the global memory.

memory. The local sort can be done in the same way as the global sort was done previously and requires each thread block to perform only a local exclusive scan instead of requiring a global one. After the local sort, each thread block writes its local buckets to the global buckets in a more coalesced way. For example, in Fig. 13.5, consider how the first thread block writes out its buckets to global memory. The first two threads both write to adjacent locations in global memory when writing the 0 bucket, while the last two threads also write to adjacent locations in global memory when writing the 1 bucket. Hence the majority of writes to global memory will be coalesced.

The main challenge in this optimization is for each thread block to identify the beginning position of each of its local buckets in the corresponding global bucket. The beginning position of a thread block's local buckets depends on the sizes of the local buckets in the other thread blocks. In particular, the position of a thread block's local 0 bucket is after all the local 0 buckets of the preceding thread blocks. On the other hand, the position of a thread block's local 1 bucket is after all the local 0 buckets of all the thread blocks and all the local 1 buckets of the preceding thread blocks. These positions can be obtained by performing an exclusive scan on the thread blocks' local bucket sizes.

Fig. 13.6 shows an example of how an exclusive scan can be used to find the position of each thread block's local buckets. After completing the local radix sort, each thread block identifies the number of keys in each of its local buckets. Next, each thread block stores these values in a table as shown in Fig. 13.6. The table is stored in row-major order, meaning that it places the sizes of the local 0 buckets for all thread blocks consecutively, followed by the sizes of the local 1 buckets. After the table has been constructed, an exclusive scan is executed on the linearized table. The resulting table consists of the beginning positions of each thread block's local buckets, which are the values we are looking for.

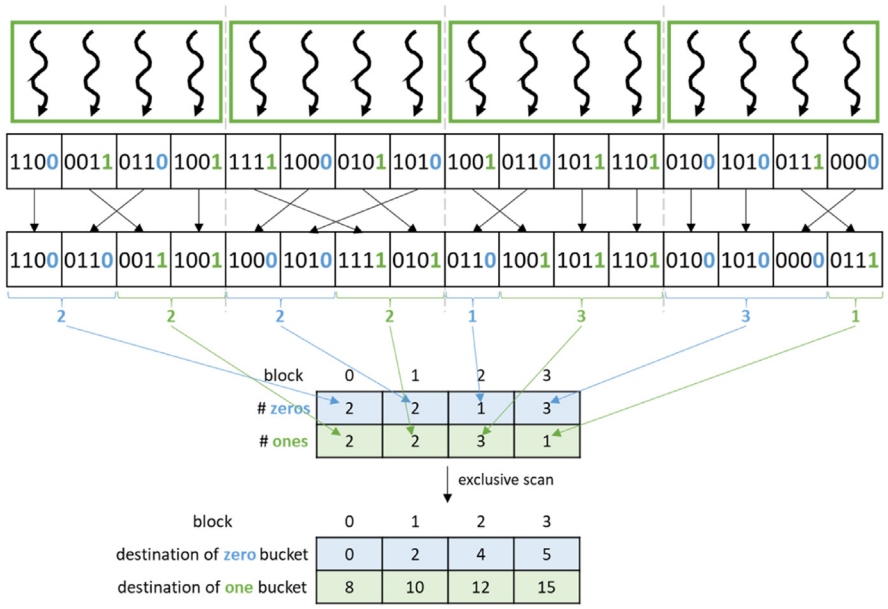


FIGURE 13.6

Finding the destination of each thread block's local buckets.

Once a thread block has identified the beginning position of its local buckets in global memory, the threads in the block can proceed to store their keys from the local buckets to the global buckets. To do so, each thread needs to keep track of the number of keys in the 0 bucket versus the 1 bucket. During the write phase, threads in each block will be writing a key in either of the buckets depending on its thread index values. For example, for block 2 in Fig. 13.6, thread 0 writes the single key in the 0 bucket, and threads 1−3 write the three keys in the 1 bucket. In comparison, for block 3 in Fig. 13.6, threads 0−2 write the three keys in the 0 bucket, and thread 3 writes the 1 key in the one bucket. Hence each thread needs to test whether it is responsible for writing a key in the local 0 bucket or the 1 bucket. Each block tracks the number of keys in each of its two local buckets so that the threads can determine where their threadIdx values fall and participate in the writing of the 0 bucket keys or 1 bucket keys. We leave the implementation of this optimization as an exercise for the reader.

13.5 Choice of radix value

So far, we have seen how radix sort can be parallelized by using a 1-bit radix as an example. For the 4-bit keys in the example, four iterations (one for each bit)

are needed for the keys to be fully sorted. In general, for N-bit keys, N iterations are needed to fully sort the keys. To reduce the number of iterations that are needed, a larger radix value can be used.

Fig. 13.7 shows an example of how radix sort can be performed using a 2-bit radix. Each iteration uses two bits to distribute the keys to buckets. Hence the 4-bit keys can be fully sorted by using only two iterations. In the first iteration the lower two bits are considered. The keys are distributed across four buckets corresponding to the keys where the lower two bits are 00, 01, 10, and 11. In the second iteration the upper two bits are considered. The keys are then distributed across four buckets based on the upper two bits. Similar to the 1-bit example, the order of the keys within each bucket is preserved from the previous iteration. Preserving the order of the keys within each bucket ensures that after the second iteration the keys are fully sorted by all four bits.

Similar to the 1-bit example, each iteration can be parallelized by assigning a thread to each key in the input list to find the key's destination index and store it in the output list. To optimize for memory coalescing, each thread block can sort its keys locally in the shared memory and then write the local buckets to global memory in a coalesced manner. An example of how to parallelize a radix sort iteration and optimize it for memory coalescing using the shared memory is shown in Fig. 13.8.

The key distinction between the 1-bit example and the 2-bit example is how to separate the keys into four buckets instead of two. For the local sort inside of each thread block, a 2-bit radix sort is performed by applying two consecutive 1-bit radix sort iterations. Each of these 1-bit iterations requires its own exclusive scan operation. However, these operations are local to the thread block, so there is no coordination across thread blocks in between the two 1-bit iterations. In general, for an r-bit radix, r local 1-bit iterations are needed to sort the keys into 2^r local buckets.

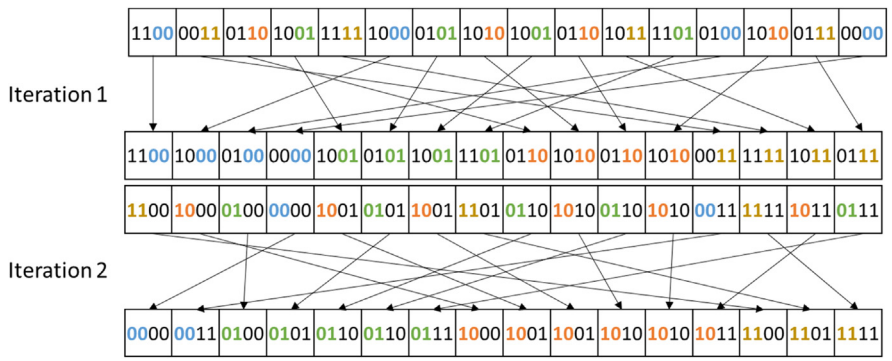


FIGURE 13.7

Radix sort example with 2-bit radix.

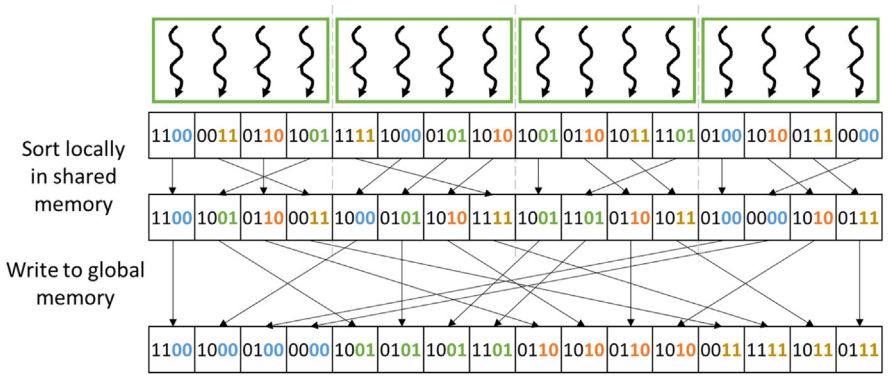


FIGURE 13.8

Parallelizing a radix sort iteration and optimizing it for memory coalescing using the shared memory for a 2-bit radix.

After the local sort is complete, each thread block must find the position of each of its local buckets in the global output list. Fig. 13.9 shows an example of how the destination of each local bucket can be found for the 2-bit radix example. The procedure is similar to the 1-bit example in Fig. 13.6. Each thread block stores the number of keys in each local bucket to a table, which is then scanned to obtain the global position of each of the local buckets. The main distinction from the 1-bit radix example is that each thread block has four local buckets instead of two, so the exclusive scan operation is performed on a table with four rows instead of two. In general, for an r-bit radix the exclusive scan operation is performed on a table with 2^r rows.

We have seen that the advantage of using a larger radix is that it reduces the number of iterations that are needed to fully sort the keys. Fewer iterations means fewer grid launches, global memory accesses, and global exclusive scan operations. However, using a larger radix also has disadvantages. The first disadvantage is that each thread block has more local buckets where each bucket has fewer keys. As a result, each thread block has more distinct global memory bucket sections that it needs to write to and less data that it needs to write to each section. For this reason, the opportunities for memory coalescing decrease as the radix gets larger. The second disadvantage is that the table on which the global exclusive scan is applied gets larger with a larger radix. For this reason, the overhead of the global exclusive scan increases as the radix increases. Therefore the radix cannot be made arbitrarily large. The choice of radix value must strike a balance between the number of iterations on one hand and the memory coalescing behavior as well as the overhead of the global exclusive scan on the other hand. We leave the implementation of radix sort with a multibit radix as an exercise for the reader.

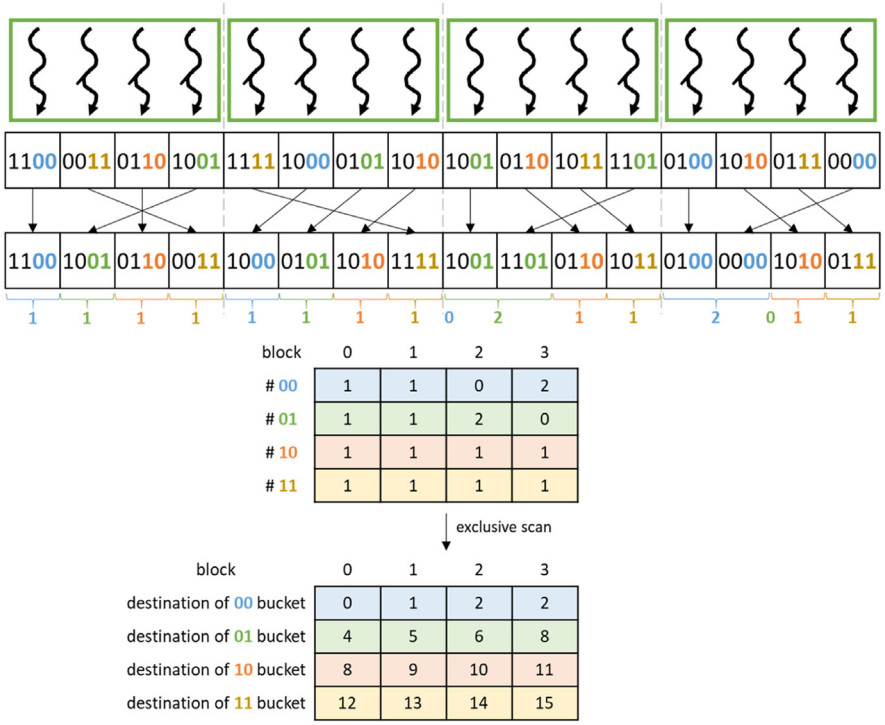


FIGURE 13.9

Finding the destination of each block's local buckets for a 2-bit radix.

13.6 Thread coarsening to improve coalescing

The price of parallelizing radix sort across many thread blocks is poor coalescing of writes to global memory. Each thread block has its own local buckets that it writes to global memory. Having more thread blocks means having fewer keys per thread block, which means that the local buckets are going to be smaller, exposing fewer opportunities for coalescing when they are written to global memory. If these thread blocks were to be executed in parallel, the price of poor coalescing might be worth paying. However, if these thread blocks were to be serialized by the hardware, the price would be paid unnecessarily.

To address this issue, thread coarsening can be applied whereby each thread is assigned to multiple keys in the input list instead of just one. Fig. 13.10 illustrates how thread coarsening can be applied to a radix sort iteration for the 2-bit radix example. In this case, each thread block is responsible for more keys than was the case in the example in Fig. 13.8. Consequently, the local buckets of each thread block are larger, exposing more opportunities for coalescing. When we compare

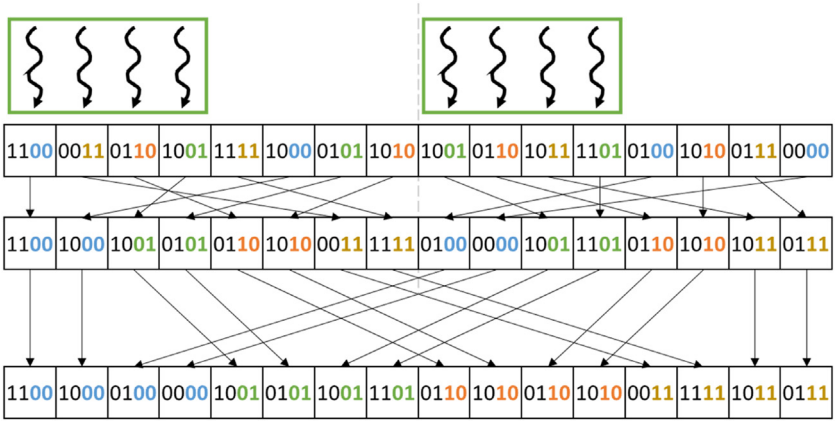


FIGURE 13.10

Radix sort for a 2-bit radix with thread coarsening to improve memory coalescing.

Fig. 13.8 and Fig. 13.10, it is clear that in Fig. 13.10 it is more likely the case that consecutive threads write to consecutive memory locations.

Another price for parallelizing radix sort across many thread blocks is the overhead of performing the global exclusive scan to identify the destination of each thread block's local buckets. Recall from Fig. 13.9 that the size of the table on which the exclusive scan is performed is proportional to the number of buckets as well as the number of blocks. By applying thread coarsening, the number of blocks is reduced, thereby reducing the size of the table and the overhead of the exclusive scan operation. We leave the application of thread coarsening to radix sort as an exercise for the reader.

13.7 Parallel merge sort

Radix sort is suitable when keys are to be sorted in lexicographic order. However, if keys are to be sorted on the basis of a complex order defined by a complex comparison operator, then radix sort is not suitable, and a comparison-based sorting algorithm is necessary. Moreover, an implementation of a comparison-based sorting algorithm can be more easily adapted to different types of keys by simply changing the comparison operator. In contrast, adapting an implementation of a noncomparison-based sorting algorithm such as radix sort to different types of keys may involve creating different versions of the implementation. These considerations may make comparison-based sorting more favorable in some cases, despite their higher complexity.

One comparison-based sort that is amenable to parallelization is merge sort. Merge sort works by dividing the input list into segments, sorting each segment

(using merge sort or another sorting algorithm), and then performing an ordered merge of the sorted segments.

Fig. 13.11 shows an example of how merge sort can be parallelized. Initially, the input list is divided into many segments, each of which are sorted independently, using some efficient sorting algorithm. After that, every pair of segments is merged into a single segment. This process is repeated until all the keys become part of the same segment.

At each stage, the computation can be parallelized by performing different merge operations in parallel as well as parallelizing within merge operations. In the earlier stages, there are more independent merge operations that can be performed in parallel. In the later stages, there are fewer independent merge operations, but each merge operation merges more keys, exposing more parallelism within the merge operation. For example, in Fig. 13.11 the first merge stage consists of four independent merge operations. Hence our grid of eight thread blocks may assign two thread blocks to process each merge operation in parallel. In the next stage, there are only two merge operations, but each operation merges twice the number of keys. Hence our grid of eight thread blocks may assign four thread blocks to process each merge operation in parallel. We saw how to parallelize a merge operation in Chapter 12, Merge. We leave the implementation of merge sort based on parallel merge as an exercise for the reader.

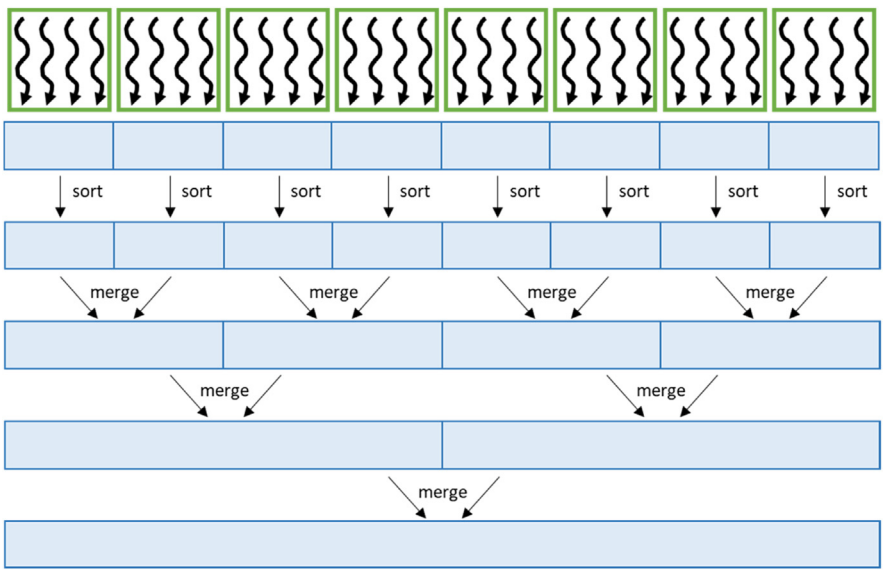


FIGURE 13.11

Parallelizing merge sort.

13.8 Other parallel sort methods

The algorithms outlined above are only two of the many possible ways to sort data in parallel. In this section, we briefly outline some of the other methods that may be of interest to readers.

Among the simplest of parallel sorting methods is the odd-even transposition sort. It begins by comparing, in parallel, every even/odd pair of keys, namely, those with indices k and k + 1 starting at the first *even* index. The position of the keys is swapped if the key at position k + 1 is less than the key at position k. This step is then repeated for every odd/even pair of keys, namely, those with indices k and k + 1 starting at the first *odd* index. These alternating phases are repeated until both are completed with no keys needing to be swapped. The odd-even transposition sort is quite similar to sequential bubble sort algorithms, and like bubble sort, it is inefficient on large sequences, since it may perform $O(N^2)$ work of a sequence of N elements.

Transposition sort uses a fixed pattern of comparisons and swaps elements when they are out of order. It is easily parallelized because each step compares pairs of keys that do not overlap. There is an entire category of sorting methods that use fixed patterns of comparison to sort sequences, often in parallel. These methods are usually referred to as *sorting networks*, and the best-known parallel sorting networks are Batcher's bitonic sort and odd-even merge sort (Batcher, 1968). Batcher's algorithms operate on sequences of fixed length and are more efficient than odd-even transposition sort, requiring only $O(N \cdot \log^2 N)$ comparisons for a sequence of N elements. Even though the cost of these algorithms is asymptotically worse than the $O(N \cdot \log N)$ cost of methods such as merge sort, in practice, they are often the most efficient methods on small sequences because of their simplicity.

Most comparison-based parallel sorts that do not use the fixed set of comparisons that are typical of sorting networks can be divided into two broad categories. The first partitions the unsorted input into tiles, sorts each tile, and then performs most of its work in combining these tiles to form the output. The merge sort that we described in this chapter is an example of such an algorithm; most of the work is performed in the merge tree that combines sorted tiles. The second category focuses most of its work on partitioning the unsorted sequence, such that combining partitions is relatively trivial. Sample sort algorithms (Frazer and McKellar, 1970) are the typical example of this category. Sample sort begins by selecting $p - 1$ keys from the input (e.g., at random), sorts them, and then uses them to partition the input into p buckets such that all keys in bucket k are greater than all keys in any bucket $j < k$ and less than all in any bucket $j > k$. This step is analogous to a p-way generalization of the two-way partitioning that is performed by quicksort. Having partitioned the data in this way, each bucket can be sorted independently, and the sorted output is formed by merely concatenating the buckets in order. Sample sort algorithms are often the most efficient choice for

extremely large sequences in which data must be distributed across multiple physical memories, including across the memories of multiple GPUs in a single node. In practice, oversampling the keys is common, since modest oversampling will result in balanced partitions with high probability (Blelloch et al., 1991).

Just as merge sort and sample sort typify bottom-up and top-down strategies for comparison-based sorting, radix sorting algorithms can be designed to follow a bottom-up or top-down strategy. The radix sort that we described in this chapter is more completely described as an LSB or, more generally, least significant digit (LSD), radix sort. The successive steps of the algorithm start with the LSD of the key and work toward the most significant digit (MSD). A MSD radix sort adopts the opposite strategy. It begins by using the MSD to partition the input into buckets that correspond to the possible values of that digit. This same partitioning is then applied independently in each bucket, using the next MSD. Upon reaching the LSD, the entire sequence will have been sorted. Like sample sort, MSD radix sort is often a better choice for very large sequences. Whereas the LSD radix sort requires global shuffling of data in each step, each step of MSD radix sort operates on progressively more localized regions of the data.

13.9 Summary

In this chapter we have seen how to sort keys (and their associated values) on GPUs in parallel. In most of the chapter we focused on radix sort, which sorts keys by distributing them across buckets. The distribution process is repeated for each digit in the key while preserving the order from the previous digit's iteration to ensure that the keys are sorted according to all the digits at the end. Each iteration is parallelized by assigning a thread to each key in the input list and having that thread find the destination of the key in the output list, which involves collaborating with other threads to perform an exclusive scan operation.

One of the key challenges in optimizing radix sort is achieving coalesced memory accesses in writing the keys to the output list. An important optimization to enhance coalescing is to have each thread block perform a local sort to local buckets in shared memory and then write each local bucket to global memory in a coalesced manner. Another optimization is to increase the size of the radix to reduce the number of iterations that are needed and thus the number of grids that are launched. However, the radix size should not be increased too much because it would result in poorer coalescing and more overhead from the global exclusive scan operation. Finally, applying thread coarsening is effective at improving memory coalescing as well as reducing the overhead of the global exclusive scan.

Radix sort has the advantage of having computation complexity that is lower than O(N*log*(N)). However, radix sort only works for limited types of keys such as integers. Therefore we also look at the parallelization of comparison-based sorting that is applicable to general types of keys. A class of comparison-based

sorting algorithms that is amenable to parallelization is merge sort. Merge sort can be parallelized by performing independent merge operations of different input segments in parallel as well as parallelizing within each merge operation, as we saw in Chapter 12, Merge.

The process of implementing and optimizing parallel sorting algorithms on GPUs is complex, and the average user is more likely to use a parallel sorting library for GPUs, such as Thrust (Bell and Hoberock, 2012), than to implement one's own sorting kernels from scratch. Nevertheless, parallel sorting remains an interesting case study of the tradeoffs that go into optimizing parallel patterns.

Exercises

1. Extend the kernel in Fig. 13.4 by using shared memory to improve memory coalescing.
2. Extend the kernel in Fig. 13.4 to work for a multibit radix.
3. Extend the kernel in Fig. 13.4 by applying thread coarsening to improve memory coalescing.
4. Implement parallel merge sort using the parallel merge implementation from Chapter 12, Merge.

References

Batcher, K.E., 1968. Sorting networks and their applications. In: Proceedings of the AFIPS Spring Joint Computer Conference.

Bell, N., Hoberock, J., 2012. Thrust: a productivity-oriented library for CUDA. GPU Computing Gems Jade Edition. Morgan Kaufmann, pp. 359–371.

Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagha, M., 1991, A comparison of sorting algorithms for the Connection Machine CM-2. In: Proceedings of the Third ACM Symposium on Parallel Algorithms and Architectures.

Frazer, W.D., McKellar, A.C., 1970. Samplesort: a sampling approach to minimal storage tree sorting. Journal of ACM 17 (3).

Satish, N., Harris M., Garland, M., 2009. Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing.

CHAPTER

14 Sparse matrix computation

Chapter Outline

Our next parallel pattern is sparse matrix computation. In a sparse matrix the majority of the elements are zeros. Storing and processing these zero elements are wasteful in terms of memory capacity, memory bandwidth, time, and energy. Many important real-world problems involve sparse matrix computation. Because of the importance of these problems, several sparse matrix storage formats and their corresponding processing methods have been proposed and widely used in the field. All these methods employ some type of compaction techniques to avoid storing or processing zero elements at the cost of introducing some level of irregularity into the data representation. Unfortunately, such irregularity can lead to underutilization of memory bandwidth, control flow divergence, and load imbalance in parallel computing. It is therefore important to strike a good balance between compaction and regularization. Some storage formats achieve a higher level of compaction at a high level of irregularity. Others achieve a more modest level of compaction while keeping the representation more regular. The relative performance of a parallel computation using each storage format is known to be heavily dependent on the distribution of nonzero elements in the sparse matrices. Understanding the wealth of work in sparse matrix storage formats and their corresponding parallel algorithms gives a parallel programmer important background for addressing compaction and regularization challenges in solving related problems.

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00010-0

14.1 Background

A sparse matrix is a matrix in which most of the elements are zeros. Sparse matrices arise in many scientific, engineering, and financial modeling problems. For example, matrices can be used to represent the coefficients in a linear system of equations. Each row of the matrix represents one equation of the linear system. In many science and engineering problems the large number of variables and equations that are involved are sparsely coupled. That is, each equation involves only a small number of variables. This is illustrated in Fig. 14.1, in which each column of the matrix corresponds to the coefficients for a variable: column 0 for x_0, column 1 for x_1, and so on. For example, the fact that row 0 has nonzero elements in columns 0 and 1 indicates that only variables x_0 and x_1 are involved in equation 0. It should be clear that variables x_0, x_2, and x_3 are present in equation 1, variables x_1 and x_2 are present in equation 2, and only variable x_3 is present in equation 3. Sparse matrices are typically stored in a format, or representation, that avoids storing zero elements.

