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.

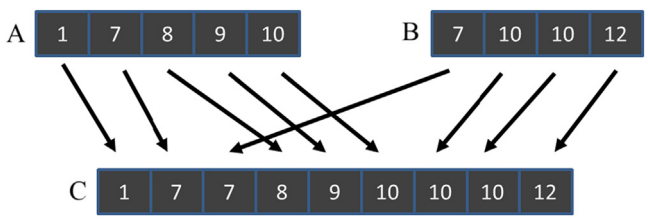


FIGURE 12.1

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

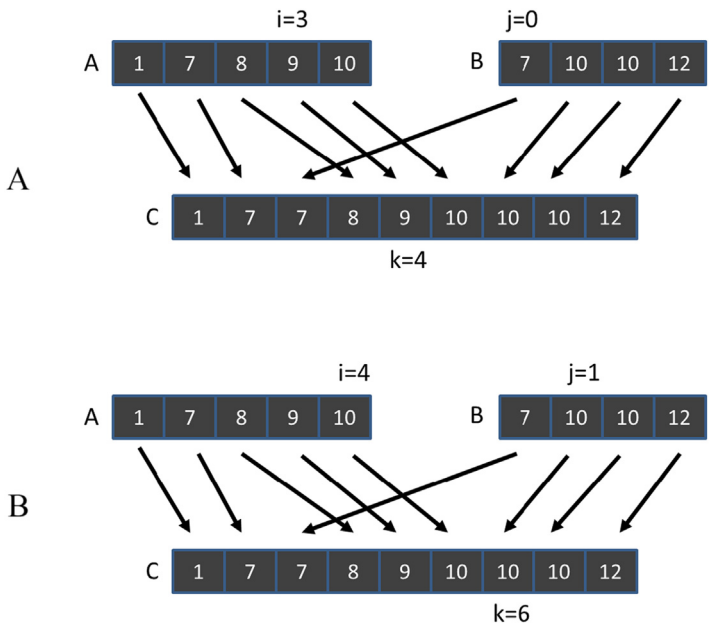


FIGURE 12.3

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,

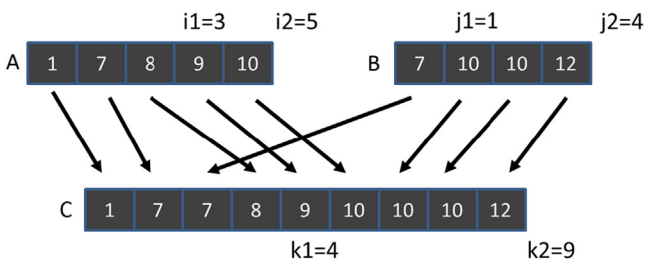


FIGURE 12.4

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.

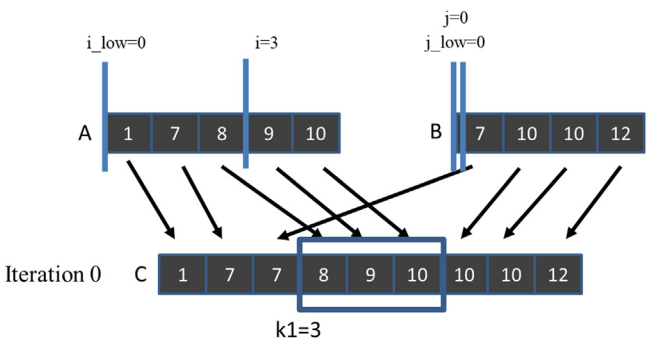


FIGURE 12.6

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

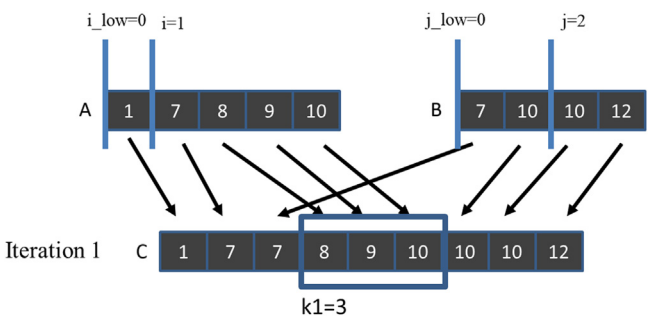


FIGURE 12.7

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

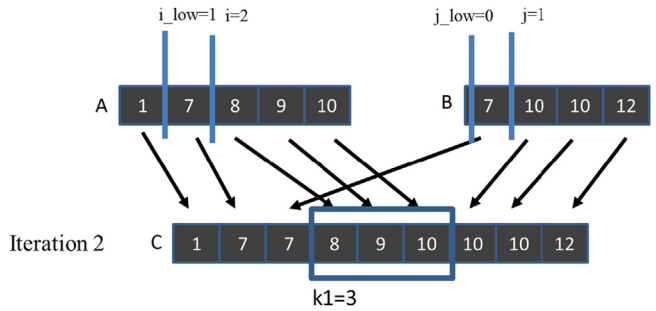


FIGURE 12.8

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

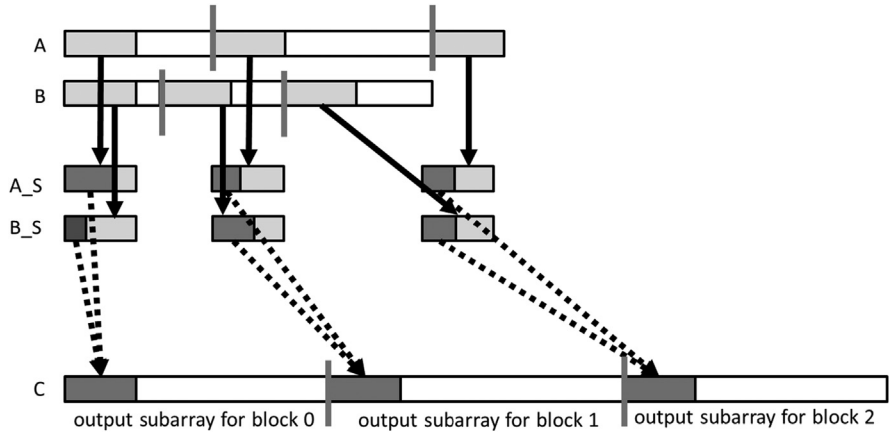


FIGURE 12.10

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

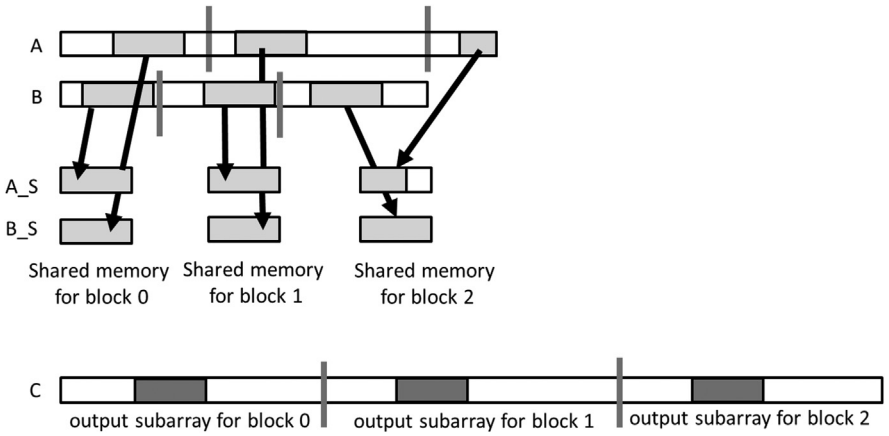


FIGURE 12.14

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

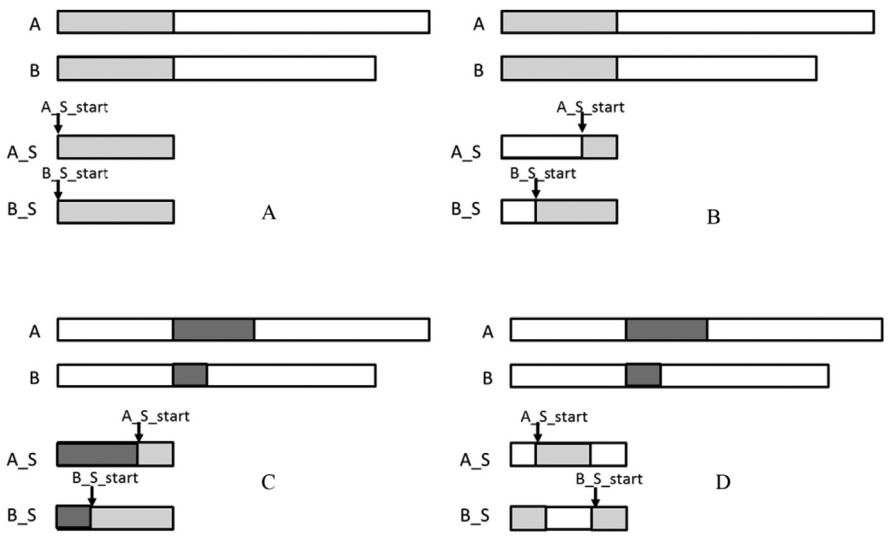


FIGURE 12.15

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.

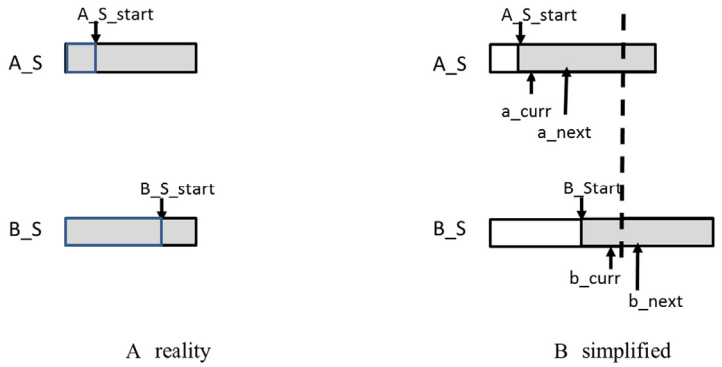


FIGURE 12.17

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.

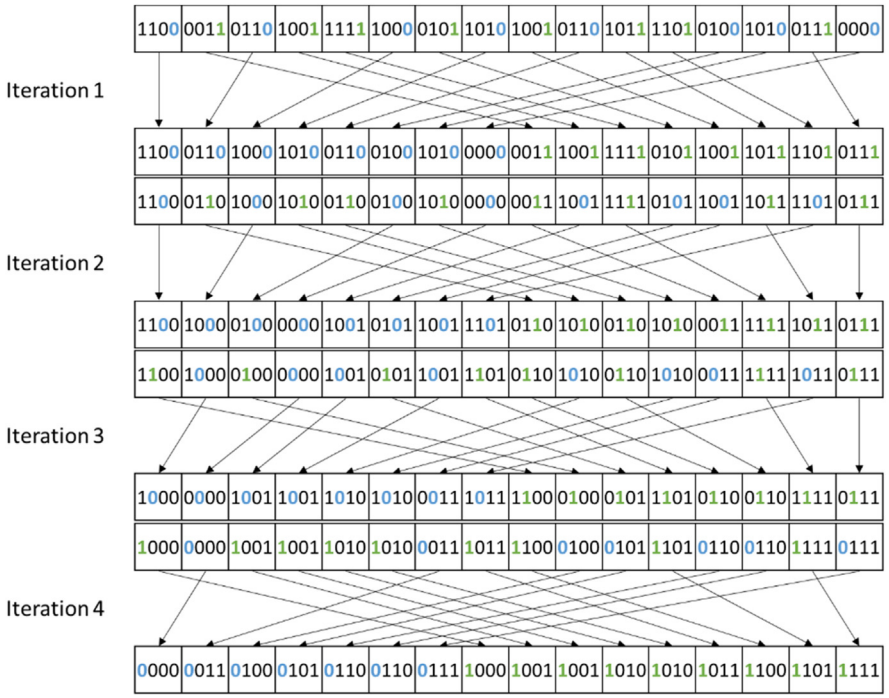


FIGURE 13.1

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

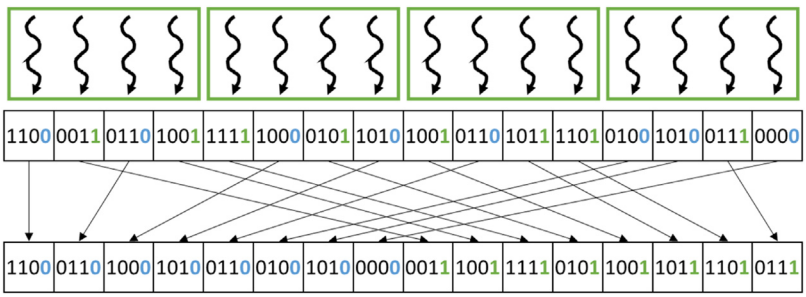


FIGURE 13.2

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.

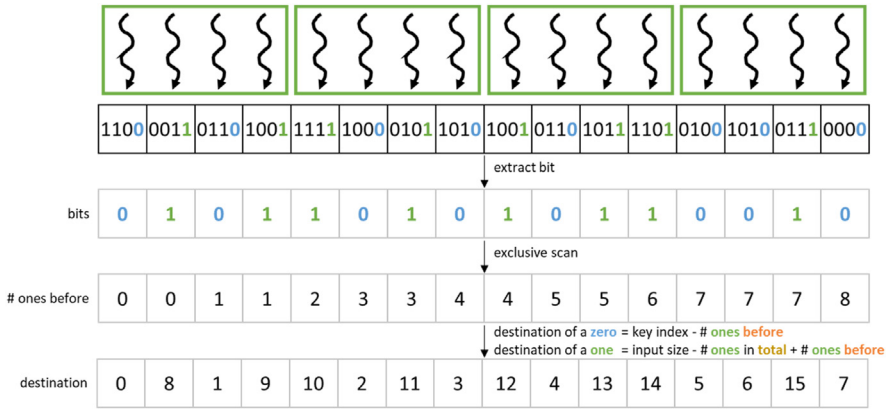


FIGURE 13.3

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

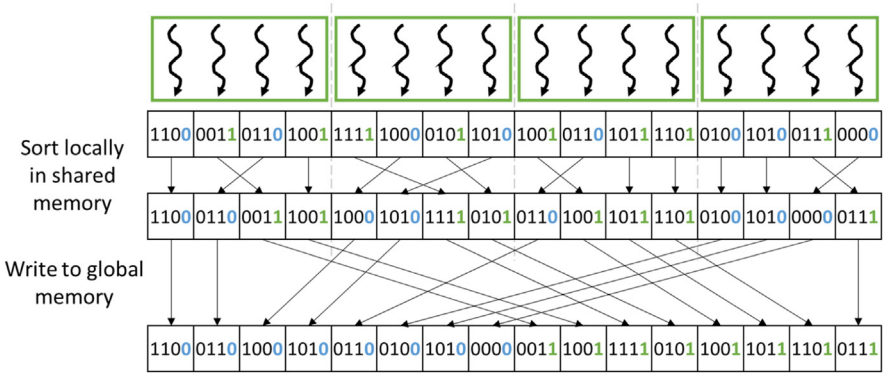


FIGURE 13.5

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.

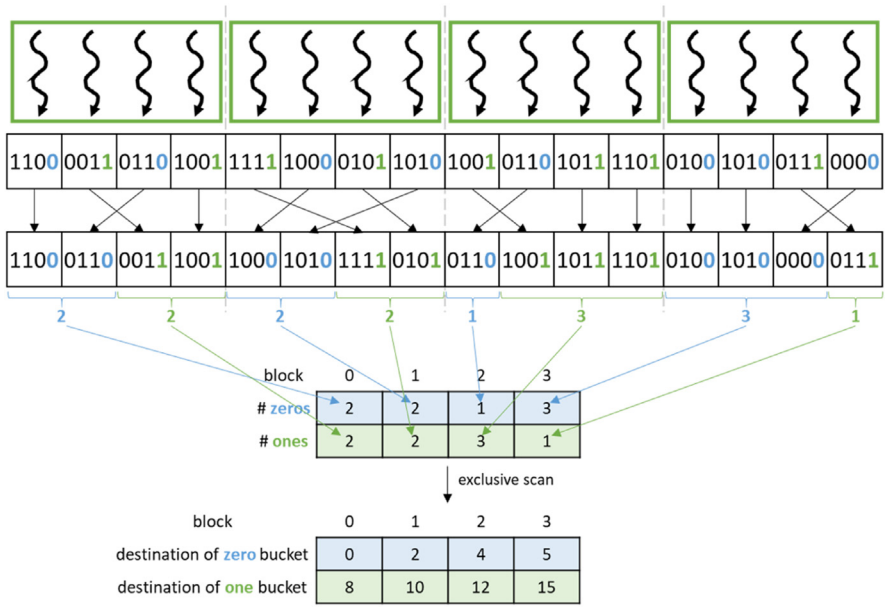


FIGURE 13.6

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.

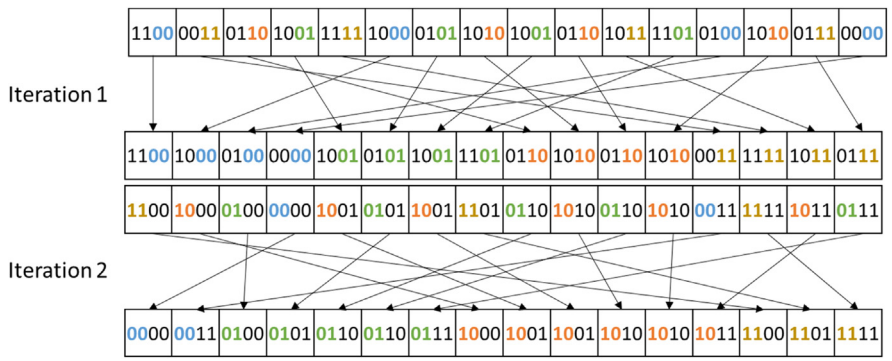


FIGURE 13.7

Radix sort example with 2-bit radix.

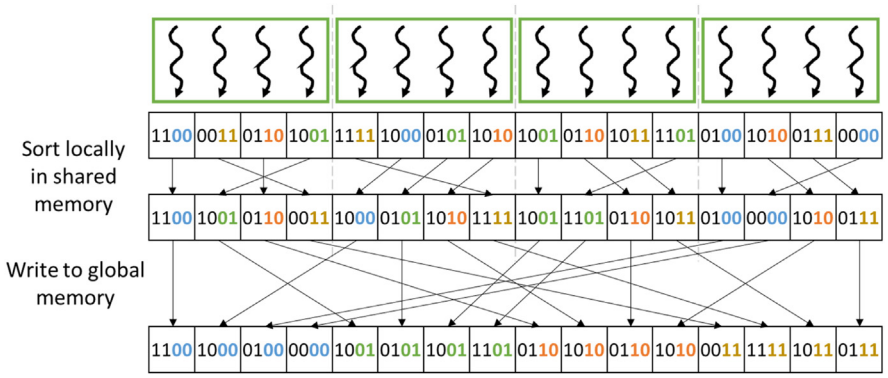


FIGURE 13.8

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.

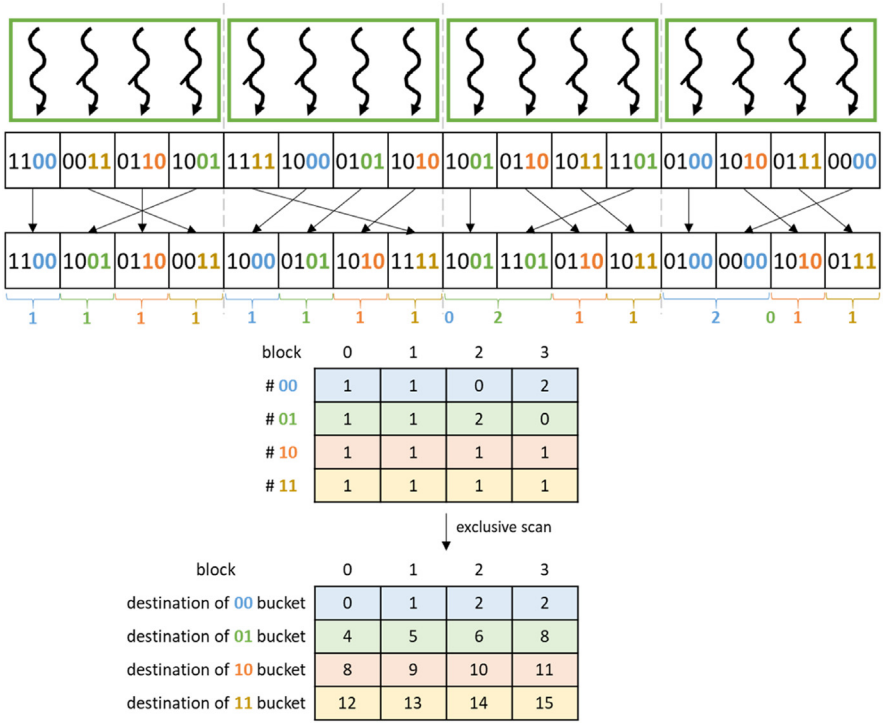


FIGURE 13.9

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

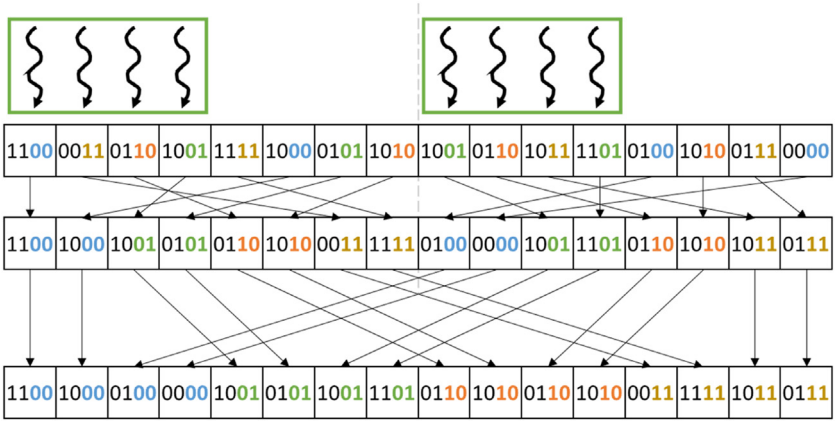


FIGURE 13.10

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.

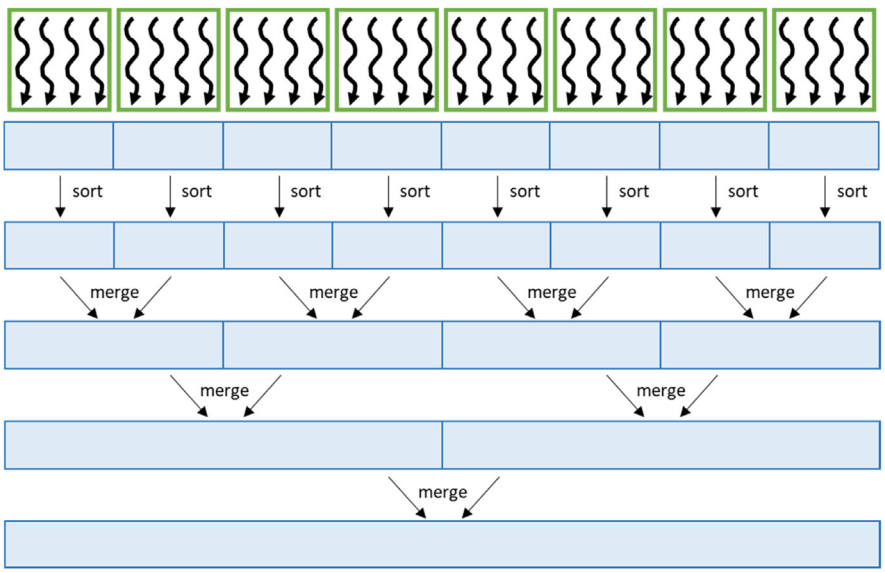


FIGURE 13.11

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

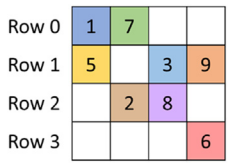


FIGURE 14.1

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)

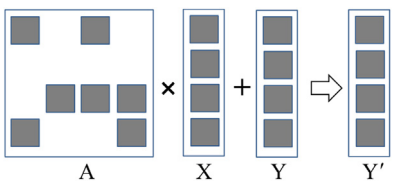


FIGURE 14.2

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

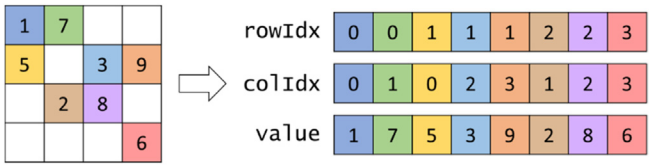


FIGURE 14.3

Example of the coordinate list (COO) format.

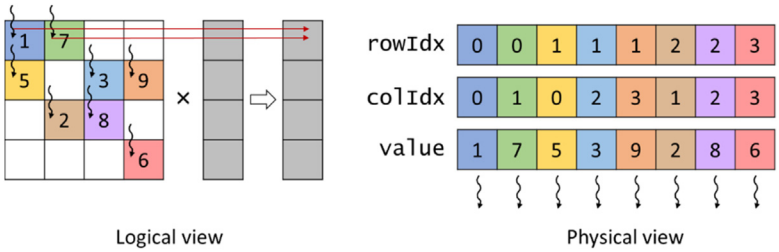


FIGURE 14.4

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.

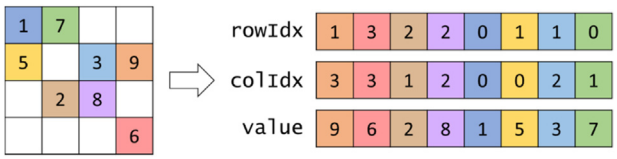


FIGURE 14.6

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

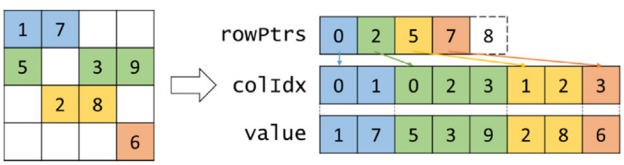


FIGURE 14.7

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.

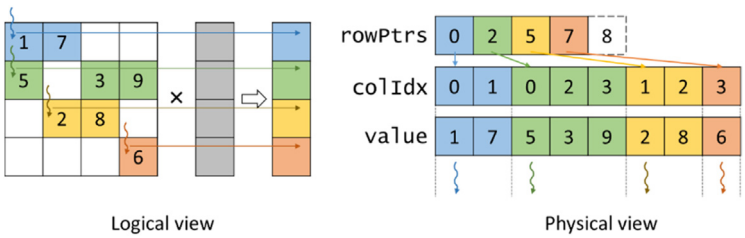


FIGURE 14.8

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

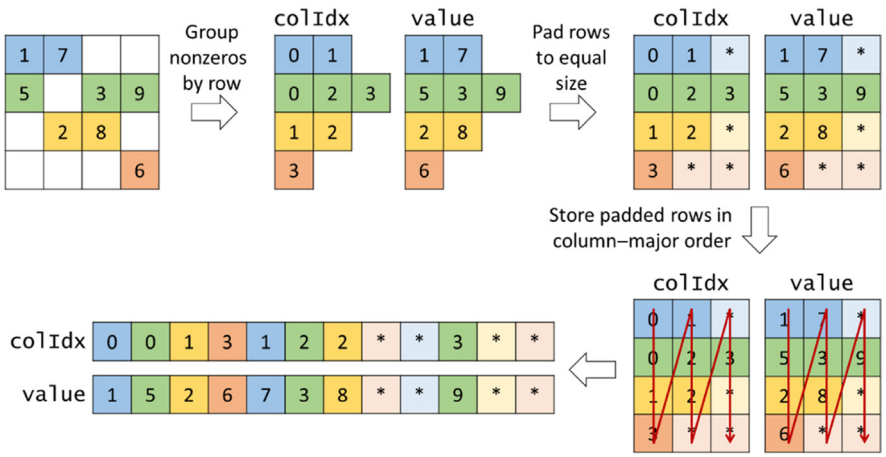


FIGURE 14.10

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

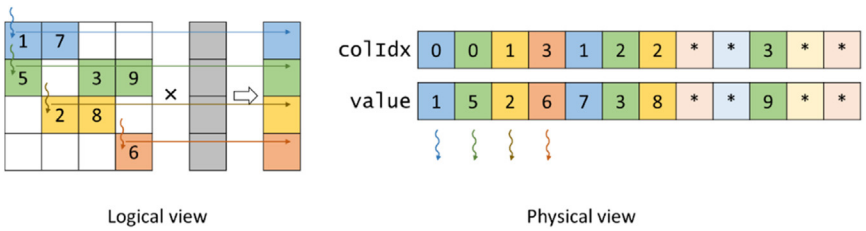


FIGURE 14.11

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

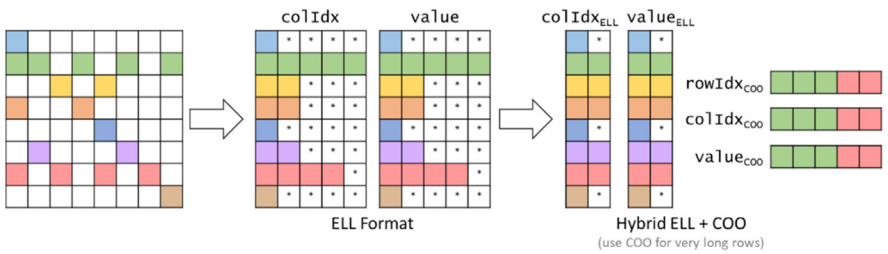


FIGURE 14.13

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

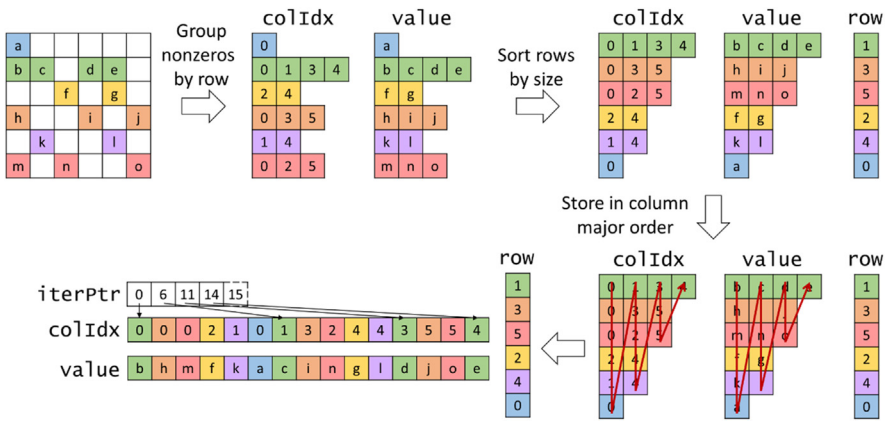


FIGURE 14.14

Example of JDS storage format.

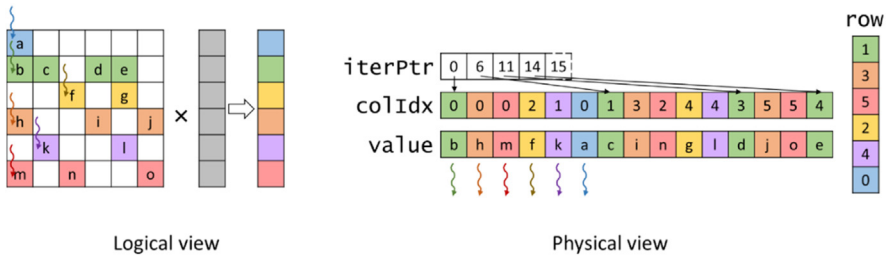


FIGURE 14.15

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.

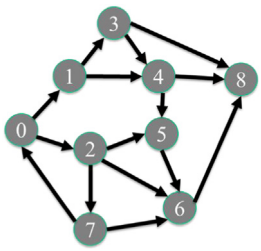


FIGURE 15.1

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

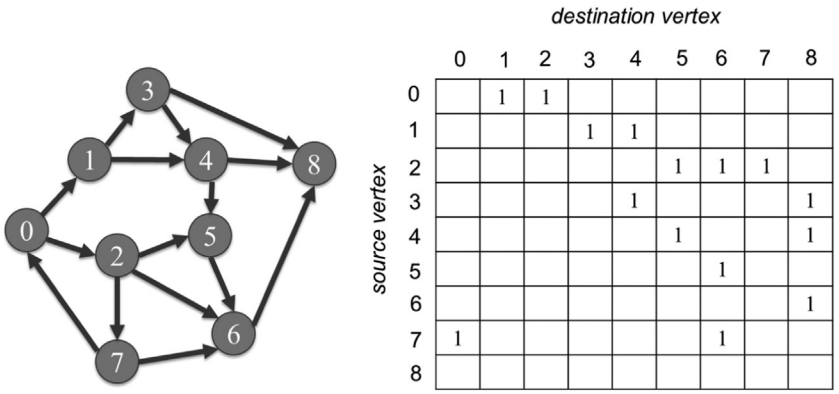


FIGURE 15.2

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.

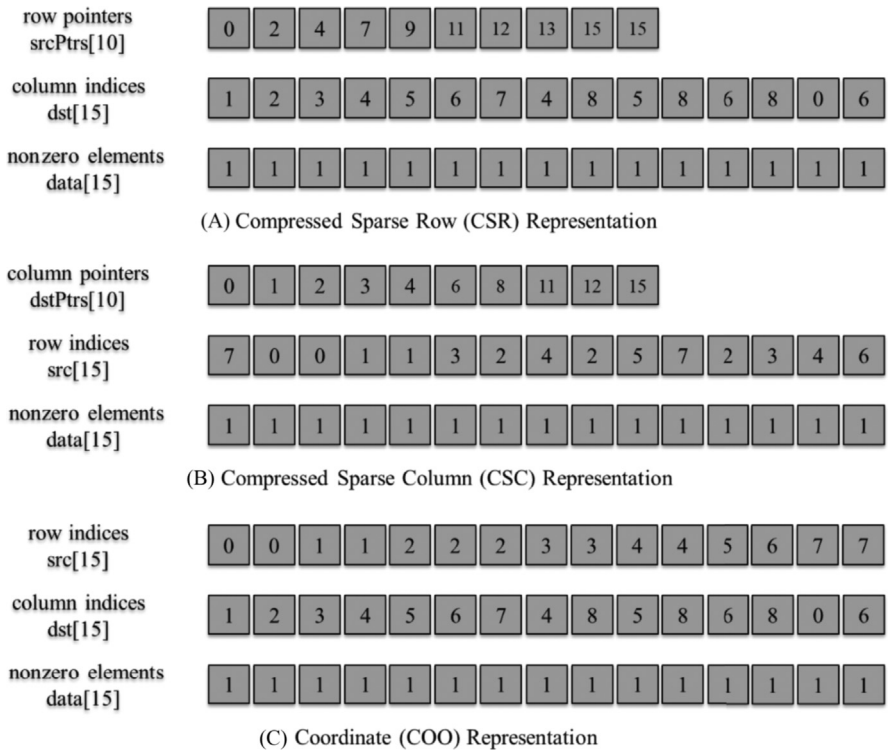


FIGURE 15.3

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).

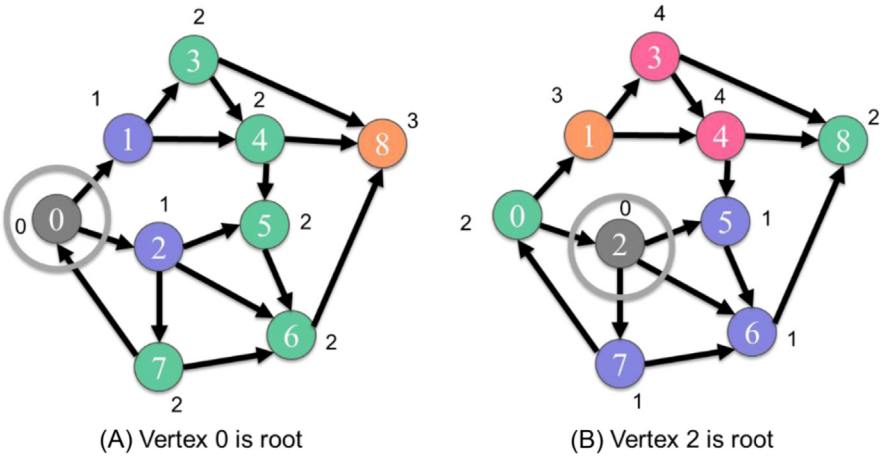


FIGURE 15.4

(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.

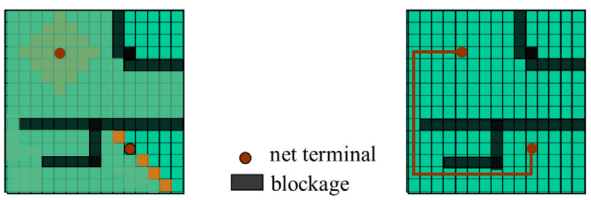
Once we have all the vertices labeled with their level, we can easily find a path from the root vertex to any of the vertices where the number of edges traveled is equivalent to the level. For example, in Fig. 15.4(B), we see that vertex 1 is labeled as level 3, so we know that the smallest number of 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.

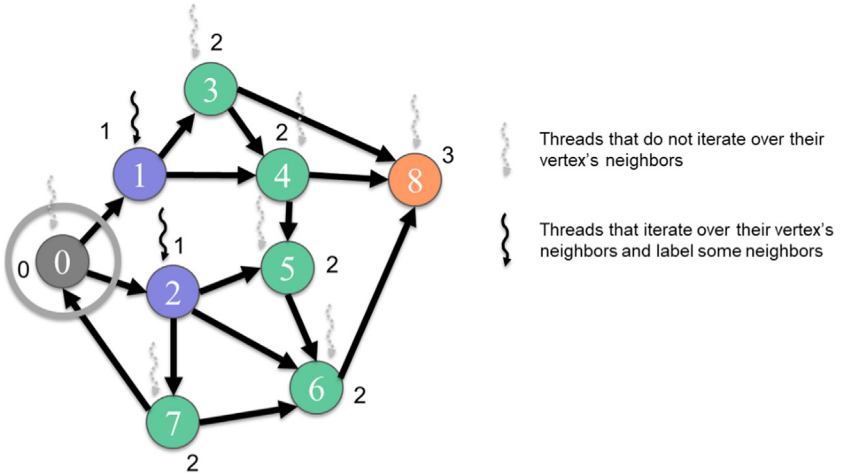


FIGURE 15.7

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.

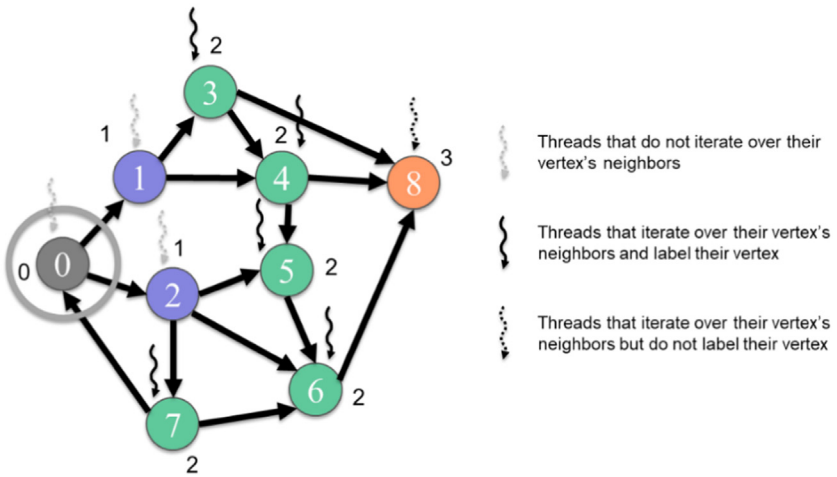


FIGURE 15.9

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.

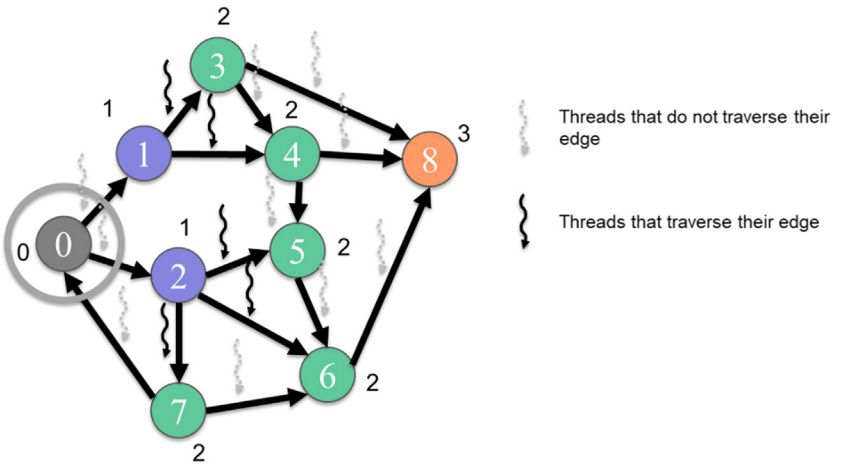


FIGURE 15.11

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.

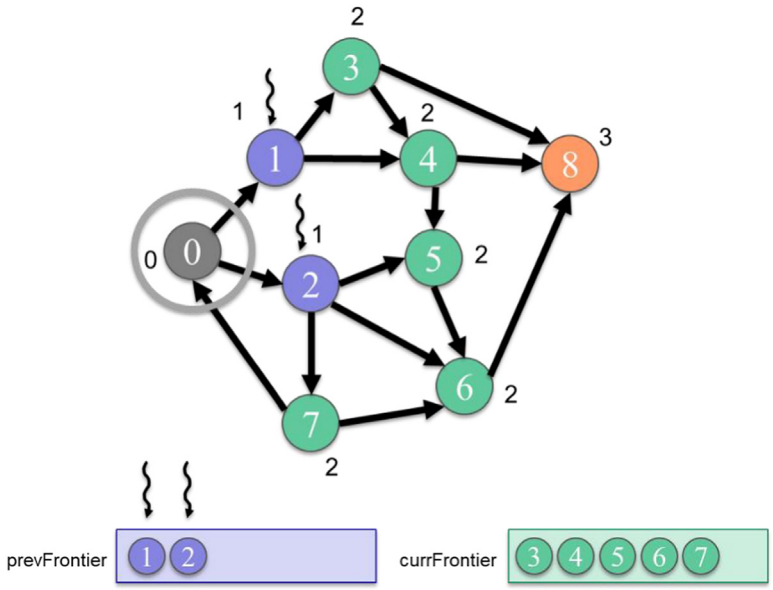


FIGURE 15.13

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.

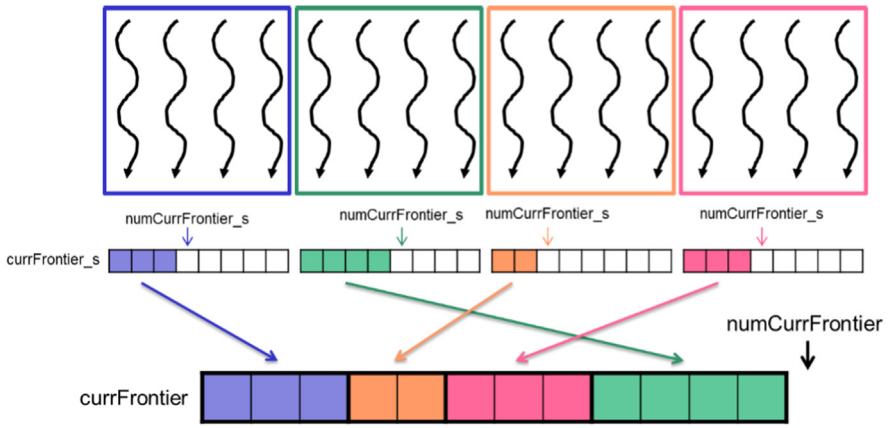


FIGURE 15.15

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

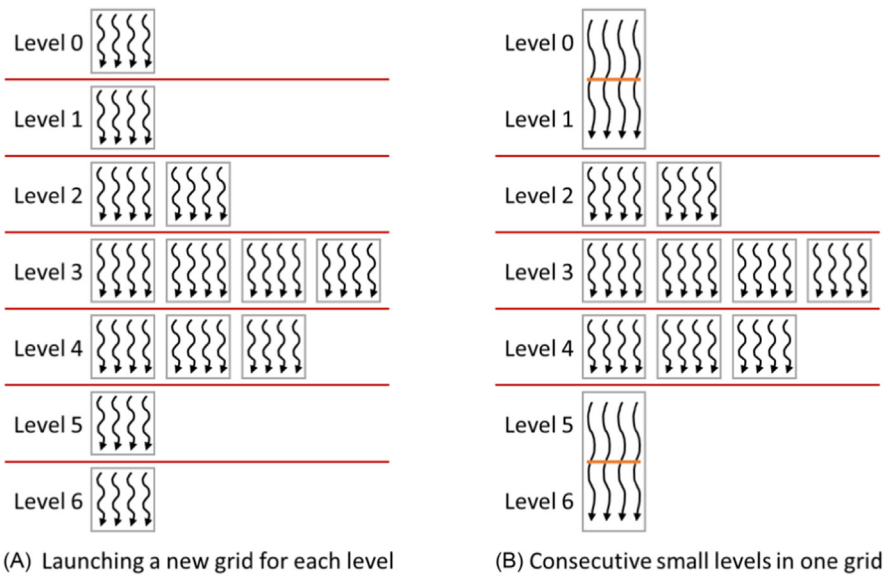


FIGURE 15.16

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

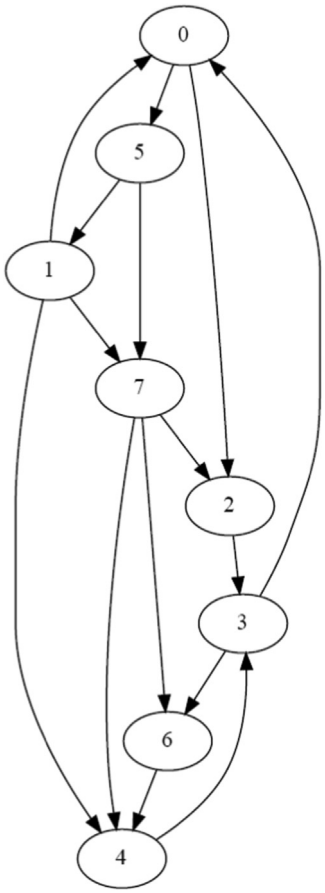
While BFS is among the simplest graph applications, it exhibits the challenges that are characteristic of more complex applications: problem decomposition for extracting parallelism, taking advantage of privatization, implementing fine-grained load balancing, and ensuring proper synchronization. Graph 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