Matrices are often used in solving linear systems of N equations of N variables in the form of $A^*X + Y = 0$, where A is an $N \times N$ matrix, X is a vector of N variables, and Y is a vector of N constant values. The objective is to solve for the X variable values that will satisfy all the equations. An intuitive approach is to invert the matrix so that $X = A^{-1} * (-Y)$. This can be done for matrices of moderate size through methods such as Gaussian elimination. While it is theoretically possible to use these methods to solve the equations that are represented by sparse matrices, the sheer size of many sparse matrices can overwhelm this intuitive approach. Furthermore, an inverse sparse matrix is often much larger than the original because the inversion process tends to generate many additional nonzero elements, which are called "fill-ins." As a result, it is often impractical to compute and store the inverse matrix in solving real-world problems.

Linear systems of equations represented in sparse matrices can be better solved with an iterative approach. When the sparse matrix A is positive-definite (i.e., $x^TAx > 0$ for all nonzero vectors x in R^n), one can use conjugate gradient methods to iteratively solve the corresponding linear system with guaranteed convergence to a solution (Hestenes and Stiefel, 1952). Conjugate gradient methods guess a solution for X, and perform $A^*X + Y$, and see whether the result is close

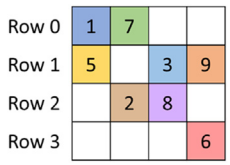


FIGURE 14.1

A simple sparse matrix example.

to a 0 vector. If not, we can use a gradient vector formula to refine the guessed X and perform another iteration of A*X + Y. These iterative solution methods for linear systems are closely related to the iterative solution methods for differential equations that we introduced in Chapter 8, Stencil.

The most time-consuming part of iterative approaches to solving linear systems of equations is in the evaluation of A*X + Y, which is a sparse matrix-vector multiplication and accumulation. Fig. 14.2 shows a small example of matrix-vector multiplication and accumulation in which A is a sparse matrix. The dark squares in A represent nonzero elements. In contrast, both X and Y are typically dense vectors. That is, most of the elements of X and Y hold nonzero values. Owing to its importance, standardized library function interfaces have been created to perform this operation under the name SpMV (sparse matrix vector multiplication and accumulation). We will use SpMV to illustrate the important tradeoffs between different storage formats in parallel sparse matrix computation.

The main objective of the different sparse matrix storage formats is to remove all the zero elements from the matrix representation. Removing all the zero elements not only saves storage but also eliminates the need to fetch these zero elements from memory and perform useless multiplication or addition operations with them. This can significantly reduce the consumption of memory bandwidth and computation resources.

There are various design considerations that go into the structure of a sparse matrix storage formats. The following is a list of some of the key considerations:

- Space efficiency (or compaction): the amount of memory capacity that is required to represent the matrix using the storage format
- Flexibility: the extent to which the storage format makes it easy to modify the matrix by adding or removing nonzeros
- Accessibility: the kinds of data that the storage format makes it easy to access
- Memory access efficiency: the extent to which the storage format enables an efficient memory access pattern for a particular computation (one facet of regularization)
- Load balance: the extent to which the storage format balances the load across different threads for a particular computation (another facet of regularization)

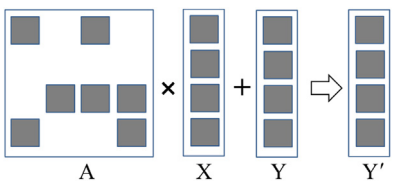


FIGURE 14.2

A small example of matrix-vector multiplication and accumulation.

Throughout this chapter we will introduce different storage formats and examine how these storage formats compare in each of these design considerations.

14.2 A simple SpMV kernel with the COO format

The first sparse matrix storage format that we will discuss is the coordinate list (COO) format. The COO format is illustrated in Fig. 14.3. COO stores the nonzero values in a one-dimensional array, which is shown as the `value` array. Each nonzero element is stored with both its column index and its row index. We have both `colIdx` and `rowIdx` arrays to accompany the `value` array. For example, A[0,0] of our small example is stored at the entry with index 0 in the `value` array (1 in `value[0]`) with both its column index (0 in `colIdx[0]`) and its row index (0 in `rowIdx[0]`) stored at the same position in the other arrays.

COO completely removes all zero elements from the storage. It does incur storage overhead by introducing the `colIdx` and `rowIdx` arrays. In our small example, in which the number of zero elements is not much larger than the number of nonzero elements, the storage overhead is actually more than the space that is saved by not storing the zero elements. However, it should be clear that for sparse matrices in which the vast majority of elements are zeros, the overhead that is introduced is far less than the space that is saved by not storing zeros. For example, in a sparse matrix in which only 1% of the elements are nonzero values, the total storage for the COO representation, including all the overhead, would be around 3% of the space required to store both zero and nonzero elements.

One approach to performing SpMV in parallel using a sparse matrix represented in the COO format is to assign a thread to each nonzero element in the matrix. An example of this parallelization approach is illustrated in Fig. 14.4, and the corresponding code is shown in Fig. 14.5. In this approach, each thread identifies the index of the nonzero element for which it is responsible (line 02) and ensures that it is within bounds (line 03). Next, the thread identifies the row index (line 04), column index (line 05), and value (line 06) of the nonzero element for which it is responsible from the `rowIdx`, `colIdx`, and `value` arrays, respectively. It then looks up the input vector value at the location corresponding to the column index, multiplies it by the nonzero value, then accumulates the result to the output

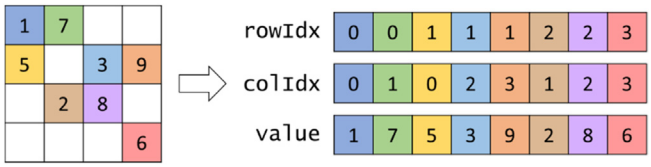


FIGURE 14.3

Example of the coordinate list (COO) format.

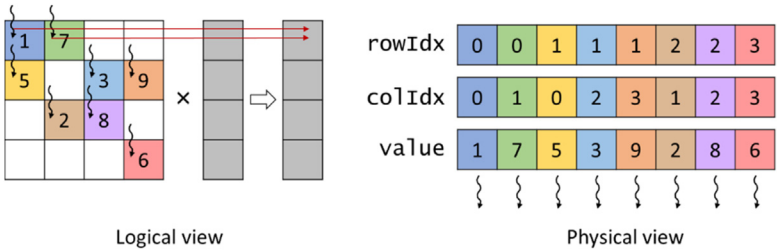


FIGURE 14.4

Example of parallelizing SpMV with the COO format.

```
01    __global__ void spmv_coo_kernel(COOMatrix cooMatrix, float* x, float* y) {
02        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03        if(i < cooMatrix.numNonzeros) {
04            unsigned int row = cooMatrix.rowIdx[i];
05            unsigned int col = cooMatrix.colIdx[i];
06            float value = cooMatrix.value[i];
07            atomicAdd(&y[row], x[col]*value);
08        }
09    }
```

FIGURE 14.5

A parallel SpMV/COO kernel.

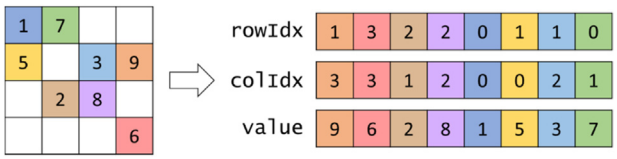


FIGURE 14.6

Reordering coordinate list (COO) format.

value at the corresponding row index (line 07). An atomic operation is used for the accumulation because multiple threads may update the same output element, as is the case with the first two threads mapped to row 0 of the matrix in Fig. 14.4. It should be obvious that any SpMV computation code will reflect the storage format assumed. Therefore we add the storage format to the name of the kernel to clarify the combination that was used. We also refer to the SpMV code in Fig. 14.5 as SpMV/COO.

Now we examine the COO format under the design considerations listed in Section 14.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency we defer the discussion for later, when we have introduced other formats. For flexibility we observe that we can arbitrarily reorder the elements in a COO format without losing any information as long as we reorder the `rowIdx`, `colIdx`, and `value` arrays in the same way. This is illustrated by using our small example in Fig. 14.6, where we have reordered the

elements of `rowIdx`, `colIdx`, and `value`. Now `value[0]` actually contains an element from row 1 and column 3 of the original sparse matrix. Because we have also shifted the row index and column index values along with the data value, we can correctly identify this element's location in the original sparse matrix.

In the COO format, we can process the elements in any order we want. The correct `y` element that is identified by `rowIdx[i]` will receive the correct contribution from the product of `value[i]` and `x[colIdx[i]]`. If we make sure that we somehow perform this operation for all elements of `value`, we will calculate the correct final answer regardless of the order in which we process these elements.

The reader may ask why we would want to reorder these elements. One reason is that the data may be read from a file that does not provide the nonzeros in a particular order, and we still need a consistent way of representing the data. For this reason, COO is a popular choice of storage format when the matrix is initially being constructed. Another reason is that not having to provide any ordering enables nonzeros to be added to the matrix by simply appending entries at the end of each of the three arrays. For this reason, COO is a popular choice of storage format when the matrix is modified throughout the computation. We will see another benefit of the flexibility of the COO format in Section 14.5.

The next design consideration that we look at is accessibility. COO makes it easy to access, for a given nonzero, its corresponding row index and column index. This feature of COO enables parallelization across nonzero elements in SpMV/COO. On the other hand, COO does not make it easy to access, for a given row or column, all the nonzeros in that row or column. For this reason, COO would not be a good choice of format if the computation required a row-wise or column-wise traversal of the matrix.

For memory access efficiency we refer to the physical view in Fig. 14.4 for how the threads access the matrix data from memory. The access pattern is such that consecutive threads access consecutive elements in each of the three arrays that form the COO format. Therefore accesses to the matrix by SpMV/COO are coalesced.

For load balance we recall that each thread is responsible for a single nonzero value. Hence all threads are responsible for the same amount of work, which means that we do not expect any control divergence to take place in SpMV/COO except for the threads at the boundary.

The main drawback of SpMV/COO is the need to use atomic operations. The reason for using atomic operations is that multiple threads are assigned to nonzeros in the same row and therefore need to update the same output value. The atomic operations can be avoided if all the nonzeros in the same row are assigned to the same thread such that the thread will be the only one updating the corresponding output value. However, recall that the COO format does not give this accessibility. In the COO format, it is not easy to access, for a given row, all the nonzeros in that row. In the next section we will see another storage format that provides this accessibility.

14.3 Grouping row nonzeros with the CSR format

In the previous section we saw that parallelizing SpMV with the COO format suffers from the use of atomic operations because the same output value is updated by multiple threads. These atomic operations can be avoided if the same thread is responsible for all the nonzeros of a row, which requires the storage format to give us the ability to access, for a given row, all the nonzeros in that row. This kind of accessibility is provided by the compressed sparse row (CSR) storage format.

Fig. 14.7 illustrates how the matrix in Fig. 14.1 can be stored by using the CSR format. Like the COO format, CSR stores the nonzero values in a one-dimensional array shown as the `value` array in Fig. 14.2. However, these nonzero values are grouped by row. For example, we store the nonzero elements of row 0 (1 and 7) first, followed by the nonzero elements of row 1 (5, 3, and 9), followed by the nonzero elements of row 2 (2 and 8), and finally the nonzero elements of row 3 (6).

Also similar to the COO format, CSR stores for each nonzero element in the `value` array its column index at the same position in the `colIdx` array. Naturally, these column indices are grouped by row as the values are. In Fig. 14.7 the nonzeros of each row are sorted by their column indices in increasing order. Sorting the nonzeros in this way results in favorable memory access patterns, but it is not necessary. The nonzeros within each row may not necessarily be sorted by their column index, and the kernel that is presented in this section will still work correctly. When the nonzeros within each row are sorted by their column index, the layout of the `value` array (and the `colIdx` array) for CSR can be viewed as the row-major layout of the matrix after eliminating all the zero elements.

The key distinction between the COO format and the CSR format is that the CSR format replaces the `rowIdx` array with a `rowPtrs` array that stores the starting offset of each row's nonzeros in the `colIdx` and `value` arrays. In Fig. 14.7 we show a `rowPtrs` array whose elements are the indices for the beginning locations of each row. That is, `rowPtrs[0]` indicates that row 0 starts at location 0 of the `value` array, `rowPtrs[1]` indicates that row 1 starts at location 2, and so on. Note that `rowPtrs[4]` stores the starting location of a nonexistent "row 4." This is for convenience, as some algorithms need to use the starting location of the next row

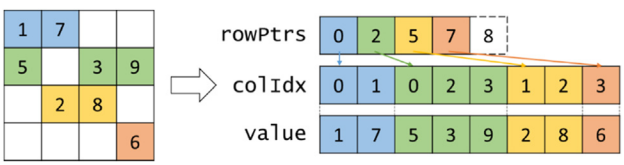


FIGURE 14.7

Example of compressed sparse row (CSR) format.

to delineate the end of the current row. This extra marker gives a convenient way to locate the ending location of row 3.

To perform SpMV in parallel using a sparse matrix represented in the CSR format, one can assign a thread to each row of the matrix. An example of this parallelization approach is illustrated in Fig. 14.8, and the corresponding code is shown in Fig. 14.9. In this approach, each thread identifies the row that it is responsible for (line 02) and ensures that it is within bounds (line 03). Next, the thread loops through the nonzero elements of its row to perform the dot product (lines 05–06). To find the row's nonzero elements, the thread looks up their starting index in the `rowPtrs` array (`rowPtrs[row]`). It also finds where they end by looking up the starting index of the next row's nonzeros (`rowPtrs[row + 1]`). For each nonzero element the thread identifies its column index (line 07) and value (line 08). It then looks up the input value at the location corresponding to the column index, multiplies it by the nonzero value, and accumulates the result to a local variable `sum` (line 09). The `sum` variable is initialized to 0 before the dot product loop begins (line 04) and is accumulated to the output vector after the loop is over (line 11). Notice that the accumulation of the `sum` to the output vector does not require atomic operations. The reason is that each row is traversed by a single thread, so each thread will write to a distinct output value, as illustrated in Fig. 14.8.

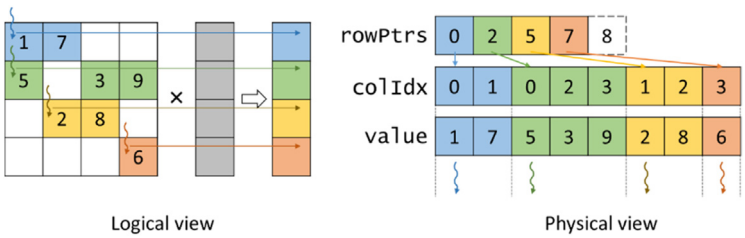


FIGURE 14.8

Example of parallelizing SpMV with the CSR format.

```
01  __global__ void spmv_csr_kernel(CSRMatrix csrMatrix, float* x, float* y) {
02     unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;
03     if(row < csrMatrix.numRows) {
04       float sum = 0.0f;
05       for(unsigned int i=csrMatrix.rowPtrs[row]; i<csrMatrix.rowPtrs[row+1];
06                                                                    ++i) {
07          unsigned int col = csrMatrix.colIdx[i];
08          float value = csrMatrix.value[i];
09          sum += x[col]*value;
10       }
11       y[row] += sum;
12     }
13  }
```

FIGURE 14.9

A parallel SpMV/CSR kernel.

Now we examine the CSR format under the design considerations listed in Section 14.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency we observe that CSR is more space efficient than COO. COO requires three arrays, `rowIdx`, `colIdx`, and `value`, each of which has as many elements as the number of nonzeros. In contrast, CSR requires only two arrays, `colIdx` and `value`, with as many elements as the number of nonzeros. The third array, `rowPtrs`, requires only as many elements as the number of rows plus one, which makes it substantially smaller than the `rowIdx` array in COO. This difference makes CSR more space efficient than COO.

For flexibility we observe that CSR is less flexible than COO when it comes to adding nonzeros to the matrix. In COO a nonzero can be added by simply appending it to the ends of the arrays. In CSR a nonzero to be added must be added to the specific row to which it belongs. This means that the nonzero elements of the later rows would all need to be shifted, and the row pointers of the later rows would all need to be incremented accordingly. For this reason, adding nonzeros to a CSR matrix is substantially more expensive than adding them to a COO matrix.

For accessibility, CSR makes it easy to access, for a given row, the nonzeros in that row. This feature of CSR enables parallelization across rows in SpMV/CSR, which is what allows it to avoid atomic operations in comparison with SpMV/COO. In a real-world sparse matrix application there are usually thousands to millions of rows, each of which contains tens to hundreds of nonzero elements. This makes parallelization across rows seem very appropriate: There are many threads, and each thread has a substantial amount of work. On the other hand, for some applications the sparse matrix may not have enough rows to fully utilize all the GPU threads. In these kinds of applications the COO format can extract more parallelism, since there are more nonzeros than rows. Moreover, CSR does not make it easy to access, for a given column, all the nonzeros in that column. Thus an application may need to maintain an additional, more column-oriented layout of the matrix if easy access to all elements of a column is needed.

For memory access efficiency we refer to the physical view in Fig. 14.8 for how the threads access the matrix data from memory during the first iteration of the dot product loop. The access pattern is such that consecutive threads access data elements that are far apart. In particular, threads 0, 1, 2, and 3 will access `value[0]`, `value[2]`, `value[5]`, and`value[7]`, respectively, in the first iteration of their dot product loop. They will then access `value[1]`, `value[3]`, `value[6]`, and no data, respectively, in the second iteration, and so on. As a result, the accesses to the matrix by the parallel SpMV/CSR kernel in Fig. 14.9 are not coalesced. The kernel does not make efficient use of memory bandwidth.

For load balance we observe that the SpMV/CSR kernel can potentially have significant control flow divergence in all warps. The number of iterations that are taken by a thread in the dot product loop depends on the number of nonzero elements in the row that is assigned to the thread. Since the distribution of nonzero elements among rows can be random, adjacent rows can have very different

number of nonzero elements. As a result, there can be widespread control flow divergence in most or even all warps.

In summary, we have seen that the advantages of CSR over COO are that it has better space efficiency and that it gives us access to all the nonzeros of a row, allowing us to avoid atomic operations by parallelizing the computation across rows in SpMV/CSR. On the other hand, the disadvantages of CSR over COO are that it provides less flexibility with adding nonzero elements to the sparse matrix, it exhibits a memory access pattern that is not amenable to coalescing, and it causes high control divergence. In the following sections we discuss additional storage formats that sacrifice some space efficiency as compared to CSR in order to improve memory coalescing and reduce control divergence. Note that converting from COO to CSR on the GPU is an excellent exercise for the reader, using multiple fundamental parallel computing primitives, including histogram and prefix sum.

14.4 Improving memory coalescing with the ELL format

The problem of noncoalesced memory accesses can be addressed by applying data padding and transposition on the sparse matrix data. These ideas were used in the ELL storage format, whose name came from the sparse matrix package in ELLPACK, a package for solving elliptic boundary value problems (Rice and Boisvert, 1984).

A simple way to understand the ELL format is to start with the CSR format, as is illustrated in Fig. 14.10. From a CSR representation that groups nonzeros by row, we determine the rows with the maximal number of nonzero elements. We

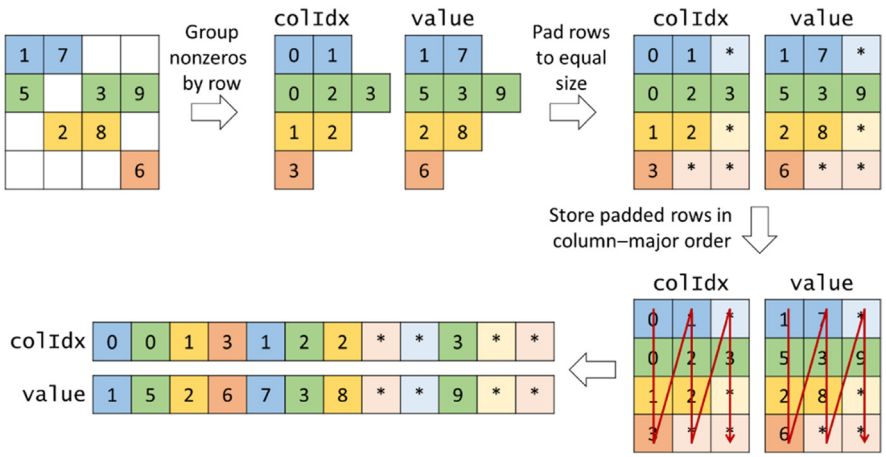


FIGURE 14.10

Example of ELL storage format.

then add padding elements to all other rows after the nonzero elements to make them the same length as the maximal rows. This makes the matrix a rectangular matrix. For our small sparse matrix example we determine that row 1 has the maximal number of elements. We then add one padding element to row 0, one padding element to row 2, and two padding elements to row 3 to make all them the same length. These additional padding elements are shown as squares with an * in Fig. 14.11. Now the matrix has become a rectangular matrix. Note that the `colIdx` array also needs to be padded the same way to preserve its correspondence to the `values` array.

We can now lay the padded matrix out in column-major order. That is, we will place all elements of column 0 in consecutive memory locations, followed by all elements of column 1, and so on. This is equivalent to transposing the rectangular matrix in the row-major order used by the C language. In terms of our small example, after the transposition, `value[0]` through `value[3]` now contain 1, 5, 2, 6, which are the 0th elements of all rows. This is illustrated in the bottom left portion of Fig. 14.10. Similarly, `colIdx[0]` through `colIdx[3]` contain the column positions of 0th elements of all rows. Note that we no longer need the `rowPtrs`, since the beginning of row `r` is now simply `value[r]`. With the padded elements it is also very easy to move from the current element of row `r` to the next element by simply adding the number of rows in the original matrix to the index. For example, the 0th element of row 2 is in `value[2]`, and the next element is in `value[2+4]`, which is equivalent to `value[6]`, where 4 is the number of rows in the original matrix in our small example.

We illustrate how to parallelize SpMV using the ELL format in Fig. 14.11, along with a parallel SpMV/ELL kernel in Fig. 14.12. Like CSR, each thread is assigned to a different row of the matrix (line 02), and a boundary check ensures that the row is within bounds (line 03). Next, a dot product loop goes through the nonzero elements of each row (line 05). Note that the SpMV/ELL kernel assumes that the input matrix has a vector `ellMatrix.nnzPerRow` that records the number of nonzeros in each row and allows each thread to iterate only through the nonzeros in its assigned row. If the input matrix does not have this vector, the kernel can simply iterate through all elements, including the padding elements, and still

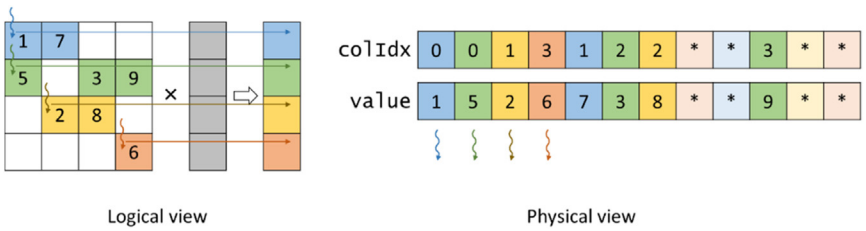


FIGURE 14.11

Example of parallelizing SpMV with the ELL format.

```
01    __global__ void spmv_ell_kernel(ELLMatrix ellMatrix, float* x, float* y) {
02        unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;
03        if(row < ellMatrix.numRows) {
04            float sum = 0.0f;
05            for(unsigned int t = 0; t < ellMatrix.nnzPerRow[row]; ++t) {
06                unsigned int i = t*ellMatrix.numRows + row;
07                unsigned int col = ellMatrix.colIdx[i];
08                float value = ellMatrix.value[i];
09                sum += x[col]*value;
10            }
11            y[row] = sum;
12        }
13    }
```

FIGURE 14.12

A parallel SpMV/ELL kernel.

execute correctly, since the padding elements have value zero and will not affect the output values. Next, since the compressed matrix is stored in column-major order, the index `i` of the nonzero element in the one-dimensional array can be found by multiplying the iteration number `t` by the number of rows and adding the row index (line 06). Next, the thread loads the column index (line 07) and nonzero value (line 08) from the ELL matrix arrays. Note that the accesses to these arrays are coalesced because the index `i` is expressed in terms of `row`, which itself is expressed in terms of `threadIdx.x`, meaning that consecutive threads have consecutive array indices. Next, the thread looks up the input value, multiplies it by the nonzero value, and accumulates the result to a local variable `sum` (line 09). The `sum` variable is initialized to 0 before the dot product loop begins (line 04) and is accumulated to the output vector after the loop is over (line 11).

Now we examine the ELL format under the design considerations listed in Section 14.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency we observe that the ELL format is less space efficient than the CSR format, owing to the space overhead of the padding elements. The overhead of the padding elements highly depends on the distribution of nonzeros in the matrix. In situations in which one or a small number of rows have an exceedingly large number of nonzero elements, the ELL format will result in an excessive number of padded elements. Consider our sample matrix; in the ELL format we have replaced a 4×4 matrix with a 4×3 matrix, and with the overhead from the column indices we are storing more data than was contained in the original 4×4 matrix. For a more realistic example, if a 1000×1000 sparse matrix has 1% of its elements of nonzero value, then on average, each row has ten nonzero elements. With the overhead, the size of a CSR representation would be about 2% of the uncompressed total size. Assume that one of the rows has 200 nonzero values while all other rows have less than 10. Using the ELL format, we will pad all other rows to 200 elements. This makes the ELL representation about 40% of the uncompressed total size and 20 times larger than the CSR representation. This calls for a method to control the number of padded elements when we convert from the CSR format to the ELL format, which we will introduce in the next section.

For flexibility we observe that ELL is more flexible than CSR when it comes to adding nonzeros to the matrix. In CSR, adding a nonzero to a row would require shifting all the nonzeros of the subsequent rows and incrementing their row pointers. However, in ELL, as long as a row does not have the maximum number of nonzeros in the matrix, a nonzero can be added to the row by simply replacing a padding element with an actual value.

For accessibility, ELL gives us the accessibility of both CSR and COO. We saw in Fig. 14.12 how ELL allows us to access, given a row index, the nonzeros of that row. However, ELL also allows us to access, given the index of a nonzero element, the row and column index of that element. The column index is trivial to find, as it can be accessed from the `colIdx` array at the same location `i`. However, the row index can also be accessed, owing to the regular nature of the padded matrix. Recall that the index `i` of the nonzero element was calculated in Fig. 14.9 as follows:

```
i = t*ellMatrix.numRows + row
```

Therefore if instead `i` is given and we would like to find `row`, it can be found as follows:

```
row = i%ellMatrix.numRows
```

because `row` is always less than `ellMatrix.numRows`, so `row%ellMatrix.numRows` is simply `row` itself. This accessibility of ELL allows parallelization across both rows as well as nonzero elements.

For memory access efficiency we refer to the physical view in Fig. 14.11 for how the threads access the matrix data from memory during the first iteration of the dot product loop. The access pattern is such that consecutive threads access consecutive data elements. With the elements arranged in column-major order, all adjacent threads are now accessing adjacent memory locations, enabling memory coalescing and thus making more efficient use of memory bandwidth. Some GPU architectures, especially in the older generations, have more strict address alignment rules for memory coalescing. One can force each iteration of the SpMV/ELL kernel to be fully aligned to architecturally specified alignment units such as 64 bytes by adding a few rows to the end of the matrix before transposition.

For load balance we observe that SpMV/ELL still exhibits the same load imbalance as SpMV/CSR because each thread still loops over the number of nonzeros in the row for which it is responsible. Therefore ELL does not address the problem of control divergence.

In summary, the ELL format improves on the CSR format by allowing more flexibility to add nonzeros by replacing padding elements, better accessibility, and, most important, more opportunities for memory coalescing in SpMV/ELL. However, ELL has worse space efficiency than CSR, and the control divergence of SpMV/ELL is as bad as that of SpMV/CSR. In the next section we will see how we can improve on the ELL format to address the problems of space efficiency and control divergence.

14.5 Regulating padding with the hybrid ELL-COO format

The problems of low space efficiency and control divergence in the ELL format are most pronounced when one or a small number of rows have exceedingly large number of nonzero elements. If we have a mechanism to "take away" some elements from these rows, we can reduce the number of padded elements in ELL and also reduce the control divergence. The answer lies in an important use case for the COO format.

The COO format can be used to curb the length of rows in the ELL format. Before we convert a sparse matrix to ELL, we can take away some of the elements from the rows with exceedingly large numbers of nonzero elements and place the elements into a separate COO storage. We can use SpMV/ELL on the remaining elements. With the excess elements removed from the extra-long rows, the number of padded elements for other rows can be significantly reduced. We can then use a SpMV/COO to finish the job. This approach of employing two formats to collaboratively complete a computation is often referred to as a hybrid method.

Fig. 14.13 illustrates how an example matrix can be represented using the hybrid ELL-COO format. We see that in the ELL format alone, rows 1 and 6 have the largest number of nonzero elements, causing the other rows to have excessive padding. To address this issue, we remove the last three nonzero elements of row 2 and the last two nonzero elements of row 6 from the ELL representation and move them into a separate COO representation. By removing these elements, we reduce the maximal number of nonzero elements among all rows in the small sparse matrix from 5 to 2. As shown in Fig. 14.13, we reduced the number of padded elements from 22 to 3. More important, all threads now need to take only two iterations.

The reader may wonder whether the additional work done to separate COO elements from an ELL format could incur too much overhead. The answer is that it depends. In situations in which a sparse matrix is used in only one SpMV calculation, this extra work can indeed incur significant overhead. However, in many real-work applications, the SpMV is performed on the same sparse kernel

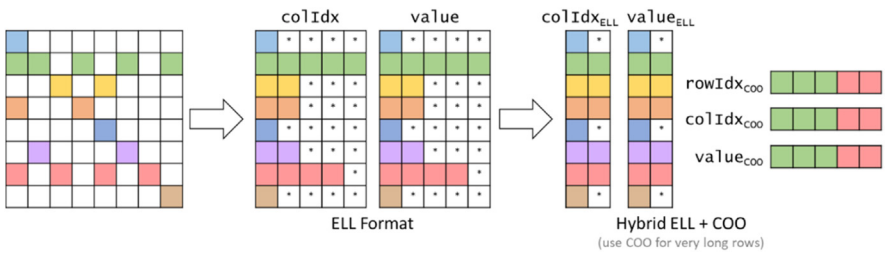


FIGURE 14.13

Hybrid ELL-COO example.

repeatedly in an iterative solver. In each iteration of the solver the x and y vectors vary, but the sparse matrix remains the same, since its elements correspond to the coefficients of the linear system of equations being solved, and these coefficients do not change from iteration to iteration. Therefore the work done to produce both the hybrid ELL and COO representation can be amortized across many iterations. We will come back to this point in the next section.

Now we examine the hybrid ELL-COO format under the design considerations listed in Section 14.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency we observe that the hybrid ELL-COO format has better space efficiency than the ELL format alone because it reduces the amount of padding used.

For flexibility we observe that the hybrid ELL-COO format is more flexible than just ELL when it comes to adding nonzeros to the matrix. With ELL we can add nonzero elements by replacing padding elements for rows that have them. With hybrid COO-ELL we can also add nonzeros by replacing padding elements. However, we can also append nonzeros to the COO part of the format if the row does not have any padding elements that can be replaced in the ELL part.

For accessibility we observe that the hybrid ELL-COO format sacrifices accessibility compared to the ELL format alone. In particular, it is not always possible to access, given a row index, all the nonzeros in that row. Such an access can be done only for the rows that fit in the ELL part of the format. If the row overflows to the COO part, then finding all the nonzeros of the row would require searching the COO part, which is expensive.

For memory access efficiency, both SpMV/ELL and SpMV/COO exhibit coalesced memory accesses to the sparse matrix. Hence their combination will also result in a coalesced access pattern.

For load balance, removing nonzeros from the long rows in the ELL part of the format reduces control divergence of the SpMV/ELL kernel. These nonzeros are placed in the COO part of the format, which does not affect control divergence, since SpMV/COO does not exhibit control divergence, as we have seen.

In summary, the hybrid ELL-COO format, in comparison with the ELL format alone, improves space efficiency by reducing padding, provides more flexibility for adding nonzeros to the matrix, retains the coalesced memory access pattern, and reduces control divergence. The price that is paid is a small limitation in accessibility, in which it becomes more difficult to access all the nonzeros of a given row if that row overflows to the COO part of the format.

14.6 Reducing control divergence with the JDS format

We have seen that the ELL format can be used to achieve coalesced memory access patterns when accessing the sparse matrix in SpMV and that the hybrid ELL-COO format can further improve space efficiency by reducing padding and

can also reduce control divergence. In this section we will look at another format that can achieve coalesced memory access patterns in SpMV and also reduce control divergence without the need to perform any padding. The idea is to sort the rows according to their length, say, from the longest to the shortest. Since the sorted matrix looks largely like a triangular matrix, the format is often referred to as the jagged diagonal storage (JDS) format.

Fig. 14.14 illustrates how a matrix can be stored using the JDS format. First, the nonzeros are grouped by row, as in the CSR and ELL formats. Next, the rows are sorted by the number of nonzeros in each row in increasing order. As we sort the rows, we typically maintain an additional `row` array that preserves the index of the original row. Whenever we exchange two rows in the sorting process, we also exchange the corresponding elements of the `row` array. Thus we can always keep track the original position of all rows. After the rows have been sorted, the nonzeros in the `value` array and their corresponding column indices in the `colIdx` array are stored in column-major order. A `iterPtr` array is added to track the beginning of the nonzero elements for each iteration.

Fig. 14.15 illustrates how SpMV can be parallelized by using the JDS format. Each thread is assigned to a row of the matrix and iterates through the nonzeros of that row, performing the dot product along the way. The threads use the `iterPtr` array to identify where the nonzeros of each iteration begin. It should be clear from the physical view on the right side of Fig. 14.15, which depicts the first iteration for each thread, that the threads access the nonzeros and column indices in the JDS arrays in a coalesced manner. The code for implementing SpMV/JDS is left as an exercise.

In another variation of the JDS format, the rows, after being sorted, can be partitioned into sections of rows. Since the rows have been sorted, all rows in a

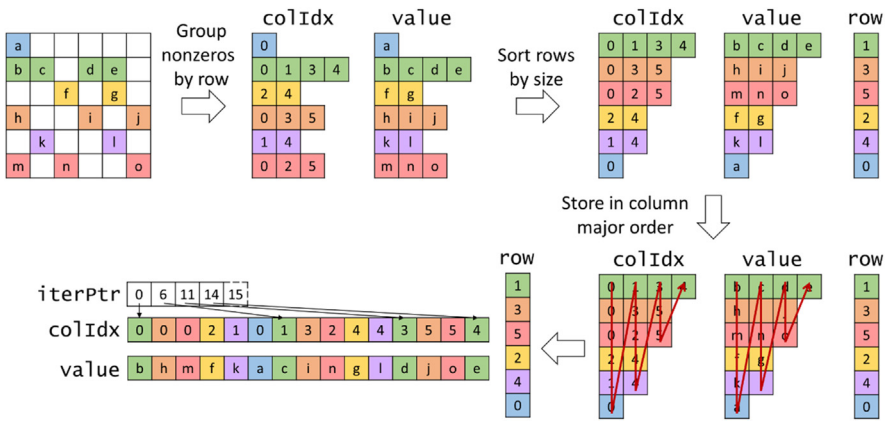


FIGURE 14.14

Example of JDS storage format.

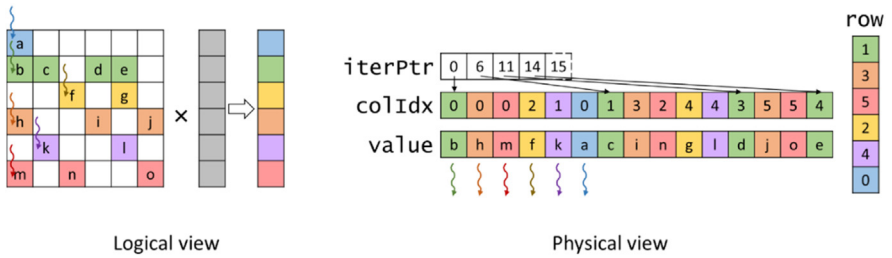


FIGURE 14.15

Example of parallelizing SpMV with the JDS format.

section will likely have more or less uniform numbers of nonzero elements. We can then generate the ELL representation for each section. Within each section we need to pad the rows only to match the row with the maximum number of elements in that section. This would reduce the number of padding elements substantially in comparison to one ELL representation of the entire matrix. In this variation of JDS, the `iterPtr` array would not be needed. Instead, one would need a section pointer array that points to the beginning of each ELL section only (as opposed to each iteration).

The reader should ask whether sorting rows will result in incorrect solutions to the linear system of equations. Recall that we can freely reorder equations of a linear system without changing the solution. As long as we reorder the y elements along with the rows, we are effectively reordering the equations. Therefore we will end up with the correct solution. The only extra step is to reorder the final solution back to the original order using the `row` array. The other question is whether sorting will incur significant overhead. The answer is similar to what we saw in the hybrid ELL-COO method. As long as the SpMV/JDS kernel is used in an iterative solver, one can afford to perform such sorting as well as the reordering of the final solution `x` elements and amortize the cost among many iterations of the solver.

Now we examine the ELL format under the design considerations listed in Section 14.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency the JDS format is more space efficient than the ELL format because it avoids padding. The variant of JDS that uses ELL for each section has padding, but the amount of padding is less than that with the ELL format.

For flexibility the JDS format does not make it easy to add nonzeros to a row of the matrix. It is even less flexible than the CSR format because adding nonzeros changes the sizes of the rows, which may require rows to be resorted.

For accessibility the JDS format is similar to the CSR format in that it allows us to access, given a row index, the nonzero elements of that row. On the other hand, it does not make it easy to access, given a nonzero, the row index and column index of that nonzero, as the COO and ELL formats do.

For memory access efficiency the JDS format is like the ELL format in that it stores the nonzeros in column-major order. Accordingly, the JDS format enables accesses to the sparse matrix to happen in a coalesced manner. Because JDS does not require padding, the starting location of memory accesses in each iteration, as shown in the physical view of Fig. 14.15, can vary in arbitrary ways. As a result, there is no simple, inexpensive way to force all iterations of the SpMV/JDS kernel to start at architecturally specified alignment boundaries. This lack of option to force alignment can make memory accesses for JDS less efficient than those in ELL.

For load balance, the unique feature of JDS is that it sorts the rows of the matrix such that threads in the same warp are likely to iterate over rows of similar length. Therefore JDS is effective at reducing control divergence.

14.7 Summary

In this chapter we presented sparse matrix computation as an important parallel pattern. Sparse matrices are important in many real-world applications that involve modeling complex phenomenon. Furthermore, sparse matrix computation is a simple example of data-dependent performance behavior of many large real-world applications. Due to the large amount of zero elements, compaction techniques are used to reduce the amount of storage, memory accesses, and computation performed on these zero elements. Using this pattern, we introduce the concept of regularization using hybrid methods and sorting/partitioning. These regularization methods are used in many real-world applications. Interestingly, some of the regularization techniques reintroduce zero elements into the compacted representations. We use hybrid methods to mitigate the pathological cases in which we could introduce too many zero elements. Readers are referred to Bell and Garland (2009) and encouraged to experiment with different sparse datasets to gain more insight into the data-dependent performance behavior of the various SpMV kernels presented in this chapter.

It should be clear that both the execution efficiency and memory bandwidth efficiency of the parallel SpMV kernels depend on the distribution of the input data matrix. This is quite different from most of the kernels we have studied so far. However, such data-dependent performance behavior is quite common in real-world applications. This is one of the reasons why parallel SpMV is such an important parallel pattern. It is simple, yet it illustrates an important behavior in many complex parallel applications.

We would like to make an additional remark on the performance of sparse matrix computation as compared to dense matrix computation. In general, the FLOPS ratings that are achieved by either CPUs or GPUs are much lower for sparse matrix computation than for dense matrix computation. This is especially true for SpMV, in which there is no data reuse in the sparse matrix. The OP/B is essentially 0.25, limiting the achievable FLOPS rate to a small fraction of the

peak performance. The various formats are important for both CPUs and GPUs, since both are limited by memory bandwidth when performing SpMV. People are often surprised by the low FLOPS rating of this type of computation on both CPUs and GPUs in the past. After reading this chapter, you should be no longer be surprised.

Exercises

1. Consider the following sparse matrix:

$$\begin{matrix} 1 & 0 & 7 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 4 & 3 & 0 \\ 2 & 0 & 0 & 1 \end{matrix}$$

   Represent it in each of the following formats: (1) COO, (2) CSR, (3) ELL, and (4) JDS.
2. Given a sparse matrix of integers with m rows, n columns, and z nonzeros, how many integers are needed to represent the matrix in (1) COO, (2) CSR, (3) ELL, and (4) JDS? If the information that is provided is not enough to allow an answer, indicate what information is missing.
3. Implement the code to convert from COO to CSR using fundamental parallel computing primitives, including histogram and prefix sum.
4. Implement the host code for producing the hybrid ELL-COO format and using it to perform SpMV. Launch the ELL kernel to execute on the device, and compute the contributions of the COO elements on the host.
5. Implement a kernel that performs parallel SpMV using a matrix stored in the JDS format.

References

Bell, N., Garland, M., 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the ACM Conference on High-Performance Computing Networking Storage and Analysis (SC'09).

Hestenes, M.R., Stiefel, E., 1952. Methods of Conjugate Gradients for Solving Linear Systems (PDF). J. Res. Nat. Bureau Stand. 49 (6).

Rice, J.R., Boisvert, R.F., 1984. Solving Elliptic Problems Using ELLPACK. Springer, Verlag, p. 497.

CHAPTER

Graph traversal 15

With special contributions from John Owens and Juan Gómez-Luna

Chapter Outline

A graph is a data structure that represents the relationships between entities. The entities involved are represented as vertices, and the relations are represented as edges. Many important real-world problems are naturally formulated as large-scale graph problems and can benefit from massively parallel computation. Prominent examples include social networks and driving direction map services. There are multiple strategies for parallelizing graph computations, some of which are centered on processing vertices in parallel, while others are centered on processing edges in parallel. Graphs are intrinsically related to sparse matrices. Thus graph computation can also be formulated in terms of sparse matrix operations. However, one can often improve the efficiency of graph computation by exploiting properties that are specific to the type of graph computation being performed. In this chapter we will focus on graph search, a graph computation that underlies many real-world applications.

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00011-2

15.1 Background

A graph data structure represents the relations between entities. For example, in social media, the entities are users, and the relations are connections between users. For another example, in driving direction map services, the entities are locations, and the relations are the roadways between the locations. Some relations are bidirectional, such as friend connections in a social network. Other relations are directional, such as one-way streets in a road network. In this chapter we will focus on directional relations. Bidirectional relations can be represented with two directional relations, one for each direction.

Fig. 15.1 shows an example of a simple graph with directional edges. A directional relation is represented as an arrowed edge going from a source vertex to a destination vertex. We assign a unique number to each vertex, also called the vertex *id*. There is one edge going from vertex 0 to vertex 1, one edge going from vertex 0 to vertex 2, and so on.

An intuitive representation of a graph is an *adjacency matrix*. If there is an edge going from a source vertex i to a destination vertex j, the value of element A[i][j] of the adjacency matrix is 1. Otherwise, it is 0. Fig. 15.2 shows the adjacency matrix for the simple graph in Fig. 15.1. We see that A[1][3] and A[4][5] are 1, since there are edges going from vertex 1 to vertex 3. For clarity we leave the 0 values out of the adjacency matrix. That is, if an element is empty, its value is understood to be 0.

If a graph with N vertices is *fully connected*, that is, every vertex is connected with all other vertices, each vertex should have (N − 1) outgoing edges. There should be a total of N(N − 1) edges, since there is no edge going from a vertex to itself. For example, if our nine-vertex graph were fully connected, there should be eight edges going out of each vertex. There should be a total of 72 edges. Obviously, our graph is much less connected; each vertex has three or fewer outgoing edges. Such a graph is referred to as being *sparsely connected*. That is, the average number of outgoing edges from each vertex is much smaller than N − 1.

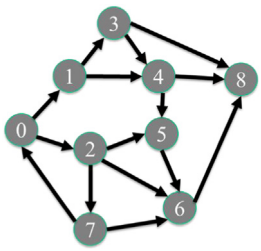


FIGURE 15.1

A simple graph example with 9 vertices and 15 directional edges.

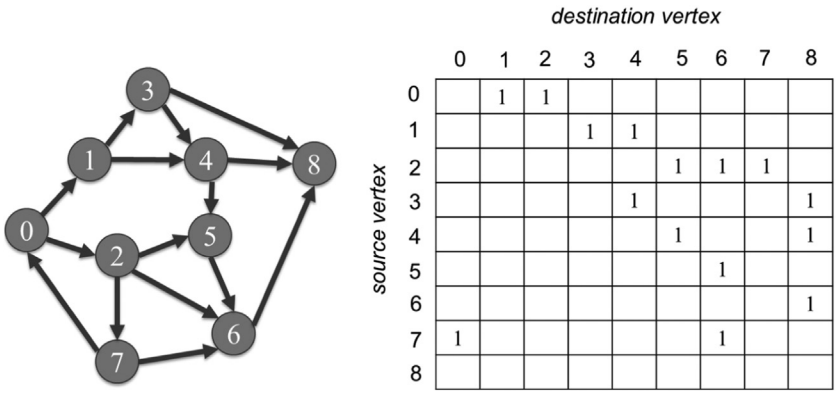


FIGURE 15.2

Adjacency matrix representation of the simple graph example.

At this point, the reader has most likely made the correct observation that sparsely connected graphs can probably benefit from a sparse matrix representation. As we have seen in Chapter 14, Sparse Matrix Computation, using a compressed representation of the matrix can drastically reduce the amount of storage required and the number of wasted operations on the zero elements. Indeed, many real-world graphs are sparsely connected. For example, in a social network such as Facebook, Twitter, or LinkedIn, the average number of connections for each user is much smaller than the total number of users. This makes the number of nonzero elements in the adjacency matrix much smaller than the total number of elements.

Fig. 15.3 shows three representations of our simple graph example using three different storage formats: compressed sparse row (CSR), compressed sparse column (CSC), and coordinate (COO). We will refer to the row indices and pointers arrays as the `src` and `srcPtrs` arrays, respectively, and the column indices and pointers arrays as the `dst` and `dstPtrs` arrays, respectively. If we take CSR as an example, recall that in a CSR representation of a sparse matrix each row pointer gives the starting location for the nonzero elements in a row. Similarly, in a CSR representation of a graph, each source vertex pointer (`srcPtrs`) gives the starting location of the outgoing edges of the vertex. For example, `srcPtrs[3]=7` gives the starting location of the nonzero elements in row 3 of the original adjacency matrix. Also, `srcPtrs[4]=9` gives the starting location of the nonzero elements in row 4 of the original matrix. Thus we expect to find the nonzero data for row 3 in `data[7]` and `data[8]` and the column indices (destination vertices) for these elements in `dst[7]` and `dst[8]`. These are the data and column indices for the two edges leaving vertex 3. The reason we call the column index array `dst` is that the column index of an element in the adjacency matrix gives the destination vertex of the represented edge. In our example, we see that the destination of the two edges for source vertex 3 are `dst[7]=4` and `dst[8]=8`. We leave it as an exercise to the reader to draw similar analogies for the CSC and COO representations.

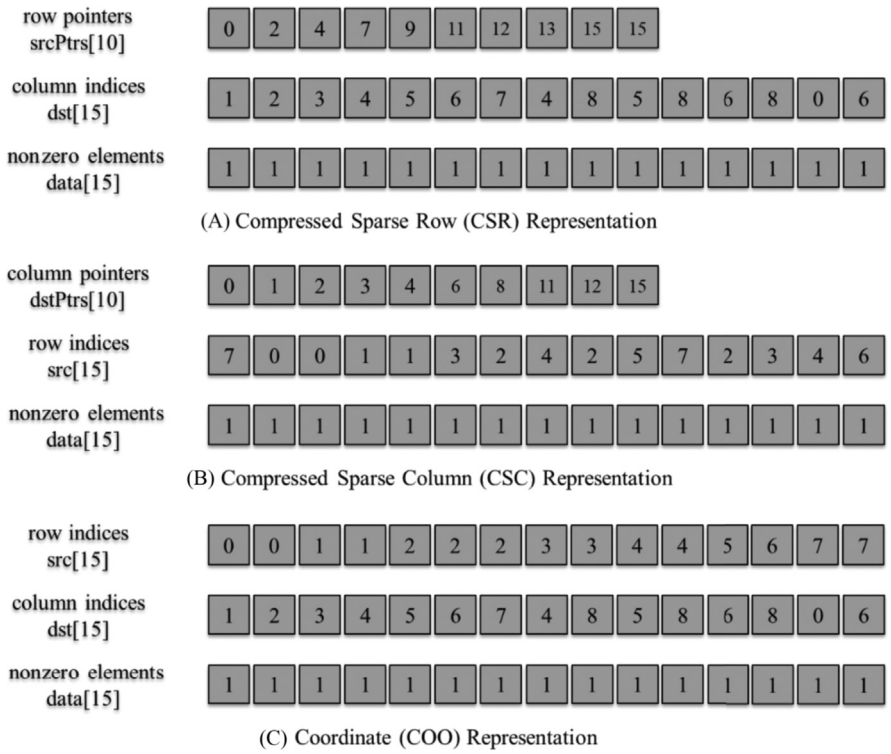


FIGURE 15.3

Three sparse matrix representations of the adjacency matrix: (A) CSR, (B) CSC, (C) COO. *COO*, coordinate; *CSC*, compressed sparse column; *CSR*, compressed sparse row.

Note that the `data` array in this example is unnecessary. Since the value of all its elements is 1, we do not need to store it. We can make the data implicit, that is, whenever a nonzero element exists, we can just assume that it is 1. For example, the existence of each column index in the destination array of a CSR representation implies that an edge exists. However, in some applications the adjacency matrix may store additional information about the relationship, such as the distance between two locations or the date on which two social network users became connected. In those applications the data array will need to be explicitly stored.

Sparse representation can lead to significant savings in storing the adjacency matrix. For our example, assuming that the `data` array can be eliminated, the CSR representation requires storage for 25 locations versus the 9^2=81 locations if we stored the entire adjacency matrix. For real-life problems in which a very small fraction of the adjacency matrix elements are nonzero, the savings can be tremendous.

Different graphs may have drastically different structures. One way to characterize these structures is to look at the distribution of the number of edges that are connected to each vertex (the *vertex degree*). A road network, expressed as a graph, would have a relatively uniform degree distribution with a low average degree per vertex because each road intersection (vertex) would typically have only a low number of roads connected to it. On the other hand, a graph of Twitter followers, in which each incoming edge represents a "follow," would have a much broader distribution of vertex degrees, with a large-degree vertex representing a popular Twitter user. The structure of the graph may influence the choice of algorithm to implement a particular graph application.

Recall from Chapter 14, Sparse Matrix Computation, that each sparse matrix representation gives different accessibility to the represented data. Hence the choice of which representation to use for the graph has implications for which information about the graph is made easily accessible to the graph traversal algorithm. The CSR representations give easy access to the outgoing edges of a given vertex. The CSC representation gives easy access to the incoming edges of a given vertex. The COO representation gives easy access to the source and destination vertices of a given edge. Therefore the choice of the graph representation goes hand-in-hand with the choice of the graph traversal algorithm. We demonstrate this concept throughout this chapter by examining different parallel implementations of breadth-first search, a widely used graph search computation.

15.2 Breadth-first search

An important graph computation is breadth-first search (BFS). BFS is often used to discover the shortest number of edges that one needs to traverse to go from one vertex to another vertex of the graph. In the graph example in Fig. 15.1 we may need to find all the alternative routes that we could take going from the location represented by vertex 0 to that represented by vertex 5. By visual inspection we see that there are three possible paths: $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5$, $0 \rightarrow 1 \rightarrow 4 \rightarrow 5$, and $0 \rightarrow 2 \rightarrow 5$, with $0 \rightarrow 2 \rightarrow 5$ being the shortest. There are different ways of summarizing the outcome of a BFS traversal. One way is, given a vertex that is referred to as the root, to label each vertex with the smallest number of edges that one needs to traverse to go from the root to that vertex.