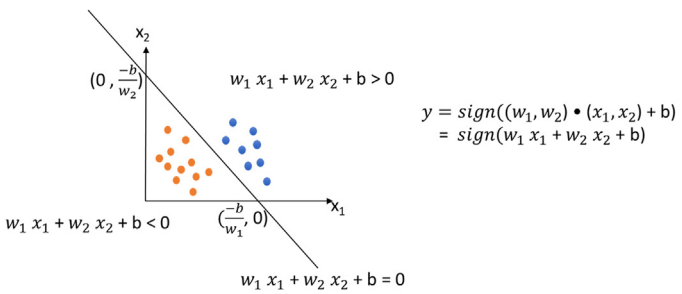


FIGURE 16.1

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

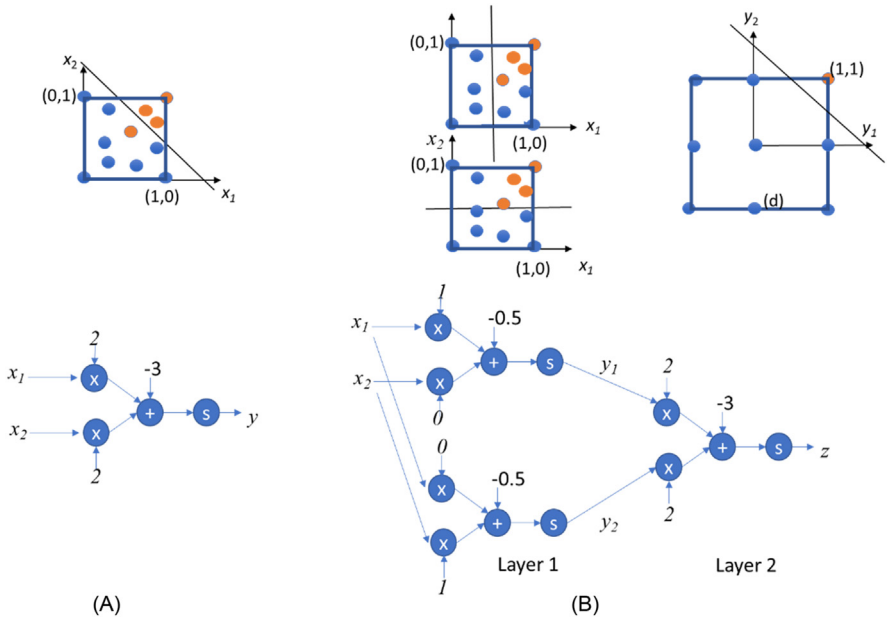


FIGURE 16.2

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

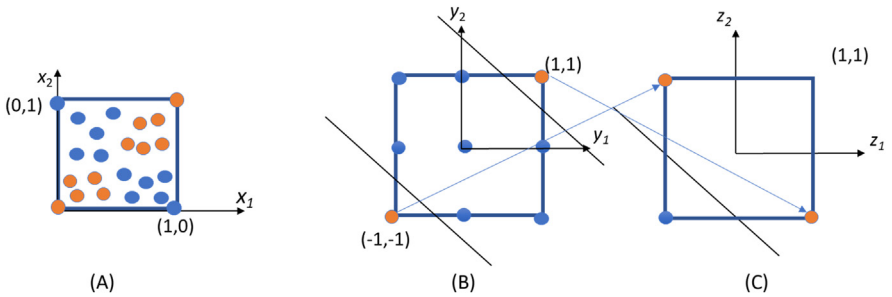


FIGURE 16.3

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

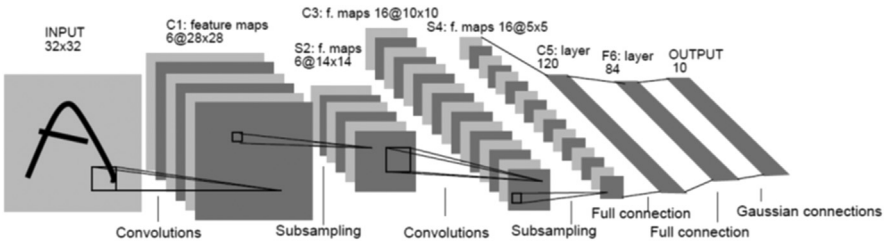


FIGURE 16.4

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

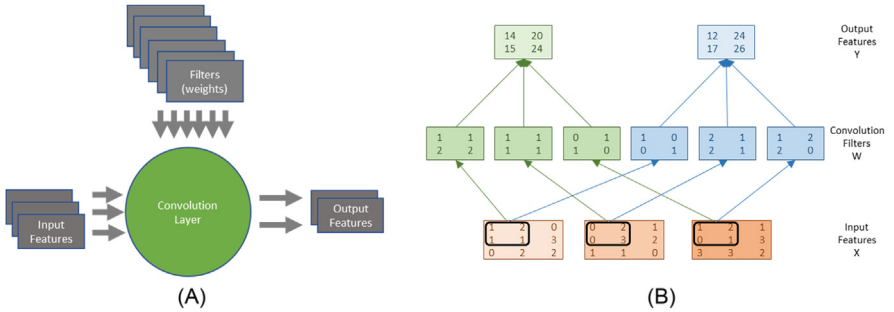


FIGURE 16.5

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

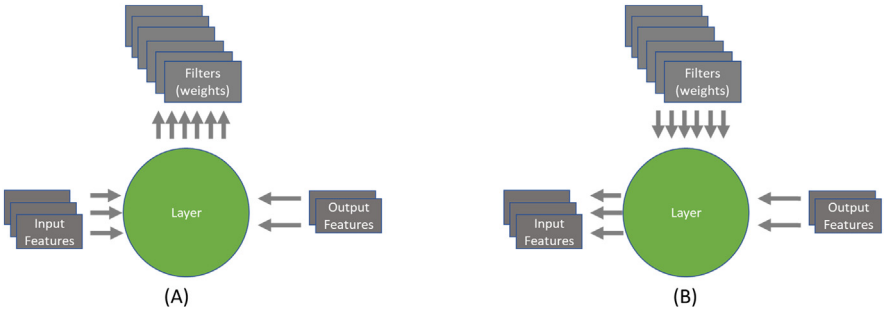


FIGURE 16.8

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 $\frac{\partial E}{\partial x}$.

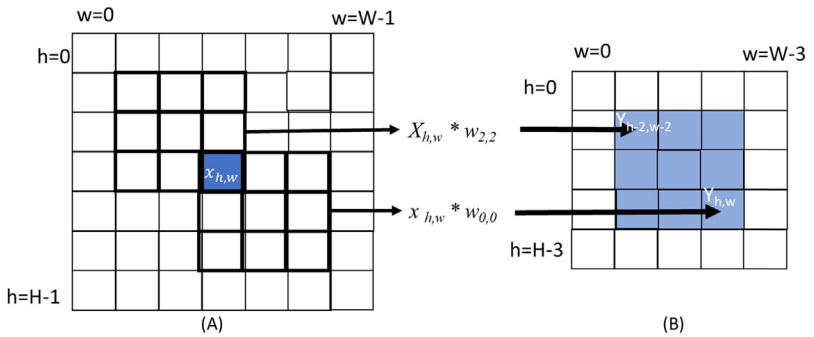


FIGURE 16.9

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.

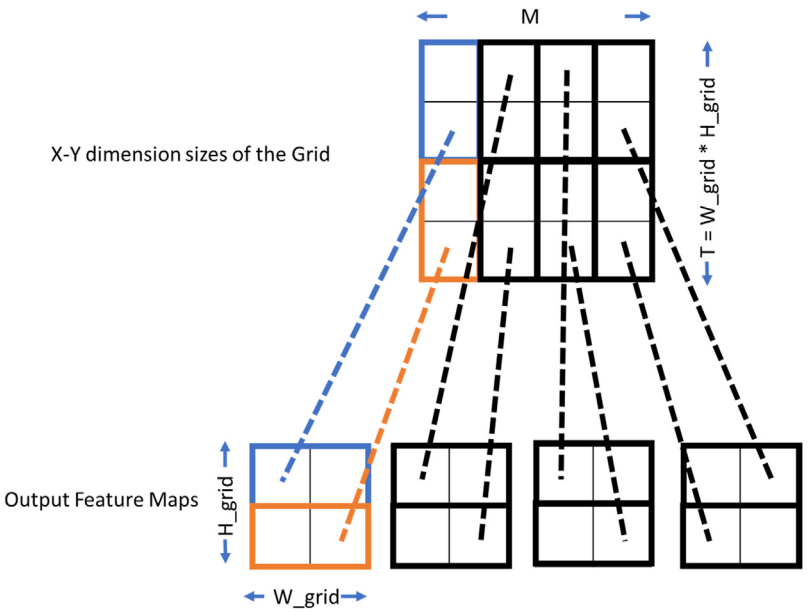


FIGURE 16.14

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.

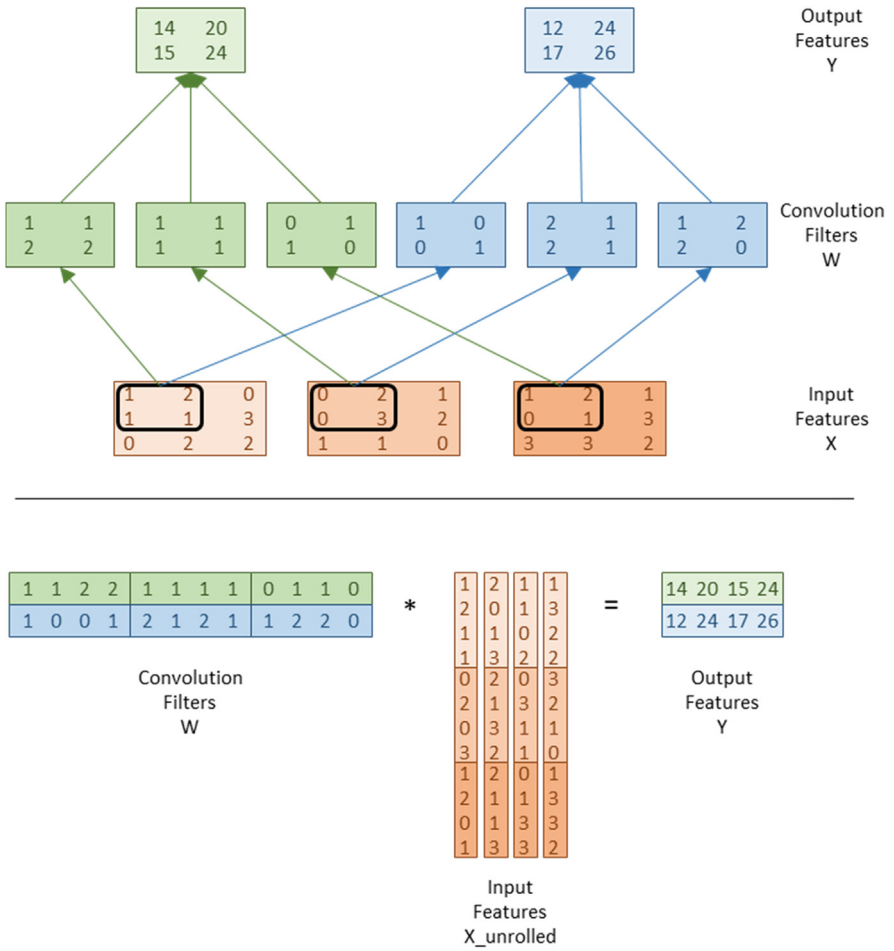


FIGURE 16.16

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.

Lavin, A., Scott G., 2016. Fast algorithms for convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

LeCun, Y., et al., 1990. Handwritten digit recognition with a back-propagation network. In Proc. Adv. Neural Inf. Process. Syst. 396–404, http://yann.lecun.com/exdb/publis/pdf/lecun-90c.pdf.

LeCun, Y., Bengio, Y., Hinton, G.E., 2015. Deep learning. Nature 521, 436–444 (28 May 2015). Available from: http://www.nature.com/nature/journal/v521/n7553/full/nature14539.html.

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998. Gradient-based learning applied to document recognition. Proc. IEEE 86 (11), 2278–2324. Available from: http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

Raina, R., Madhavan, A., Ng, A.Y., 2009. Large-scale deep unsupervised learning using graphics processors. Proc. 26th ICML 873–880. Available from: http://www.andrewng.org/portfolio/large-scale-deep-unsupervised-learning-using-graphics-processors/.

Rosenblatt, F., 1957. The Perceptron—A Perceiving and Recognizing Automaton. Report 85–460-1. Cornell Aeronautical Laboratory.

Rumelhart, D.E., Hinton, G.E., Williams, R.J., 1986. Learning representations by back-propagating errors. Nature 323 (6088), 533–536.

Samuel, A., 1959. Some studies in machine learning using the game of checkers. IBM J. Res. Dev. 3 (3), 210–229. Available from: https://doi.org/10.1147/rd.33.0210. CiteSeerX 10.1.1.368.2254.

Tan, G., Li, L., Treichler, S., Phillips, E., Bao, Y., Sun, N., 2011. Fast implementation of DGEMM on Fermi GPU. Supercomputing 11.

Vasilache N., Johnson J., Mathieu M., Chintala S., Piantino S, LeCun Y., 2014. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation, http://arxiv.org/pdf/1412.7580v3.pdf.

CHAPTER

17 Iterative magnetic resonance imaging reconstruction

Chapter Outline

In this chapter we start with the background and problem formulation of a relatively simple application that has traditionally been constrained by the limited capabilities of mainstream computing systems. We show that parallel execution not only speeds up the existing approaches but also allows the applications experts to pursue an approach that has been known to provide benefit but was previously ignored because of the excessive computational requirements. This approach represents an increasingly important class of computational methods that derive statistically optimal estimation of unknown values from a very large amount of observational data. We use an example algorithm and its implementation source code from such an approach to illustrate how a developer can systematically determine the kernel parallelism structure, assign variables into different types of memories, steer around limitations of the hardware, validate results, and assess the impact of performance improvements.

17.1 Background

Magnetic resonance imaging (MRI) is commonly used as a medical procedure to safely and noninvasively probe the structure and function of biological tissues in all regions of the body. Images that are generated by using MRI have had a profound impact in both clinical and research settings. MRI consists of two phases: acquisition (scan) and reconstruction. During the acquisition phase, the scanner samples data in the k-space domain (i.e., the spatial frequency domain or Fourier transform domain) along a predefined trajectory. These samples are then

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00012-4

transformed into the desired image during the reconstruction phase. Intuitively, the reconstruction phase estimates the shape and texture of the tissues on the basis of the observation k-space data collected from the scanner.

The application of MRI is often limited by high noise levels, significant imaging artifacts, and/or long data acquisition times. In clinical settings, short scan times not only increase scanner throughput but also reduce patient discomfort, and this tends to mitigate motion-related artifacts. High image resolution and fidelity are important because they enable early detection of pathology, leading to improved prognoses for patients. However, the goals of short scan time, high resolution, and high signal-to-noise ratio (SNR) often conflict; improvements in one metric tend to come at the expense of one or both of the others. New technological breakthroughs are needed to enable simultaneous improvement on all of three dimensions. This study presents a case in which massively parallel computing provides such a breakthrough.

The reader is referred to MRI textbooks such as Liang and Lauterbur (1999) for the physics principles behind MRI. For this case study, we will focus on the computational complexity in the reconstruction phase and how the complexity is affected by the k-space sampling trajectory. The k-space sampling trajectory used by the MRI scanner can significantly affect the quality of the reconstructed image, the time complexity of the reconstruction algorithm, and the time required for the scanner to acquire the samples. Eq. (17.1) shows a formulation that relates the k-space samples to the reconstructed image for a class of reconstruction methods:

$$\widehat{m}(\mathrm{r}) = \sum_{j} W(\mathrm{k_j}) s(\mathrm{k_j}) ei2\pi \mathrm{k_j}\ \mathrm{r} \tag{17.1}$$

In Eq. (17.1), $m(\mathbf{r})$ is the reconstructed image, $s(\mathbf{k})$ is the measured k-space data, and $W(\mathbf{k})$ is the weighting function that accounts for nonuniform sampling; that is, $W(\mathbf{k})$ decreases the influence of data from k-space regions where a higher density of samples points are taken. For this class of reconstructions, $W(\mathbf{k})$ can also serve as an *apodization* filtering function that reduces the influence of noise and reduces artifacts due to finite sampling.

If data are acquired at uniformly spaced Cartesian grid points in the k-space under ideal conditions, then the $W(\mathbf{k})$ weighting function is a constant and can thus be factored out of the summation in Eq. (17.1). Furthermore, with the uniformly spaced Cartesian grid samples, the exponential terms in Eq. (17.1) are uniformly spaced in the k-space. As a result, the reconstruction of $m(\mathbf{r})$ becomes an inverse fast Fourier transform (FFT) on $s(\mathbf{k})$, an extremely efficient computation method. A collection of data measured at such uniformed spaced Cartesian grid points is referred to as a *Cartesian scan trajectory*. Fig. 17.1A depicts a Cartesian scan trajectory. In practice, Cartesian scan trajectories allow straightforward implementation on scanners and are widely used in clinical settings today.

Although the inverse FFT reconstruction of Cartesian scan data is computationally efficient, non-Cartesian scan trajectories often have an advantage in reduced sensitivity to patient motion, better ability to provide self-calibrating field

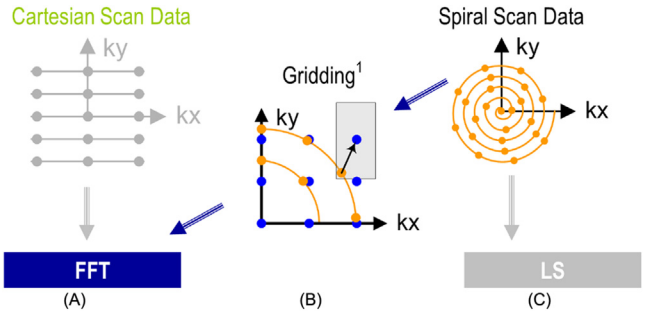
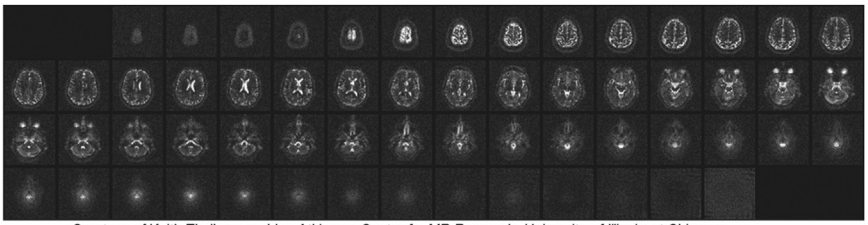


FIGURE 17.1

Scanner k-space trajectories and their associated reconstruction strategies: (A) Cartesian trajectory with FFT reconstruction, (B) spiral (or non-Cartesian trajectory in general) followed by gridding to enable FFT reconstruction, (C) spiral (non-Cartesian) trajectory with a linear solver–based reconstruction.



Courtesy of Keith Thulborn and Ian Atkinson, Center for MR Research, University of Illinois at Chicago

FIGURE 17.2

Non-Cartesian k-space sample trajectory and accurate linear solver–based reconstruction enable new capabilities with exciting medical applications.

inhomogeneity information, and reduced requirements for scanner hardware performance. As a result, non-Cartesian scan trajectories such as spirals (shown in Fig. 17.1(C)), radial lines (also known as projection imaging), and rosettes have been proposed to reduce motion-related artifacts and address scanner hardware performance limitations. These improvements have recently allowed the reconstructed image pixel values to be used for measuring subtle phenomenon such as tissue chemical anomalies before they become anatomical pathology.

Fig. 17.2 shows such an MRI reconstruction-based measurement that generates a map of sodium, a heavily regulated substance in normal human tissues. The information can be used to track tissue health in stroke and cancer treatment processes. Because sodium is much less abundant than water molecules in human tissues, measuring sodium levels reliably requires a higher SNR through a higher number of samples and therefore needs to mitigate the extra scan time with non-Cartesian scan trajectories. The improved SNR enables reliable collection of

in vivo concentration data on chemical substances, such as sodium, in human tissues. The variation or shifting of the sodium concentration suggests early signs of disease development or tissue death. For example, the sodium map of a human brain shown in Fig. 17.2 can be used to give early indication of brain tumor tissue responsiveness to chemotherapy protocols, enabling individualized medicine.

Image reconstruction from non-Cartesian trajectory data presents both challenges and opportunities. The main challenge arises from the fact that the exponential terms are no longer uniformly spaced; the summation does not have the form of an FFT anymore. Therefore one can no longer perform reconstruction by directly applying an inverse FFT to the k-space samples. In a commonly used approach called gridding, the samples are first interpolated onto a uniform Cartesian grid and then reconstructed by using the FFT (see Fig. 17.1B). For example, a convolution approach to gridding takes a k-space data point, convolves it with a gridding convolution mask, and accumulates the results on a Cartesian grid. As we saw in Chapter 7, Convolution, is quite computationally intensive and is an important pattern for massively parallel computing. The reader already has the skills for accelerating convolution gridding computation with parallel computing and thus facilitating the application of the current FFT approach to non-Cartesian trajectory data.

In this chapter we will cover an iterative, statistically optimal image reconstruction method that can accurately model imaging physics and bound the noise error in the resulting image pixel values. Such statistically optimal methods are gaining importance in the wake of big data analytics. However, such iterative reconstruction methods have been impractical for large-scale three-dimensional (3D) problems, owing their excessive computational requirements compared to gridding. Recently, these reconstructions have become viable in clinical settings because of the wide availability of GPUs. An iterative reconstruction algorithm that used to take hours using high-end sequential CPUs to reconstruct an image of moderate resolution now takes only minutes using both CPUs and GPUs, a delay that is acceptable in clinical settings.

17.2 Iterative reconstruction

Haldar and Liang proposed a linear solver–based iterative reconstruction algorithm (Stone et al., 2008) for non-Cartesian scan data, as shown in Fig. 17.1C. The algorithm allows for explicitly modeling the physics of the scanner data acquisition process and can thus reduce the artifacts in the reconstructed image. However, it is computationally expensive. We use this as an example of innovative methods that have required too much computation time to be considered practical. We will show that massively parallel execution can reduce the reconstruction time to the order of a minute so that the new imaging capabilities, such as sodium imaging, can be deployed in clinical settings.

Fig. 17.3 shows a solution of the quasi-Bayesian estimation problem formulation of the iterative linear solver–based reconstruction approach, where ρ is a vector containing voxel values for the reconstructed image, F is a matrix that models the physics of imaging process, D is a vector of data samples from the scanner, and W is a matrix that can incorporate prior information such as anatomical constraints. F^H and W^H are the Hermitian transpose (or conjugate transpose) of F and W, respectively, by taking the transpose and then taking the complex conjugate of each entry (the complex conjugate of $a + ib$ being $a - ib$). In clinical settings, the anatomical constraints represented in W are derived from one or more high-resolution, high-SNR water molecule scans of the patient. These water molecule scans reveal features such as the location of anatomical structures. The matrix W is derived from these reference images. The problem is to solve for ρ given all the other matrices and vectors.

On the surface, the computational solution to the problem formulation in Fig. 17.3 should be very straightforward. It involves matrix multiplication and addition ($F^HF+\lambda W^HW$), matrix-vector multiplication (F^HD), matrix inversion ($F^HF+\lambda W^HW)^{-1}$, and finally matrix multiplication ($(F^HF+\lambda W^HW)^{-1}*F^HD$). However, the sizes of the matrices make this straightforward approach extremely time-consuming. The dimensions of F^H and F matrices are determined by the number of voxels in the 3D reconstructed image and the number of k-space samples used in the reconstruction. Even in a modest resolution 128^3-voxel reconstruction, there are 128^3=2 million columns in F with N elements in each column, where N is the number of k-space samples that are used (size of D). Obviously, F is extremely large. Such massive dimensions are commonly encountered in big data analytics when one tries to use iterative solver methods to estimate the major contributing factors of a massive amount of noisy observational data.

The sizes of the matrices that are involved are so large that the matrix operations that are involved in a direct solution of the equation in Fig. 17.3 using

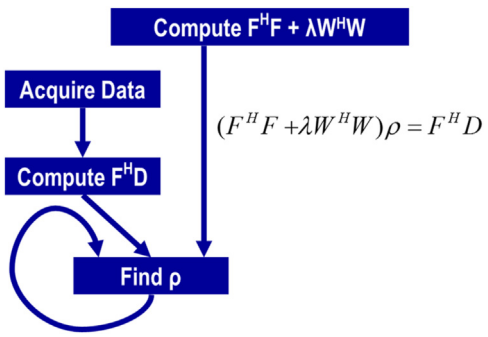


FIGURE 17.3

An iterative linear solver–based approach to reconstructing non-Cartesian k-space sample data.

methods such as Gaussian elimination are practically intractable. An iterative method for matrix inversion, such as the conjugate gradient (CG) algorithm, is therefore preferred. The CG algorithm reconstructs the image by iteratively solving the equation in Fig. 17.3 for ρ. During each iteration the CG algorithm updates the current image estimate ρ to improve the value of the quasi-Bayesian cost function. The computational efficiency of the CG technique is determined largely by the efficiency of matrix-vector multiplication operations involving $F^HF+\lambda W^HW$ and ρ, as these operations are required during each iteration of the CG algorithm.