Fig. 15.4(A) shows the desired BFS result with vertex 0 as the root. Through one edge, we can get to vertices 1 and 2. Thus we label these vertices as belonging to level 1. By traversing another edge, we can get to vertices 3 (through vertex 1), 4 (through vertex 1), 5 (through vertex 2), 6 (through vertex 2), and 7 (through vertex 2). Thus we label these vertices as belonging to level 2. Finally, by traversing one more edge, we can get to vertex 8 (through any of vertices 3, 4, or 6).

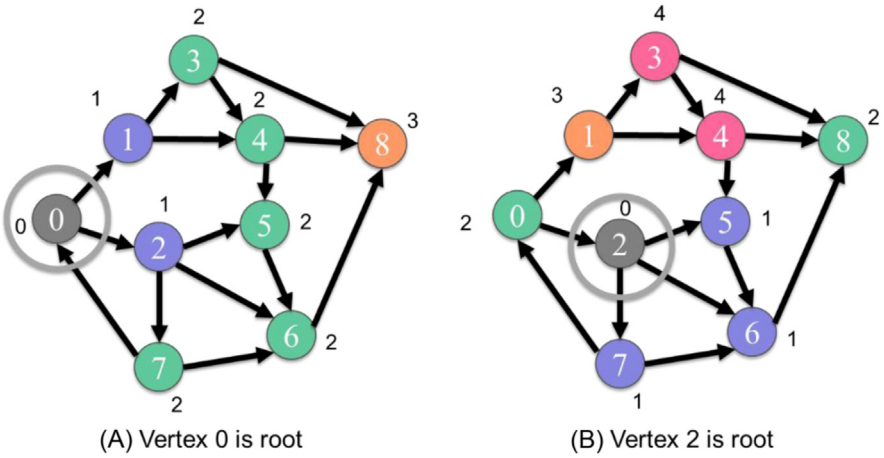


FIGURE 15.4

(A and B) Two examples of breadth-first search results for two different root vertices. The labels adjacent to each vertex indicate the number of hops (depth) from the root vertex.

The BFS result would be quite different with another vertex as the root. Fig. 15.4(B) shows the desired result of BFS with vertex 2 as the root. The level 1 vertices are 5, 6, and 7. The level 2 vertices are 8 (through vertex 6) and 0 (through vertex 7). Only vertex 1 is at level 3 (through vertex 0). Finally, the level 4 vertices are 3 and 4 (both through vertex 1). It is interesting to note that the outcome is quite different for each vertex even though we moved the root to a vertex that is only one edge away from the original root.

One can view the labeling actions of BFS as constructing a BFS tree that is rooted in the root node of the search. The tree consists of all the labeled vertices and only the edges traversed during the search that go from a vertex at one level to vertices at the next level.

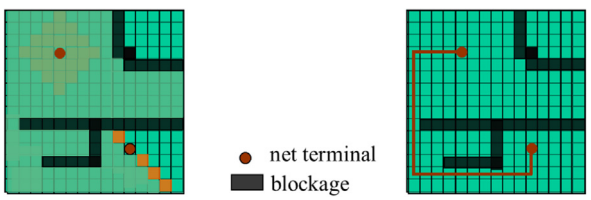
Once we have all the vertices labeled with their level, we can easily find a path from the root vertex to any of the vertices where the number of edges traveled is equivalent to the level. For example, in Fig. 15.4(B), we see that vertex 1 is labeled as level 3, so we know that the smallest number of edges between the root (vertex 2) and vertex 1 is 3. If we need to find the path, we can simply start from the destination vertex and trace back to the root. At each step, we select the predecessor whose level is one less than that of the current vertex. If there are multiple predecessors with the same level, we can randomly pick one. Any vertex thus selected would give a sound solution. The fact that there are multiple predecessors to choose from means that there are multiple equally good solutions to the problem. In our example we can find a shortest path from vertex 2 to vertex 1 by starting from vertex 1, choosing vertex 0, then vertex 7, and then vertex 2. Therefore a solution path is $2 \rightarrow 7 \rightarrow 0 \rightarrow 1$. This of course assumes that each

vertex has a list of the source vertices of all the incoming edges so that one can find the predecessors of a given vertex.

Fig. 15.5 shows an important application of BFS in computer-aided design (CAD). In designing an integrated circuit chip, there are many electronic components that need to be connected to complete the design. The connectors of these components are called net terminals. Fig. 15.5(A) shows two such net terminals as round dots; one belongs to a component in the upper left part, and the other belongs to another component in the lower right part of the chip. Assume that the design requires that these two net terminals be connected. This is done by running, or routing, a wire of a given width from the first net terminal to the second net terminal.

The routing software represents the chip as a grid of wiring blocks in which each block can potentially serve as a piece of a wire. A wire can be formed by extending in either the horizontal or the vertical direction. For example, the black J-shape in the lower half of the chip consists of 21 wiring blocks and connects three net terminals. Once a wiring block is used as part of a wire, it can no longer be used as part of any other wires. Furthermore, it forms a blockage for wiring blocks around it. No wires can be extended from a used block's lower neighbor to its upper neighbor or from its left neighbor to its right neighbor, and so on. Once a wire is formed, all other wires must be routed around it. Routing blocks can also be occupied by circuit components, which impose the same blockage constraint as when they are used as part of a wire. This is why the problem is called a maze routing problem. The previously formed circuit components and wires form a maze for the wires that have yet to be formed. The maze routing software finds a route for each additional wire given all the constraints from the previously formed components and wires.

The maze routing application represents the chip as a graph. The routing blocks are vertices. An edge from vertex i to vertex j indicates that one can extend a wire from block i to block j. Once a block is occupied by a wire or a component, it is either marked as a blockage vertex or taken away from the graph, depending on the design of the application. Fig. 15.5 shows that the



(A) Breadth-first search (B) Identifying a routing path

FIGURE 15.5

Maze routing in integrated circuits—an application for breadth-first search: (A) breadth-first search, (B) identifying a routing path.

application solves the maze routing problem with a BFS from the root net terminal to the target net terminal. This is done by starting with the root vertex and labeling the vertices into levels. The immediate vertical or horizontal neighbors (a total of four) that are not blockages are marked as level 1. We see that all four neighbors of the root are reachable and will be marked as level 1. The neighbors of level 1 vertices that are neither blockages nor visited by the current search will be marked as level 2. The reader should verify that there are four level 1 vertices, eight level 2 vertices, twelve level 3 vertices, and so on in Fig. 15.5(A). As we can see, the BFS essentially forms a wavefront of vertices for each level. These wavefronts start small for level 1 but can grow very large very quickly in a few levels.

Fig. 15.5(B) shows that once the BFS is complete, we can form a wire by finding a shortest path from the root to the target. As was explained earlier, this can be done by starting with the target vertex and tracing back to the predecessors whose levels are one lower than that of the current vertex. Whenever there are multiple predecessors that have equivalent levels, there are multiple routes that are of the same length. One could design heuristics to choose the predecessor in a way that minimizes the difficulty of constraints for wires that have yet to be formed.

15.3 Vertex-centric parallelization of breadth-first search

A natural way to parallelize graph algorithms is to perform operations on different vertices or edges in parallel. In fact, many parallel implementations of graph algorithms can be classified as vertex-centric or edge-centric. A vertex-centric parallel implementation assigns threads to vertices and has each thread perform an operation on its vertex, which usually involves iterating over the neighbors of that vertex. Depending on the algorithm, the neighbors of interest may be those that are reachable via the outgoing edges, the incoming edges, or both. In contrast, an edge-centric parallel implementation assigns threads to edges and has each thread perform an operation on its edge, which usually involves looking up the source and destination vertices of that edge. In this section we look at two different vertex-centric parallel implementations of BFS: one that iterates over outgoing edges and one that iterates over incoming edges. In the next section we look at an edge-centric parallel implementation of BFS and compare.

The parallel implementations that we will look at follow the same strategy when it comes to iterating over levels. In all implementations we start by labeling the root vertex as belonging to level 0. We then call a kernel to label all the neighbors of the root vertex as belonging to level 1. After that, we call a kernel to label all the unvisited neighbors of the level 1 vertices as belonging to level 2. Then we call a kernel to label all the unvisited neighbors of the level 2 vertices as belonging to level 3. This process continues until no new vertices are visited and labeled.

```
01  __global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* level,
02                  unsigned int* newVertexVisited, unsigned int currLevel) {
03      unsigned int vertex = blockIdx.x*blockDim.x + threadIdx.x;
04      if(vertex < csrGraph.numVertices) {
05          if(level[vertex] == currLevel - 1) {
06              for(unsigned int edge = csrGraph.srcPtrs[vertex];
07                      edge < csrGraph.srcPtrs[vertex + 1]; ++edge) {
08                  unsigned int neighbor = csrGraph.dst[edge];
09                  if(level[neighbor] == UINT_MAX) { // Neighbor not visited
10                      level[neighbor] = currLevel;
11                      *newVertexVisited = 1;
12                  }
13              }
14          }
15      }
16  }
```

FIGURE 15.6

A vertex-centric push (top-down) BFS kernel. *BFS*, breadth-first search.

The reason a separate kernel is called for each level is that we need to wait until all vertices in a previous level have been labeled before proceeding to label vertices in the next level. Otherwise, we risk labeling a vertex incorrectly. In the rest of this section we focus on implementing the kernel that is called for each level. That is, we will implement a BFS kernel that, given a level, labels all the vertices that belong to that level on the basis of the labels of the vertices from previous levels.

The first vertex-centric parallel implementation assigns each thread to a vertex to iterate over the vertex's outgoing edges (Harish and Narayanan, 2007). Each thread first checks whether its vertex belongs to the previous level. If so, the thread will iterate over the outgoing edges to label all the unvisited neighbors as belonging to the current level. This vertex-centric implementation is often referred to as a *top-down* or *push* implementation.[1] Since this implementation requires accessibility to the outgoing edges of a given source vertex (i.e., nonzero elements of a given row of the adjacency matrix), a CSR representation is needed.

Fig. 15.6 shows the kernel code for the vertex-centric push implementation, and Fig. 15.7 shows an example of how this kernel performs a traversal from level 1 (previous level) to level 2 (current level). The kernel starts by assigning a thread to each vertex (line 03), and each thread ensures that its vertex id is within bounds (line 04). Next, each thread checks whether its vertex belongs to the previous level (line 05). In Fig. 15.7, only the threads assigned to vertices 1 and 2 will pass this check. The threads that pass this check will then use the CSR `srcPtrs` array to locate the outgoing edges of the vertex and iterate over them (lines 06–07). For each outgoing edge, the thread finds the neighbor at the destination of the edge using the CSR `dst` array (line 08). The thread then checks

[1] If we are constructing a BFS tree, this implementation can be seen as assigning threads to parent vertices in the BFS tree in search of their children, hence the name *top-down*. This terminology assumes that the root of the tree is on the top and the leaves of the tree are at the bottom. *Push* refers to each active vertex's action of pushing its depth via its outgoing edges to all its neighbors.

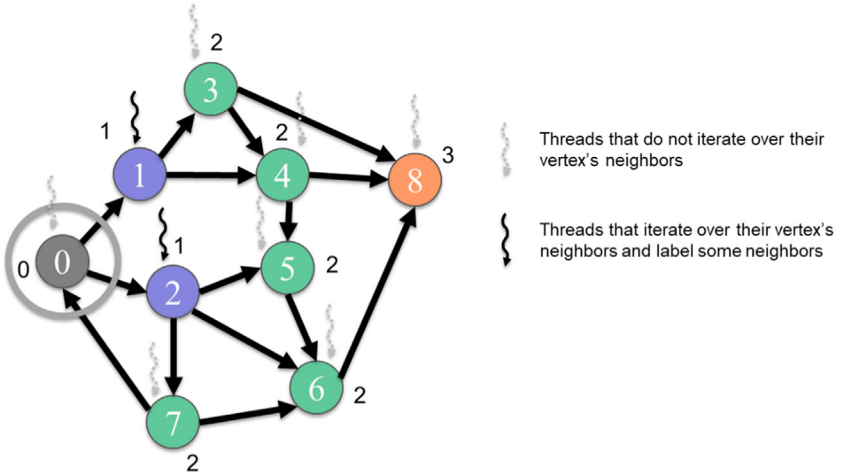


FIGURE 15.7

Example of a vertex-centric push BFS traversal from level 1 to level 2. *BFS*, breadth-first search.

whether the neighbor has not been visited by checking whether the neighbor has been assigned to a level yet (line 09).

Initially, all vertex levels are set to `UINT_MAX`, which means that the vertex is unreachable. Hence a neighbor has not been visited if its level is still `UINT_MAX`. If the neighbor has not been visited, the thread will label the neighbor as belonging to the current level (line 10). Finally, the thread will set a flag indicating that a new vertex has been visited (line 11). This flag is used by the launching code to decide whether a new grid needs to be launched to process a new level or we have reached the end. Note that multiple threads can assign 1 to the flag and the code will still execute correctly. This property is termed *idempotence*. In an idempotent operation such as this one, we do not need an atomic operation because the threads are not performing a read-modify-write operation. All threads write the same value, so the outcome is the same regardless of how many threads perform a write operation.

The second vertex-centric parallel implementation assigns each thread to a vertex to iterate over the vertex's incoming edges. Each thread first checks whether its vertex has been visited yet. If not, the thread will iterate over the incoming edges to find whether any of the neighbors belong to the previous level. If the thread finds a neighbor that belongs to the previous level, the thread will label its vertex as belonging to the current level. This vertex-centric implementation is often referred to as a *bottom-up* or *pull* implementation.[2] Since this

[2] If we are constructing a BFS tree, this implementation can be seen as assigning threads to potential child vertices in the BFS tree in search of their parents, hence, the name *bottom-up*. *Pull* refers to each vertex's action of reaching back to its predecessors and pulling active status from them.

implementation requires accessibility to the incoming edges of a given destination vertex (i.e., nonzero elements of a given column of the adjacency matrix), a CSC representation is needed.

Fig. 15.8 shows the kernel code for the vertex-centric pull implementation, and Fig. 15.9 shows an example of how this kernel performs a traversal from level 1 to level 2. The kernel starts by assigning a thread to each vertex (line 03), and each thread ensures that its vertex id is within bounds (line 04). Next, each thread checks whether its vertex has not been visited yet (line 05). In Fig. 15.9 the threads that are assigned to vertices 3−8 all pass this check. The threads that

```
01  __global__ void bfs_kernel(CSCGraph cscGraph, unsigned int* level,
02                  unsigned int* newVertexVisited, unsigned int currLevel) {
03      unsigned int vertex = blockIdx.x*blockDim.x + threadIdx.x;
04      if(vertex < cscGraph.numVertices) {
05          if(level[vertex] == UINT MAX) { // Vertex not yet visited
06              for(unsigned int edge = cscGraph.dstPtrs[vertex];
07                      edge < cscGraph.dstPtrs[vertex + 1]; ++edge) {
08                  unsigned int neighbor = cscGraph.src[edge];
09                  if(level[neighbor] == currLevel - 1) {
10                      level[vertex] = currLevel;
11                      *newVertexVisited = 1;
12                      break;
13                  }
14              }
15          }
16      }
17  }
```

FIGURE 15.8

A vertex-centric pull (bottom-up) BFS kernel. *BFS*, breadth-first search.

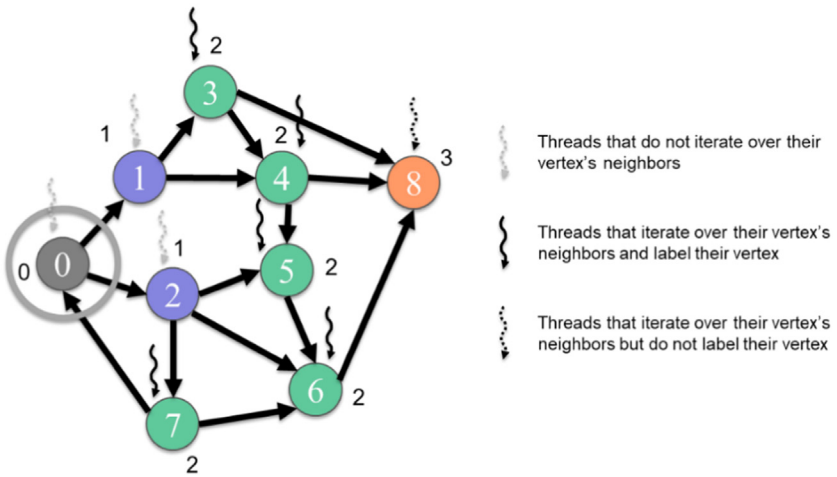


FIGURE 15.9

Example of a vertex-centric pull (bottom-up) traversal from level 1 to level 2.

pass this check will then use the CSC `dstPtrs` array to locate the incoming edges of the vertex and iterate over them (lines 06–07). For each incoming edge, the thread finds the neighbor at the source of the edge, using the CSC `src` array (line 08). The thread then checks whether the neighbor belongs to the previous level (line 09). If so, the thread will label its vertex as belonging to the current level (line 10) and set a flag indicating that a new vertex has been visited (line 11). The thread will also break out of the loop (line 12).

The justification for breaking out of the loop is as follows. For a thread to establish that its vertex is in the current level, it is sufficient for the thread's vertex to have one neighbor in the previous level. Therefore it is unnecessary for the thread to check the rest of the neighbors. Only the threads whose vertices do not have any neighbors in the previous level will end up looping over the entire neighbors list. In Fig. 15.9, only the thread assigned to vertex 8 will loop over the entire neighbor list without breaking.

In comparing the push and pull vertex-centric parallel implementations, there are two key differences to consider that have an important impact on performance. The first difference is that in the push implementation a thread loops over its vertex's entire neighbor list, whereas in the pull implementation a thread may break out of the loop early. For graphs with low degree and low variance, such as road networks or CAD circuit models, this difference may not be important because the neighbor lists are small and similar in size. However, for graphs with high degree and high variance, such as social networks, the neighbor lists are long and may vary substantially in size, resulting in high load imbalance and control divergence across threads. For this reason, breaking out of the loop early can provide substantial performance gains by reducing load imbalance and control divergence.

The second important difference between the two implementations is that in the push implementation, only the threads assigned to vertices in the previous level loop over their neighbor list, whereas in the pull implementation, all the threads assigned to any unvisited vertex loop over their neighbor list. For earlier levels, we expect to have a relatively small number of vertices per level and a large number of unvisited vertices in the graph. For this reason, the push implementation typically performs better for earlier levels because it iterates over fewer neighbor lists. In contrast, for later levels, we expect to have more vertices per level and fewer unvisited vertices in the graph. Moreover, the chances of finding a visited neighbor in the pull approach and exiting the loop early are higher. For this reason the pull implementation typically performs better for later levels.

Based on this observation, a common optimization is to use the push implementation for earlier levels, then switch to the pull implementation for later levels. This approach is often referred to as a *direction-optimized* implementation. The choice of when to switch between implementations usually depends on the type of graph. Low-degree graphs usually have many levels, and it takes a while to reach a point at which the levels have many vertices and a substantial number of vertices have already been visited. On the other hand, high-degree graphs

usually have few levels, and the levels grow very quickly. The high-degree graphs in which it takes only a few levels to get from any vertex to any other vertex are usually referred to as *small world graphs*. Because of these properties, switching from the push implementation to the pull implementation usually happens much earlier for high-degree graphs than for low-degree graphs.

Recall that the push implementation uses a CSR representation of the graph, whereas the pull implementation uses a CSC representation of the graph. For this reason, if a direction-optimized implementation is to be used, both a CSR and a CSC representation of the graph need to be stored. In many applications, such as social networks or maze routing, the graph is undirected, which means that the adjacency matrix is symmetric. In this case, the CSR and CSC representations are equivalent, so only one of them needs to be stored and can be used by both implementations.

15.4 Edge-centric parallelization of breadth-first search

In this section we look at an edge-centric parallel implementation of BFS. In this implementation, each thread is assigned to an edge. It checks whether the source vertex of the edge belongs to the previous level and whether the destination vertex of the edge is unvisited. If so, it labels the unvisited destination vertex as belonging to the current level. Since this implementation requires accessibility to the source and destination vertices of a given edge (i.e., row and column indices of a given nonzero), a COO data structure is needed.

Fig. 15.10 shows the kernel code for the edge-centric parallel implementation, while Fig. 15.11 shows an example of how this kernel performs a traversal from level 1 to level 2. The kernel starts by assigning a thread to each edge (line 03), and each thread ensures that its edge id is within bounds (line 04). Next, each thread finds the source vertex of its edge, using the COO `src` array (line 05), and checks whether the vertex belongs to the previous level (line 06). In Fig. 15.11, only the threads assigned to the outgoing edges of vertices 1 and 2 will pass this

```
01  __global__ void bfs_kernel(COOGraph cooGraph, unsigned int* level,
02                              unsigned int* newVertexVisited, unsigned int currLevel) {
03      unsigned int edge = blockIdx.x*blockDim.x + threadIdx.x;
04      if(edge < cooGraph.numEdges) {
05          unsigned int vertex = cooGraph.src[edge];
06          if(level[vertex] == currLevel - 1) {
07              unsigned int neighbor = cooGraph.dst[edge];
08              if(level[neighbor] == UINT_MAX) { // Neighbor not visited
09                  level[neighbor] = currLevel;
10                  *newVertexVisited = 1;
11              }
12          }
13      }
14  }
```

FIGURE 15.10

An edge-centric BFS kernel. *BFS*, breadth-first search.

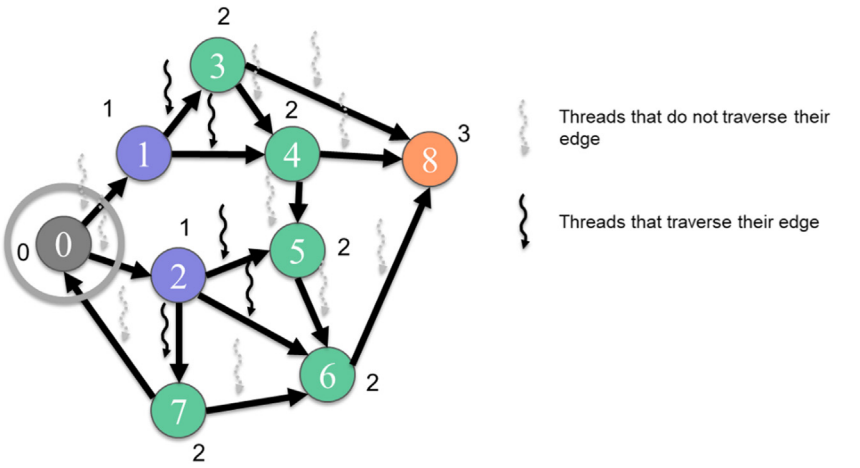


FIGURE 15.11

Example of an edge-centric traversal from level 1 to level 2.

check. The threads that pass this check will locate the neighbor at the destination of the edge, using the COO `dst` array (line 07), and check whether the neighbor has not been visited (line 08). If not, the thread will label the neighbor as belonging to the current level (line 09). Finally, the thread will set a flag indicating that a new vertex has been visited (line 10).

The edge-centric parallel implementation has two main advantages over the vertex-centric parallel implementations. The first advantage is that the edge-centric implementation exposes more parallelism. In the vertex-centric implementations, if the number of vertices is small, we may not launch enough threads to fully occupy the device. Since a graph typically has many more edges than vertices, the edge-centric implementation can launch more threads. Hence the edge-centric implementation is usually more suitable for small graphs.

The second advantage of the edge-centric implementation over the vertex-centric implementations is that it exhibits less load imbalance and control divergence. In the vertex-centric implementations, each thread iterates over a different number of edges, depending on the degree of the vertex to which it is assigned. In contrast, in the edge-centric implementation, each thread traverses only one edge. With respect to the vertex-centric implementation the edge-centric implementation is an example of rearranging the mapping of threads to work or data to reduce control divergence, as was discussed in Chapter 6, Performance Considerations. The edge-centric implementation is usually more suitable for high-degree graphs that have a large variation in the degrees of vertices.

The disadvantage of the edge-centric implementation is that it checks every edge in the graph. In contrast, the vertex-centric implementations can skip an entire edge list if the implementation determines that a vertex is not relevant for

the level. For example, consider the case in which some vertex v has n edges and is not relevant for a particular level. In the edge-centric implementation our launch includes n threads, one for each edge, and each of these threads independently inspects v and discovers that the edge is irrelevant. In contrast, in the vertex-centric implementations, our launch includes only one thread for v that skips all n edges after inspecting v once to determine that it is irrelevant. Another disadvantage of the edge-centric implementation is that it uses COO, which requires more storage space to store the edges compared to CSR and CSC, which are used by the vertex-centric implementations.

The reader may have noticed that the code examples in the previous section and this one resemble our implementations of sparse matrix-vector multiplication (SpMV) in Chapter 14, Sparse Matrix Computation. In fact, with a slightly different formulation we can express a BFS level iteration entirely in terms of SpMV and a few other vector operations, in which the SpMV operation is the dominant operation. Beyond BFS, many other graph computations can also be formulated in terms of sparse matrix computations, using the adjacency matrix (Jeremy and Gilbert, 2011). Such a formulation is often referred to as the *linear-algebraic formulation* of graph problems and is the focus of an API specification known as GraphBLAS. The advantage of linear-algebraic formulations is that they can leverage mature and highly optimized parallel libraries for sparse linear algebra to perform graph computations. The disadvantage of linear-algebraic formulations is that they may miss out on optimizations that take advantage of specific properties of the graph algorithm in question.

15.5 Improving efficiency with frontiers

In the approaches that we discussed in the previous two sections, we checked every vertex or edge in every iteration for their relevance to the level in question. The advantage of this strategy is that the kernels are highly parallel and do not require any synchronization across threads. The disadvantage is that many unnecessary threads are launched and a lot of wasted work is performed. For example, in the vertex-centric implementations we launch a thread for every vertex in the graph, many of which simply discover that the vertex is not relevant and do not perform any work. Similarly, in the edge-centric implementation we launch a thread for every edge in the graph; many of the threads simply discover that the edge is not relevant and do not perform any useful work.

In this section we aim to avoid launching unnecessary threads and eliminate the redundant checks that they perform in each iteration. We will focus on the vertex-centric push approach that was presented in Section 15.3. Recall that in the vertex-centric push approach, for each level, a thread is launched for each vertex in the graph. The thread checks whether its vertex is in the previous level, and if so, it labels all the vertex's unvisited neighbors as belonging to the current level.