Fortunately, the matrix W often has a sparse structure that permits efficient implementation of W^HW, and the matrix F^HF is Toeplitz, which enables efficient matrix-vector multiplication via the FFT. Stone et al. (2008) present a GPU-accelerated method for calculating Q, a data structure that allows us to quickly calculate matrix-vector multiplication involving F^HF without actually calculating F^HF itself. The calculation of Q can take days on a high-end CPU core. Since F models the physics of the image process, it needs to be done only once for a given scanner and planned trajectory. Thus Q needs to be calculated only once and is used for multiple scans using the same scan trajectory.

The matrix-vector multiply to calculate F^HD takes about one order of magnitude less time than Q but can still take about 3 hours for a 128^3-voxel reconstruction on a high-end sequential CPU. Recall that D is the vector of data samples from the scanner. Thus since F^HD needs to be computed for every image acquisition, it is desirable to reduce the computation time of F^HD to minutes.[1] We will show the details of this process. As it turns out, the core computational structure of Q is identical to that of F^HD; Q just involves much more computation because it deals with matrix multiplication rather than just matrix-vector multiplication. Thus it suffices to discuss one of them from the parallelization perspective. We will focus on F^HD, since this is the one that will need to be run for each data acquisition.

The "find ρ" step in Fig. 17.3 performs the actual CG based on F^HD. As we explain earlier, precalculation of Q makes this step much less computationally intensive than F^HD, accounting for less than 1% of the execution of the reconstruction of each image on a sequential CPU. As a result, we will leave the CG solver out of the parallelization scope and focus on F^HD in this chapter. However, we will revisit its status at the end of the chapter.

17.3 Computing F^HD

Fig. 17.4 shows a sequential C implementation of the computations for the core step of computing a data structure for computing F^HD. The computations

[1] Note that the F^HD computation can be approximated with gridding and can run in a few seconds with perhaps reduced quality of the final reconstructed image.

```
01  for (int m = 0; m < M; m++) {
02    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
03    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
04    for (int n = 0; n < N; n++) {
05      float expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
06      float cArg = cos(expFhD);
07      float sArg = sin(expFhD);
08      rFhD[n] +=  rMu[m]*cArg - iMu[m]*sArg;
09      iFhD[n] +=  iMu[m]*cArg + rMu[m]*sArg;
10    }
11  }
```

FIGURE 17.4

Computation of F^HD.

start with an outer loop that iterates through the k-space samples (line 01). A quick glance at Fig. 17.4 shows that the C implementation of F^HD is an excellent candidate for acceleration because it exhibits substantial data parallelism. The algorithm first computes the real and imaginary components of Mu (rMu and iMu) at the current sample point in the k-space. It then enters an inner n-loop that computes the contribution of the current k-space sample to the real and imaginary components of F^HD at each voxel in the image space. Keep in mind that M is the total number of k-space samples and N is the total number of voxels in the reconstructed image. The value of F^HD at any voxel depends on the values of all k-space sample points. However, no voxel elements of F^HD depend on any other voxel elements of F^HD. Therefore all elements of F^HD can be computed in parallel. Specifically, all iterations of the outer loop can be done in parallel, and all iterations of the inner loop can be done in parallel. However, the calculations of the inner loop have a dependence on the calculation done by the preceding statements in the same iteration of the outer loop.

Despite the algorithm's abundant inherent parallelism, potential performance bottlenecks are evident. First, in the loop that computes the elements of F^HD, the ratio of floating-point operations to memory accesses is at best 0.75 OP/B and at worst 0.25 OP/B. The best case assumes that the sin and cos trigonometry operations are computed by using five-element Taylor series that require 13 and 12 floating-point operations, respectively. The worst case assumes that each trigonometric operation is computed as a single operation in hardware. As we saw in Chapter 5, Memory Architecture and Data Locality, a substantially higher floating-point arithmetic to global memory access ratio is needed for the kernel not to be limited by memory bandwidth. Thus the memory accesses will clearly limit the performance of the kernel unless the ratio is drastically increased.

Second, the ratio of floating-point arithmetic to floating-point trigonometry functions is only 13:2. Thus a GPU-based implementation must tolerate or avoid stalls due to the long latency and low throughput of the sin and cos operations. Without a good way to reduce the cost of trigonometry functions, the performance will likely be dominated by the time that is spent in these functions.

We are now ready to take the steps in converting F^HD from sequential C code to a CUDA kernel.

Step 1: Determine the kernel parallelism structure

The conversion of the loops in Fig. 17.4 into a CUDA kernel is conceptually straightforward. Since all iterations of the outer loop of Fig. 17.4 can be executed in parallel, we can simply convert the outer loop into a CUDA kernel by mapping its iterations to CUDA threads. Fig. 17.5 shows a kernel from such a straightforward conversion. Each thread implements an iteration of the original outer loop; that is, we use each thread to calculate the contribution of one k-space sample to all F^HD elements. The original outer loop has M iterations, and M can be in the millions. We obviously need to have a large number of thread blocks to generate enough threads to implement all these iterations.

To make performance tuning easy, we declare a constant `FHD_THREADS_PER_BLOCK` that defines the number of threads in each thread block when we invoke the cmpFhD kernel. Thus we will use `M/FHD_THREADS_PER_BLOCK` for the grid size and `FHD_THREADS_PER_BLOCK` for the block size when invoking the kernel. Within the kernel, each thread calculates the original iteration of the outer loop that it is assigned to cover, using the familiar formula `blockIdx.x*FHD_THREADS_PER_BLOCK + threadIdx.x`. For example, assume that there are 1,000,000 k-space samples and we decide to use 1024 threads per block. The grid size at kernel innovation will be 1,000,000/1024=977 blocks. The block size will be 1,024. The calculation of m for each thread will be equivalent to `blockIdx.x*1024+threadIdx`.

While the kernel of Fig. 17.5 exploits ample parallelism, it suffers from a major problem: All threads write into all rFhD and iFhD voxel elements. This means that the kernel must use atomic operations in the global memory in the inner loop in order to keep threads from trashing each other's contributions to the voxel value (lines 10–11). As we saw in Chapter 9, Parallel Histogram, heavy

```
01  #define FHD_THREADS_PER_BLOCK 1024
02  __global__ void cmpFhD(float* rPhi, iPhi, rD, iD,
03       kx, ky, kz, x, y, z, rMu, iMu, rFhD, iFhD, int N) {
04    int m = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;
05    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
06    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
07    for (int n = 0; n < N; n++) {
08       float expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
09       float cArg = cos(expFhD);  float sArg = sin(expFhD);
10       atomicAdd(&rFhD[n],rMu[m]*cArg - iMu[m]*sArg);
11       atomicAdd(&iFhD[n],iMu[m]*cArg + rMu[m]*sArg);
12    }
13  }
```

FIGURE 17.5

First version of the F^HD kernel.

use of atomic operations on global memory data can seriously reduce the performance of parallel execution. Furthermore, the size of the rFhD and iFhD arrays, which is the total number of voxels in the reconstructed image, makes privatization in shared memory infeasible. We need to explore other options.

The arrangement for each thread to take an input element (k-space sample in this application) and update all or many output elements (voxels of the reconstructed image in this application) is referred to as the *scatter approach*. Intuitively, each thread scatters the effect of an input to many output values. Unfortunately, threads in the scatter approach can update the same output elements and potentially trash each other's contributions. Thus atomic operations are needed for a scatter approach in general and tend to negatively impact the performance of parallel execution.

A potentially better alternative to the scatter approach is to use each thread to calculate one output element by collecting the contributions from all input elements, which is referred to as the *gather approach*. The gather approach ensures that the threads update only their designated output elements and never interfere with each other. In our application, parallelization based on the gather approach assigns each thread to calculate one pair of rFhD and iFhD elements from all k-space samples. As a result, there is no interference between threads and no need for atomic operations.

To adopt the gather approach, we need to make the n-loop the outer loop so that we can assign each iteration of the n-loop to a thread. This can be done by swapping the inner loop and the outer loop in Fig. 17.4 so that each of the new outer loop iterations processes one rFhD/iFhD element pair. That is, each of the new outer loop iterations will execute the new inner loop that accumulates the contribution of all k-space samples to the rFhD/iFhD element pair handled by the outer loop iteration. This transformation of the loop structure is called *loop interchange*. It requires a perfectly nested loop, meaning that there is no statement between the outer for-loop statement and the inner for-loop statement. However, this is not true for the FhD code in Fig. 17.4. We need to find a way to move the calculation of rMu and iMu elements out of the way.

From a quick inspection of Fig. 17.4 we see that the F^HD calculation can be split into two separate loops, as is shown in Fig. 17.6, using a technique called *loop fission* or loop splitting. This transformation takes the body of a loop and splits it into two loops. In the case of F^HD the outer loop consists of two parts: the statements before the inner loop and the inner loop itself. As is shown in Fig. 17.6, we can perform loop fission on the outer loop by placing the statements before the inner loop into a first loop and the inner loop into a second loop.

An important consideration in loop fission is that the transformation changes the relative execution order of the two parts of the original outer loop. In the original outer loop, both parts of the first iteration execute before the second iteration. After fission the first part of all iterations will execute; they are then followed by the second part of all iterations. The reader should be able to verify that this change of execution order does not affect the execution results for F^HD. This is

```
01  for (int m = 0; m < M; m++) {
02     rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
03     iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
04  }
05  for (int m = 0; m < M; m++) {
06     for (int n = 0; n < N; n++) {
07        float expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
08        float cArg = cos(expFhD);
09        float sArg = sin(expFhD);
10        rFhD[n] +=  rMu[m]*cArg - iMu[m]*sArg;
11        iFhD[n] +=  iMu[m]*cArg + rMu[m]*sArg;
12     }
13  }
```

FIGURE 17.6

Loop fission on the F^HD computation.

```
01  #define MU_THREADS_PER_BLOCK 1024
02  __global__ void cmpMu(float* rPhi, iPhi, rD, iD, rMu, iMu)  {
03     int m = blockIdx.x*MU_THREAEDS_PER_BLOCK + threadIdx.x;
04     rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
05     iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
06  }
```

FIGURE 17.7

The cmpMu kernel.

because the execution of the first part of each iteration does not depend on the result of the second part of any preceding iterations of the original outer loop. Loop fission is a transformation that is often done by advanced compilers that are capable of analyzing the (lack of) dependence between statements across loop iterations.

With loop fission the F^HD computation is now done in two steps. The first step is a single-level loop that calculates the rMu and iMu elements for use in the second loop. The second step corresponds to the second loop that calculates the F^HD elements based on the rMu and iMu elements calculated in the first step. Each step can now be converted into its own CUDA kernel. The two CUDA kernels will execute sequentially with respect to each other. Since the second loop needs to use the results from the first loop, separating these two loops into two kernels that execute in sequence does not sacrifice any parallelism.

The cmpMu() kernel in Fig. 17.7 implements the first loop. The conversion of the first loop from sequential C code to a CUDA kernel is straightforward: Each thread executes one iteration of the original C code. Since the M value can be very big, reflecting the large number of k-space samples, such a mapping can result in a large number of threads. Since each thread block can have up to 1024 threads in each block, we will need to use multiple blocks to allow the large number of threads. This can be accomplished by having a number of threads in each block, specified by `MU_THREADS_PER_BLOCK` in Fig. 17.7, and by employing `M/MU_THREADS_PER_BLOCK` blocks needed to cover all M iterations of

the original loop. For example, if there are 1,000,000 k-space samples, the kernel could be invoked with a configuration of 1024 threads per block and 1,000,000/1024 = 977 blocks. This is done by defining `MU_THREADS_PER_BLOCK` as 1024 and using it as the block size and `M/MU_THREADS_PER_BLOCK` as the grid size during kernel innovation.

Within the kernel, each thread can identify the iteration assigned to it by using its `blockIdx` and `threadIdx` values. Since the threading structure is one-dimensional, only `blockIdx.x` and `threadIdx.x` need to be used. Because each block covers a section of the original iterations, the iteration covered by a thread is `blockIdx.x*MU_THREADS_PER_BLOCK + threadIdx`. For example, assume that `MU_THREADS_PER_BLOCK = 1024`. The thread with `blockIdx.x=0` and `threadIdx.x=37` covers the 37th iteration of the original loop, whereas the thread with `blockIdx.x=5` and `threadIdx.x=2` covers the 5122nd (5*1024+2) iteration of the original loop. Using this iteration number to access the Mu, Phi, and D arrays ensures that the arrays are covered by the threads in the same way they were covered by the iterations of the original loop. Because every thread writes into its own Mu element, there is no potential conflict between any of these threads.

Determining the structure of the second kernel requires a little more work. An inspection of the second loop in Fig. 17.6 shows that there are at least three options in designing the second kernel. In the first option, each thread corresponds to one iteration of the inner loop. This option creates the most number of threads and thus exploits the largest amount of parallelism. However, the number of threads would be N*M, with N in the millions and M in hundreds of thousands. Their product would result in too many threads in the grid, more than are needed to fully utilize the device.

A second option is to use each thread to implement an iteration of the outer loop. This option employs fewer threads than the first option. Instead of generating N*M threads, this option generates M threads. Since M corresponds to the number of k-space samples and a large number of samples, on the order of a hundred thousand, are typically used to calculate F^HD, this option still exploits a large amount of parallelism. However, this kernel suffers the same problem as the kernel in Fig. 17.5. That is, each thread will write into all rFhD and iFhD elements, thus creating an extremely large number of conflicts between threads. As is the case of Fig. 17.5, the code in Fig. 17.8 requires atomic operations that will significantly slow down the parallel execution. Thus this option does not work well.

A third option is to use each thread to compute one pair of rFhD and iFhD output elements. This option requires us to interchange the inner and outer loops and then use each thread to implement an iteration of the new outer loop. The transformation is shown in Fig. 17.9. Loop interchange is necessary because the loop being implemented by the CUDA threads must be the outer loop. Loop interchange makes each of the new outer loop iterations process a pair of rFhD and iFhD output elements and makes each inner loop collect the contributions of all input elements to this pair of output elements.

```
01  #define FHD_THREADS_PER_BLOCK 1024
02  __global__ void cmpFhD(float* rPhi, iPhi, phiMag,
03           kx, ky, kz, x, y, z, rMu, iMu, int N) {

04    int m = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;
05    for (int n = 0; n < N; n++) {
06       float expFhD = 2*PI*(kx[m]*x[n]+ky[m]*y[n]+kz[m]*z[n]);
07       float cArg = cos(expFhD);
08       float sArg = sin(expFhD);
09       atomicAdd(&rFhD[n],rMu[m]*cArg - iMu[m]*sArg);
10       atomicAdd(&iFhD[n],iMu[m]*cArg + rMu[m]*sArg);
11    }
12  }
```

FIGURE 17.8

Second option of the F^HD kernel.

```
01  for (int n = 0; n < N; n++) {
02     for (int m = 0; m < M; m++) {
03        float expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
04        float cArg = cos(expFhD);
05        float sArg = sin(expFhD);
06        rFhD[n] +=  rMu[m]*cArg - iMu[m]*sArg;
07        iFhD[n] +=  iMu[m]*cArg + rMu[m]*sArg;
08     }
09  }
```

FIGURE 17.9

Loop interchange of the F^HD computation.

Loop interchange is permissible here because all iterations of both levels of loops are independent of each other. They can be executed in any order relative to one another. Loop interchange, which changes the order of the iterations, is allowed when these iterations can be executed in any order. This option allows us to convert the new outer loop into a kernel to be executed by N threads. Since N corresponds to the number of voxels in the reconstructed image, the N value can be very large for higher-resolution images. For a 128^3 image, there are 128^3 = 2,097,152 threads, resulting in a large amount of parallelism. For higher resolutions, such as 512^3, we may need to either use multidimensional grids, launch multiple grids, or assign multiple voxels to a single thread. The threads in this third option all accumulate into their own rFhD and iFhD elements, since every thread has a unique n value. There is no conflict between threads. This makes this third option the best choice among the three options.

The kernel that is derived from the interchanged loops is shown in Fig. 17.10. The outer loop has been stripped away; each thread covers an iteration of the outer (n) loop, where n is equal to `blockIdx.x*FHD_THREADS_PER_BLOCK + threadIdx.x`. Once this iteration (n) value has been identified, the thread executes the inner (m) loop based on that n value. This kernel can be invoked with a number of threads in each block, specified by a global constant `FHD_THREADS_PER_BLOCK`. Assuming that N is the variable that stores the number of voxels in the reconstructed image,

```
01  #define FHD_THREADS_PER_BLOCK 1024
02  __global__ void cmpFhD(float* rPhi, iPhi, phiMag,
03           kx, ky, kz, x, y, z, rMu, iMu, int M) {

04    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;
05    for (int m = 0; m < M; m++) {
06       float expFhD = 2*PI*(kx[m]*x[n]+ky[m]*y[n]+kz[m]*z[n]);
07       float cArg = cos(expFhD);
08       float sArg = sin(expFhD);
09       rFhD[n] +=  rMu[m]*cArg - iMu[m]*sArg;
10       iFhD[n] +=  iMu[m]*cArg + rMu[m]*sArg;
11    }
12 }
```

FIGURE 17.10

Third option of the FHD kernel.

`N/FHD_THREADS_PER_BLOCK` blocks cover all N iterations of the original loop. For example, if there are 2,097,152 voxels, the kernel could be invoked with a configuration of 1024 threads per block and 2,097,152/1024 = 2048 blocks. In Fig. 17.10 this is done by assigning 1024 to `FHD_THREADS_PER_BLOCK` and using it as the block size and `N/FHD_THREADS_PER_BLOCK` as the grid size during kernel innovation.

Step 2: Getting around the memory bandwidth limitation

The simple cmpFhD kernel in Fig. 17.10 will perform significantly better than the kernels in Figs. 17.5 and 17.8 but will still result in limited speedup, owing to memory bandwidth limitations. A quick analysis shows that the execution is limited by the low compute to global memory access ratio of each thread. In the original loop, each iteration performs at least 14 memory accesses: `kx[m]`, `ky[m]`, `kz[m]`, `x[n]`, `y[n]`, `z[n]`, `rMu[m]` twice, `iMu[m]` twice, `rFhD[n]` read and write, and iFhD[n] read and write. Meanwhile, about 13 floating-point multiplication, addition, or trigonometry operations are performed in each iteration. Therefore the compute to global memory access ratio is 13/(14*4)=0.23 OP/B, which is too low according to our analysis in Chapter 5, Memory Architecture and Data Locality. We can immediately improve the compute to global memory access ratio by assigning some of the array elements to automatic variables. As we discussed in Chapter 5, Memory Architecture and Data Locality, the automatic variables will reside in registers, thus converting reads and writes to the global memory into reads and writes to on-chip registers. A quick review of the kernel in Fig. 17.10 shows that for each thread, the same `x[n]`, `y[n]`, and `z[n]` elements are used across all iterations of the for-loop (lines 05−06). This means that we can load these elements into automatic variables before the execution enters the loop. The kernel can then use the automatic variables inside the loop, thus converting global memory accesses to register accesses. Furthermore, the loop repeatedly reads from and writes into rFhD[n] and iFhD[n]. We can have the iterations read from and write into two automatic variables and write only the contents of these

```
01  #define FHD_THREADS_PER_BLOCK 1024
02  __global__ void cmpFhD(float* rPhi, iPhi, phiMag,
                  kx, ky, kz, x, y, z, rMu, iMu, int M) {

03    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;
                 // assign frequently accessed coordinate and output
                 // elements into registers
04    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
05    float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];
06    for (int m = 0; m < M; m++) {
07       float expFhD = 2*PI*(kx[m]*xn_r+ky[m]*yn_r+kz[m]*zn_r);
08       float cArg = cos(expFhD);
09       float sArg = sin(expFhD);
10       rFhDn_r +=  rMu[m]*cArg - iMu[m]*sArg;
11       iFhDn_r +=  iMu[m]*cArg + rMu[m]*sArg;
12    }
13    rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
14  }
```

FIGURE 17.11

Using registers to reduce memory accesses in the F^HD kernel.

automatic variables into rFhD[n] and iFhD[n] after the execution exits the loop. The resulting code is shown in Fig. 17.11. By increasing the number of registers used by 5 for each thread, we have reduced the memory access done in each iteration from 14 to 7. Thus we have increased the compute to global memory access ratio from 0.23 OP/B to 0.46 OP/B. This is a good improvement and a good use of the precious register resource.

Recall that the register usage can limit the occupancy, that is, the number of blocks that can run in a streaming multiprocessor (SM). By increasing the register usage by 5 in the kernel code, we increase the register usage of each thread block by `5*FHD_THREADS_PER_BLOCK`. Assuming that we have 1024 threads per block, we just increased the block register usage by 5120. Since each SM can accommodate a combined register usage of 65,536 registers among all blocks assigned to it (in SM Version 3.5 or higher), we need to be careful, as any further increase of register usage can begin to limit the number of blocks that can be assigned to an SM. Fortunately, the register usage is not a limiting factor to parallelism for this kernel.

We want to further improve the compute to global memory access ratio by eliminating more global memory accesses in the cmpFhD kernel. The next candidates to consider are the k-space samples kx[m], ky[m], and kz[m]. These array elements are accessed differently than the x[n], y[n], and z[n] elements: Different elements of kx, ky, and kz are accessed in each iteration of the loop in Fig. 17.11. This means that we cannot load a k-space element into a register and expect to access that element off a register through all the iterations. Therefore registers will not help here. However, we should notice that the k-space elements are not modified by the kernel. Also, each k-space element is used across all threads in the grid. This means that we might be able to place the k-space elements into the constant memory. Perhaps the constant cache can eliminate most of the DRAM accesses.

An analysis of the loop in Fig. 17.11 reveals that the k-space elements are indeed excellent candidates for the constant memory. The index used for accessing kx, ky, and kz is m. We know that m is independent of threadIdx, which implies that all threads in a warp will be accessing the same element of kx, ky, and kz. This is an ideal access pattern for cached constant memory: Every time an element is brought into the cache, it will be used at least by all 32 threads in a warp for a current generation device. This means that for every 32 accesses to the constant memory, at least 31 of them will be served by the cache. This allows the cache to effectively eliminate 96% or more of the accesses to the global memory. Better yet, each time when a constant is accessed from the cache, it can be broadcast to all the threads in a warp. This makes constant memory almost as efficient as registers for accessing k-space elements.[2]

However, there is a technical issue involved in placing the k-space elements into the constant memory. Recall that constant memory has a capacity of 64KB. However, the size of the k-space samples can be much larger, on the order of millions. A typical way of working around the limitation of constant memory capacity is to break a large dataset down into chunks of 64KB or smaller. The developer must reorganize the kernel so that the kernel will be invoked multiple times, with each invocation of the kernel consuming only a chunk of the large dataset. This turns out to be quite easy for the cmpFhD kernel.

A careful examination of the loop in Fig. 17.11 reveals that all threads will sequentially march through the k-space sample arrays. That is, all threads in the grid access the same k-space element during each iteration. For large datasets the loop in the kernel simply iterates more times. This means that we can divide the loop into sections, with each section processing a chunk of the k-space elements that fit into the 64KB capacity of the constant memory.[3] The host code now invokes the kernel multiple times. Each time the host invokes the kernel, it places a new chunk into the constant memory before calling the kernel function. This is illustrated in Fig. 17.12. (For more recent devices and CUDA versions, a "const __restrict__" declaration of kernel parameters makes the corresponding input data available in the read-only data cache, which is a simpler way of getting the same effect as using constant memory.)

In Fig. 17.12 the `cmpFhD` kernel is called from a loop. The code assumes that kx, ky, and kz arrays are in the host memory. The dimension of kx, ky, and kz is given by M. At each iteration the host code calls the `cudaMemcpyToSymbol()` function to transfer a chunk of the k-space data into the device constant memory, as was discussed in Chapter 7 Convolution, The kernel is then invoked to process

[2] The reason why a constant memory access is not exactly as efficient as a register access is that a memory load instruction is still needed for access to the constant memory.