On the other hand, the threads whose vertices are not in the current level do not do anything. Ideally, these threads should not even be launched. To avoid launching these threads, we can have the threads processing the vertices in the previous level collaborate to construct a *frontier* of the vertices that they visit. Hence for the current level, threads need to be launched only for the vertices in that frontier (Luo et al., 2010).

Fig. 15.12 shows the kernel code for the vertex-centric push implementation that uses frontiers, and Fig. 15.13 shows an example of how this kernel performs a traversal from level 1 to level 2. A key distinction from the previous approach is that the kernel takes additional parameters to represent the frontiers. The additional parameters include the arrays `prevFrontier` and `currFrontier` to store the vertices in the previous and current frontiers, respectively. They also include pointers to the counters `numPrevFrontier` and `numCurrFrontier` that store the number of vertices in each frontier. Note that the flag for indicating that a new vertex has been visited is no longer needed. Instead, the host can tell that the end has been reached when the number of vertices in the current frontier is 0.

We now look at the body of the kernel in Fig. 15.12. The kernel starts by assigning a thread to each element of the previous frontier (line 05), and each thread ensures that its element id is within bounds (line 06). In Fig. 15.13, only vertices 1 and 2 are in the previous frontier, so only two threads are launched. Each thread loads its element from the previous frontier, which contains the index of the vertex that it is processing (line 07). The thread uses the CSR `srcPtrs` array to locate the outgoing edges of the vertex and iterate over them (lines 08–09). For each outgoing edge, the thread finds the neighbor at the destination of the edge, using the CSR `dst` array (line 10). The thread then checks whether the neighbor has not been visited; if not, it labels the neighbor as belonging to the current level (line 11). An important distinction from the previous implementation is that an atomic operation is used to perform the checking and labeling operation. The reason will be explained shortly. If a thread succeeds in labeling the

```
01  __global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* level,
02              unsigned int* prevFrontier, unsigned int* currFrontier,
03              unsigned int numPrevFrontier, unsigned int* numCurrFrontier,
04              unsigned int currLevel) {
05      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
06      if(i < numPrevFrontier) {
07          unsigned int vertex = prevFrontier[i];
08          for(unsigned int edge = csrGraph.srcPtrs[vertex];
09                  edge < csrGraph.srcPtrs[vertex + 1]; ++edge) {
10              unsigned int neighbor = csrGraph.dst[edge];
11              if(atomicCAS(&level[neighbor],UINT_MAX,currLevel) == UINT_MAX) {
12                  unsigned int currFrontierIdx = atomicAdd(numCurrFrontier, 1);
13                  currFrontier[currFrontierIdx] = neighbor;
14              }
15          }
16      }
17  }
```

FIGURE 15.12

A vertex-centric push (top-down) BFS kernel with frontiers. *BFS*, breadth-first search.

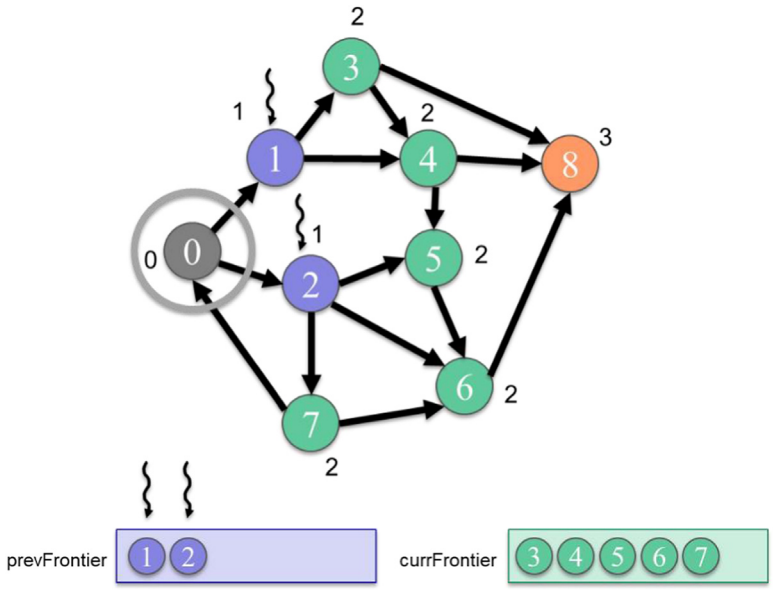


FIGURE 15.13

Example of a vertex-centric push (top-down) BFS traversal from level 1 to level 2 with frontiers. *BFS*, breadth-first search.

neighbor, it must add the neighbor to the current frontier. To do so, the thread increments the size of the current frontier (line 12) and adds the neighbor to the corresponding location (line 13). The size of the current frontier needs to be incremented atomically (line 12) because multiple threads may be incrementing it simultaneously, so we need to ensure that no race condition takes place.

We now turn our attention to the atomic operation on line 11. As a thread iterates through the neighbors of its vertex, it checks whether the neighbor has been visited; if not, it labels the neighbor as belonging to the current level. In the vertex-centric push kernel without frontiers in Fig. 15.12, this checking and labeling operation is performed without atomic operations (09−10). In that implementation, if multiple threads check the old label of the same unvisited neighbor before any of them are able to label it, multiple threads may end up labeling the neighbor. Since all threads are labeling the neighbor with the same label (the operation is idempotent), it is okay to allow the threads to label the neighbor redundantly. In contrast, in the frontier-based implementation in Fig. 15.12, each thread not only labels the unvisited neighbor but also adds it to the frontier. Hence if multiple threads observe the neighbor as unvisited, they will all add the neighbor to the frontier, causing it to be added multiple times. If the neighbor is added multiple times to the frontier, it will be processed multiple times in the next level, which is redundant and wasteful.

To avoid having multiple threads observe the neighbor as unvisited, the checking and updating of the neighbor's label should be performed atomically. In other words, we must check whether the neighbor has not been visited, and if not, label it as part of the current level all in one atomic operation. An atomic operation that can perform all of these steps is *compare-and-swap*, which is provided by the `atomicCAS` intrinsic function. This function takes three parameters: the address of the data in memory, the value to which we want to compare the data, and the value to which we would like to set the data if the comparison succeeds. In our case (line 11), we would like to compare `level[neighbor]` to `UINT_MAX` to check whether the neighbor is unvisited and set `level[neighbor]` to `currLevel` if the comparison succeeds. As with other atomic operations, `atomicCAS` returns the old value of the data that was stored. Therefore we can check whether the compare-and-swap operation succeeded by comparing the return value of `atomicCAS` with the value that `atomicCAS` compared with, which in this case is `UINT_MAX`.

As was mentioned earlier, the advantage of this frontier-based approach over the approach described in the previous section is that it reduces redundant work by only launching threads to process the relevant vertices. The disadvantage of this frontier-based approach is the overhead of the long-latency atomic operations, especially when these operations contend on the same data. For the `atomicCAS` operation (line 11) we expect the contention to be moderate because only some threads, not all, will visit the same unvisited neighbor. However, for the `atomicAdd` operation (line 12) we expect the contention to be high because all threads increment the same counter to add vertices to the same frontier. In the next section we look at how this contention can be reduced.

15.6 Reducing contention with privatization

Recall from Chapter 6, Performance Considerations, that one optimization that can be applied to reduce the contention of atomic operations on the same data is privatization. Privatization reduces contention of atomics by applying partial updates to a private copy of the data, then updating the public copy when done. We saw an example of privatization in the histogram pattern in Chapter 9, Parallel Histogram, where threads in the same block updated a local histogram that was private to the block, then updated the public histogram at the end.

Privatization can also be applied in the context of concurrent frontier updates (increments to `numCurrFrontier`) to reduce the contention on inserting into the frontier. We can have each thread block maintain its own local frontier throughout the computation and update the public frontier when done. Hence threads will contend on the same data only with other threads in the same block. Moreover, the local frontier and its counter can be stored in shared memory, which enables lower-latency atomic operations on the counter and stores to the local frontier. Furthermore, when the local frontier in shared memory is stored to the public frontier in global memory, the accesses can be coalesced.

Fig. 15.14 shows the kernel code for the vertex-centric push implementation that uses privatized frontiers, while Fig. 15.15 illustrates the privatization of the frontiers. The kernel starts by declaring a private frontier for each thread block in shared memory (lines 07–08). One thread in the block initializes the frontier's counter to 0 (lines 09–11), and all threads in the block wait at the `__syncthreads` barrier for the initialization to complete before they start using the counter (line 12). The next part of the code is similar to the previous version: Each thread loads its vertex from the frontier (line 17), iterates over its outgoing edges (lines 18–19), finds the neighbor at the destination of the edge (line 20), and atomically checks whether the neighbor is unvisited and visits it if it is unvisited (line 21).

```
01  __global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* level,
02              unsigned int* prevFrontier, unsigned int* currFrontier,
03              unsigned int numPrevFrontier, unsigned int* numCurrFrontier,
04              unsigned int currLevel) {
05
06      // Initialize privatized frontier
07      __shared__ unsigned int currFrontier_s[LOCAL_FRONTIER_CAPACITY];
08      __shared__ unsigned int numCurrFrontier_s;
09      if(threadIdx.x == 0) {
10          numCurrFrontier_s = 0;
11      }
12      __syncthreads();
13
14      // Perform BFS
15      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
16      if(i < numPrevFrontier) {
17          unsigned int vertex = prevFrontier[i];
18          for(unsigned int edge = csrGraph.srcPtrs[vertex];
19                  edge < csrGraph.srcPtrs[vertex + 1]; ++edge) {
20              unsigned int neighbor = csrGraph.dst[edge];
21              if(atomicCAS(&level[neighbor], UINT_MAX, currLevel) == UINT_MAX) {
22                  unsigned int currFrontierIdx_s = atomicAdd(&numCurrFrontier_s, 1);
23                  if(currFrontierIdx_s < LOCAL_FRONTIER_CAPACITY) {
24                      currFrontier_s[currFrontierIdx_s] = neighbor;
25                  } else {
26                      numCurrFrontier_s = LOCAL_FRONTIER_CAPACITY;
27                      unsigned int currFrontierIdx = atomicAdd(numCurrFrontier, 1);
28                      currFrontier[currFrontierIdx] = neighbor;
29                  }
30              }
31          }
32      }
33      __syncthreads();
34
35      // Allocate in global frontier
36      __shared__ unsigned int currFrontierStartIdx;
37      if(threadIdx.x == 0) {
38          currFrontierStartIdx = atomicAdd(numCurrFrontier, numCurrFrontier_s);
39      }
40      __syncthreads();
41
42      // Commit to global frontier
43      for(unsigned int currFrontierIdx_s = threadIdx.x;
44              currFrontierIdx_s < numCurrFrontier_s; currFrontierIdx_s += blockDim.x) {
45          unsigned int currFrontierIdx = currFrontierStartIdx + currFrontierIdx_s;
46          currFrontier[currFrontierIdx] = currFrontier_s[currFrontierIdx_s];
47      }
48
49  }
```

FIGURE 15.14

A vertex-centric push (top-down) BFS kernel with privatization of frontiers. *BFS*, breadth-first search.

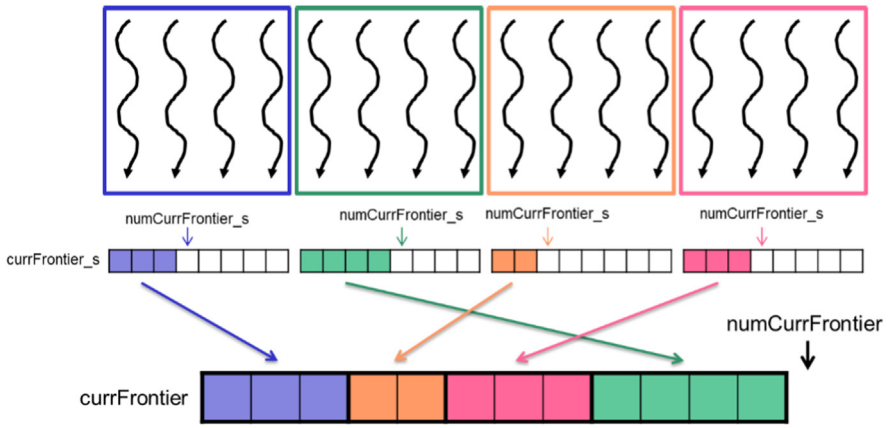


FIGURE 15.15

Privatization of frontiers example.

If the thread succeeds in visiting the neighbor, that is, the neighbor is unvisited, it adds the neighbor to the local frontier. The thread first atomically increments the local frontier counter (line 22). If the local frontier is not full (line 23), the thread adds the neighbor to the local frontier (line 24). Otherwise, if the local frontier has overflowed, the thread restores the value of the local counter (line 26) and adds the neighbor in the global frontier by atomically incrementing the global counter (line 27) and storing the neighbor at the corresponding location (line 28).

After all threads in a block have iterated over their vertices' neighbors, they need to store the privatized local frontier to the global frontier. First, the threads wait for each other to complete to ensure that no more neighbors will be added to the local frontier (line 33). Next, one thread in the block acts on behalf of the others to allocate space in the global frontier for all the elements in the local frontier (lines 36–39) while all the threads wait for it (line 40). Finally, the threads iterate over the vertices in the local frontier (line 43–44) and store them in the public frontier (line 45–46). Notice that the index into the public frontier `currFrontierIdx` is expressed in terms of `currFrontierIdx_s`, which is expressed in terms of `threadIdx.x`. Therefore threads with consecutive thread index values store to consecutive global memory locations, which means that the stores are coalesced.

15.7 Other optimizations

Reducing launch overhead

In most graphs, the frontiers of the initial iterations of a BFS can be quite small. The frontier of the first iteration has only the neighbors of the source. The frontier

of the next iteration has all the unvisited neighbors of the current frontier vertices. In some cases, the frontiers of the last few iterations can also be small. For these iterations the overhead of terminating a grid and launching a new one may outweigh the benefit of parallelism. One way to deal with these iterations with small frontiers is to prepare another kernel that uses only one thread block but may perform multiple consecutive iterations. The kernel uses only a local block-level frontier and uses `__syncthreads()` to synchronize across all threads in between levels.

This optimization is illustrated in Fig. 15.16. In this example, levels 0 and 1 can each be processed by a single thread block. Rather than launching a separate grid for levels 0 and 1, we launch a single-block grid and use `__syncthreads()` to synchronize between levels. Once the frontier reaches a size that overflows the block-level frontier, the threads in the block copy the block-level frontier contents to the global frontier and return to the host code. The host code will then call the regular kernel in the subsequent level iterations until the frontier is small again. The single-block kernel thus eliminates the launch overhead for the iterations with small frontiers. We leave its implementation as an exercise for the reader.

Improving load balance

Recall that in the vertex-centric implementations the amount of work to be done by each thread depends on the connectivity of the vertex that is

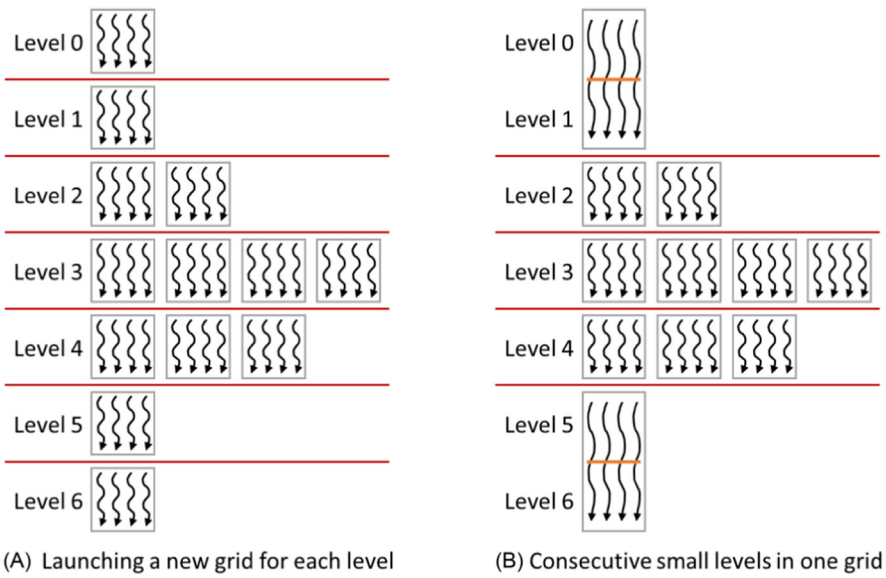


FIGURE 15.16

Executing multiple levels in one grid for levels with small frontiers: (A) launching a new grid for each level, (B) consecutive small levels in one grid.

assigned to it. In some graphs, such as social network graphs, some vertices (celebrities) may have degrees that are several orders of magnitude higher than those of other vertices. When this happens, one or a few of the threads can take excessively long and slow down the execution of the entire grid. We have seen one way to address this issue, which is by using an edge-centric parallel implementation instead. Another way in which we can potentially address this issue is by sorting the vertices of a frontier into *buckets* depending on their degree and processing each bucket in a separate kernel with an appropriately sized group of processors. One notable implementation (Merrill and Garland, 2012) uses three different buckets for vertices with small, medium, and large degrees. The kernel processing the small buckets assigns each vertex to a single thread; the kernel processing the medium buckets assigns each vertex to a single warp; and the kernel processing the large buckets assigns each vertex to an entire thread block. This technique is particularly useful for graphs with a high variation in vertex degrees.

Further challenges

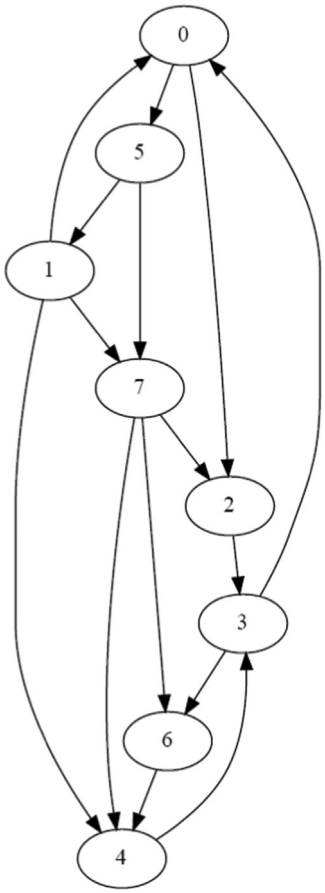
While BFS is among the simplest graph applications, it exhibits the challenges that are characteristic of more complex applications: problem decomposition for extracting parallelism, taking advantage of privatization, implementing fine-grained load balancing, and ensuring proper synchronization. Graph computation is applicable to a wide range of interesting problems, particularly in the areas of making recommendations, detecting communities, finding patterns within a graph, and identifying anomalies. One significant challenge is to handle graphs whose size exceeds the memory capacity of the GPU. Another interesting opportunity is to preprocess the graph into other formats before beginning computation in order to expose more parallelism or locality or to facilitate load balancing.

15.8 Summary

In this chapter we have seen the challenges that are associated with parallelizing graph computations, using breadth-first search as an example. We started with a brief introduction to the representation of graphs. We discussed the differences between vertex-centric and edge-centric parallel implementations and observed the tradeoffs between them. We also saw how to eliminate redundant work by using frontiers and optimized the use of frontiers by using privatization. We also briefly discussed other advanced optimizations to reduce synchronization overhead and improve load balance.

Exercises

1. Consider the following directed unweighted graph:



   a. Represent the graph using an adjacency matrix.
   b. Represent the graph in the CSR format. The neighbor list of each vertex must be sorted.
   c. Parallel BFS is executed on this graph starting from vertex 0 (i.e., vertex 0 is in level 0). For each iteration of the BFS traversal:
      i. If a vertex-centric push implementation is used:
         1. How many threads are launched?
         2. How many threads iterate over their vertex's neighbors?
      ii. If a vertex-centric pull implementation is used:
         1. How many threads are launched?

   2. How many threads iterate over their vertex's neighbors?
   3. How many threads label their vertex?
iii. If an edge-centric implementation is used:
   1. How many threads are launched?
   2. How many threads may label a vertex?
iv. If a vertex-centric push frontier-based implementation is used:
   1. How many threads are launched?
   2. How many threads iterate over their vertex's neighbors?

2. Implement the host code for the direction-optimized BFS implementation described in Section 15.3.
3. Implement the single-block BFS kernel described in Section 15.7.

References

Harish, P., Narayanan, P.J., 2007. Accelerating large graph algorithms on the GPU using CUDA. In: International Conference on High-Performance Computing (HiPC), India.

Jeremy, K., Gilbert, J. (Eds.), 2011. Graph Algorithms in the Language of Linear Algebra. Society for Industrial and Applied Mathematics.

Luo, L., Wong, M., Hwu, W., 2010. An effective GPU implementation of breadth-first search. In: ACM/IEEE Design Automation Conference (DAC).

Merrill, D., Garland, M., 2012. Scalable GPU graph traversal. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).

CHAPTER

Deep learning 16

With special contributions from Carl Pearson and Boris Ginsburg

Chapter Outline

This chapter presents an application case study on deep learning, a recent branch of machine learning using artificial neural networks. Machine learning has been used in many application domains to train or adapt application logic according to the experience gleaned from datasets. To be effective, one often needs to conduct such training with a massive amount of data. While machine learning has existed as a subject of computer science for a long time, it has recently gained a great deal of practical industry acceptance for two reasons. The first reason is the massive amounts of data available from the pervasive use of the internet. The second reason is the inexpensive, massively parallel GPU computing systems that can effectively train application logic with these massive datasets. We will start with a brief introduction to machine learning and deep learning and then consider in more detail one of the most popular deep learning algorithms: convolutional neural networks (CNN). CNN have a high compute to memory access ratio and high levels of parallelism, which make them a perfect candidate for GPU acceleration. We will first present a basic implementation of a convolutional neural network. Next, we will show how we can improve this basic implementation with shared memory. We will then show how one can formulate the convolutional

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00024-0

layers as matrix multiplication, which can be accelerated by using highly optimized hardware and software in modern GPUs.

16.1 Background

Machine learning, a term coined by Arthur Samuel of IBM in 1959 (Samuel, 1959), is a field of computer science that studies methods for learning application logic from data rather than designing explicit algorithms. Machine learning is most successful in computing tasks in which designing explicit algorithms is infeasible, mostly because there is not enough knowledge in the design of such explicit algorithms. That is, one can give examples of what should happen in various situations but not general rules for making such decisions for all possible inputs. For example, machine learning has contributed to the recent improvements in application areas such as automatic speech recognition, computer vision, natural language processing, and recommender systems. In these application areas, one can provide many input examples and what should come out for each input, but there is no algorithm that can correctly process all possible inputs.

The kinds of application logic that are created with machine learning can be organized according to the types of tasks that they perform. There is a wide range of machine learning tasks. Here, we show a few out of a large number:

1. Classification: to determine to which of the k categories an input belongs. An example is object recognition, such as determining which type of food is shown in a photo.
2. Regression: to predict a numerical value given some inputs. An example is to predict the price of a stock at the end of the next trading day.
3. Transcription: to convert unstructured data into textual form. An example is optical character recognition.
4. Translation: to convert a sequence of symbols in one language to a sequence of symbols in another. An example is translating from English to Chinese.
5. Embedding: to convert an input to a vector while preserving relationships between entities. An example is to convert a natural language sentence into a multidimensional vector.

The reader is referred to a large body of literature on the mathematical background and practical solutions to the various tasks of machine learning. The purpose of the chapter is to introduce the computation kernels that are involved in the neural network approach to the classification task. A concrete understanding of these kernels will allow the reader to understand and develop kernels for deep learning approaches to other machine learning tasks. Therefore in this section we will go into details about the classification task to establish the background

knowledge that is needed to understand neural networks. Mathematically, a classifier is a function f that maps an input to k categories or labels:

$$f: R^n \rightarrow \{1, 2, \ldots, k\}$$

The function f is parameterized by θ that maps input vector x to numerical code y, that is,

$$y = f(x, \theta)$$

The parameter θ is commonly referred to as the *model*. It encapsulates weights that are learned from data. This definition of θ is best illustrated with a concrete example. Let us consider a linear classifier called a *perceptron* (Rosenblatt, 1957): $y = sign(W \cdot x + b)$, where W is vector of weights of the same length as x and b is a bias constant. The sign function returns value 1 if its input is positive, 0 if its input is 0, and -1 if its input is negative. That is, the sign function as a classifier activates, that is, finalizes, the mapping of the input value into three categories:$\{-1, 0, 1\}$; therefore it is often called the *activation* function. Activation functions introduce nonlinearity into an otherwise linear function of a perceptron. In this case, the model θ is the combination of the vector W and the constant b. The structure of the model is a sign function whose input is a linear expression of input x elements where the coefficients are elements of W, and the constant is b.

Fig. 16.1 shows a perceptron example in which each input is a two-dimensional (2D) vector (x_1, x_2). The linear perceptron's model θ consists of a weight vector (w_1, w_2) and a bias constant b. As shown in Fig. 16.1, the linear expression $w_1x_1 + w_2 \cdot x_2 + b$ defines a line in the $x_1 - x_2$ space that cuts the space into two parts: the part in which all points make the expression greater than zero and the part in which all points make the expression less than zero. All points on the line make the expression equal to 0.

Visually, given a combination of (w_1, w_2) and b values, we can draw a line in the (x_1, x_2) space, as shown in Fig. 16.1. For example, for a perceptron whose model is $(w_1, w_2) = (2, 3)$ and $b = -6$, we can easily draw a line by connecting the two intersection points with the x_1 axis ($(\frac{-b}{w_1}, 0) = (3, 0)$) and the x_2 axis

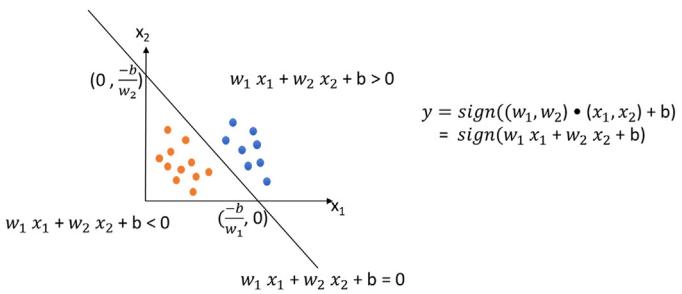


FIGURE 16.1

A perceptron linear classifier example in which the input is a two-dimensional vector.