[3] Note that not all accesses to read-only data are as favorable for constant memory as what we have here. In some applications, threads in different blocks access different input elements in the same iteration. Such more diverged access pattern makes it much harder to fit enough of the data into the constant memory for a kernel launch.

```
__constant__ float kx_c[CHUNK_SIZE], ky_c[CHUNK_SIZE], kz_c[CHUNK_SIZE];
…

void main() {
  for (int i = 0; i < M/CHUNK_SIZE; i++);
    cudaMemcpyToSymbol(kx_c,&kx[i*CHUNK_SIZE],4*CHUNK_SIZE,
              cudaMemCpyHostToDevice);
    cudaMemcpyToSymbol(ky_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE,
              cudaMemCpyHostToDevice);
    cudaMemcpyToSymbol(kz_c,&kz[i*CHUNK_SIZE],4*CHUNK_SIZE,
              cudaMemCpyHostToDevice);
    …
    cmpFhD<<<FHD_THREADS_PER_BLOCK, N/FHD_THREADS_PER_BLOCK>>>(rPhi,
                        iPhi, phiMag, x, y, z, rMu, iMu, CHUNK_SIZE);
  }
  /* Need to call kernel one more time if M is not */
  /* perfect multiple of CHUNK SIZE */
}
```

FIGURE 17.12

Host code sequence for chunking k-space data to fit into constant memory.

```
01  #define FHD_THREADS_PER_BLOCK 1024
02  __global__ void cmpFhD(float* rPhi, iPhi, phiMag,
03                         x, y, z, rMu, iMu, int M) {
04    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;
05    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
06    float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];
07    for (int m = 0; m < M; m++) {
08       float expFhD = 2*PI*(kx_c[m]*xn_r+ky_c[m]*yn_r+kz_c[m]*zn_r);
09       float cArg = cos(expFhD);
10       float sArg = sin(expFhD);
11       rFhDn_r +=  rMu[m]*cArg - iMu[m]*sArg;
12       iFhDn_r +=  iMu[m]*cArg + rMu[m]*sArg;
13    }
14    rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
15  }
```

FIGURE 17.13

Revised F^HD kernel to use constant memory.

the chunk. Note that when M is not a perfect multiple of `CHUNK_SIZE`, the host code will need to have an additional round of `cudaMemcpyToSymbol()` and one more kernel invocation to finish the remaining k-space data.

Fig. 17.13 shows a revised kernel that accesses the k-space data from the constant memory. Note that pointers to kx, ky, and kz are no longer in the parameter list of the kernel function. The `kx_c`, `ky_c`, and `kz_c` arrays are accessed as global variables declared under __constant__ keyword, as shown in Fig. 17.12. By accessing these elements from the constant cache, the kernel now has effectively only four global memory accesses to the rMu and iMu arrays. The compiler will typically recognize that the four array accesses are made to only two locations. It will

perform only two global accesses, one to `rMu[m]` and one to `iMu[m]`. The values will be stored in temporary register variables for use in the other two. This makes the final number of memory accesses equal to 2. The compute to memory access ratio is up to 1.63 OP/B. This is still not quite ideal but is sufficiently high that the memory bandwidth limitation is no longer the only factor that limits performance. As we will see, we can perform a few other optimizations that make the computation more efficient and further improve performance.

If we ran the code in Figs. 17.12 and 17.13, we would have found out that the performance enhancement was not as high as we expected for some devices. As it turns out, the code shown in these figures does not result in as much memory bandwidth reduction as we expected. The reason is that the constant cache does not perform very well for the code. This has to do with the design of the constant cache and the memory layout of the k-space data. As is shown in Fig. 17.14A, each constant cache entry is designed to store multiple consecutive words. This design reduces the cost of constant cache hardware. When an element is brought into the cache, several elements around it are also brought into the cache. This is illustrated as shaded sections surrounding kx[i], ky[i], and kz[i], which are shown as dark boxes in Fig. 17.14. Three cache lines in the constant cache are needed to support the efficient execution of each iteration of a warp.

In a typical execution we will have a fairly large number of warps that are concurrently executing on an SM. Since different warps can be at very different iterations, they may require many constant cache entries altogether. For example, if we define each thread block to have 1024 threads and expect to assign two blocks to execute concurrently in each SM, we will have (1024/32) $\times$ 2=64 warps executing concurrently in an SM. If each of them requires a minimum of three cache lines in the constant cache to sustain efficient execution, in the worst case, we need a total of 64 $\times$ 3=192 cache lines. Even if we assume that, on average, three warps will be executing at the same iteration

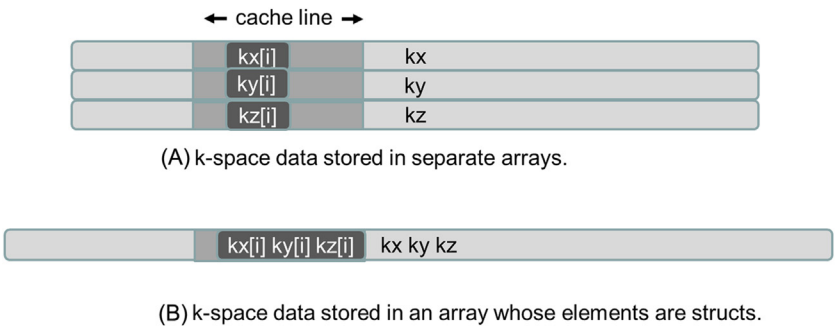


FIGURE 17.14

Effect of k-space data layout on constant cache efficiency: (A) k-space data stored in separate arrays, (B) k-space data stored in an array whose elements are structs.

and thus can share cache lines, we still need 64 cache lines. This is referred to as the working set of all the active warps.

Because of cost constraints, the constant caches of some devices have a small number of cache lines, such as 32. When there are not enough cache lines to accommodate the entire working set, the data that are being accessed by different warps begin to compete with each other for the cache lines. By the time a warp moves to its next iteration, the next elements to be accessed have already been purged to make room for the elements that have been accessed by other warps. As it turns out, the constant cache capacity in some devices is indeed insufficient to accommodate the entries for all the warps that are active in an SM. As a result, the constant cache fails to eliminate many of the global memory accesses.

The problem of inefficient use of cache entries has been well studied in the literature and can be solved by adjusting the memory layout of the k-space data. The solution is illustrated in Fig. 17.14B, and the code based on this solution is shown in Figs. 17.15 and 17.16. Rather than having the x, y, and z components of the k-space data stored in three separate arrays, the solution stores these components in an array whose elements make up a struct. In the literature this style of declaration is often referred to as *array of structures*. The declaration of the array is shown in Fig. 17.15 (lines 01–03). We assume that the memory allocation and initialization code (not shown) has placed the x, y, and z components of the k-

```
01  struct kdata {
02    float x, float y, float z;
03  };

04  __constant__ struct kdata k_c[CHUNK_SIZE];
05  …
06  void main() {
07    for (int i = 0; i < M/CHUNK_SIZE; i++){
08      cudaMemcpyToSymbol(k_c,k,12*CHUNK_SIZE,cudaMemCpyHostToDevice);
10      cmpFhD<<<FHD THREADS PER BLOCK, N/FHD THREADS PER BLOCK>>>(…);
11    }
12  }
```

FIGURE 17.15

Adjusting k-space data layout to improve cache efficiency.

```
01  __global__ void cmpFhD(float* rPhi, iPhi, phiMag,
02                          x, y, z, rMu, iMu, int M) {
03    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;
04    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
05    float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];

06    for (int m = 0; m < M; m++) {
07       float expFhD = 2*PI*(k[m].x*xn_r + k[m].y*yn_r + k[m].z*zn_r);
08       float cArg = cos(expFhD);
09       float sArg = sin(expFhD);
10       rFhDn_r +=  rMu[m]*cArg - iMu[m]*sArg;
11       iFhDn_r +=  iMu[m]*cArg + rMu[m]*sArg;
12    }
13    rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
14  }
```

FIGURE 17.16

Adjusting for the k-space data memory layout in the F^HD kernel.

space data into the fields properly. By storing the x, y, and z components in the three fields of an array element, the developer forces these components to be stored in consecutive locations of the constant memory. Therefore all three components that are used by an iteration of a warp can now fit into one cache entry, reducing the number of entries needed to support the execution of all the active warps. Note that since we have only one array to hold all k-space data, we can just use one CUDA Memcpy To Symbol to copy the entire chunk to the device constant memory. Assuming that each k-space sample is a single-precision floating-point number, the size of the transfer is adjusted from `4*CHUNK_SIZE` to `12*CHUNK_SIZE` to reflect the transfer of all the three components in one CUDA Memcpy To Symbol call.

With the new data structure layout, we also need to revise the kernel so that the access is done according to the new layout. The new kernel is shown in Fig. 17.16. Note that kx[m] has become k[m].x, ky[m] has become k[m].y, and so on. This small change to the code can result in significant enhancement of its execution speed on some devices.[4]

Step 3: Using hardware trigonometry functions

CUDA offers hardware implementations of mathematical functions that provide much higher throughput than their software counterparts. A motivation for GPUs to offer such hardware implementations for trigonometry functions such as `sin( )` and `cos( )` is to improve the speed of view angle transformations in graphics applications. These functions are implemented as hardware instructions executed by the SFU (special function units). The procedure for using these functions is quite easy. In the case of the `cmpFhD` kernel, what we need to do is to change the calls to `sin( )` and `cos( )` functions into their hardware versions: `__sin()` and `__cos()` (two "_" characters before the function name). These are intrinsic functions that are recognized by the compiler and translated into SFU instructions. Because these functions are called in a heavily executed loop body, we expect that the change will result in a significant performance improvement. The resulting `cmpFhD` kernel is shown in Fig. 17.17.

However, we need to be careful about the reduced accuracy in switching from software functions to hardware functions. Hardware implementations currently have less accuracy than software libraries (the details are available in the CUDA C Programming Guide). In the case of MRI we need to make sure that the hardware implementation provides enough accuracy, as defined in Fig. 17.18. The testing process involves a "perfect" image (I_0) of a fictitious object, sometimes referred to as a

[4] The reader might notice that the adjustment from multiple arrays to an array of structures is the opposite of what is often done to global memory data. When adjacent threads in a warp access consecutive elements of an array of structures in global memory, it is much better to store the fields of the structure into multiple arrays so that the global memory accesses are coalesced. The key difference here is that all threads in a warp are accessing the same elements, not consecutive ones.

```
01  #define FHD_THREADS_PER_BLOCK 1024
02  __global__ void cmpFhD(float* rPhi, iPhi, phiMag,
03          x, y, z, rMu, iMu, int M) {

04    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;
05    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
06    float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];
07    for (int m = 0; m < M; m++) {
08       float expFhD = 2*PI*(k[m].x*xn_r+k[m].y*yn_r+k[m].z*zn_r);
09       float cArg = __cos(expFhD);
10       float sArg = __sin(expFhD);
11       rFhDn_r +=  rMu[m]*cArg - iMu[m]*sArg;
12       iFhDn_r +=  iMu[m]*cArg + rMu[m]*sArg;
13    }
14    rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
15  }
```

FIGURE 17.17

Using hardware `__sin()` and `__cos()` functions.

A.N. Netravali and B.G. Haskell, Digital Pictures: Representation, Compression, and Standards (2nd Ed), Plenum Press, New York, NY (1995).

$$MSE = \frac{1}{mn}\sum_i \sum_j (I(i,j) - I_0(i,j))^2 \qquad PSNR = 20\log_{10}\left(\frac{\max(I_0(i,j))}{\sqrt{MSE}}\right)$$

FIGURE 17.18

Metrics used to validate the accuracy of hardware functions. I_0 is the perfect image. I is the reconstructed image. PSNR is peak signal-to-noise ratio.

phantom object. We use a reverse process to generate a corresponding "scanned" k-space data that is synthesized. The synthesized scanned data is then processed by the proposed reconstruction system to generate a reconstructed image (I). The values of the voxels in the perfect and reconstructed images are then fed into the peak signal-to-noise ratio (PSNR) formula in Fig. 17.18.

The criteria for passing the test depend on the application for which the image is intended. In our case, we worked with experts in clinical MRI to ensure that the PSNR changes due to hardware functions were well within the accepted limits for their applications. In applications in which the images are used by physicians to form an impression of injury or to evaluate a disease, one also needs to have visual inspection of the image quality. Fig. 17.19 shows the visual comparison of the original "true" image. It then shows that the PSNR that is achieved by CPU double-precision and single-precision implementations are both 27.6 dB, well above the acceptable level for the application. A visual inspection also shows that the reconstructed image indeed corresponds well with the original image.

The advantage of iterative reconstruction compared to a simple bilinear interpolation gridding/iFFT is also obvious in Fig. 17.19. The image reconstructed with the simple gridding/iFFT has a PSNR of only 16.8 dB, substantially lower than the PSNR of 27.6 dB that is achieved by the iterative reconstruction method.

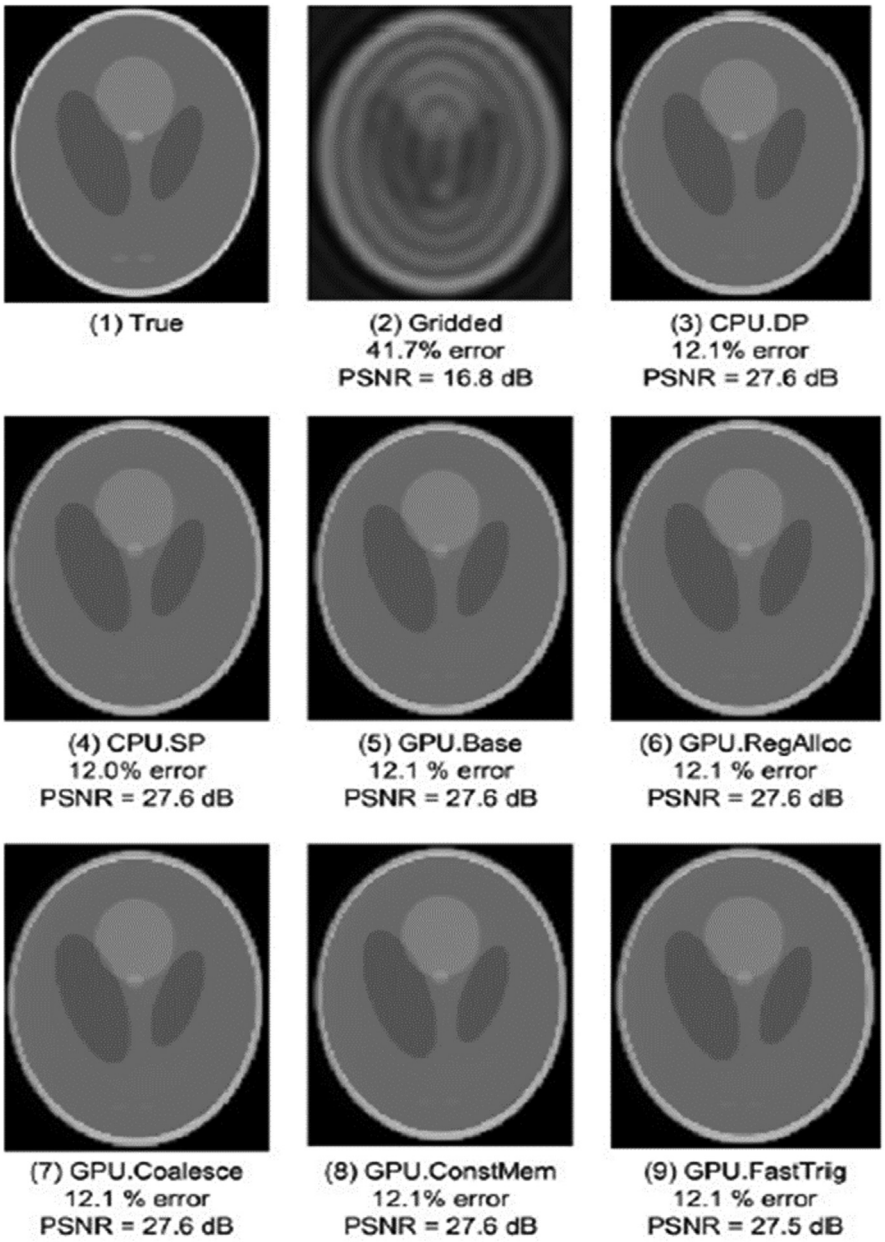


FIGURE 17.19

Validation of floating-point precision and accuracy of the different F^HD implementations.

A visual inspection of the gridding/iFFT image in Fig. 17.19 (image 2) shows that there are severe artifacts that can significantly affect the usability of the image for diagnostic purposes. These artifacts do not occur in the images from the iterative reconstruction method.

When we moved from double-precision arithmetic to single-precision arithmetic on the CPU, there was no measurable degradation of PSNR, which remained at 27.6 dB. When we moved the trigonometry function from the software library to the hardware units, we observed a negligible degradation of PSNR, from 27.6 dB to 27.5 dB. The slight loss of PSNR is within an acceptable range for the application. A visual inspection confirms that the reconstructed image does not have significant artifacts compared to the original image.

Step 4: Experimental performance tuning

Up to this point, we have not determined the appropriate values for the configuration parameters of the kernel. One kernel configuration parameter is the number of threads per block. Using an adequate number of threads per block is needed to fully utilize the thread capacity of each SM. Another kernel configuration parameter is the number of times one should unroll the body of the for-loop in Fig. 17.17 (line 07). This can be set by using a "#pragma unroll" followed by the number of unrolls that we want the compiler to perform on a loop. On one hand, unrolling the loop can reduce the number of overhead instructions and potentially reduce the number of clock cycles to process each k-space sample data. On the other hand, too much unrolling can potentially increase the usage of registers and reduce the number of blocks that can fit into an SM.

Note that the effects of these configurations are not isolated from each other. Increasing one parameter value can potentially use the resource that could have been used to increase another parameter value. As a result, one needs to evaluate these parameters jointly in an experimental manner. There can be a large number of combinations to try. In the case of F^HD, the performance improves about 20% by systematically searching all the combinations and choosing the one with the best measured runtime, as compared to a heuristic tuning search effort that explores only some promising trends. Ryoo et al. (2008) present a Pareto optimal curve–based method to screen away most of the inferior combinations.

17.4 Summary

In this chapter we presented the key steps for parallelizing and optimizing a loop-intensive application—iterative reconstruction of MRI images—from its sequential form. We started with the appropriate organization of parallelization: the scatter approach versus the gather approach. We showed that transforming from a scatter approach to a gather approach is key to avoiding atomic operations, which

can significantly reduce the performance of parallel execution. We discussed the practical techniques, that is, loop fission and loop interchange, that are needed to enable the gather approach to parallelization.

We further presented the application of optimization techniques, such as promoting array elements into registers, using constant memory/cache for input elements, and using hardware functions to improve the performance of the parallel kernel. There is about a 10× speed improvement going from the basic version to the final optimized version, as was discussed.

Before parallelization and optimization, F^HD used to account for nearly 100% of the execution time. An interesting observation is that in the end, the CG solver (the "find ρ" step in Fig. 17.3) can actually take more time than F^HD. This is because we have accelerated F^HD dramatically. Any further acceleration will now require acceleration of the CG solver. After successful parallelization and optimization, F^HD accounts for only about 50%. The other 50% is largely spent in the CG solver. This is a well-known phenomenon in parallelizing real applications. Because some time-consuming phases of the execution are accelerated by successful parallelization efforts, the execution time becomes dominated by other phases that used to account for insignificant portions of the execution.

Exercises

1. Loop fission splits a loop into two loops. Use the F^HD code in Fig. 17.4 and enumerate the execution order of the two parts of the outer loop body: (1) the statements before the inner loop and (2) the inner loop.
   a. List the execution order of these parts from different iterations of the outer loop before fission.
   b. List the execution order of these parts from the two loops after fission.
   c. Determine whether the execution results in parts (a) and (b) of this exercise will be identical. The execution results are identical if all data required by a part are properly generated and preserved for its consumption before that part executes and the execution result of the part is not overwritten by other parts that should come after the part in the original execution order.
2. Loop interchange swaps the inner loop into the outer loop and vice versa. Use the loops from Fig. 17.9 and enumerate the execution order of the instances of loop body before and after the loop exchange.
   a. List the execution order of the loop body from different iterations before loop interchange. Identify these iterations with the values of m and n.
   b. List the execution order of the loop body from different iterations after loop interchange. Identify these iterations with the values of m and n.
   c. Determine whether the execution results in parts (a) and (b) of this exercise will be identical. The execution results are identical if all data required by a

part are properly generated and preserved for its consumption before that part executes and the execution result of the part is not overwritten by other parts that should come after the part in the original execution order.

3. In Fig. 17.11, identify the difference between the access to x[] and kx[] in the nature of indices used. Use the difference to explain why it does not make sense to try to load kx[n] into a register for the kernel shown in Fig. 17.11.

References

Liang, Z.P., Lauterbur, P., 1999. Principles of Magnetic Resonance Imaging: A Signal Processing Perspective. John Wiley and Sons.

Ryoo, S., Ridrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, Z., Baghsorkhi, S.S., et al., 2008. Program optimization carving for GPU computing. J. Parallel Distrib. Comput. Available from: https://doi.org/10.1016/j.jpdc.2008.050.011.

Stone, S.S., Haldar, J.P., Tsao, S.C., Hwu, W.W., Sutton, B.P., Liang, Z.P., 2008. Accelerating advanced MRI reconstruction on GPUs. J. Parallel Distrib. Comput. 68 (10), 1307–1318. Available from: https://doi.org/10.1016/j.jpdc.2008.050.013.

CHAPTER

Electrostatic potential map 18

With special contributions from John Stone

Chapter Outline

The previous case study used a statistical estimation application to illustrate the process of selecting an appropriate level of a loop nest for parallel execution, transforming the loops for reduced memory access interference, using constant memory for magnifying the memory bandwidth for read-only data, using registers to reduce the consumption of memory bandwidth, and using special hardware functional units to accelerate trigonometry functions. In this case study, we use a molecular dynamics application based on regular grid data structures to illustrate the use of optimization techniques that achieve global memory access, coalescing and improved computation throughput. As we did in the previous case study, we present a series of implementations of an electrostatic potential map calculation kernel in which each version improves on the previous one. Each version adopts one or more practical techniques from Chapter 6, Performance Considerations. Some of the techniques were used in the previous case study, but some are different: systematic reuse of computational results, thread granularity coarsening, and fast boundary condition checking. This application case study shows that the effective use of these practical techniques can significantly improve the execution throughput of the application.

18.1 Background

This case study is based on visual molecular dynamics (VMD) (Humphrey et al., 1996), a popular software system that was designed for displaying, animating, and

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00009-4

analyzing biomolecular systems. VMD has more than 200,000 registered users. It is an important foundation for a modern "computational microscope" with which biologists can observe tiny life forms, such as viruses, that are too small for traditional microscopy techniques. While it has strong built-in support for analyzing biomolecular systems, such as calculating electrostatic potential values at spatial grid points of a molecular system (the focus of this chapter), it has also been a popular tool for displaying other large datasets, such as sequencing data, quantum chemistry simulation data, and volumetric data, owing to its versatility and user extensibility.

While VMD is designed to run on a diverse range of hardware, including laptops, desktops, clusters, and supercomputers, most users use VMD as a desktop science application for interactive three-dimensional (3D) visualization and analysis. For computation that runs too long for interactive use, VMD can also be used in a batch mode to render movies for later use. A motivation for accelerating VMD is to make batch mode jobs fast enough for interactive use. This can drastically improve the productivity of scientific investigations. With CUDA devices widely available in desktop PCs, such acceleration can have broad impact on the VMD user community. To date, multiple aspects of VMD have been accelerated with CUDA, including electrostatic potential map calculation, ion placement, molecular orbital calculation and display, and imaging of gas migration pathways in proteins.

The computation covered in this case study is the calculation of electrostatic potential maps in a grid space. This calculation is often used in the placement of ions into a molecular structure for molecular dynamics simulation. Fig. 18.1

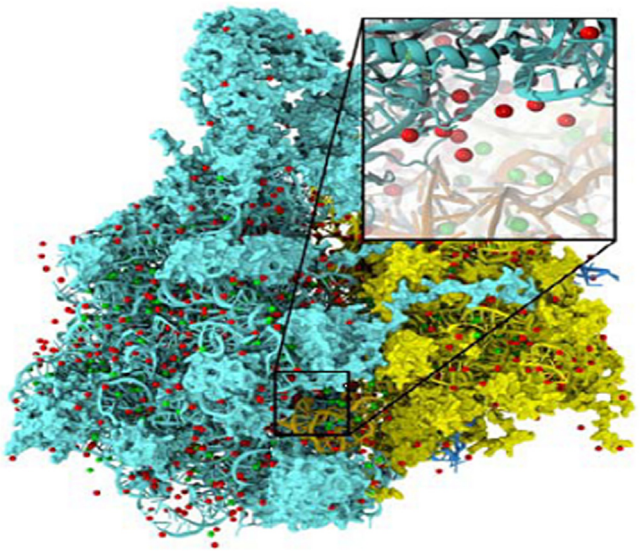


FIGURE 18.1

Electrostatic potential map used in building stable structures for molecular dynamics simulation.

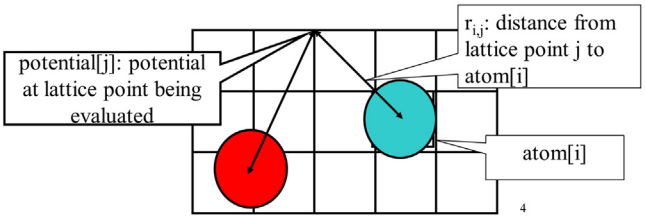


FIGURE 18.2

The contribution of atom[i] to the electrostatic potential at lattice point j (potential[j]) is atom[i]. charge/r_{ij}. In the direct coulomb summation method, the total potential at lattice point j is the sum of contributions from all atoms in the system.

shows the placement of ions into a protein structure in preparation for a molecular dynamics simulation. In this application the electrostatic potential map is used to identify spatial locations where ions (red dots) can fit in according to physical laws. The function can also be used to calculate time-averaged electrical field potential maps during molecular dynamics simulation, which is useful for the simulation process as well as the visualization and analysis of simulation results.

There are several methods for calculating electrostatic potential maps. Among them, direct Coulomb summation (DCS) is a highly accurate method that is particularly suitable for GPUs (Stone et al., 2007). The DCS method calculates the electrostatic potential value of each grid point as the sum of contributions from all atoms in the system. This is illustrated in Fig. 18.2. The contribution of atom i to a lattice point j is the charge of atom i divided by the distance from lattice point j to atom i. Since this needs to be done for all grid points and all atoms, the number of calculations is proportional to the product of the total number of atoms in the system and the total number of grid points. For a realistic molecular system this product can be very large. Therefore the calculation of the electrostatic potential map has been traditionally done as a batch job in VMD.

18.2 Scatter versus gather in kernel design

Fig. 18.3 shows the base C code of the DCS code. The function is written to process a two-dimensional (2D) slice of a 3D grid. The function will be called repeatedly for all the slices of the modeled space. The structure of the function is quite simple with three levels of `for` loops. The outer two levels iterate over the *y* dimension and the *x* dimension of the grid point space. For each grid point, the innermost for loop iterates over all atoms, calculating the contribution of electrostatic potential energy from all atoms to the grid point. Note that each atom is represented by four consecutive elements of the atoms[] array. The first three elements store the *x*, *y*, and *z* coordinates of the atom and the fourth element the electrical charge of the atom. At the end of the innermost loop, the accumulated

```
01 void cenergy(float *energygrid, dim3 grid, float gridspacing, float z,
02                 const float *atoms, int numatoms) {
03   int atomarrdim = numatoms * 4; //x,y,z, and charge info for each atom
04   for (int j=0; j<grid.y; j++) {
       // calculate y coordinate of the grid point based on j
05     float y = gridspacing * (float) j;
06     for (int i=0; i<grid.x; i++) {
         // calculate x coordinate based on i
07       float x = gridspacing * (float) i;
08       float energy = 0.0f;
09       for (int n=0; n<atomarrdim; n+=4) {
10         float dx = x - atoms[n  ];
11         float dy = y - atoms[n+1];
12         float dz = z - atoms[n+2];
13         energy += atoms[n+3] / sqrtf(dx*dx + dy*dy+ dz*dz);
14       }
15       energygrid[grid.x*grid.y*z + grid.x*j + i] = energy;
16     }
17   }
18 }
```

FIGURE 18.3

An unoptimized direct Coulomb summation C code for a two-dimensional slice.

value of the grid point is written out to the grid data structure. The outer loops then iterate and take the execution to the next grid point.

Note that the DCS function in Fig. 18.3 calculates the x and y coordinates of each grid point on the fly by multiplying the grid point index values by the spacing between grid points. This is a uniform grid method in which all grid points are spaced at the same distance in all three dimensions. The function takes advantage of the fact that all the grid points in the same slice have the same z coordinate. This value is precalculated by the caller of the function and passed in as a function parameter (z). There are, however, several optimizations that can be done to the sequential C code in Fig. 18.3 to significantly improve its execution speed.

Fig. 18.4 shows a C code for DCS with a few optimizations to improve its execution speed and efficiency. First, the innermost loop (n-loop) in Fig. 18.3 has been exchanged into the outermost loop (line 05 in Fig. 18.4). Thus the code iterates over all atoms. For each atom the inner loops (i-loop and j-loop) scatter the contribution of the atom to all the grid points. As we discussed in Chapter 17, Iterative Magnetic Resonance Imaging Reconstruction, the loop interchange is permissible because the three levels of loops in Fig. 18.3 are perfectly nested and all the iterations are independent of each other.

The loop interchange enables two optimizations. First, the z components of the distance between an atom and all grid points in the plane are identical and can be calculated once for the entire slice of the grid point. Therefore the calculation can be done outside the two inner loops (lines 6−7). Similarly, the y components of distance between an atom and all grid points in the same row are identical and can be done outside the innermost loop (lines 11−12). In comparison, the y and z components of the distance were both calculated in the innermost loop in Fig. 18.3. This drastic reduction in the number of calculations makes the

```
01 void cenergy(float *energygrid, dim3 grid, float gridspacing, float z,
02              const float *atoms, int numatoms) {
03   int atomarrdim = numatoms * 4; //x,y,z, and charge info for each atom
     // starting point of the slice in the energy grid
04   int grid_slice_offset = (grid.x*grid.y*z) / gridspacing;
     //  calculate potential contribution of each atom
05   for (int n=0; n<atomarrdim; n+=4) {
06      float dz = z - atoms[n+2];  // all grid points in a slice have the same
07      float dz2 = dz*dz;          // z  value, no recalculation in inner loops
08      float charge = atoms[n+3];
09      for (int j=0; j<grid.y; j++) {
10         float y = gridspacing * (float) j;
11         float dy = y - atoms[n+1];  // all grid points in a row have the same
12         float dy2 = dy*dy;          // y value
13         int grid_row_offset =  grid_slice_offset+ grid.x*j;
14         for (int i=0; i<grid.x; i++) {
15            float x = gridspacing * (float) i;
16            float dx = x - atoms[n    ];
17            energygrid[grid_row_offset+i] += charge / sqrtf(dx*dx + dy2+ dz2);
18         }
19      }
20    }
21  }
```

FIGURE 18.4

An optimized direct Coulomb summation C code for a two-dimensional slice.

C code in Fig. 18.4 much faster. These optimizations cannot be done in Fig. 18.3 because the innermost loop iterates over all atoms, so one must recalculate the x, y, and z components of the distance as they change from atom to atom.

For GPU execution we assume that the host program inputs and maintains the atomic charges and their coordinates in the system memory. It also maintains the grid point data structure in the system memory. The DCS kernel is designed to process a 2D slice of the electrostatic potential grid point structure (not to be confused with thread grids). These grid points are like the grid points for discretization that were discussed in Chapter 8, Stencil. For each 2D slice, the CPU transfers its grid data to the device global memory. Similar to the k-space data (Chapter 17, Iterative Magnetic Resonance Imaging Reconstruction), the atom information is divided into chunks to fit into the constant memory. For each chunk of the atom information, the CPU transfers the chunk into the device constant memory, invokes the DCS kernel to calculate the contribution of the current chunk to the current slice, and prepares to transfer the next chunk. After all chunks of the atom information have been processed for the current slice, the slice is transferred back to update the grid point data structure in the CPU system memory. The system then moves on to the next slice.

Let us now focus on the design of DCS kernel. It is natural to parallelize the optimized C code in Fig. 18.4. The resulting kernel is shown in Fig. 18.5. The defined constant `CHUNK_SIZE` specifies the number of atoms that should be transferred into the GPU constant memory for each kernel call. The value of `CHUNK_SIZE*4` should be less than or equal to 64K. The kernel uses each thread to implement an iteration of the outermost loop in Fig. 18.4 and scatters the contribution of its assigned atom to

```
01    constant   float atoms[CHUNK SIZE*4];
02  void  __global__ cenergy(float *energygrid, dim3 grid, float gridspacing,
03                           float z) {
04    int n = (blockIdx.x * blockDim .x + threadIdx.x) * 4;
05    float dz = z-atoms[n+2];  // all grid points in a slice have the same
06    float dz2 = dz*dz;        // z value
      // starting position of the slice in the energy grid
07    int grid_slice_offset = (grid.x*grid.y*z) / gridspacing;
08    float charge = atoms[n+3];
09    for (int j=0; j<grid.y; j++) {
10       float y = gridspacing * (float) j;
11       float dy = y-atoms[n+1];   // all grid points in a row have the same
12       float dy2 = dy*dy;         // y value
         // starting position of the row in the energy grid
13       int grid_row_offset =  grid_slice_offset+ grid.x*j;
14       for (int i=0; i<grid.x; i++) {
15           float x = gridspacing * (float) i;
16           float dx = x - atoms[n    ];
17           atomicAdd(&energygrid[grid_row_offset+i],
18                     charge / sqrtf(dx*dx+dy2+dz2));
19       }
20    }
21  }
```

FIGURE 18.5

Direct Coulomb summation kernel using the scatter approach.

all grid points. Unfortunately, as we learned in Chapter 17, Iterative Magnetic Resonance Imaging Reconstruction, this scatter approach to parallelization requires atomic operations for updating the energy grid points (lines 17–18), which significantly reduces the speed of parallel execution.

As we learned in Chapter 17, Iterative Magnetic Resonance Imaging Reconstruction, we can instead use a gather approach in which each thread calculates the accumulated contributions of all atoms to one grid point. This is a preferred approach, since each thread will be writing into its own grid point and there is no need to use atomic operations. However, this requires the loops to be arranged in the ordering of the unoptimized C code in Fig. 18.3; that is, we would be parallelizing a slower C implementation. This exemplifies a frequently experienced dilemma in parallelizing applications: The optimized sequential code is not as amenable to parallelization as the unoptimized sequential code is. The downside is that we can end up with drastically slower execution within each thread, which can reduce the speed benefit of parallelization. We will return to this point later in this chapter.

Fig. 18.6 shows a kernel based on the gather approach. The kernel is based on the unoptimized C code in Fig. 18.3. We form a 2D thread grid that matches the 2D potential grid point organization. To do so, we need to modify the two outer loops in lines 04–06 of Fig. 18.3 into perfectly nested loops so that we can use each thread to execute one iteration of the two-level loop. We can either perform a loop fission (as we did in the previous case study) or move the calculation of the y coordinate (line 05 of Fig. 18.3) into the inner loop. The former would require us to create a new array to hold all y values and would result in two kernels communicating data through the

```
01    constant   float atoms[CHUNK SIZE*4];
02  void __global__ cenergy(float *energygrid, dim3 grid, float gridspacing,
03                          float z, int numatoms) {
04     int i = blockIdx.x * blockDim.x + threadIdx.x;
05     int j = blockIdx.y * blockDim.y + threadIdx.y;
06     int atomarrdim = numatoms * 4;
07     int k = z / gridspacing;
08     float y = gridspacing * (float) j;
09     float x = gridspacing * (float) i;
10     float energy = 0.0f;
       // calculate potential contribution from all atoms
11     for (int n=0; n<atomarrdim; n+=4) {
12        float dx = x - atoms[n  ];
13        float dy = y - atoms[n+1];
14        float dz = z - atoms[n+2];
15        energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
16     }
17     energygrid[grid.x*grid.y*k + grid.x*j + i] += energy;
18   }
```

FIGURE 18.6

Direct Coulomb summation kernel using the gather approach.

global memory. The latter increases the number of times that the y coordinate will be calculated. In this case, we choose to perform the latter, since there is only a small amount of calculation that can be easily accommodated in the inner loop without significantly increasing the execution time of the inner loop. The amount of work to be absorbed into the inner loop is much smaller than that in Chapter 17, Iterative Magnetic Resonance Imaging Reconstruction. The former would have added a kernel launch overhead for a kernel in which threads do little work. The selected transformation allows all i and j iterations to be executed in parallel. This is a tradeoff between the amount of calculation done and the level of parallelism achieved.

Inside the kernel code in Fig. 18.6, the outer two levels of the loop in Fig. 18.3 have been removed and are replaced by the execution configuration parameters in the kernel invocation (lines 04–05 of Fig. 18.6). Within each thread grid, the thread blocks are organized to calculate the electrostatic potential of tiles of the grid structure. In the simplest kernel, each thread calculates the value at one grid point. In more sophisticated kernels, each thread calculates multiple grid points and exploits the redundancy between the calculations of the grid points to improve execution speed. This is an example of the thread coarsening optimization discussed in Chapter 6, Performance Considerations and will be discussed in the next section.

The performance of the kernel in Fig. 18.6 is quite good, since its execution speed is not hampered by atomic operations. Also, a quick glance over the code shows that each thread does nine floating-point operations for every four memory elements accessed. These atoms[] array elements for each atom are cached in a hardware constant cache memory in each streaming multiprocessor (SM) and are broadcast to many threads. The massive reuse of these constant memory elements across threads makes the constant cache extremely effective, eliminating the vast majority of the DRAM accesses. As a result, global memory bandwidth is not a limiting factor for this kernel.

18.3 Thread coarsening

Although the kernel in Fig. 18.6 avoids the global memory bottleneck through constant caching, it still needs to execute four constant memory access instructions for every nine floating-point operations performed. These memory access instructions consume hardware resources that could otherwise be used to increase the execution throughput of floating-point instructions. Furthermore, the execution of these memory access instructions consumes energy, an important limiting factor for many large-scale parallel computing systems. This section shows that we can use the thread coarsening technique to fuse several threads together so that the atoms[] data can be fetched once from the constant memory, stored in registers, and used for multiple grid points.

Furthermore, as shown in Fig. 18.7, all energy grid points along the same row (*y* dimension) have the same *y* coordinate. Therefore the difference between the *y* coordinate of an atom and the *y* coordinate of any grid point along a row has the same value. In the DCS kernel in Fig. 18.6, this calculation is redundantly done by all threads for all grid points in a row when calculating the distance between the atom and the grid points. We can eliminate this redundancy and improve the execution efficiency.

The idea is to have each thread calculate the electrostatic potential for multiple energy grid points in the same row. The kernel in Fig. 18.8 has each thread calculate four grid points. For each atom the code calculates dy, the difference of the y coordinates, only once (line 21). It then calculates the expression dy*dy + dz*dz and saves it to the automatic variable dysqdzsq, which is assigned to a register (line 23). This value is the same for all four grid points. The electrical charge information is also accessed from constant memory and stored in the automatic variable charge (line 24). Therefore the calculation of energy0 through energy3 can all just use the values stored in the registers.

Similarly, the *x* coordinate of the atom is also read from constant memory and used to calculate dx0 through dx3 (lines 17−20). Altogether, this kernel eliminates three accesses to constant memory for the *y* coordinate of its atom, three accesses for the *x* coordinate of its atom, three accesses for the charge of the atom, three floating-point subtraction operations, five floating-point multiply operations, and nine floating-point add operations in processing an atom for four grid points. A quick inspection of the kernel code in Fig. 18.8 shows that each

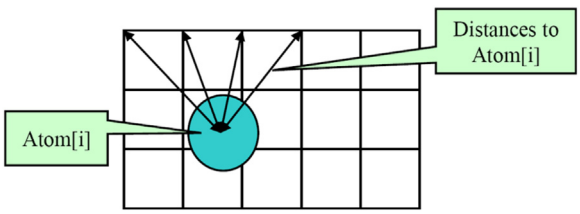


FIGURE 18.7

Reusing computation results among multiple grid points.

```
01    constant   float atoms[CHUNK SIZE*4];
02  #define COARSEN_FACTOR 4
03  void __global__ cenergy(float *energygrid, dim3 grid, float gridspacing,
04                          float z, int numatoms) {
05     int i = blockIdx.x * blockDim.x*COARSEN_FACTOR + threadIdx.x;
06     int j = blockIdx.y * blockDim.y + threadIdx.y;
07     int atomarrdim = numatoms * 4;
08     int k = z / gridspacing;
09     float y = gridspacing * (float) j;
10     float x = gridspacing * (float) i;
11     float energy0 = 0.0f;
12     float energy1 = 0.0f;
13     float energy2 = 0.0f;
14     float energy3 = 0.0f;
15     // calculate potential contribution from all atoms
16     for (int n=0; n<atomarrdim; n+=4) {
17        float dx0 = x - atoms[n  ];
18        float dx1 = dx0 +   gridspacing;
19        float dx2 = dx0 + 2*gridspacing;
20        float dx3 = dx0 + 3*gridspacing;
21        float dy = y - atoms[n+1];
22        float dz = z - atoms[n+2];
23        float dysqdzsq = dy*dy + dz*dz;
24        float charge = atoms[n+3];
25        energy0 += charge / sqrtf(dx0*dx0 + dysqdzsq);
26        energy1 += charge / sqrtf(dx1*dx1 + dysqdzsq);
27        energy2 += charge / sqrtf(dx2*dx2 + dysqdzsq);
28        energy3 += charge / sqrtf(dx3*dx3 + dysqdzsq);
29     }
30     energygrid[grid.x*grid.y*k + grid.x*j + i  ] += energy0;
31     energygrid[grid.x*grid.y*k + grid.x*j + i+1] += energy1;
32     energygrid[grid.x*grid.y*k + grid.x*j + i+2] += energy2;
33     energygrid[grid.x*grid.y*k + grid.x*j + i+3] += energy3;
34   }
```

FIGURE 18.8

Direct Coulomb summation kernel with thread coarsening.

iteration of the loop performs four constant memory accesses, three floating-point subtractions, eleven floating-point additions, six floating-point multiplications, and four floating-point divisions for four grid points.

The reader should also verify that the version of DCS kernel in Fig. 18.6 performs 16 constant memory accesses, 12 floating-point subtractions, 12 floating-point additions, 12 floating-point multiplications, and 12 floating-point divisions—a total of 48 floating-point operations for the same four grid points. Going from Fig. 18.6—Fig. 18.8, there is a total reduction from 16 constant memory accesses and 48 operations down to 4 constant memory accesses and 24 floating-point operations, a sizable reduction. One can expect a sizable improvement in both the execution time and the energy consumption of the kernel.

The cost of the optimization is that more registers are used by each thread. This can potentially reduce the number of threads that can be accommodated by each SM. However, since the number of registers stays within the permissible limit, it does not limit the occupancy of GPU execution resources.

18.4 Memory coalescing

While the performance of the DCS kernel in Fig. 18.8 is quite high, a quick profiling run reveals that the threads perform memory writes inefficiently. In lines 30–33, each thread writes four neighboring grid points. Unfortunately, the write pattern of adjacent threads in each warp will result in uncoalesced global memory writes. There are two problems that cause the uncoalesced write pattern in the kernel. First, each thread calculates four adjacent neighboring grid points. Thus for each statement that writes to the energygrid[] array, the threads in a warp are not accessing adjacent locations. Note that two adjacent threads access memory locations that are four elements apart. Thus the 32 locations to be written by all the threads in a warp are spread out, with three elements in between the loaded/written locations. This problem can be solved by assigning adjacent grid points to adjacent threads in each block. We first assign blockDim.x consecutive grid points in the *x* dimension to the threads. We then assign the next blockDim.x consecutive grid points to the same threads. We repeat the assignment until each thread has the number of grid points desired. This assignment is illustrated in Fig. 18.9.

The kernel code with coarsened thread granularity and coalescing-aware assignment of grid points to threads is shown in Fig. 18.10. Note that the x coordinates that are used to calculate the atom-to-grid-point distances for a thread's assigned grid points are offset by blockDim.x*gridspacing. This reflects the fact that the *x* coordinates of the four grid points assigned to a thread are blockDim.x grid points away from each other. After the end of the loop, the indices of the memory writes to the energygrid array are blockDim.x away from each other as well. Hence all writes to the energygrid array will be coalesced, and the performance of the kernel will be improved over that of Fig. 18.8.

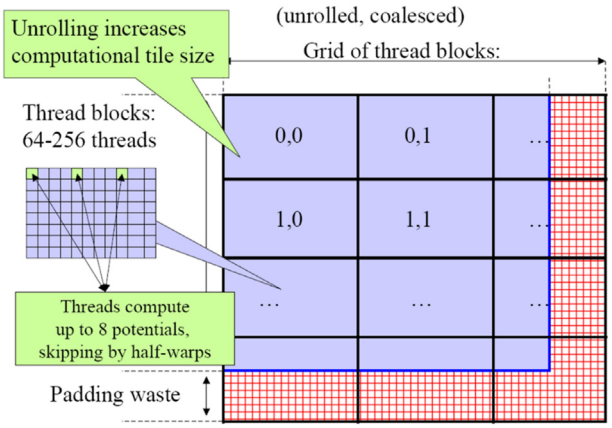


FIGURE 18.9

Organizing threads and memory layout for coalesced writes.

```
01    constant   float atoms[CHUNK SIZE*4];
02  #define COARSEN_FACTOR 4
03  void __global__ cenergy(float *energygrid, dim3 grid, float gridspacing,
04                          float z, int numatoms) {
05     int i = blockIdx.x * blockDim.x*COARSEN_FACTOR + threadIdx.x;
06     int j = blockIdx.y * blockDim.y + threadIdx.y;
07     int atomarrdim = numatoms * 4;
08     int k = z / gridspacing;
09     float y = gridspacing * (float) j;
10     float x = gridspacing * (float) i;
11     float energy0 = 0.0f;
12     float energy1 = 0.0f;
13     float energy2 = 0.0f;
14     float energy3 = 0.0f;
15     // calculate potential contribution from all atoms
16     for (int n=0; n<atomarrdim; n+=4) {
17        float dx0 = x - atoms[n  ];
18        float dx1 = dx0 +   blockDim.x * gridspacing;
19        float dx2 = dx0 + 2*blockDim.x * gridspacing;
20        float dx3 = dx0 + 3*blockDim.x * gridspacing;
21        float dy = y - atoms[n+1];
22        float dz = z - atoms[n+2];
23        float dysqdzsq = dy*dy + dz*dz;
24        float charge = atoms[n+3];
25        energy0 += charge / sqrtf(dx0*dx0 + dysqdzsq);
26        energy1 += charge / sqrtf(dx1*dx1 + dysqdzsq);
27        energy2 += charge / sqrtf(dx2*dx2 + dysqdzsq);
28        energy3 += charge / sqrtf(dx3*dx3 + dysqdzsq);
29     }
30     energygrid[grid.x*grid.y*k + grid.x*j + i              ] += energy0;
31     energygrid[grid.x*grid.y*k + grid.x*j + i +   blockDim.x] += energy1;
32     energygrid[grid.x*grid.y*k + grid.x*j + i + 2*blockDim.x] += energy2;
33     energygrid[grid.x*grid.y*k + grid.x*j + i + 3*blockDim.x] += energy3;
34   }
```

FIGURE 18.10

Direct Coulomb summation kernel with thread coarsening and memory coalescing.