$((0, \frac{-b}{w_2}) = (0, 2))$. The line thus drawn corresponds to the equation $2x_1 + 3x_2 - 6 = 0$ With this drawing, we can easily visualize the outcome of input points: Any point above the line (shown as blue dots in Fig. 16.1) is classified as class 1, any point on the line is classified as class 0, and any point below the line (shown as orange dots in Fig. 16.1) is classified as class -1.

The process of computing the class for an input is commonly referred to as *inference* for the classifier. In the case of a perceptron we simply plug the input coordinate values into $y = sign\ (W \cdot x + b)$. In our example, if the input point is $(5, -1)$, we can perform inference by plugging its coordinates into the perceptron function:

$$y = sign(2 * 5 + 3 * (-1) + 6) = \text{sign}(13) = 1$$

Therefore $(5, -1)$ is classified to class 1, that is, it is among the blue dots.

Multilayer classifiers

Linear classifiers are useful when there is a way to draw hyperplanes (i.e., lines in a 2D space and planes in a three-dimensional [3D] space) that partition the space into regions and thus define each class of data points. Ideally, each class of data points should occupy exactly one such region. For example, in a 2D, 2-class classifier, we need to be able to draw a line that separates points of one class from those of the other. Unfortunately, this is not always feasible.

Consider the classifiers in Fig. 16.2. Assume that all input's coordinates fall in the range of [0, 1]. The classifier should classify all points whose x_1 and x_2 values are both greater than 0.5 (points that fall in the upper right quadrant of the domain) as class 1 and the rest as class -1. This classifier could be approximately implemented with a line like that shown in Fig. 16.2(A). For example, a line $2x_1 + 2x_2 - 3 = 0$ would classify most of the points properly. However, some of the orange points whose x_1 and x_2 are both greater than 0.5 but the sum is less than 1.5, for example, (0.55, 0.65), would be misclassified into class -1 (blue). This is because any line will necessarily either cut away part of the upper right quadrant or include part of the rest of the domain. There is no single line that can properly classify all possible inputs.

A *multilayer perceptron* (MLP) allows the use of multiple lines to implement more complex classification patterns. In a multilayer perceptron, each layer consists of one or more perceptrons. The outputs of perceptrons in one layer are the inputs to those in the next layer. An interesting and useful property is that while the inputs to the first layer have an infinite number of possible values, the output of the first layer and thus the input to the second layer can have only a modest number of possible values. For example, if the perceptron from Fig. 16.1 was used as a first layer, its outputs would be restricted to $\{-1, 0, 1\}$.

Fig. 16.2(B) shows a two-layer perceptron that can precisely implement the desired classification pattern. The first layer consists of two perceptrons. The first one, $y_1 = sign(x_1 - 0.5)$, classifies all points whose x_1 coordinate is greater than

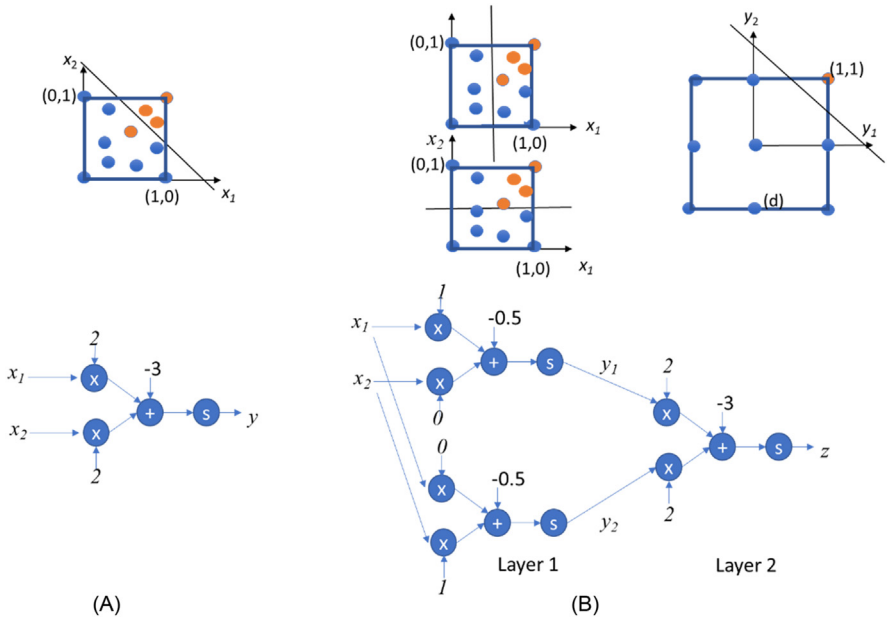


FIGURE 16.2

A multilayer perceptron example.

0.5 as class 1; that is, the output value is 1. The rest of the points are classified as either class -1 or class 0. The second classifier in the first layer, $y_2 = sign(x_2 - 0.5)$, classifies all points whose x_2 coordinate is greater than 0.5 into class 1. The rest of the points are classified as class -1.

Therefore the output of the first layer (y_1, y_2) can only be one of the following nine possibilities: (-1, -1), (-1, 0), (-1,1), (0, -1) (0, 0) (0, 1) (1, -1), (1, 0), (1,1). That is, there are only nine possible input pair values to the second layer. Out of these nine possibilities, (1, 1) is special. All the original input points in the orange category are mapped to (1, 1) by the first layer. Therefore we can use a simple perceptron in the second layer to draw a line between (1, 1) and the eight other possible points in the y_1-y_2 space, shown in Fig. 16.2B. This can be done by a line $2y_1 + 2y_2 - 3 = 0$ or many other lines that are small variations of it.

Let us use the (0.55, 0.65) input that was misclassified by the single-layer perceptron in Fig. 16.2A. When processed by the two-layer perceptron, the upper perceptron of first layer in Fig. 16.2B generates $y_1 = 1$, and the lower perceptron generates $y_2 = 1$. On the basis of these input values, the perceptron in the second layer generates $z = 1$, the correct classification for (0.55, 0.65).

Note that a two-layer perceptron still has significant limitations. For example, assume that we need to build a perceptron to classify the input points shown in Fig. 16.3A. The values of the orange input points can result in input

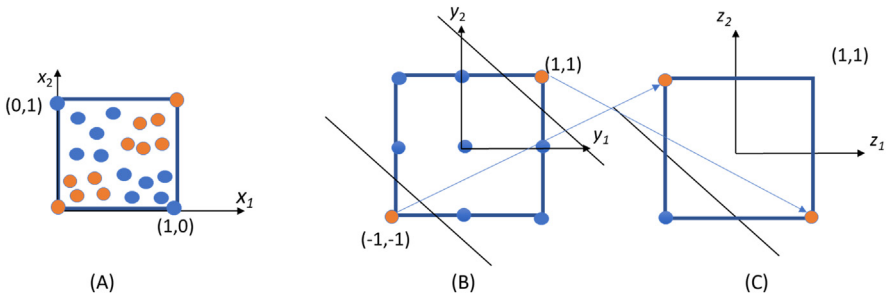


FIGURE 16.3

Need for perceptrons with more than two layers.

values $(-1, -1)$ or $(1, 1)$ to the second layer. We see that there is no way to draw a single line to properly classify the points in the second layer. We show in Fig. 16.2B that we can add another line by adding another perceptron in the second layer. The function would be $z_2 = sign(-2y_1 - 2y_2 - 3)$ or small variations of it. The reader should verify that all blue points in Fig. 16.3B will be mapped to the $(-1, -1)$ in the z_1-z_2 space. Whereas $(1, 1)$ and $(-1, -1)$ in the $y_1 - y_2$ space are mapped to $(1, -1)$ and $(-1, 1)$ in the $z_1 - z_2$ space. Now we can draw a line $z_1 + z_2 + 1 = 0$ or small variations of it to properly classify the points, as shown in Fig. 16.3C. Obviously, if we need to partition the input domain into more regions, we might need even more layers to perform proper classification.

Layer 1 in Fig. 16.2B is a small example of a fully connected layer, in which every output (i.e., y_1, y_2) is a function of every input (i.e., x_1, x_2). In general, in a fully connected layer, every one of the m outputs is a function of all the n inputs. All the weights of a fully connected layer form an m $\times$ n weight matrix W, where each of the m rows is the weight vector (of size n elements) to be applied to the input vector (of size n elements) to produce one of the m outputs. Therefore the process of evaluating all the outputs from the inputs of a fully connected layer is a matrix-vector multiplication. As we will see, fully connected layers are core components of many types of neural networks, and we will further study the GPU implementations.

Fully connected layers become extremely expensive when m and n become large. The main reason is that a fully connected layer requires an m $\times$ n weight matrix. For example, in an image recognition application, n is the number of pixels in the input image, and m is the number of classifications that need to be performed on the input pixels. In this case, n is in the millions for high-resolution images, and m can be in the hundreds or more depending on the variety of objects that need to be recognized. Also, the objects can be of different scales and orientations in the images; many classifiers may need to be in place to deal with these variations. Feeding all these classifiers with all inputs is both expensive and likely wasteful.

Convolutional layers reduce the cost of fully connected layers by reducing the number of inputs each classifier takes and sharing the same weights across classifiers. In a convolutional layer, each classifier takes only a patch of the input image and performs convolution on the pixels in the patch based on the weights. The output is called an output feature map, since each pixel in the output is the activation result of a classifier. Sharing weights across classifiers allows the convolutional layers to have large numbers of classifiers, that is, large m values, without an excessive number of weights. Computationally, this can be implemented as a 2D convolution. However, this approach effectively applies the same classifier to a different part of an image. One can have different sets of weights applied to the same input and generate multiple output feature maps, as we will see later in this chapter.

Training models

So far, we have assumed that the model parameters used by a classifier are somehow available. Now we turn to training, or the process of using data to determine the values of the model parameters θ, including the weights (w_1, w_2) and the bias b. For simplicity we will assume supervised training, in which input data labeled with desired output values are used to determine the weight and bias values. Other training modalities, such as semisupervised and reinforcement learning, have also been developed to reduce the reliance on labeled data. The reader is referred to the literature to understand how training can be accomplished under such circumstances.

Error function

In general, training treats the model parameters as unknown variables and solves an inverse problem given the labeled input data. In the perceptron example in Fig. 16.1, each data point that is used for training would be labeled with its desired classification result: −1, 0, or 1. The training process typically starts with an initial guess of the (w_1, w_2) and b values and performs inference on the input data and generates classification results. These classification results are compared to the labels. An error function, sometimes referred to as a cost function, is defined to quantify the difference between the classification result and the corresponding label for each data point. For example, assume that y is the classification output class and t is the label. The following is an example error function:

$$E = \frac{(y-t)^2}{2}$$

This error function has the nice property that the error value is always positive as long as there is any difference, positive or negative, between the values of y and t. If we need to sum up the error across many input data points, both positive and negative differences will contribute to the total rather than canceling each other out. One can also define the error as the absolute value of the difference,

among many other options. As we will see, the coefficient $\frac{1}{2}$ simplifies the computation involved in solving the model parameters.

Stochastic gradient descent

The training process will attempt to find the model parameter values that minimize the sum of the error function values for all the training data points. This can be done with a *stochastic gradient descent* approach, which repeatedly runs different permutations of the input dataset through the classifier, evolves the parameter values, and checks whether the parameter values have converged in that their values have stabilized and changed less than a threshold since the last iteration. Once the parameter values converge, the training process ends.

Epoch

During each iteration of the training process, called an *epoch*, the training input dataset is first randomly shuffled, that is, permutated, before it is fed to the classifier. This randomization of the input data ordering helps to avoid suboptimal solutions. For each input data element, its classifier output y value is compared with the label data to generate the error function value. In our perceptron example, if a data label is (class) 1 and the classifier output is (class) -1, the error function value using $E = \frac{(y-t)^2}{2}$ would be 2. If the error function value is larger than a threshold, a backpropagation operation is activated to make changes to the parameters so that the inference error can be reduced.

Backpropagation

The idea of backpropagation is to start with the error function and look back into the classifier and identify the way in which each parameter contributes to the error function value (LeCun et al., 1990). If the error function value increases when a parameter's value increases for a data element, we should decrease the parameter value so that the error function value can decrease for this data point. Otherwise, we should increase the parameter value to reduce the error function value for the data point. Mathematically, the rate and direction in which a function's value changes as one of its input variables changes are the partial derivative of the function over the variable. For a perceptron the model parameters and the input data points are considered input variables for the purpose of calculating the partial derivatives of the error function. Therefore the backpropagation operation will need to derive the partial derivative values of the error function over the model parameters for each input data element that triggers the backpropagation operation.

Let us use the perceptron $y = sign(w_1x_1 + w_2x_2 + \mathrm{b})$ to illustrate the backpropagation operation. Assume error function $E = \frac{(y-t)^2}{2}$ and that the backpropagation is triggered by a training input data element (5, 2). The goal is to modify the w_1, w_2, and b values so that the perceptron will more likely classify (5, 2) correctly. That is, we need to derive the values of partial derivatives $\frac{\partial E}{\partial w_1}$, $\frac{\partial E}{\partial w_2}$, and $\frac{\partial E}{\partial b}$ in order to make changes to the w_1, w_2, and b values.

Chain rule

We see that E is a function of y and y is a function of w_1, w_2, and b. Thus we can use the chain rule to derive these partial derivatives. For w_1,

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial w_1}$$

$\frac{\partial E}{\partial y}$ is straightforward:

$$\frac{\partial E}{\partial y} = \frac{\partial \frac{(y-t)^2}{2}}{\partial y} = y - t$$

However, we face a challenge with $\frac{\partial y}{\partial w_1}$. Note that the sign function is not a differentiable function, as it is not continuous at 0. To solve this problem, the machine learning community commonly use a smoother version of the sign function that is differentiable near zero and close to the sign function value for x values away from 0. A simple example of such a smoother version is the sigmoid function $s = \frac{1-e^{-x}}{1+e^{-x}}$. For x values that are negative with large absolute value, the sigmoid expression is dominated by the e^{-x} terms, and the sigmoid function value will be approximately -1. For x values that are positive with large absolute values, the e^{-x} terms diminish, and the function value will be approximately 1. For x values that are close to 0, the function value increases rapidly from near -1 to near 1. Thus the sigmoid function closely approximates the behavior of a sign function and yet is continuous differentiable for all x values. With this change from sign to sigmoid, the perceptron is $y = sigmoid(w_1x_1 + w_2x_2 + \text{b})$. We can express $\frac{\partial y}{\partial w_1}$ as $\frac{\partial sigmoid(k)}{\partial k}\frac{\partial k}{\partial w_1}$ using the chain rule with an intermediate variable $k = w_1x_1 + w_2x_2 + \text{b}$. Based on calculus manipulation, $\frac{\partial k}{\partial w_1}$ is simply x_1 and

$$\frac{\partial sigmoid(k)}{\partial k} = \left(\frac{1-e^{-k}}{1+e^{-k}}\right)' = \left(1-e^{-k}\right)'\left(\frac{1}{1+e^{-k}}\right) + \left(1-e^{-k}\right)\left(\frac{1}{1+e^{-k}}\right)'$$

$$= e^{-k}\left(\frac{1}{1+e^{-k}}\right) + \left(1-e^{-k}\right)\left(-1\times\left(\frac{1}{1+e^{-k}}\right)^2\left(-e^{-k}\right)\right) = \frac{2e^{-k}}{(1+e^{-k})^2}$$

Putting it all together, we have

$$\frac{\partial E}{\partial w_1} = (y-t)\frac{2e^{-k}}{(1+e^{-k})^2}x_1$$

Similarly,

$$\frac{\partial E}{\partial w_2} = (y-t)\frac{2e^{-k}}{(1+e^{-k})^2}x_2$$

$$\frac{\partial E}{\partial b} = (y-t)\frac{2e^{-k}}{(1+e^{-k})^2}$$

where

$$k = w_1 x_1 + w_2 x_2 + b$$

It should be clear that all the three partial derivative values can be completely determined by the combination of the input data (x_1, x_2 and t) and the current values of the model parameters (w_1, w_2 and b). The final step for the backpropagation is to modify the parameter values. Recall that the partial derivative of a function over a variable gives the direction and rate of change in the function value as the variable changes its value. If the partial derivative of the error function over a parameter has a positive value given the combination of input data and current parameter values, we want to decrease the value of the parameter so that the error function value will decrease. On the other hand, if the partial derivative of the error function of the variable has a negative value, we want to increase the value of the parameter so that the error function value will decrease.

Learning rate

Numerically, we would like to make bigger changes to the parameters to whose change the error function is more sensitive, that is, when the absolute value of the partial derivative of the error function over this parameter is a large value. These considerations lead to us to subtract from each parameter a value that is proportional to the partial derivative of the error function over that parameter. This is accomplished by multiplying the partial derivatives with a constant ε, called the learning rate constant in machine learning, before it is subtracted from the parameter value. The larger ε is, the faster the values of the parameters evolve so the solution can potentially be reached with fewer iterations. However, large ε also increases the chance of instability and prevents the parameter values from converging to a solution. In our perceptron example, the modifications to the parameters are as follows:

$$w_1 \leftarrow w_1 - \varepsilon \frac{\partial E}{\partial w_1}$$

$$w_2 \leftarrow w_2 - \varepsilon \frac{\partial E}{\partial w_2}$$

$$b \leftarrow b - \varepsilon \frac{\partial E}{\partial b}$$

For the rest of this chapter we will use a generic symbol θ to represent the model parameters in formula and expressions. That is, we will represent the three expressions above with one generic expression:

$$\theta \leftarrow \theta - \varepsilon \frac{\partial E}{\partial}$$

The reader should understand that for each of these generic expressions one can replace θ with any of the parameters to apply the expression to the parameter.

Minibatch

In practice, because the backtracking process is quite expensive, it is not triggered by individual data points whose inference result differs from its label. Rather, after the inputs are randomly shuffled in an epoch, they are divided into segments called *minibatches*. The training process runs an entire minibatch through the inference and accumulates their error function values. If the total error in the minibatch is too large, backpropagation is triggered for the minibatch. During backpropagation the inference results of each data point in the minibatch are checked, and if it is not correct, the data is used to derive partial derivative values that are used to modify the model parameter values as described above.

Training multilayer classifiers

For multilayer classifiers the backpropagation starts with the last layer and modifies the parameter values in that layer as we discussed above. The question is how we should modify the parameters of the previous layers. Keep in mind that we can derive $\frac{\partial E}{\partial \theta}$ based on $\frac{\partial E}{\partial y}$, as we have demonstrated for the final layer. Once we have $\frac{\partial E}{\partial y}$ for the previous layer, we have everything we need to calculate the modifications to the parameters in that layer.

A simple and yet important observation is that the output of the previous layer is also the input to the final layer. Therefore $\frac{\partial E}{\partial y}$ of the previous layer is really the $\frac{\partial E}{\partial x}$ of the final layer. Therefore the key is to derive $\frac{\partial E}{\partial x}$ for the final layer after we modify the parameter values of the final layer. As we can see below, $\frac{\partial E}{\partial x}$ is not that different from $\frac{\partial E}{\partial \theta}$, that is, $\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial x}$.

$\frac{\partial E}{\partial y}$ can be simply reused from the derivations for the parameters. $\frac{\partial y}{\partial x}$ is also quite straightforward since the inputs play the same role as the parameters as far as y is concerned. We simply need to do a partial derivative of the intermediate function k with respect to the inputs. For our perceptron example, we have

$$\frac{\partial E}{\partial x_1} = (y - t)\frac{2e^{-k}}{(1 + e^{-k})^2}w_1$$

$$\frac{\partial E}{\partial x_2} = (y - t)\frac{2e^{-k}}{(1 + e^{-k})^2}w_2$$

where $k = w_1x_1 + w_2x_2 + \mathrm{b}$. In the perceptron example in Fig. 16.2*B*, x_1 of the final layer (layer 2) is y_1, output of the top perceptron of layer 1 and x_2 is y_2, output of the bottom perceptron of layer 1. Now we are ready to proceed with the calculation of $\frac{\partial E}{\partial \theta}$ for the two perceptrons in the previous layer. Obviously, this process can be repeated if there are more layers.

Feedforward networks

By connecting layers of classifiers and feeding the output of each layer to the next, we form a feedforward network. Fig. 16.2*B* shows an example of a two-layer feedforward network. All our discussions on inference with and training of multilayer perceptrons (MLP) assume this property. In a feedforward network, all

outputs of an earlier layer go to one or more of the later layers. There are no connections from a later layer output to an earlier layer input. Therefore the backpropagation can simply iterate from the final stage backwards with no complications caused by feedback loops.

16.2 Convolutional neural networks

A deep learning procedure (LeCun et al., 2015) uses a hierarchy of feature extractors to learn complex features, which can achieve more accurate pattern recognition results if there is enough training data to allow the system to properly train the parameters of all the layers of feature extractors to automatically discover an adequate number of relevant patterns. There is one category of deep learning procedures that are easier to train and that can be generalized much better than others. These deep learning procedures are based on a particular type of feedforward network called the convolutional neural network (CNN).

The CNN was invented in late 1980s (LeCun et al., 1998). By the early 1990s, CNNs had been applied to automated speech recognition, optical character recognition (OCR), handwriting recognition, and face recognition (LeCun et al., 1990). However, until the late 1990s the mainstream of computer vision and that of automated speech recognition had been based on carefully engineered features. The amount of labeled data was insufficient for a deep learning system to compete with recognition or classification functions crafted by human experts. It was a common belief that it was computationally infeasible to automatically build hierarchical feature extractors that have enough layers to perform better than human-defined application-specific feature extractors.

Interest in deep feedforward networks was revived around 2006 by a group of researchers who introduced unsupervised learning methods that could create multilayer, hierarchical feature detectors without requiring labeled data (Hinton et al., 2006, Raina et al., 2009). The first major application of this approach was in speech recognition. The breakthrough was made possible by GPUs that allowed researchers to train networks ten times faster than traditional CPUs. This advancement, coupled with the massive amount of media data available online, drastically elevated the position of deep learning approaches. Despite their success in speech, CNN were largely ignored in the field of computer vision until 2012.

In 2012 a group of researchers from the University of Toronto trained a large, deep convolutional neural network to classify 1000 different classes in the ILSVRC contest (Krizhevsky et al., 2012). The network was huge by the norms of the time: It had approximately 60 million parameters and 650,000 neurons. It was trained on 1.2 million high-resolution images from the ImageNet database. The network was trained in only one week on two GPUs using a CUDA-based

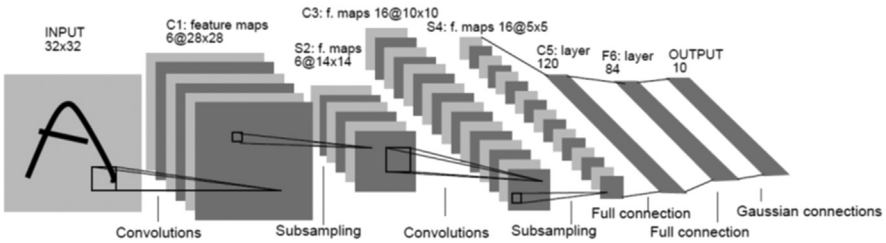


FIGURE 16.4

LeNet-5, a convolutional neural network for handwritten digit recognition. The letter A in the input should be classified as none of the ten classes (digits).

convolutional neural network library written by Alex Krizhevsky (Krizhevsky). The network achieved breakthrough results with a winning test error rate of 15.3%. In comparison, the second-place team that used the traditional computer vision algorithms had an error rate of 26.2%. This success triggered a revolution in computer vision, and CNN became a mainstream tool in computer vision, natural language processing, reinforcement learning, and many other traditional machine learning areas.

This section presents the sequential implementation of CNN inference and training. We will use LeNet-5, the network that was designed in the late 1980s for digit recognition (LeCun et al., 1990). As shown in Fig. 16.4, LeNet-5 is composed of three types of layers: convolutional layers, subsampling layers, and fully connected layers. These three types of layers continue to be the key components of today's neural networks. We will consider the logical design and sequential implementation of each type of layer. The input to the network is shown as a gray image with a handwritten digit represented as a 2D 32×32 pixel array. The last layer computes the output, which is the probability that the original image belongs to each one of the ten classes (digits) that the network is set up to recognize.

Convolutional neural network inference

The computation in a convolutional network is organized as a sequence of layers. We will call inputs to and outputs from layers *feature maps* or simply *features*. For example, in Fig. 16.4 the computation of the C1 convolutional layer at the input end of the network is organized to generate six output feature maps from the INPUT pixel array. The output to be produced for input feature maps consists of pixels, each of which is produced by performing a convolution between a small local patch of the feature map pixels produced by the previous layer (INPUT in the case of C1) and a set of weights (i.e., convolution filters as defined in Chapter 7: Convolution) called a *filter bank*. The convolution result is then fed into an activation function such as sigmoid to produce an output pixel in the

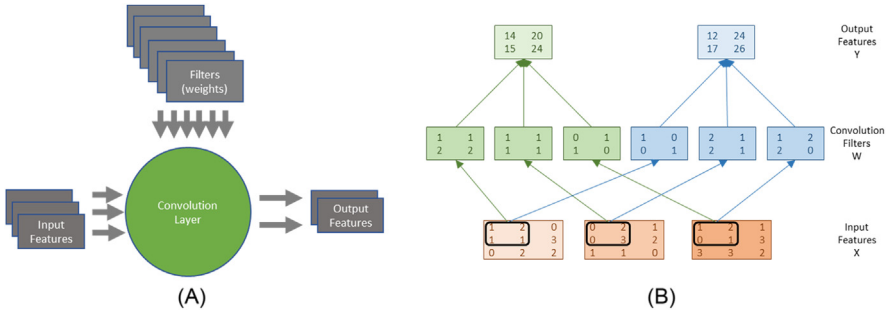


FIGURE 16.5

Forward propagation path of a convolutional layer.

output feature map. One can think of the convolutional layer for each pixel of an output feature map as a perceptron whose inputs are the patch of pixels in the input feature maps. That is, the value of each output pixel is the sum of convolution results from the corresponding patches in all input feature maps.

Fig. 16.5 shows a small convolutional layer example. There are three input feature maps, two output feature maps, and six filter banks. Different pairs of input and output feature map pairs in a layer use different filter banks. Since there are three input feature maps and two output feature maps in Fig. 16.5, we need $3 \times 2 = 6$ filter banks. For the C3 layer of LeNet in Fig. 16.4 there are six input feature maps and 16 output feature maps. Thus a total of $6 \times 16 = 96$ filter banks are used in C3.

Fig. 16.5B illustrates more details of the calculations done by a convolutional layer. We omitted the activation function for the output pixels for simplicity. We show that each output feature map is the sum of convolutions of all input feature maps. For example, the upper left corner element of output feature map 0 (value 14) is calculated as the convolution between the circled patch of input feature maps and the corresponding filter banks:

$$\begin{aligned}&(1,2,1,1)\cdot(1,1,2,2)+(0,2,0,3)\cdot(1,1,1,1)+(1,2,0,1)\cdot(0,1,1,0)\\&=1+2+2+2+0+2+0+3+0+2+0+0\\&=14\end{aligned}$$

One can also think of the three input maps as a 3D input feature map and the three filter banks as a 3D filter bank. Each output feature map is simply the 3D convolution result of the 3D input feature map and the 3D filter bank. In Fig. 16.5B the three 2D filter banks on the left form a 3D filter bank, and the three on the right form a second 3D filter bank. In general, if a convolutional layer has n input feature maps and m output feature maps, n^*m different 2D filter banks will be used. One can also think about these filter banks as m 3D filter banks. Although not shown in Fig. 16.4, all 2D filter banks used in LeNet-5 are 5×5 convolution filters.

Recall from Chapter 7, Convolution, that generating a convolution output image from an input image and a convolution filter requires one to make assumptions about the "ghost cells." Instead of making such assumptions, the LeNet-5 design simply uses two elements at the edge of each dimension as ghost cells. This reduces the size of each dimension by four: two at the top, two at the bottom, two at the left, and two at the right. As a result, we see that for layer C1, the 32 × 32 INPUT image results in an output feature map that is a 28 × 28 image. Fig. 16.4 illustrates this computation by showing that a pixel in the C1 layer is generated from a square (5 × 5, although not explicitly shown) patch of INPUT pixels.

We assume that the input feature maps are stored in a 3D array X[C, H, W], where C is the number of input feature maps, H is the height of each input map image, and W is the width of each input map image. That is, the highest-dimension index selects one of the feature maps (often referred to as channels), and the indices of the lower two dimensions select one of the pixels in an input feature map. For example, the input feature maps for the C1 layer are stored in X [1, 32, 32], since there is only one input feature map (INPUT in Fig. 16.4) that consists of 32 pixels in each of the x and y dimensions. This also reflects the fact that one can think of the 2D input feature maps to a layer altogether as forming a 3D input feature map.

The output feature maps of a convolutional layer are also stored in a 3D array Y[M, H − K + 1, W − K + 1], where M is the number of output feature maps and K is the height (and width) of each 2D filter. For example, the output feature maps for the C1 layer are stored in Y[6, 28, 28], since C1 generates six output feature maps using 5 × 5 filters. The filter banks are stored in a four-dimensional array W[M, C, K, K].[1] There are M × C filter banks. Filter bank W[m, c,_,_] is used when using input feature map X[c,_,_] to calculate output feature map Y[m, _,_]. Recall that each output feature map is the sum of convolutions of all input feature maps. Therefore we can consider the forward propagation path of a convolutional layer as set of M 3D convolutions in which each 3D convolution is specified by a 3D filter bank that is a C × K × K submatrix of W.

Fig. 16.6 shows a sequential C implementation of the forward propagation path of a convolutional layer. Each iteration of the outermost (*m*) for-loop (lines 04−12) generates an output feature map. Each iteration of the next two levels (*h* and *w*) of for-loops (lines 05−12) generates one pixel of the current output feature map. The innermost three loop levels (lines 08−11) perform the 3D convolution between the input feature maps and the 3D filter banks.

The output feature maps of a convolutional layer typically go through a subsampling layer (also known as a pooling layer). A subsampling layer reduces the size of image maps by combining pixels. For example, in Fig. 16.4, subsampling layer S2 takes the six input feature maps of size 28 × 28 and generates six

[1] Note that W is used for both the width of images and the name of the filter bank (weight) matrix. In each case the usage should be clear from the context.

```
01  void convLayer_forward(int M, int C, int H, int W, int K, float* X, float* W,
                           float* Y) {
02      int H_out = H - K + 1;
03      int W_out = W - K + 1;

04      for(int m = 0;  m < M;  m++)            // for each output feature map
05         for(int h = 0; h < H_out; h++)       // for each output element
06            for(int w = 0; w < W_out; w++) {
07               Y[m, h, w] = 0;
08               for(int c = 0;  c < C; c++)   // sum over all input feature maps
09                  for(int p = 0; p < K; p++) // KxK  filter
10                     for(int q = 0; q < K; q++)
11                        Y[m, h, w] +=  X[c, h + p, w + q] * W[m, c, p, q];
12            }
13   }
```

FIGURE 16.6

A C implementation of the forward propagation path of a convolutional layer.

```
01  void subsamplingLayer forward(int M, int H, int W, int K, float* Y, float*
S){
02     for(int m = 0;  m < M;  m++)             // for each output feature map
03        for(int h = 0; h < H/K; h++)          // for each output element,
04           for(int w = 0; w < W/K; w++) {     // this code assumes that H and W
05              S[m, x, y] = 0.;                // are multiples of K
06              for(int p = 0; p < K; p++) {    // loop over KxK input samples
07                 for(int q = 0; q < K; q++)
08                     S[m, h, w] += Y[m, K*h + p, K*w+ q] /(K*K);
09              }
                                 // add bias and apply non-linear activation
10              S[m, h, w] = sigmoid(S[m, h, w] + b[m]);
11           }
12  }
```

FIGURE 16.7

A sequential C implementation of the forward propagation path of a subsampling layer. The layer also includes an activation function, which is included in a convolutional layer if there is no subsampling layer after the convolutional layer.

feature maps of size 14 $\times$ 14. Each pixel in a subsampling output feature map is generated from a 2 $\times$ 2 neighborhood in the corresponding input feature map. The values of these four pixels are averaged to form one pixel in the output feature map. The output of a subsampling layer has the same number of output feature maps as the previous layer, but each map has half the number of rows and columns. For example, the number of output feature maps (six) of the subsampling layer S2 is the same as the number of its input feature maps, or the output feature maps of the convolutional layer C1.

Fig. 16.7 shows a sequential C implementation of the forward propagation path of a subsampling layer. Each iteration of the outermost (m) for-loop (lines 02−11) generates an output feature map. The next two levels (h and w) of for-loops (lines 03−11) generates individual pixels of the current output map. The two innermost for-loops (lines 06−09) sum up the pixels in the neighborhood. K is equal to 2 in our LeNet-5 subsampling example in Fig. 16.4. A bias value b[m] that is specific to each output feature map is then added to each output feature

map, and the sum goes through a sigmoid activation function. The reader should recognize that each output pixel is generated by the equivalent of a perceptron that takes four of the input pixels in each feature map as its input and generates a pixel in the corresponding output feature map. ReLU is another frequently used activation function that is a simple nonlinear filter that passes only nonnegative values: Y = X, if X $\geq$ 0 and 0 otherwise.

To complete our example, convolutional layer C3 has 16 output feature maps, each of which is a 10 $\times$ 10 image. This layer has 6 $\times$ 16 = 96 filter banks, and each filter bank has 5 $\times$ 5 = 25 weights. The output of C3 is passed into the subsampling layer S4, which generates 16 5 $\times$ 5 output feature maps. Finally, the last convolutional layer C5 uses 16 $\times$ 120 = 1920 5 $\times$ 5 filter banks to generate 120 one-pixel output features from its 16 input feature maps.

These feature maps are passed through fully connected layer F6, which has 84 output units, in which each output is fully connected to all inputs. The output is computed as a product of a weight matrix W with an input vector X, and then a bias is added and the output is passed through sigmoid. For the F6 example, W is a 120 $\times$ 84 matrix. In summary, the output is an 84-element vector Y6 = sigmoid (W * X + b). The reader should recognize that this is equivalent to 84 perceptrons, and each perceptron takes all 120 one-pixel x values generated by the C5 layer as its input. We leave the detailed implementation of a fully connected layer as an exercise.

The final stage is an output layer that uses Gaussian filters to generate a vector of ten elements, which correspond to the probability that the input image contains one of the ten digits.

Convolutional neural network backpropagation

Training of CNNs is based on the stochastic gradient descent method and the backpropagation procedure that were discussed in Section 16.1 (Rumelhart et al., 1986). The training dataset is labeled with the "correct answer." In the handwriting recognition example the labels give the correct letter in the image. The label information can be used to generate the "correct" output of the last stage: the correct probability values of the ten-element vector, where the probability of the correct digit is 1.0 and those for all other digits are 0.0.

For each training image, the final stage of the network calculates the loss (error) function as the difference between the generated output probability vector element values and the "correct" output vector element values. Given a sequence of training images, we can numerically calculate the gradient of the loss function with respect to the elements of the output vector. Intuitively, it gives the rate at which the loss function value changes when the values of the output vector elements change.

The backpropagation process starts by calculating the gradient of loss function $\frac{\partial E}{\partial y}$ for the last layer. It then propagates the gradient from the last layer toward the first layer through all layers of the network. Each layer receives as its input $\frac{\partial E}{\partial y}$

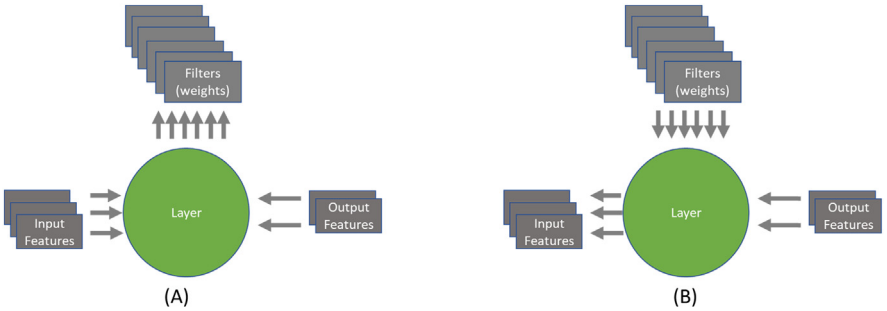


FIGURE 16.8

Backpropagation of (*A*) $\frac{\partial \mathbf{E}}{\partial \mathbf{W}}$ and (*B*) $\frac{\partial \mathbf{E}}{\partial \mathbf{X}}$ for a layer in CNN.

gradient with respect to its output feature maps (which is just the $\frac{\partial E}{\partial x}$ of the later layer) and computes its own $\frac{\partial E}{\partial x}$ gradient with respect to its input feature maps, as shown in Fig. 16.8B. This process repeats until it finishes adjusting the input layer of the network.

If a layer has learned parameters ("weights") w, then the layer also computes its $\frac{\partial E}{\partial w}$ gradient of loss with respect to its weights, as shown in Fig. 16.8A. For example, the fully connected layer is given as $y = w \cdot x$. The backpropagation of the gradient $\frac{\partial E}{\partial y}$ is given by the following equation:

$$\frac{\partial E}{\partial x} = w^T \frac{\partial E}{\partial y} \quad \text{and} \quad \frac{\partial E}{\partial w} = \frac{\partial E}{\partial y} x^T$$

This equation can be derived on an element-by-element basis, as we did for the two-layer perceptron example. Recall that each fully connected layer output pixel is calculated by a perceptron that takes the pixels in the input feature map as input. As we showed for training MLP in Section 16.1, $\frac{\partial E}{\partial x}$ for one of the inputs x is the sum of products between $\frac{\partial E}{\partial y}$ for each output y element to which the input element contributes and the w value via which the x value contributes to the y value. Because each row of the w matrix relates all the x elements (columns) to a y element (one of the rows) for the fully connected layer, each column of w (i.e., row of w^T) relates all y (i.e., $\frac{\partial E}{\partial y}$) elements back to an x (i.e., $\frac{\partial E}{\partial x}$) element, since transposition switches the roles of rows and columns. Thus the matrix-vector multiplication $w^T \frac{\partial E}{\partial y}$ results in a vector that has the $\frac{\partial E}{\partial x}$ values for all input x elements.

Similarly, since each w element is multiplied by one x element to generate a y element, the $\frac{\partial E}{\partial w}$ of each w element can be calculated as the product of an element of $\frac{\partial E}{\partial y}$ with an x element. Thus the matrix multiplication between $\frac{\partial E}{\partial y}$ (a single-column matrix) and x^T (a single-row matrix) results in a matrix of $\frac{\partial E}{\partial w}$ values for all w elements of the fully connected layer. This can also be seen as an outer product between the $\frac{\partial E}{\partial y}$ and x vectors.

Let's turn our attention to the backpropagation for a convolutional layer. We will start from the calculation of $\frac{\partial E}{\partial x}$ from $\frac{\partial E}{\partial y}$, which will ultimately be used to

calculate the gradients for the previous layer. The gradient $\frac{\partial E}{\partial x}$ with respect to the channel c of input x is given as the sum of the "backward convolution" with the corresponding $W(m, c)$ over all the m layer outputs:

$$\frac{\partial E}{\partial x}(c,h,w) = \sum_{m=0}^{M-1}\sum_{p=0}^{K-1}\sum_{q=0}^{K-1}\left(\frac{\partial E}{\partial y}(m,h-p,w-q) * w(m,c,k-p,k-q)\right)$$

The backward convolution through the $h-p$ and $w-q$ indexing allows the gradients of all output y elements that received contributions from an x element in the forward convolution to contribute to the gradient of that x element through the same weights. This is because in the forward inference of the convolutional layer, any change in the value of the x element is multiplied by these w elements and contributes to the change in the loss function value through these y elements. Fig. 16.9 shows the indexing pattern using a small example with 3 × 3 filter banks. The nine shaded y elements in the output feature map are the y elements that receive contributions from $x_{h,w}$ in forward interference. For example, input element $x_{h,w}$ contributes to $y_{h-2,w-2}$ through multiplication with $w_{2,2}$ and to $y_{h,w}$ through multiplication with $w_{0,0}$. Therefore during backpropagation, $\frac{\partial E}{\partial x_{h,w}}$ should receive contribution from the $\frac{\partial E}{\partial y}$ values of these nine elements, and the computation is equivalent to a convolution with a transposed filter bank w^T.

Fig. 16.10 shows the C code for calculating each element of $\frac{\partial E}{\partial x}$ for each input feature map. Note that the code assumes that $\frac{\partial E}{\partial y}$ has been calculated for all the output feature maps of the layer and passed in with a pointer argument dE_dY. This is a reasonable assumption, since $\frac{\partial E}{\partial y}$ for the current layer is the $\frac{\partial E}{\partial x}$ for its immediate next layer, whose gradients should have been calculated in the backpropagation before reaching the current layer. It also assumes that the space of $\frac{\partial E}{\partial x}$ has been allocated in the device memory whose handle is passed in as a pointer argument dE_dX. The function generates all the elements of $\frac{\partial E}{\partial x}$.

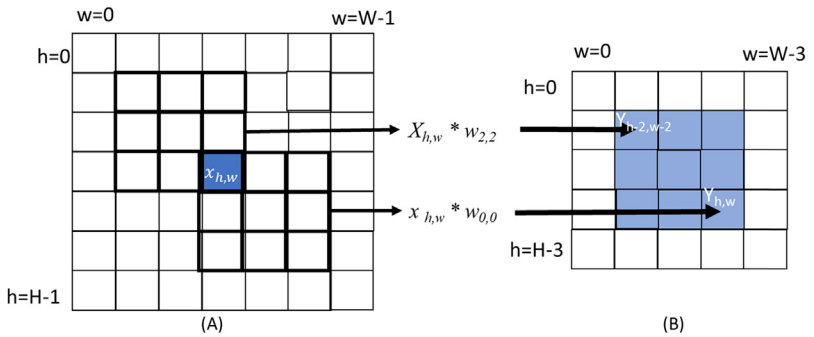


FIGURE 16.9

Convolutional layer. Backpropagation of (*A*) $\partial E/\partial w$ and (*B*) $\partial E/\partial x$.

```
01   void convLayer_backward_x_grad(int M, int C, int H_in, int W_in, int K,
                                    float* dE_dY, float* W, float* dE_dX) {
02      int H_out = H_in - K + 1;
03      int W_out = W_in - K + 1;
04      for(int c = 0;  c < C; c++)
05         for(int h = 0; h < H_in; h++)
06            for(int w = 0; w < W_in; w++)
07                dE_dX[c, h, w] = 0;

08      for(int m = 0;  m < M;  m++)
09        for(int h = 0; h < H-1; h++)
10          for(int w = 0; w < W-1; w++)
11            for(int c = 0;  c < C; c++)
12              for(int p = 0; p < K; p++)
13                for(int q = 0; q < K; q++)
14                  if(h-p >= 0 && w-p >=0 && h-p < H_out and w-p < W_OUT)
15                     dE_dX[c, h, w] += dE_dY[m, h-p, w-p] * W[m, c, k-p, k-q];
16    }
```

FIGURE 16.10

$\frac{\partial E}{\partial X}$ calculation of the backward path of a convolutional layer.

```
01  void convLayer_backward_w_grad(int M, int C, int H, int W, int K, float*
                                    dE_dY, float* X, float* dE_dW) {
02     int H_out = H - K + 1;
03     int W_out = W - K + 1;
04     for(int m = 0; m < M; m++)
05       for(int c = 0; c < C; c++)
06         for(int p = 0; p < K; p++)
07           for(int q = 0; q < K; q++)
08              dE_dW[m, c, p, q] = 0.;

09      for(int m = 0;  m < M;  m++)
10        for(int h = 0; h < H_out; h++)
11          for(int w = 0; w < W_out; w++)
12            for(int c = 0;  c < C; c++)
13              for(int p = 0; p < K; p++)
14                for(int q = 0; q < K; q++)
15                   dE_dW[m, c, p, q] += X[c, h+p, w+q] * dE_dY[m, c, h, w];
16    }
```

FIGURE 16.11

$\frac{\partial E}{\partial W}$ calculation of the backward path of a convolutional layer.

The sequential code for calculating $\frac{\partial E}{\partial w}$ for a convolutional layer computation is similar to that of $\frac{\partial E}{\partial x}$ and is shown in Fig. 16.11. Since each $W(m, c)$ affects all elements of output $Y(m)$, we should accumulate gradients for each $W(m, c)$ over all pixels in the corresponding output feature map:

$$\frac{\partial E}{\partial W}(m,c,p,q) = \sum_{h=0}^{H_{out}-1}\sum_{w=0}^{W_{out}-1}\left(X(c,h+p,w+q) * \frac{\partial E}{\partial Y}(m,h,w)\right)$$

Note that while the calculation of $\frac{\partial E}{\partial x}$ is important for propagating the gradient to the previous layer, the calculation of the $\frac{\partial E}{\partial w}$ is key to the adjustments to the weight values of the current layer.

```
01  void convLayer_batched(int N, int M, int C, int H, int W, int K, float* X,
                           float* W, float* Y) {
02    int H_out = H - K + 1;
03    int W_out = W - K + 1;
04    for(int n = 0;  n < N;  n++)       // for each sample in the mini-batch
05       for(int m = 0;  m < M;  m++)          // for each output feature map
06          for(int h = 0; h < H_out; h++)     // for each output element
07             for(int w = 0; w < W_out; w++) {
08                Y[n, m, h, w] = 0;
09                for (int c = 0;  c < C; c++ // sum over all input feature maps
10                  for (int p = 0; p < K; p++)            // KxK  filter
11                    for (int q = 0; q < K; q++)
12                       Y[n,m,h,w] = Y[n,m,h,w] + X[n, c, h+p, w+q]*W[m,c,p,q];
13             }
14      }
```

FIGURE 16.12

Forward path of a convolutional layer with minibatch training.

After the $\frac{\partial E}{\partial w}$ values at all filter bank element positions have been computed, weights are updated to minimize the expected error using the formula presented in Section 16.1: $w \leftarrow w - \varepsilon^* \frac{\partial E}{\partial w}$, where ε is the learning rate constant. The initial value of ε is set empirically and reduced through the epochs according to the rule defined by user. The value of ε is reduced through the epochs to ensure that the weights converge to a minimal error. Recall that the negative sign of the adjustment term causes the change to be opposite to the direction of the gradient so that the change will likely reduce the error. Recall also that the weight values of the layers determine how the input is transformed through the network. The adjustment of these weight values of all the layers adapts the behavior of the network. That is, the network "learns" from a sequence of labeled training data and adapts its behavior by adjusting all weight values at all its layers for inputs whose inference results were incorrect and triggered backpropagation.

As we discussed in Section 16.1, backpropagation is typically triggered after a forward pass has been performed on a minibatch of N images from the training dataset, and the gradients have been computed for this minibatch. The learned weights are updated with the gradients that are calculated for the minibatch, and the process is repeated with another minibatch.[2] This adds one additional dimension to all previously described arrays, indexed with n, the index of the sample in the minibatch. It also adds one additional loop over samples.

Fig. 16.12 shows the revised forward path implementation of a convolutional layer. It generates the output feature maps for all the samples of a minibatch.

[2] If we work by the "optimization book," we should return used samples back to the training set and then build a new minibatch by randomly picking the next samples. In practice, we iterate sequentially over the whole training set. In machine learning, a pass through the training set is called an *epoch*. Then we shuffle the whole training set and start the next epoch.

16.3 Convolutional layer: a CUDA inference kernel

The computation pattern in training a convolutional neural network is like matrix multiplication: It is both compute intensive and highly parallel. We can process different samples in a minibatch, different output feature maps for the same sample, and different elements for each output feature map in parallel. In Fig. 16.12 the n-loop (line 04, over samples in a minibatch), the m-loop (line 05, over output feature maps), and the nested h-w-loops (lines 06−07, over pixels of each output feature map) are all parallel loops in that their iterations can be executed in parallel. These four loop levels together offer a massive level of parallelism.

The innermost three loop levels, the c-loop (over the input feature maps or channels) and the nested p-q-loops (over the weights in a filter bank), also offer a significant level of parallelism. However, to parallelize them, one would need to use atomic operations in accumulating into the *Y* elements, since different iterations of these loop levels can perform read-modify-write on the same *Y* elements. Therefore we will keep these loops serial unless we really need more parallelism.

Assuming that we exploit the four levels of "easy" parallelism (*n*, *m*, *h*, *w*) in the convolutional layer, the total number of parallel iterations is the product N*M*H_out*W_out. This high degree of available parallelism makes the convolutional layer an excellent candidate for GPU acceleration. We can easily design a kernel with thread organizations that are designed to capture the parallelism.

We first need to make some high-level design decisions about the thread organization. Assume that we will have each thread compute one element of one output feature map. We will use 2D thread blocks, in which each thread block computes a tile of TILE_WIDTH × TILE_WIDTH pixels in one output feature map. For example, if we set TILE_WIDTH = 16, we would have a total of 256 threads per block. This captures part of the nested h-w-loop level parallelism in processing the pixels of each output feature map.

Blocks can be organized into a 3D grid in several different ways. Each option designates the grid dimensions to capture the *n*, *m*, and *h*-*w* parallelism in different combinations. We will present the details of one of the options and leave it as an exercise for the reader to explore different options and evaluate the potential pros and cons of each option. The option that we present in detail is as follows:

1. The first dimension (*X*) corresponds to the (*M*) output features maps covered by each block.
2. The second dimension (*Y*) reflects the location of a block's output tile inside the output feature map.
3. The third dimension (*Z*) in the grid corresponds to samples (N) in the minibatch.

Fig. 16.13 shows the host code that launches a kernel based on the thread organization proposed above. The number of blocks in the *X* and *Z* dimensions of the grid are straightforward; They are simply *M*, the number of output feature

```
01   # define TILE WIDTH 16
02   W_grid = W_out/TILE_WIDTH;   // number of horizontal tiles per output map
03   H_grid = H_out/TILE_WIDTH;   // number of vertical tiles per output map
04   T = H grid * W grid;
05   dim3 blockDim(TILE WIDTH, TILE WIDTH, 1);
06   dim3 gridDim(M, T, N);
07   ConvLayerForward_Kernel<<< gridDim, blockDim>>>(…);
```

FIGURE 16.13

Host code for launching a convolutional layer kernel.

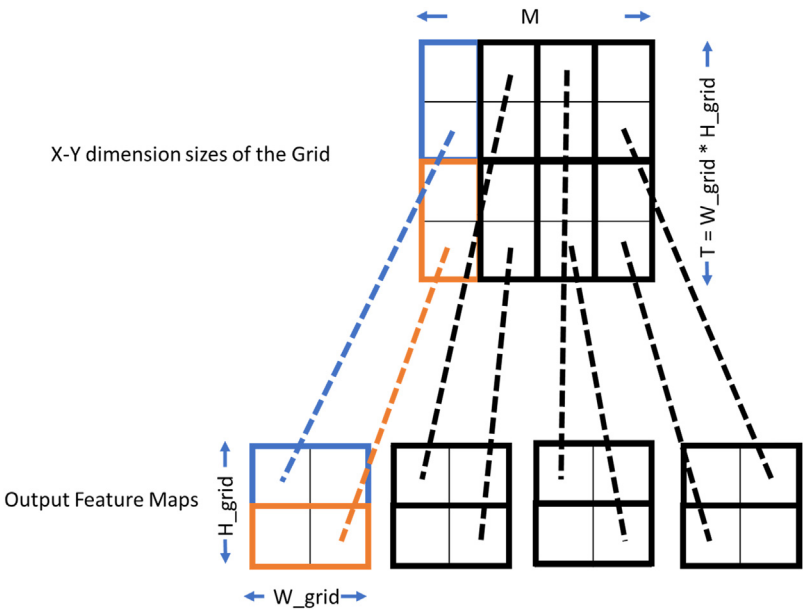


FIGURE 16.14

Mapping output feature map tiles to blocks in the *X-Y* dimension of the grid.

maps, and *N*, the number of samples in a minibatch. The arrangement in the *Y* dimension is a little more complex and is illustrated in Fig. 16.14. Ideally, we would like to dedicate two dimensions of the grid indices to the vertical and horizontal tile indices for simplicity. However, we have only one dimension for both, since we are using *X* for the output feature map index and *Z* for the sample index in a minibatch. Therefore we linearize the tile indices to encode both the horizontal and vertical tile indices of output feature map tiles.