18.5 Cutoff binning for data size scalability

We can typically come up with multiple algorithms to solve a given problem. Some algorithms require fewer steps of computation than others, some expose a higher degree of parallel execution than others, some have better numerical stability than others, and some consume less memory bandwidth than others. Unfortunately, there is often not a single algorithm that is better than others in all the four aspects. Given a problem and a decomposition strategy, a parallel programmer often needs to select an algorithm that achieves the best compromise for a given hardware system.

In general, alternative algorithms for solving the same problem should reach the same solution. Under this requirement, one can optimize the computation for better efficiency and/or more parallelism. In some applications, one can often come up with even more aggressive algorithm strategies if the problem can be solved with slight variation in the final solution. An important algorithm strategy,

referred to as *cutoff summation*, can significantly improve the execution efficiency of grid algorithms such as the electrostatic potential calculation by sacrificing a small amount of accuracy. This is based on the observation that many grid calculation problems are based on physical laws in which numerical contributions from particles or samples that are far away from a grid point can be collectively treated with an implicit method at much lower computational complexity.

The cutoff summation strategy is illustrated for the electrostatic potential calculation in Fig. 18.11. Fig. 18.11(A) shows the direct summation algorithms that were discussed in the previous sections of this chapter. Each grid point receives contributions from all atoms. While this is a very parallel approach and achieves excellent speedup, it does not scale well to very large energy grid systems in which the number of atoms increases proportionally to the volume of the system. The amount of computation increases with the square of the volume. For large volume systems, such increase makes the computation excessively long even for massively parallel devices.

We know that each grid point needs to receive accurate contributions from atoms that are close to it. The atoms that are far away from a grid point will have a tiny contribution to the energy value at the grid point because the contribution is inversely proportional to the distance. Fig. 18.11(B) illustrates this observation with a circle drawn around a grid point. The contributions to the grid point energy from atoms outside the circle (dark atoms) are small and can be handled with a separate implicit method. If we can devise an algorithm in which each grid point receives contributions only from atoms within a fixed radius of its coordinate, the computational complexity of the algorithm will be reduced to being linearly proportional to the volume of the system. This would make the computation time of the algorithm linearly proportional to the volume of the system. Such algorithms have been used extensively in sequential computation.

In sequential computing, a simple cutoff algorithm handles one atom at a time. For each atom the algorithm iterates through the grid points that fall within a radius of the atom's coordinate. This is a straightforward procedure, since the

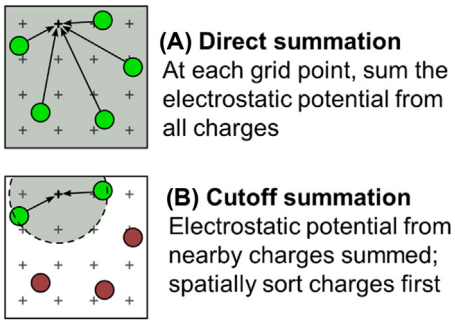


FIGURE 18.11

(A) Cutoff summation versus (B) direct summation.

grid points are in an array that can be easily indexed as a function of their coordinates. A C implementation of the cutoff algorithm can be derived with minor modifications to the C implementation of DCS in Fig. 18.4 by restricting the range of i and j of the inner loops to the energy grid points that fall into the radius. However, this simple procedure does not carry over easily to parallel execution. The reason is what we discussed in Section 18.2: The atom-centric parallelization does not work well, owing to its scatter memory update behavior.

Therefore we need to find a cutoff binning algorithm based on the grid-centric decomposition: Each thread calculates the energy value at one grid point. Fortunately, there is a well-known approach to adapting a direct summation algorithm, such as the one in Fig. 18.10, into a cutoff binning algorithm. Rodrigues et al. presents such an algorithm for the electrostatic potential problem (Rodrigues et al., 2008).

The key idea of the algorithm is to first sort the input atoms into bins according to their coordinates. Each bin corresponds to a box in the energy grid space, and it contains all atoms whose coordinates fall into the box. These bins are implemented as multidimensional arrays: the x, y, and z dimensions as well as the fourth dimension that is a vector of atoms in the bin.

We define a "neighborhood" of bins for a grid point as the collection of bins that contain all the atoms that can contribute to the energy value of the grid point. Fig. 18.12 shows an example of the neighborhood bins for a grid point. Note that nine bins around the grid point overlap with the circle of cutoff distance. For the correct cutoff summation we need to make sure that all atoms in these nine bins are considered for contribution to the grid point. Note that some of the atoms in the neighborhood bins may not fall into the radius. Therefore in processing an atom from one of the neighborhood bins, all threads need to check whether the atom falls into its radius. This can cause some control divergence among threads in a warp.

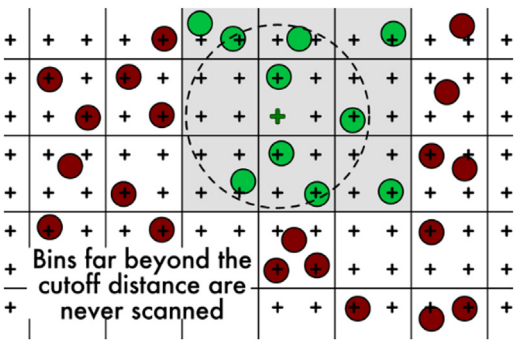


FIGURE 18.12

Neighborhood bins for a grid point.

Although Fig. 18.12 shows only one layer (2D) of bins that immediately surround the bin containing a grid point as its neighborhood, a real algorithm will typically have multiple 3D bins in a grid point's neighborhood. In this approach, all threads iterate through their own neighborhood. They use their block and thread indices to identify the coordinates of their assigned grid point and use these coordinates to identify the appropriate bins to examine. One can think of the neighborhood bins as a stencil in the energy grid space. However, the calculations that are involved in determining the neighborhood bins given a cutoff radius can be a complex geometric problem whose solution can be time consuming. Therefore the neighborhood bins are usually defined for all threads in a block and are prepared before launching the grid.

Fig. 18.13 shows a per-block design of neighborhood bins. Based on the block dimensions and the grid spacing, one can calculate the area (volume in 3D) of energy grid space covered by each block. We show the areas covered by the blocks as squares in Fig. 18.13. For simplicity we assume that each of these areas is also covered by a bin; that is, all atoms that fall into the same square will be collected into the same bin. For example, if the grid spacing is 0.5 Å and the blocks are $8 \times 8 \times 8$, each block would cover a 4 Å $\times$ 4 Å $\times$ 4Å cube in the energy grid space. If we assume a typical cutoff distance of 12 Å for molecular-level force calculation, we need to identify all bins that can potentially be fully or partially covered by any of the 512 circles, each of which is centered at one of the grid points covered by one of the threads in the block. We can also use a conservative approximation by drawing a supercircle that is centered at the center of the bin with a radius that is the cutoff distance plus half of the bin diagonal. The rationale is that this supercircle would cover all the circles that are centered at the corners of the bin. We can simply create a list of the relative positions of the bins that are fully or partially covered by the supercircle.

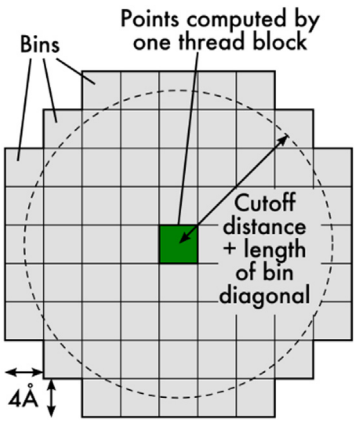


FIGURE 18.13

Identifying the neighborhood bins for all grid points processed by a block.

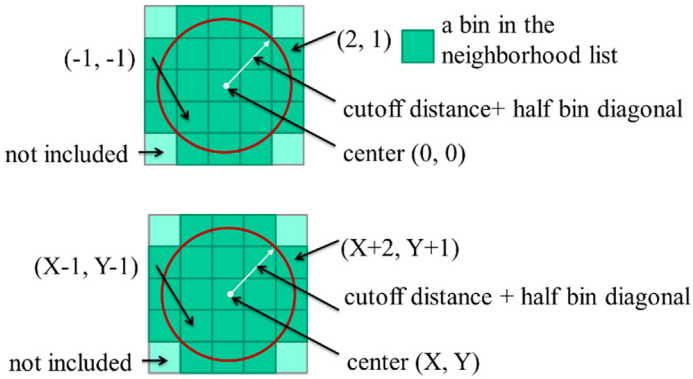


FIGURE 18.14

Neighborhood list using relative offsets.

Fig. 18.14 shows a small example of identifying neighborhood bins. We see that for each bin, there are 9 bins that are fully covered by the supercircle and 12 bins that are partially covered. We can generate a list of 9 + 12=21 neighborhood bins that each thread in a block needs to examine for atoms that are within the cutoff distance of the grid point covered by the thread. These bins are expressed as the relative offset of the bin coordinates. For example, the 9 bins that are fully covered by the supercircle can be expressed as the list $(-1, -1)$, $(0, -1)$, $(1, -1)$, $(-1, 0)$, $(0, 0)$, $(1, 0)$, $(-1, 1)$, $(0, 1)$, and $(1, 1)$, as illustrated in the top portion of Fig. 18.14. The list would be supplied to the kernel, most likely as a constant memory array. During kernel execution, all threads in a block would iterate through the neighborhood list. For each neighborhood bin, the threads apply the offsets to the coordinates of the bin covered by the block and derive the coordinate of the neighbor bin, as illustrated in the bottom portion of Fig. 18.14. They collaborate to load the atoms in the bin into the shared memory and then each thread individually checks whether these atoms fall within the cutoff distance of its assigned grid point. Each thread could make different decisions about including or excluding each atom for contributing to the energy value at its assigned grid point.

The improvement in computational complexity in a cutoff binning algorithm mainly comes from the fact that each thread now examines a much smaller subset of atoms, defined by the neighborhood bins, in a large grid system. However, this makes constant memory much less attractive for holding the atoms. Since thread blocks will be accessing different neighborhoods, the limited-sized constant memory will unlikely be able to hold all the atoms that are needed by all active thread blocks. This motivates the use of global memory to hold a much larger set of atoms. To mitigate the bandwidth consumption, threads in a block collaborate in loading the atom information in each common neighborhood bin into the shared memory. All threads then examine the atoms out of shared memory.

One subtle issue with binning is that bins may end up with different numbers of atoms. Since the atoms are statistically distributed in the grid system, some bins may have lots of atoms, and some bins may end up with no atoms at all. To guarantee memory coalescing, it is important that all bins be of the same size and aligned at appropriate coalescing boundaries. This would require us to fill many bins with dummy atoms whose electrical charge is 0, which causes two negative effects. First, the dummy atoms still occupy global memory and shared memory storage. They also consume data transfer bandwidth to the device. Second, the dummy atoms extend the execution time of the thread blocks whose bins have few real atoms.

A good approach is setting the bin size at a reasonable level that covers the number of atoms in the vast majority of the bins, typically much smaller than the largest possible number of atoms in a bin. The binning process maintains an overflow list. In processing an atom, if the atom's home bin is full, the atom is added to the overflow list instead. After the device completes a kernel, the resulting grid point energy values are transferred back to the host. The host executes a sequential cutoff algorithm on the atoms in the overflow list to complete the missing contributions from these overflow atoms.

As long as the overflow atoms account for only a small percentage (e.g., less than 3%) of the atoms, the additional sequential processing time of the overflow atoms is typically shorter than that of the device execution time. One can also design the kernel so that each kernel invocation calculates the energy values for a subvolume of grid points. After each kernel completes, the host launches the next kernel and processes the overflow atoms for the completed kernel. Thus the host will be processing the overflow atoms while the device executes the next kernel. This approach can hide most, if not all, of the delays in processing overflow atoms, since it is done in parallel with the execution of the next kernel.

18.6 Summary

This chapter presents a series of decisions and tradeoffs in parallelizing the calculation of electrostatic potential energy in a regularly spaced energy grid. We show that parallelizing a highly optimized sequential C implementation of the DCS method leads to a slow scatter kernel that requires heavy use of atomic operations. We then show that one can parallelize a less optimized sequential C code into a gather kernel that has much higher parallel execution speed. We also show that through thread coarsening, we can reclaim much of the efficiency of the optimized sequential execution. We further demonstrate that by carefully choosing the grid points to fold into each thread, the kernel can have completely coalesced memory write patterns.

While DCS is a highly accurate method for calculating the electrostatic potential energy map of a molecular system, it is not a scalable method. The number of

operations that are performed by the method grows proportionally with the number of atoms and the number of grid points. When we increase the physical volume of the molecular system to be simulated, we should expect both the number of grid points and the number of atoms to increase proportionally to the physical size. As a result, the number of operations to be performed will be approximately proportional to the square of the physical volume; that is, the number of operations to be performed will grow quadratically with the volume of the system being simulated. This makes the use of DCS method not suitable for simulating realistic biological systems. We show that by slightly reducing the accuracy, a cutoff summation method implemented with the binning technique can dramatically improve the complexity of the computation while retaining a high level of parallelism.

Exercises

1. Complete the host code for configuring the grid and calling the kernel in Fig. 18.6 with all the execution configuration parameters.
2. Compare the number of operations (memory loads, floating-point arithmetic, branches) executed in each iteration of the kernel in Fig. 18.8 with that in Fig. 18.6 for a coarsening factor of 8. Keep in mind that each iteration of the former corresponds to eight iterations of the latter.
3. Give two potential disadvantages associated with increasing the amount of work done in each CUDA thread, as shown in Section 18.3.
4. Use Fig. 18.13 to explain how control divergence can arise when threads in a block process a bin in the neighborhood list.

References

Humphrey, W., Dalke, A., Schulten, K., 1996. VMD – visual molecular dyanmics. J. Mole. Graph. 14, 33–38.

Rodrigues, C.I., Hardy, D.J., Stone, J.E., Schulten, K., Hwu, W.M., 2008. GPU acceleration of cutoff pair potentials for molecular modeling applications. In *Proceedings of the Fifth Conference on Computing Frontiers* (pp. 273–282).

Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G., Schulten, K., 2007. Accelerating molecular modeling applications with graphics processors. J. Comput. Chem. 28, 2618–2640.

CHAPTER

Parallel programming and computational thinking 19

Chapter Outline

We have so far concentrated on practical knowledge in parallel programming, which consists of CUDA programming interface features, the GPU architecture, performance optimization techniques, parallel patterns, and application case studies. In this chapter we switch our discussions to concepts that are more abstract. We generalize parallel programming into a computational thinking process of designing or selecting parallel algorithms and decomposing a domain problem into well-defined and coordinated work units that can each be efficiently performed by the selected algorithms. A programmer with strong computational thinking skills not only analyzes but also transforms the structure of a domain problem: which parts are inherently serial, which parts are amenable to high-performance parallel execution, and the domain-specific tradeoffs that are involved in moving parts from the former category to the latter. With good algorithm selection and problem decomposition the programmer can achieve an appropriate compromise between parallelism, work efficiency, and resource consumption. A strong combination of domain knowledge and computational thinking skills is often needed for creating successful computational solutions to challenging domain problems. This chapter will give the reader more insight into parallel programming and computational thinking in general.

19.1 Goals of parallel computing

Before we discuss the fundamental concepts of parallel programming, it is important for us to first review the three main reasons why people pursue parallel computing. The first goal is to solve a given problem in less time. For example, an

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00002-1

investment firm may need to run a financial portfolio scenario risk analysis package on all its portfolios during after-trading hours. Such an analysis could require 200 hours on a sequential computer. However, the portfolio management process may require that analysis be completed in 4 hours to be in time for major decisions based on the resulting information. Using parallel computing may speed up the analysis and allow it to complete within the required time window.

The second goal of using parallel computing is to solve bigger problems within a given amount of time. In our financial portfolio analysis example the investment firm may be able to run the portfolio scenario risk analysis on its current portfolio within a given time window using sequential computing. However, the firm is planning on expanding the number of holdings in its portfolio. The enlarged problem size would cause the sequential analysis to exceed the allowed time window. Parallel computing that reduces the running time of the bigger problem size can help to accommodate the planned expansion to the portfolio.

The third goal of using parallel computing is to achieve better solutions for a given problem and a given amount of time. The investment firm may have been using an approximate model in its portfolio scenario risk analysis. Using a more accurate model may increase the computational complexity and increase the running time on a sequential computer beyond the allowed window. For example, a more accurate model may require consideration of interactions between more types of risk factors using a more numerically complex formula. Parallel computing that reduces the running time of the more accurate model may complete the analysis within the allowed time window.

In practice, parallel computing may be driven by a combination of these three goals.

It should be clear from our discussion that parallel computing is motivated primarily by increased speed. The first goal is achieved by increased speed in running the existing model on the current problem size. The second goal is achieved by increased speed in running the existing model on a larger problem size. The third goal is achieved by increased speed in running a more complex model on the current problem size. Obviously, the increased speed through parallel computing can be used to achieve a combination of these goals. For example, parallel computing can reduce the runtime of a more complex model on a larger problem size.

It should also be clear from our discussion that applications that are good candidates for parallel computing typically involve large problem sizes and high complexity. That is, these applications process a large amount of data, require much computation in each iteration, and/or perform many iterations on the data. Applications that do not involve large problem sizes or incur high modeling complexity tend to complete within a small amount of time and do not offer much motivation for increased speed.

A real application often consists of multiple modules that work together. Fig. 19.1 shows an overview of the major modules of a molecular dynamics application. For each atom in the system the application needs to calculate the various forms of forces, such as vibrational, rotational, and nonbonded, that are

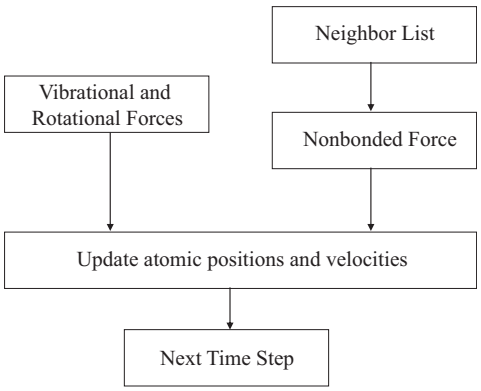


FIGURE 19.1

Major tasks of a molecular dynamics application.

exerted on the atom. Each form of force is calculated with a different method. At the high level, a programmer needs to decide how the work is organized. Note that the amount of work can vary dramatically between these modules. The nonbonded force calculation typically involves interactions among many atoms and incurs many more calculations than the vibrational and rotational forces. Therefore these modules tend to be realized as separate passes over the force data structure.

The programmer needs to decide whether each pass is worth implementing in a CUDA device. For example, the programmer may decide that the vibrational and rotational force calculations do not involve sufficient amount of work to warrant execution on a GPU device. Such a decision would lead to a CUDA program that launches a kernel that calculates nonbonding force fields for all the grid points while continuing to calculate the vibrational and rotational forces for the grid points on the host. The module that updates atom positions and velocities may also run on the host. It first combines the vibrational and rotational forces from the host and the nonbonding forces from the device. It then uses the combined forces to calculate the new atomic positions and velocities.

The portion of work done by the device will ultimately decide the application-level speedup that is achieved by parallelization. For example, assume that the nonbonding force calculation accounts for 95% of the original sequential execution time and it is accelerated by 100× using a GPU. Further assume that the rest of the application remains on the host and receives no speedup. The application-level speedup is $1/(5\% + 95\%/100) = 1/(5\% + 0.95\%) = 1/(5.95\%) = 17\times$. In the case in which the execution of the host and the CUDA device can be overlapped, the execution time of the parallel section is completely hidden in the host execution time. The application-level speedup would be $1/(5\%) = 20\times$. This is a demonstration of Amdahl's law: The application speedup due to parallel computing is limited by the sequential portion of the application. In this case, even though the

sequential portion of the application is quite small (5%), it limits the application-level speedup to 20× despite the nonbonding force calculation having a speedup of 100× and being completely hidden in the shadow of the host execution of the sequential portion. This example illustrates a major challenge in accelerating large applications: The accumulated execution time of small activities that are not worth parallel execution on a CUDA device can become a limiting factor in the speedup that is seen by the end users. We saw this phenomenon in the iterative MRI application study in Chapter 17, Iterative Magnetic Resonance Imaging Reconstruction, in which the CG computation becomes a limiting factor of speedup even though it accounts for only about 1% of the execution time of the original application.

Amdahl's law often motivates task-level parallelization. Although some of these smaller activities do not warrant fine-grained massive parallel execution, it may be desirable to execute some of these activities in parallel with each other when the dataset is large enough. This could be achieved by using a multicore host to execute such tasks in parallel. Alternatively, we could try to simultaneously execute multiple small kernels, each corresponding to one task. CUDA devices support task parallelism with streams, which will be discussed in Chapter 20, Programming a Heterogeneous Computing Cluster.

An alternative approach to reducing the effect of sequential tasks is to exploit data parallelism in a hierarchical manner. For example, in a Message Passing Interface (MPI, 2009) implementation a molecular dynamics application would typically distribute large chunks of the spatial grids and their associated atoms to nodes of a networked computing cluster. By using the host of each node to calculate the vibrational and rotational force for its chunk of atoms, we can take advantage of multiple host CPUs and achieve speedup for these lesser modules. Each node can use a CUDA device to calculate the nonbonding force at a higher level of speedup. The nodes will need to exchange data to accommodate forces that go across chunks and atoms that move across chunk boundaries. We will discuss more details of joint MPI-CUDA programming in Chapter 20, Programming a Heterogeneous Computing Cluster. The main point here is that MPI and CUDA can be used in a complementary, hierarchical way in applications to jointly achieve a higher level of speed with large datasets.

The process of parallel programming can typically be divided into three steps: algorithm selection, problem decomposition, and performance optimization and tuning. The last step was the focus of previous chapters and received general treatment in Chapter 6, Performance Considerations. In the next two sections of this chapter we will discuss the first two steps with generality as well as depth.

19.2 Algorithm selection

An algorithm is a step-by-step procedure in which each step is precisely stated and can be carried out by a computer. An algorithm must exhibit three essential

properties: definiteness, effective computability, and finiteness. Definiteness refers to the notion that each step is precisely stated; there is no room for ambiguity as to what is to be done. Effective computability refers to the fact that each step can be carried out by a computer. Finiteness means that the algorithm must be guaranteed to terminate.

Given a problem, one can typically come up with multiple algorithms to solve the problem. Some require fewer steps of computation than others (i.e., have lower algorithmic complexity), some expose a higher degree of parallel execution than others, some are more generally applicable than others, and some have better accuracy or numerical stability than others. Unfortunately, there is often not a single algorithm that is better than others in all these aspects. A parallel programmer often needs to select an algorithm that achieves the best compromise for a given hardware system.

We have seen examples of assessing the tradeoffs between different algorithms for a computation in several chapters throughout this book. For the prefix sum computation in Chapter 11, Prefix Sum (Scan), we compared two different algorithms for performing parallel prefix sum, namely, the Kogge-Stone algorithm and the Brent-Kung algorithm. Our analysis showed that the Brent-Kung algorithm has lower algorithmic complexity. It requires fewer operations to perform the same computation, which makes it more work efficient. However, we also showed that the Kogge-Stone algorithm exposes more parallelism than the Brent-Kung algorithm, allowing it to complete in fewer iterations. This tradeoff between algorithmic complexity and the amount of parallelism exposed by an algorithm is a classic tradeoff that is encountered by parallel programmers. The best algorithm usually depends on the characteristics of the parallel hardware being targeted as well as the extent to which the higher complexity of an algorithm can be mitigated with hybrid approaches that combine two parallel algorithms or combine a parallel algorithm with a lower-complexity sequential algorithm via thread coarsening.

For the sorting pattern in Chapter 13, Sorting, we compared two different algorithms for performing parallel sorting, namely, radix sort and merge sort. Radix sort can achieve lower algorithmic complexity than merge sort because it is a noncomparison sorting algorithm. It is also highly amenable to parallelization. However, radix sort is not generally applicable because it can be used only with certain types of keys. Merge sort, which is a comparison-based sort, is more generally applicable and can be used with any type of key that has a well-defined comparison operator. The tradeoff between generality and parallel execution efficiency is yet another tradeoff that parallel programmers encounter when selecting a parallel algorithm.