In the example in Fig. 16.14, each sample has four output feature maps ($M = 4$), and each output feature map consists of 2×2 tiles (H_grid = 2 in line 02 and W_grid = 2 in line 03) of $16 \times 16 = 256$ pixels each. The grid organization assigns each block to calculate one of these tiles.

```
01  __global__ void
02  ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W,
                            float* Y) {
03      int m = blockIdx.x;
04      int h = (blockIdx.y / W_grid)*TILE_WIDTH + threadIdx.y;
05      int w = (blockIdx.y % W_grid)*TILE_WIDTH + threadIdx.x;
06      int n = blockIdx.z;
07      float acc = 0.;
08      for (int c = 0;  c < C; c++) {          // sum over all input channels
09          for (int p = 0; p < K; p++)         // loop over KxK  filter
10              for (int q = 0; q < K; q++)
11                  acc += X[n, c, h + p, w + q] * W[m, c, p, q];
12      }
13      Y[n, m, h, w] = acc;
14  }
```

FIGURE 16.15

Kernel for the forward path of a convolutional layer.

We have already assigned each output feature map to the *X* dimension, which is reflected as the four blocks in the *X* dimension, each corresponding to one of the output feature maps. As shown in the bottom of Fig. 16.14, we linearize the four tiles in each output feature map and assign them to the blocks in the *Y* dimension. Thus tiles (0, 0), (0, 1), (1, 0), and (1, 1) are mapped using row-major order to the blocks with blockIdx.y values 0, 1, 2, and 3, respectively. Thus the total number of bocks in the *Y* dimension is 4 (T = H_grid*W_grid = 4 in line 04). Thus we will launch a grid with gridDim (4, 4, *N*) in lines 06−07.

Fig. 16.15 shows a kernel based on the thread organization above. Note that in the code, we use multidimensional indices in array accesses for clarity. We leave it to the reader to translate this pseudo-code into regular C, assuming that *X*, *Y*, and *W* must be accessed via linearized indexing based on row-major layout (Chapter 3, Multidimensional Grids and Data).

Each thread starts by generating the *n* (batch), *m* (feature map), *h* (vertical), and *w* (horizontal) indices of its assigned output feature map pixel. The *n* (line 06) and *m* (line 03) indices are straightforward, given the host code. For the *h* index calculation in line 04, the blockIdx.y value is first divided by W_grid to recover the tile index in the vertical direction, as illustrated in Fig. 16.13. This tile index is then expanded by the TILE_WIDTH and added to the threadIdx.y to form the actual vertical pixel index into the output feature map (line 04). The derivation of the horizontal pixel index is similar (line 05).

The kernel in Fig. 16.15 has a high degree of parallelism but consumes too much global memory bandwidth. As in the convolution pattern discussions in Chapter 7, Convolution, the execution speed of the kernel will be limited by the global memory bandwidth. As we also saw in Chapter 7, Convolution, we can use constant memory caching and shared memory tiling to dramatically reduce the global memory traffic and improve the execution speed of the kernel. These optimizations to the convolution inference kernel are left as an exercise for the reader.

16.4 Formulating a convolutional layer as GEMM

We can build an even faster convolutional layer by representing it as an equivalent matrix multiplication operation and then using a highly efficient GEMM (general matrix multiply) kernel from the CUDA linear algebra library cuBLAS. This method was proposed by Chellapilla et al. (2006). The central idea is unfolding and duplicating input feature map pixels in such a way that all elements that are needed to compute one output feature map pixel will be stored as one sequential column of the matrix that is thus produced. This formulates the forward operation of the convolutional layer to one large matrix multiplication.[3]

Consider a small example convolutional layer that takes as input $C = 3$ feature maps, each of which is of size 3×3, and produces $M = 2$ output features, each of which is of size 2×2, as shown in Fig. 16.5 and again, for convenience, at the top of Fig. 16.16. It uses $M \times C = 6$ filter banks, each of which is 2×2. The matrix version of this layer will be constructed in the following way.

First, we will rearrange all input pixels. Since the results of the convolutions are summed across input features, the input features can be concatenated into one large matrix. Each input feature map becomes a section of rows in the large matrix. As shown in Fig. 16.16, input feature maps 0, 1, and 2 become the top, middle, and bottom sections, respectively, of the "input features X_unrolled" matrix.

The rearrangement is done so that each column of the resulting matrix contains all the input values necessary to compute one element of an output feature. For example, in Fig. 16.16, all the input feature pixels that are needed for calculating the value at (0, 0) of output feature map 0 are circled in the input feature maps:

$$\begin{aligned} Y_{0,0,0} &= (1,2,1,1)\cdot(1,1,2,2) + (0,2,0,3)\cdot(1,1,1,1) + (1,2,0,1)\cdot(0,1,1,0) \\ &= 1+2+2+2+0+2+0+3+0+2+0+0 \\ &= 14 \end{aligned}$$

where the first term of each inner product is a vector formed by linearizing the patch of x pixels circled in Fig. 16.16. The second term is a vector that is formed by linearizing the filter bank that is used for the convolution. In both cases, linearization is done by using the row-major order. It is also clear that we can reformulate the three inner products into one inner product:

$$\begin{aligned} Y_{0,0,0} &= (1,2,1,1,0,2,0,3,1,2,0,1)\cdot(1,1,2,2,1,1,1,1,0,1,1,0) \\ &= 1+2+2+2+0+2+0+3+0+2+0+0 \\ &= 14 \end{aligned}$$

[3] See also https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/ for a very detailed explanation.

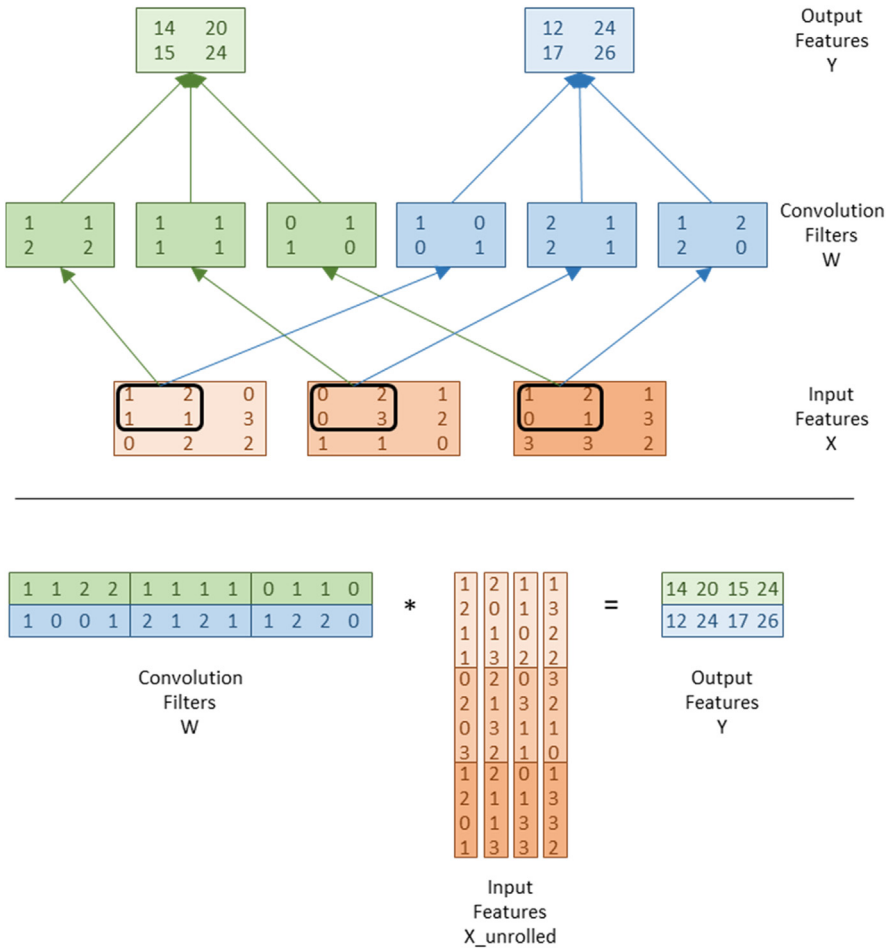


FIGURE 16.16

Formulation of convolutional layer as GEMM.

As shown in the bottom of Fig. 16.16, the concatenated vector from the filter banks becomes row 0 of the filter matrix, and the concatenated vector from the input feature maps becomes column 0 of the input feature map unrolled matrix. During matrix multiplication the row of the filter bank matrix and the column of the input feature matrix will produce one pixel of the output feature map.

Note that matrix multiplication of the 2×12 filter matrix and the 12×8 input feature map matrix produces a 2×8 output feature map matrix. The top section of the output feature map matrix is the linearized form of output feature map 0, and the bottom is output feature map 1. Both are already in row-major order, so they can be used as individual input feature maps for the next layer. As

for the filter banks, each row of the filter matrix is simply the row-major order view of the original filter bank. Thus the filter matrix is simply the concatenation of all the original filter banks. There is no physical rearrangement or relocation of filter elements that are involved.

We make an important observation that the patches of input feature map pixels for calculating different pixels of the output feature map overlap with each other, owing to the nature of convolution. This means that each input feature map pixel is replicated multiple times as we produce the expanded input feature matrix. For example, the center pixel of each 3 × 3 input feature map is used four times to compute the four pixels of an output feature, so it will be duplicated four times. The middle pixel on each edge is used two times, so it will be duplicated two times. The four pixels at corners of each input feature are used only one time and will not need to be duplicated. Therefore the total number of pixels in the expanded input feature matrix section is $4 * 1 + 2 * 4 + 1 * 4 = 16$. Since each original input feature map has only nine pixels, the GEMM formulation incurs an expansion ratio of $16/9 = 1.8$ for representing input feature maps.

In general, the size of the unrolled input feature map matrix can be derived from the number of input feature map elements that are required to generate each output feature map element. The height, or the number of rows, of the expanded matrix is the number of input feature elements contributing to each output feature map element, which is C*K*K: each output element is the convolution of K*K elements from each input feature map and there are C input feature maps. In our example, the K is 2 since the filter bank is 2 × 2 and there are three input feature maps. Thus the height of the expanded matrix should be $3 * 2 * 2 = 12$, which is exactly the height of the matrix shown in Fig. 16.16.

The width, or the number columns, of the expanded matrix is the number of elements in each output feature map. If each output feature map is an H_out × W_out matrix, the number of columns of the expanded matrix is H_out*W_out. In our example, each output feature map is a 2 × 2 matrix, yielding four columns in the expanded matrix. Note that the number of output feature maps *M* does not play into the duplication. This is because all output feature maps are computed from the same expanded input feature map matrix.

The ratio of expansion for the input feature maps is the size of the expanded matrix over the total size of the original input feature maps. The reader should verify that the expansion ratio is as follows:

$$\frac{C * K * K * H\_out * W\_out}{C * H\_in * W\_in}$$

where H_in and W_in are the height and width, respectively, of each input feature map. In our example the ratio is $(3 * 2 * 2 * 2 * 2)/(3 * 3 * 3) = 16/9$. In general, if the input feature maps and output feature maps are much larger than the filter banks, the ratio of expansion will approach K*K.

The filter banks are represented as a filter bank matrix in a fully linearized layout, in which each row contains all weight values that are needed to produce one output feature map. The height of the filter bank matrix is the number of output feature maps (*M*). Computing different output feature maps involves sharing a single expanded input feature map matrix. The width of the filter bank matrix is the number of weight values that are needed for generating each output feature map element, which is C*K*K. Recall that there is no duplication when placing the weight values into the filter bank matrix. For example, the filter bank matrix is simply a concatenated arrangement of the six filter banks in Fig. 16.16.

When we multiply the filter bank matrix *W* by the expanded input matrix X_unrolled, the output feature maps are computed as a matrix *Y* of height *M* and width H_out*W_out. That is, each row of *Y* is a complete output feature map.

Let's discuss now how we can implement this algorithm in CUDA. Let's first discuss the data layout. We can start from the layout of the input and output matrices.

1. We assume that the input feature map samples in a minibatch will be supplied in the same way as those for the basic CUDA kernel. It is organized as an N $\times$ C $\times$ H $\times$ W array, where N is the number of samples in a minibatch, C is the number of input feature maps, H is the height of each input feature map, and W is the width of each input feature map.
2. As we showed in Fig. 16.16, the matrix multiplication will naturally produce an output Y stored as an M $\times$ (H_out*W_out) array. This is what the original basic CUDA kernel would produce.
3. Since the filter bank matrix does not involve duplication of weight values, we assume that it will be prepared ahead of time and organized as an M $\times$ C $\times$ K^2 array as illustrated in Fig. 16.16.

The preparation of the unrolled input feature map matrix X_unroll is more complex. Since each expansion increases the size of input by up to K^2 times, the expansion ratio can be very large for typical K values of 5 or larger. The memory footprint for keeping all sample input feature maps for a minibatch can be prohibitively large. To reduce the memory footprint, we will allocate only one buffer for X_unrolled [C * K * K* H_out * W_out]. We will reuse this buffer by looping over samples in the minibatch. During each iteration we convert the sample input feature map from its original form into the unrolled matrix.

Fig. 16.17 shows a sequential function that produces the X_unroll array by gathering and duplicating the elements of an input feature map X. The function uses five levels of loops. The innermost two levels of for-loop (w and h, lines 08−13) place one input feature map element for each of the output feature map elements. The next two levels (p and q, lines 06−14) repeat the process for each of the K*K filter matrix elements. The outermost loop repeats the process of all

```
01    void unroll(int C, int H, int W, int K, float* X, float* X unroll) {
02        int H_out = H - K + 1;
03        int W_out = W - K + 1;
04        for(int c = 0; c < C; c++) {
              // Beginning row index of the section for channel C input feature
              // map in the unrolled matrix
05            w_base = c * (K*K);
06            for(int p = 0; p < K; p++) {
07                for(int q = 0; q < K; q++) {
08                    for(int h = 0; h <  H_out; h++) {
09                        int h unroll = w base + p*K + q;
10                        for(int w = 0; w < W out; w ++) {
11                            int w_unroll = h * W_out + w;
12                            X_unroll[h_unroll, w_unroll) = X(c, h + p, w + q);
13                        }
14                    }
15                }
16            }
17        }
18    }
```

FIGURE 16.17

A C function that generates the unrolled X matrix. The array accesses are in multidimensional indexing form for clarity and need to be linearized for the code to be compilable.

input feature maps. This implementation is conceptually straightforward and can be quite easily parallelized since the loops do not impose dependencies among their iterations. Also, successive iterations of the innermost loop (w, lines 10−13) read from a localized tile of one of the input feature maps in X and write into sequential locations (same row in X_unroll) in the expanded matrix X_unroll. This should result in efficient memory bandwidth usage on a CPU.

We are now ready to design a CUDA kernel which implements the input feature map unrolling. Each CUDA thread will be responsible for gathering (K*K) input elements from one input feature map for one element of an output feature map. The total number of threads will be (C * H_out * W_out). We will use one-dimensional thread blocks and extract multidimensional indices from the linearized thread index.

Fig. 16.18 shows an implementation of the unroll kernel. Note that each thread will build a K*K section of a column, shown as a shaded box in the Input Features X_Unrolled array in Fig. 16.16. Each such section contains all elements of a patch of the input feature map X from channel c, required for performing a convolution operation with the corresponding filter to produce one element of output Y.

Comparing the loop structures of Figs. 16.17 and 16.18 shows that the innermost two loop levels in Fig. 16.17 have been changed into outer level loops in Fig. 16.18. This interchange allows the work for collecting the input elements that are needed for calculating output elements to be done in parallel by multiple threads. Furthermore, having each thread collect all input feature map elements from an input feature map that are needed for generating an output generates a coalesced memory write pattern. As illustrated in Fig. 16.16, adjacent threads will

```
01      __global__ void
02    unroll_Kernel(int C, int H, int W, int K, float* X, float* X_unroll) {
03        int t = blockIdx.x * blockDim.x + threadIdx.x;
04        int H_out = H - K + 1;
05        int W_out = W - K + 1;
          // Width of the unrolled input feature matrix
06        int W_unroll = H_out * W_out;
07        if (t < C * W_unroll) {
              // Channel of the input feature map being collected by the thread
08            int c = t / W_unroll;
              // Column index of the unrolled matrix to write a strip of
              // input elements into (also, the linearized index of the output
              // element for which the thread is collecting input elements)
09            int w_unroll = t % W_unroll;
              // Horizontal and vertical indices of the output element
10            int h_out = w_unroll / W_out;
11            int w_out = w_unroll % W_out;
              // Starting row index for the unrolled matrix section for channel c
12            int w_base = c * K * K;
13            for(int p = 0; p < K; p++)
14                for(int q = 0; q < K; q++) {
                      // Row index of the unrolled matrix for the thread to write
                      // the input element into for the current iteration
15                    int h_unroll = w_base + p*K + q;
16                    X_unroll[h_unroll, w_unroll] = X[c, h_out + p, w_out + q];
17            }
18        }
19    }
```

FIGURE 16.18

A CUDA kernel implementation for unrolling input feature maps. The array accesses are in multidimensional indexing form for clarity and need to be linearized for the code to be compilable.

be writing adjacent X_unroll elements in a row as they all move vertically to complete their sections. The read access patterns to X are similar and can be analyzed by an inspection of the w_out values for adjacent threads. We leave the detailed analysis of the read access pattern as an exercise.

An important high-level assumption is that we keep the input feature maps, filter bank weights, and output feature maps in the device memory. The filter bank matrix is prepared once and stored in the device global memory for use by all input feature maps. For each sample in the minibatch, we launch the unroll_Kernel to prepare an expanded matrix and launch a matrix multiplication kernel, as outlined in Fig. 16.16.

Implementing convolutions with matrix multiplication can be very efficient, since matrix multiplication is highly optimized on all hardware platforms. Matrix multiplication is especially fast on GPUs because it has a high ratio of floating-point operations per byte of global memory data access. This ratio increases as the matrices get larger, meaning that matrix multiplication is less efficient on small matrices. Accordingly, this approach to convolution is most effective when it creates large matrices for multiplication.

As we mentioned earlier, the filter bank matrix is an $M \times (C * K * K)$ matrix and the expanded input feature map matrix is a $(C * K * K) \times (H\_out * W\_out)$

matrix. Note that except for the height of the filter bank matrix, the sizes of all dimensions depend on products of the parameters to the convolution, not the parameters themselves. While individual parameters can be small, their products tend to be large. For example, it is often true that in early layers of a convolutional network, C is small, but H_out and W_out are large. On the other hand, at the end of the network, C is large, but H_out and W_out are small. Hence the product C*H_out*W_out is usually large for all layers. This means that the sizes of the matrices tend to be consistently large for all layers, and so the performance using this approach tends to be high.

One disadvantage of forming the expanded input feature map matrix is that it involves duplicating the input data up to K*K times, which can require the allocation of a prohibitively large amount of memory. To work around this limitation, implementations such as the one shown in Fig. 16.16 materialize the X_unroll matrix piece by piece, for example, by forming the expanded input feature map matrix and calling matrix multiplication iteratively for each sample of the minibatch. However, this limits the parallelism in the implementation, and can sometimes lead to cases where the matrix multiplications are too small to effectively utilize the GPU. Another disadvantage of this formulation is that it lowers the computational intensity of the convolutions because X_unroll must be written and read, in addition to reading X itself, requiring significantly more memory traffic than the direct approach. Accordingly, the highest performance implementation has even more complex arrangements in realizing the unrolling algorithm to both maximize GPU utilization while keeping the reading from DRAM minimal. We will come back to this point when we present the CUDNN approach in the next section.

16.5 CUDNN library

CUDNN is a library of optimized routines for implementing deep learning primitives. It was designed to make it much easier for deep learning frameworks to take advantage of GPUs. It provides a flexible and easy-to-use C-language deep learning API that integrates neatly into existing deep learning frameworks (e.g., Caffe, Tensorflow, Theano, Torch). The library requires that input and output data be resident in the GPU device memory, as we discussed in the previous section. This requirement is analogous to that of cuBLAS.

The library is thread-safe in that its routines can be called from different host threads. Convolutional routines for the forward and backward paths use a common descriptor that encapsulates the attributes of the layer. Tensors and filters are accessed through opaque descriptors, with the flexibility to specify the tensor layout using arbitrary strides along each dimension. The most important computational primitive in CNN is a special form of batched convolution.

Table 16.1 Convolution parameters for CUDNN. Note that the CUDNN naming convention is slightly different from what we used in previous sections.

Parameter	Meaning
N	Number of images in minibatch
C	Number of input feature maps
H	Height of input image
W	Width of input image
K	Number of output feature maps
R	Height of filter
S	Width of filter
u	Vertical stride
v	Horizontal stride
pad_h	Height of zero padding
pad_w	Width of zero padding

In this section we describe the forward form of this convolution. The CUDNN parameters that govern this convolution are listed Table 16.1.

There are two inputs to the convolution:

1. D is a four-dimensional N × C × H × W tensor, which contains the input data.[4]
2. F is a four-dimensional K × C × R × S tensor, which contains the convolutional filters.

The input data array (tensor) D ranges over N samples in a minibatch, C input feature maps per sample, H rows per input feature map, and W columns per input feature map. The filters range over K output feature maps, C input feature maps, R rows per filter bank, and S columns per filter bank. The output is also a four-dimensional tensor O that ranges over N samples in the minibatch, K output feature maps, P rows per output feature map, and Q columns per output feature map, where P = f(H; R; u; pad_h) and Q = f(W; S; v; pad_w), meaning that the height and width of the output feature maps depend on the input feature map and filter bank height and width, along with padding and striding choices. The striding parameters u and v allow the user to reduce the computational load by computing only a subset of the output pixels. The padding parameters allow the user to specify how many rows or columns of 0 entries are appended to each feature map for improved memory alignment and/or vectorized execution.

[4] *Tensor* is a mathematical term for arrays that have more than two dimensions. In mathematics, matrices have only two dimensions. Arrays with three or more dimensions are called tensors. For the purpose of this book, a T-dimensional tensor can be treated simply as a T-dimensional array.

CUDNN (Chetlur et al., 2014) supports multiple algorithms for implementing a convolutional layer: matrix multiplication–based GEMM (Tan et al., 2011) and Winograd (Lavin & Scott, 2016), FFT-based (Vasilache et al., 2014), and so on. The GEMM-based algorithm to implement the convolutions with a matrix multiplication is similar to the approach presented in Section 16.4. As we discussed at the end of Section 16.4, materializing the expanded input feature matrix in global memory can be costly in terms of both global memory space and bandwidth consumption. CUDNN avoids this problem by lazily generating and loading the expanded input feature map matrix X_unroll into on-chip memory only, rather than by gathering it in off-chip memory before calling a matrix multiplication routine. NVIDIA provides a matrix multiplication–based routine that achieves a high utilization of the maximal theoretical floating-point throughput on GPUs. The algorithm for this routine is similar to the algorithm described by Tan et al. (2011). Fixed-size submatrices of the input matrices A and B are successively read into on-chip memory and are then used to compute a submatrix of the output matrix C. All indexing complexities that are imposed by the convolution are handled in the management of tiles in this routine. We compute on tiles of A and B while fetching the next tiles of A and B from off-chip memory into on-chip caches and other memories. This technique hides the memory latency that is associated with the data transfer, allowing the matrix multiplication computation to be limited only by the time it takes to perform the arithmetic calculations.

Since the tiling that is required for the matrix multiplication routine is independent of any parameters from the convolution, the mapping between the tile boundaries of X_unroll and the convolution problem is nontrivial. Accordingly, the CUDNN approach entails computing this mapping and using it to load the correct elements of A and B into on-chip memories. This happens dynamically as the computation proceeds, which allows the CUDNN convolution implementation to exploit optimized infrastructure for matrix multiplication. It requires additional indexing arithmetic compared to a matrix multiplication, but it fully leverages the computational engine of matrix multiplication to perform the work. After the computation is complete, CUDNN performs the required tensor transposition to store the result in the user's desired data layout.

16.6 Summary

This chapter started with a brief introduction to machine learning. It then dove more deeply into the classification task and introduced perceptrons, a type of linear classifier that is foundational for understanding modern CNN. We discussed how the forward inference and backward propagation training passes are implemented for both single-layer and MLP. In particular, we discussed the need for differentiable activation functions and how the model parameters can be updated through chain rules in a multilayer perceptron network during the training process.

Based on the conceptual and mathematical understanding of perceptrons, we presented a basic convolutional neural network and the implementation of its major types of layers. These layers can be viewed as special cases and/or simple adaptations of perceptrons. We then built on the convolution pattern in Chapter 7, Convolution, to present a CUDA kernel implementation of the convolutional layer, the most computationally intensive layer of CNN.

We then presented techniques for formulating convolutional layers as matrix multiplications by unrolling the input feature maps into a matrix. The conversion allows the convolutional layers to benefit from highly optimized GEMM libraries for GPUs. We also presented the C and CUDA implementations of the unrolling procedure for the input matrix and discussed the pros and cons of the unrolling approach.

We ended the chapter with an overview of the CUDNN library, which is used by most deep learning frameworks. Users of these frameworks can benefit from the highly optimized layer implementations without writing CUDA kernels themselves.

Exercises

1. Implement the forward pass for the pooling layer described in Section 16.2.
2. We used an [N × C × H × W] layout for input and output features. Can we reduce the memory bandwidth by changing it to an [N × H × W × C] layout? What are potential benefits of using a [C × H × W × N] layout?
3. Implement the backward pass for the convolutional layer described in Section 16.2.
4. Analyze the read access pattern to X in the unroll_Kernel in Fig. 16.18 and show whether the memory reads that are done by adjacent threads can be coalesced.

References

Chellapilla K., Puri S., Simard P., 2006. High Performance Convolutional Neural Networks for Document Processing, https://hal.archives-ouvertes.fr/inria-00112631/document.

Chetlur S., Woolley C., Vandermersch P., Cohen J., Tran J., 2014. cuDNN: Efficient Primitives for Deep Learning NVIDIA.

Hinton, G.E., Osindero, S., Teh, Y.-W., 2006. A fast learning algorithm for deep belief nets. Neural Comp. 18, 1527−1554. Available from: https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf.

Krizhevsky, A., Cuda-convnet, https://code.google.com/p/cuda-convnet/.

Krizhevsky, A., Sutskever, I., Hinton, G., 2012. ImageNet classification with deep convolutional neural networks. In Proc. Adv. NIPS 25 1090−1098, https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.