For the electrostatic potential map computation in Chapter 18, Electrostatic Potential Map, we compared two different algorithms for performing the computation, namely, direct Coulomb summation (DCS) and cutoff summation (Rodrigues et al., 2008). These two approaches expose the same amount of parallelism and have the same level of generality; however, they present the classic tradeoff

between algorithmic complexity and accuracy. In Chapter 18, Electrostatic Potential Map, we showed that the cutoff summation algorithm can significantly improve the execution efficiency of grid computation by sacrificing a small amount of accuracy. The challenge that is addressed by this technique is that the amount of computation performed by fully accurate methods such as DCS increases with the square of the volume. For large volume systems, such increase makes the computation excessively long even for massively parallel devices. The cutoff summation method is based on the observation that many grid calculation problems are based on physical laws in which numerical contributions from particles or samples that are far away from a grid point are tiny and can be collectively treated with an implicit method at much lower algorithmic complexity. Although this tradeoff between algorithmic complexity and accuracy is not unique to parallel programming and is also encountered with sequential implementations, it may present additional challenges to parallel programmers. We show an example of these additional challenges for the cutoff summation algorithm in the rest of this section.

In sequential computing, a simple cutoff algorithm handles one atom at a time. For each atom the algorithm iterates through the grid points that fall within a radius of the atom's coordinate. This is a straightforward procedure, since the grid points are in an array that can be easily indexed as a function of their coordinates. However, this simple procedure does not carry easily to parallel execution. The reason is that the atom-centric decomposition does not work well, owing to its scatter memory access behavior. Therefore we need to use a cutoff binning algorithm based on the grid-centric decomposition: Each thread calculates the energy value at one grid point. The key idea of the algorithm is to first sort the input atoms into bins according to their coordinates. Each bin corresponds to a box in the grid space and contains all atoms whose coordinate falls into the box. We define a "neighborhood" of bins for a grid point to be the collection of bins that contain all the atoms that can contribute to the energy value of a grid point. We described an efficient way of managing neighborhood bins for all grid points in Chapter 18, Electrostatic Potential Map. In this algorithm, all blocks iterate through their own neighborhood bins, while all threads in a block scan through the atoms in these neighborhood bins together but make individual decisions on whether each atom falls within its cutoff radius.

One subtle issue with binning is that bins may end up with different numbers of atoms. Some bins may have many atoms, and some bins may have no atom at all. A well-known solution is to set the bin size at a reasonable level, typically much smaller than the largest possible number of atoms in a bin. The binning process maintains an overflow list. When atoms are sorted into bins, if an atom's home bin is full, the atom is added to the overflow list instead. After the device has completed executing a cutoff summation kernel for calculating an electrostatic potential map, atoms in the overflow list need to be processed to complete the missing contributions.

Fig. 19.2 shows a comparison of scalability and performance of the various electrostatic potential map algorithms and implementations. Note that the CPU-SSE3 curve is based on a sequential cutoff summation algorithm. For a map with

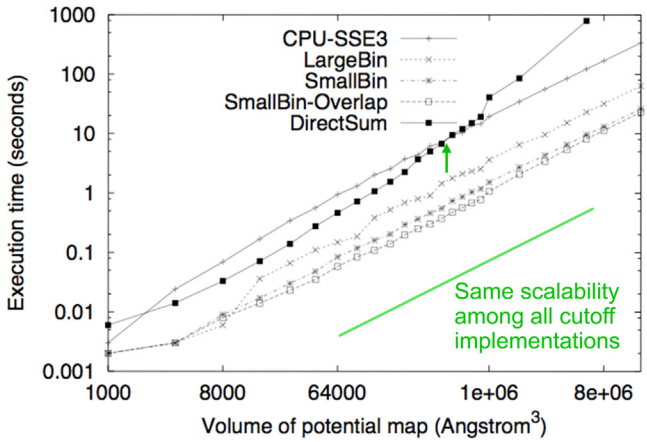


FIGURE 19.2

Scalability and performance of DCS versus cutoff binning.

small volumes, around 1000 $Å^3$, the host (CPU with SSE) executes faster than the DCS kernel, as shown in Fig. 19.2. This is because there is not enough work to fully utilize a CUDA device for such a small volume. However, for moderate volumes, between 2000 and 500,000 $Å^3$, the direct summation kernel performs significantly better than the host, owing to its massive parallelism. However, as we anticipated, the direct summation kernel scales poorly when the volume size reaches about 1,000,000 $Å^3$ and runs longer than the sequential algorithm on the CPU! This is because the algorithmic complexity of the DCS kernel is higher than the sequential cutoff algorithm and thus the amount of work done by kernel grows much faster than that done by the sequential algorithm. For volume sizes larger than 1,000,000 $Å^3$ the amount of work is so large that it swamps the hardware execution resources.

Fig. 19.2 also shows the running time of three binning-based implementations of the cutoff summation algorithm. The SmallBin implementation corresponds to the neighborhood bin list approach discussed in Chapter 18, Electrostatic Potential Map, and allows the blocks running the same kernel to process different neighborhoods of atoms. The SmallBin implementation does incur more instruction overhead for loading atoms from global memory into shared memory. For a moderate volume there is a limited number of atoms in the entire system. The ability to examine a smaller number of atoms does not provide sufficient advantage to overcome the additional instruction overhead. The SmallBin-Overlap implementation overlaps the sequential overflow atom processing with the next kernel execution. It provides a slight but noticeable improvement in running time over the SmallBin implementation. The SmallBin-Overlap implementation achieves a 17× speedup over an efficiently implemented sequential CPU-SSE cutoff summation implementation and maintains the same scalability for large volumes.

19.3 Problem decomposition

After an appropriate algorithm has been selected, for a problem to be solved with parallel computing, the problem must be formulated in such a way that it can be decomposed into subproblems that can be safely solved at the same time. Under such formulation and decomposition the programmer writes code and organizes data to solve these subproblems concurrently. Finding parallelism in large computational problems is often conceptually simple but can be challenging in practice. The key is to identify the work to be performed by each unit of parallel execution so that the inherent parallelism of the problem is well utilized.

The two most common strategies for decomposing a problem for parallel execution are the output-centric and input-centric decompositions. As the names imply, an *output-centric* decomposition assigns threads to process different units of the output data in parallel, whereas an *input-centric* decomposition assigns threads to process different units of the input data in parallel. These decomposition strategies are illustrated in Fig. 19.3. While both decomposition strategies lead to the same execution results, they can exhibit very different performance in a given hardware system. The output-centric decomposition usually exhibits the gather memory access behavior, in which each thread gathers or collects the effect of input values into an output value. Fig. 19.3A illustrates the gather access behavior. Gather-based access patterns are usually more desirable in CUDA devices because the threads can accumulate their results in their private registers. Also, multiple threads can share input values and can effectively use constant memory caching or shared memory to conserve global memory bandwidth.

The input-centric decomposition, by contrast, usually exhibits the scatter memory access behavior, in which each thread scatters or distributes the effect of an input value into the output values. The scatter behavior is illustrated in Fig. 19.3B. Scatter-based access patterns are usually undesirable in CUDA devices because multiple threads can update the same grid point at the same time. The grid points must be stored in a memory that can be written by all the threads involved. Atomic operations must be used to prevent race conditions and loss of values during simultaneous writes to an output value by multiple threads. These atomic operations are significantly slower than the register accesses that are used in the output-centric decomposition.

However, aside from the gather versus scatter access patterns, other considerations may go into deciding whether an output-centric or input-centric decomposition (or another decomposition) is more suitable for a particular application. These considerations include the amount of parallelism that is exposed by the decomposition, the ease of identifying which input data contributes to which output data, the load balance induced by the decomposition, and others, depending on the application.

The distinction between input-centric and output-centric decompositions was most evident in Chapters 17, Iterative Magnetic Resonance Imaging Reconstruction, and 18, Electrostatic Potential Map, where the two strategies were both implemented and

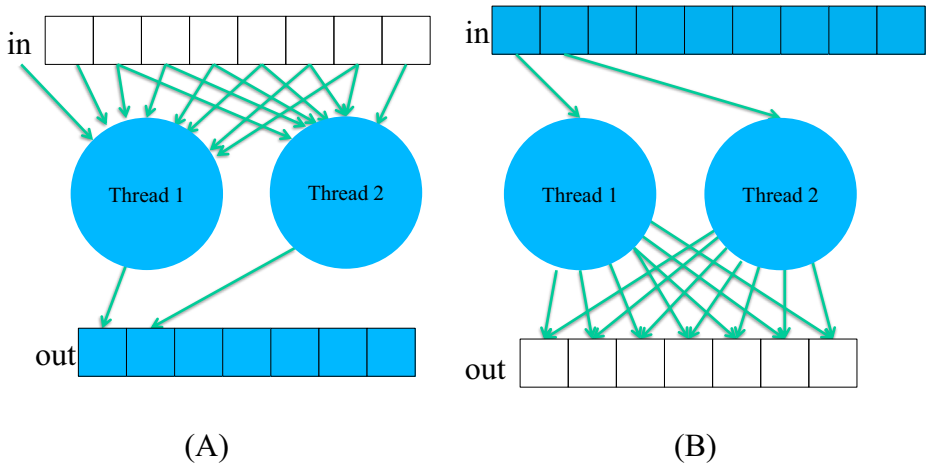


FIGURE 19.3

Problem decomposition strategies. (A) Output-centric. (B) Input-centric.

explicitly compared. However, problem decomposition is an implicit design decision that has been made throughout many computations discussed in this book. We revisit these computations to highlight the problem decomposition that was chosen in each case and why it is advantageous over alternative decompositions.

The image processing (Chapter 3, Multidimensional Grids and Data), matrix multiplication (Chapters 3, 5, and 6), Convolution (Chapter 7), and Stencil (Chapter 8) computations were all parallelized by using an output-centric decomposition. That is, the threads in these computations are assigned to output elements (image pixels, matrix entries, or grid points) and iterate over the input elements that contribute to them. Alternatively, an input-centric decomposition would assign threads to the input elements and have each thread iterate over the output elements to which its input element contributes and make an update using atomic operations. The clear advantage of the output-centric decomposition over the input-centric decomposition in these cases is avoiding atomics by using a gather access pattern instead of a scatter access pattern. On the other hand, none of the other considerations make the input-centric decomposition favorable. There are enough output elements to expose a high degree of parallelism, identifying which input elements are needed by each output element is straightforward, and all output elements require the same amount of work to be computed, so there is no load imbalance.

The histogram computation (Chapter 9, Parallel Histogram) was parallelized by using an input-centric decomposition. That is, each thread is assigned to an input element (or chunk of elements) and updates the output bins on the basis of the input value(s). Since multiple threads may update the same output bin, atomic operations are needed (scatter access pattern). Alternatively, an output-centric decomposition would assign threads to output bins and have each thread search for the inputs that map to the bin and update the bin accordingly. This decomposition would eliminate the atomic operations because each bin would be updated by a single thread (gather access pattern). However, the output-centric decomposition would create many other problems. First, the number of output bins is typically much smaller than the number of input values, so parallelism is substantially reduced. Second, a thread cannot know which input values map to its output bin without inspecting those input values, so each thread will need to iterate over every input value, which is not work efficient. Third, even if threads had some way of quickly identifying which input values mapped to their output bins, each bin would have a different number of input values mapping to it, which would create load imbalance across threads. All these considerations make the input-centric decomposition more favorable for the histogram computation.

The merge operation (Chapter 12, Merge) was parallelized by using an output-centric decomposition; that is, each thread is assigned to an element (or chunk of elements) in the output array and performs a binary search to find the corresponding input element(s) and merge them. While many other computations that favor the output-centric decomposition do so because of the benefit of gathering over scattering, this consideration is not relevant for the merge operation. Every output element in the merge operation is contributed to by

only one input element, so an input-centric merge operation would not need to use atomics. However, an input-centric merge operation either would make it more expensive to find the mapping between the input element(s) for which a thread is responsible and the corresponding output element(s) or would incur high load imbalance by having each input thread be responsible for a different number of input elements. For this reason, the output-centric decomposition is preferred.

For sparse matrix computations (Chapter 14, Sparse Matrix Computation) the SpMV/COO kernel used input-centric decomposition, in which threads were assigned to nonzeros in the input matrix and atomically updated the corresponding element in the output vector (scatter access pattern). The remaining kernels used output-centric decomposition, in which threads were assigned to output vector elements and iterated over input nonzeros (gather access pattern). Although the input-centric SpMV/COO kernel performed atomics, it had other advantages over the output-centric kernels, such as extracting more parallelism and having better load balance. Ultimately, the best decomposition for this computation depends on the input dataset. Moreover, the hybrid ELL-COO format showcases an example in which a hybrid output-centric and input-centric decomposition may be beneficial.

For graph traversal (Chapter 15, Graph Traversal) the vertex-centric push implementation and the edge-centric implementation used input-centric decomposition. That is, threads were assigned to vertices or edges and updated the levels of neighboring vertices. In contrast, the vertex-centric pull implementation used output-centric decomposition in which each thread updated only the level of the vertex to which it was assigned. The tradeoff between scatter and gather access patterns is not a concern in selecting between different decompositions because the updates to the level values are idempotent and do not require atomics. However, the amount of parallelism that is extracted and the load balance that is achieved play an important role in deciding which decomposition is more favorable, and the best decomposition ultimately depends on the input dataset. The reader is encouraged to review Chapter 15, Graph Traversal, for a more detailed handling of this topic.

The iterative MRI reconstruction problem (Chapter 17, Iterative Magnetic Resonance Imaging Reconstruction) and the electrostatic potential calculation problem (Chapter 18, Electrostatic Potential Map) both favor the output-centric decomposition because of the benefit over the gather pattern of the scatter pattern in avoiding atomic operations. These chapters already handle the distinction between the two decomposition strategies in depth. The iterative MRI reconstruction problem (Chapter 17, Iterative Magnetic Resonance Imaging Reconstruction) processes a large amount of k-space sample data. Each k-space sample data is also used many times for calculating its contributions to the reconstructed voxel data. For high-resolution reconstruction, each sample data is used a large number of times. We showed that a good decomposition of the FhD problem in MRI reconstruction is an output-centric decomposition that forms subproblems, each of which calculates the value of an FhD element using the gather strategy.

Similarly, the electrostatic potential calculation problem (Chapter 18, Electrostatic Potential Map) involves the calculation of the contribution of many

input atoms to the potential energy at a large number of output grid points. A realistic molecular system model typically involves at least hundreds of thousands of atoms and millions of energy grid points. The electrostatic charge information of each atom is used many times in calculating its contributions to the energy grid points. We showed that the decomposition of the electrostatic potential map calculation problem can be atom-centric (i.e., input-centric) or grid-centric (i.e., output-centric). In an atom-centric decomposition, each thread is responsible for calculating the effect of one atom on all grid points. In contrast, a grid-centric decomposition uses each thread to calculate the effect of all atoms on a grid point. The grid-centric (i.e., output-centric) decomposition proved to be better because it uses the more favorable gather access pattern. The amount of parallelism that is exposed by both decompositions is sufficient. For the cutoff summation algorithm the difficulty of mapping input data to output data in the grid-centric decomposition is overcome with the use of binning, and the load imbalance caused by binning is overcome with the SmallBin-Overlap implementation that was discussed in the previous section.

19.4 Computational thinking

Computational thinking is arguably the most important aspect of parallel application development (Wing, 2006). We define computational thinking as the thought processes of formulating domain problems in terms of computation steps and algorithms. Like any other thought processes and problem-solving skills, computational thinking is an art. As we mentioned in Chapter 1, Introduction, we believe that computational thinking is best taught with an iterative approach in which students bounce back and forth between practical experience and abstract concepts.

There is a very large volume of literature on a wide range of algorithms, problem decompositions, and optimization strategies that can be hard to understand. It is beyond the scope of this book to provide comprehensive coverage of all the available techniques. We discuss a substantial set of techniques in each of the algorithm, problem decomposition, and optimization steps that have broad applicability. While these techniques are demonstrated using CUDA C implementations, they help the readers build up the foundation for computational thinking in general. We believe that humans understand best when we learn from the bottom up. That is, we first learn the concepts in the context of a particular programming model, which provides us with solid footing before we generalize our knowledge to other programming models. An in-depth experience with the CUDA C implementations also enables us to gain maturity, which will help us learn parallel programming and computational thinking concepts that may not even be pertinent to the GPUs.

There is a myriad of skills that are needed for a parallel programmer to be an effective computational thinker. We summarize these foundational skills as follows:

- Computer architecture: memory organization, caching and locality, memory bandwidth, SIMT versus SPMD versus SIMD execution, and floating-point precision versus accuracy. These concepts are critical in understanding the tradeoffs between algorithms, problem decompositions, and optimizations.
- Programming interfaces and compilers: parallel execution models, types of available memories, types of synchronization support, array data layout, and thread granularity transformation. These concepts are needed for thinking through the arrangements of data structures and loop structures to achieve better performance.
- Domain knowledge: problem formulation, hard versus soft constraints, numerical methods, precision, accuracy, and numerical stability. Understanding these ground rules allows a developer to be much more creative in applying algorithm techniques.

Our goal for this book is to provide a solid foundation for all these areas. Readers should continue to broaden their knowledge in these areas after finishing this book. Most important, the best way of building up more computational thinking skills is to keep solving challenging problems with excellent computational solutions.

A good goal for effective use of computing is making science better, not just faster. This requires reexamining prior assumptions and really thinking about how to apply the big hammer of massively parallel processing. Put another way, there will probably be no Nobel Prizes or Turing Awards awarded for "just recompile" or using more threads with the same computational approach. Truly important scientific discoveries will more likely come from fresh computational thinking. Consider this an exhortation to use this bonanza of computing power to solve new problems in new ways.

There are three approaches for attacking computation-hungry applications with increasing order of difficulty, complexity, and, not surprisingly, potential for payoff. Let us call these "good," "better," and "best."

The "good" approach is simply to "accelerate" legacy program codes. The most basic effort is simply to recompile and run on a new platform or architecture without adding any domain insight or expertise in parallelism. This can be enhanced by using optimized libraries, tools, or directives, such as CuBLAS, CuFFT, Thrust, Matlab, or OpenACC. This approach does not require any algorithm selection, problem decomposition, or optimization and tuning efforts. It can be immediately rewarding for domain scientists, since minimal computer science knowledge or programming skills are required to obtain decent speedups. However, it does not realize the full potential of parallel computing.

The "better" approach involves rewriting existing codes using parallel programming skills to take advantage of new architectures or creating new codes from scratch. This approach is an opportunity for clever thinking about problem decomposition and optimization selection and is good work for nondomain computer scientists, as minimal domain knowledge is required. However, it also does

not realize the full potential of parallel computing because of the absence of domain-specific knowledge, which is needed for effective algorithm selection.

The "best" approach involves a holistic attempt at application parallelization involving all three key steps: algorithm selection, problem decomposition, and optimization and tuning. We wish not only to map a known algorithm to a parallel program and optimize it, but also to rethink the numerical methods and algorithms that are used in the solution. The cutoff binning approach in Chapter 18, Electrostatic Potential Map, is a good example. The approach requires domain expertise to trade accuracy for dramatically reduced algorithmic complexity and requires problem decomposition and optimization skills to use the grid-centric decomposition with the binning techniques designed by computer scientists. In this approach, there is the potential for the biggest performance advantage and fundamental new discoveries and capabilities. For example, one may be able to perform high-fidelity simulation of a biochemical system whose size is considered beyond reach with traditional methods. This approach is interdisciplinary and requires both computer science *and* domain insight, but the payoff is worth the effort. It is truly an exciting time to be a computational scientist!

19.5 Summary

In summary, we have discussed the main steps of parallel programming and computational thinking, namely, algorithm selection, problem decomposition, and optimization and tuning. Given a computational problem, the programmer will typically have to select from a variety of algorithms. Some of these algorithms achieve different tradeoffs while maintaining the same numerical accuracy. Others involve sacrificing some level of accuracy to achieve much more scalable running times. For the selected algorithm, different choices of problem decomposition can result in different levels of interference between threads, parallelism exposed, load imbalance, and other performance considerations during parallel execution. Computational thinking skills allow an algorithm designer to work around the roadblocks and reach a good solution.

References

Message Passing Interface Forum, MPI—A Message Passing Interface Standard Version 2.2, http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, September 4, 2009.

Rodrigues, C.I., Stone, J., Hardy, D., Hwu, W.W. 2008. GPU acceleration of cutoff-based potential summation. In: ACM Computing Frontier Conference 2008, Italy, May 2008.

Wing, J., 2006. Computational thinking. Commun. ACM 49, 3.

CHAPTER

Programming a heterogeneous computing cluster 20

An introduction to CUDA streams

With special contributions from Isaac Gelado and Javier Cabezas

Chapter Outline

So far, we have focused on programming a heterogeneous computing system with one host and one device. In high-performance computing (HPC), applications require the aggregate computing power of a cluster of computing nodes. Many of the HPC clusters today have one or more hosts and one or more devices in each node. Historically, these clusters have been programmed predominately with Message Passing Interface (MPI). In this chapter we will present an introduction to joint MPI/CUDA programming. We will present only the MPI concepts that programmers need to understand to scale their heterogeneous applications to multiple nodes in a cluster environment. In particular, we will focus on domain partitioning, point-to-point communication, and collective communication in the context of scaling a CUDA kernel into multiple nodes.

20.1 Background

Although practically no top supercomputers used GPUs before 2009, the need for better energy efficiency has led to fast adoption of GPUs in recent years. Many of

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00007-0

the top supercomputers in the world today use both CPUs and GPUs in each node. The effectiveness of this approach is validated by their high rankings in the Green500 list, which reflects their high energy efficiency.

The dominating programming interface for computing clusters today is MPI (Gropp et al., 1999), which is a set of API functions for communication between processes running in a computing cluster. MPI assumes a distributed memory model in which processes exchange information by sending messages to each other. When an application uses API communication functions, it does not need to deal with the details of the interconnect network. The MPI implementation allows the processes to address each other using logical numbers, in much the same way as using phone numbers in a telephone system: Telephone users can dial each other using phone numbers without knowing exactly where the called person is and how the call is routed.

In a typical MPI application, data and work are partitioned among processes. As is shown in Fig. 20.1, each node can contain one or more processes, shown as clouds within nodes. As these processes progress, they may need data from each other. This need is satisfied by sending and receiving messages. In some cases, the processes also need to synchronize with each other and generate collective results when collaborating on a large task. This is done with collective communication API functions.

20.2 A running example

As a running example we will use a three-dimensional (3D) stencil computation that was introduced in Chapter 8, Stencil. We assume that the computation calculates heat transfer based on a finite difference method for solving a partial differential equation that describes the physical laws of heat transfer. In particular, we will use the Jacobi iterative method, in which in each iteration or time step, the value of a grid point is calculated as a weighted sum of neighbors (north, east, south, west, up, down) and its own value from the previous time step. To achieve high numerical stability, multiple indirect neighbors in each direction are also used in the computation of a grid point. This is a *higher-order stencil*

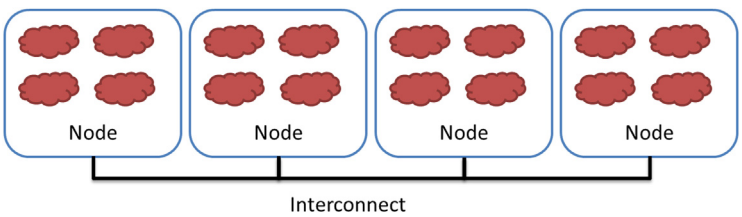


FIGURE 20.1

Programmer's view of MPI processes. *MPI*, Message Passing Interface.

computation, as we discussed in Chapter 8, Stencil. For the purpose of this chapter we assume that four points in each direction will be used.

As is shown in Fig. 20.2, there are a total of 24 neighbor points for calculating the next step value of a grid point. Each point in the grid has x, y, and z coordinates. For a grid point where the coordinate value is $x = i$, $y = j$, and $z = k$, or (i, j, k), its 24 neighbors are ($i - 4$, j, k), ($i - 3$, j, k), ($i - 2$, j, k), ($i - 1$, j, k), ($i + 1$, j, k), ($i + 2$, j, k), ($i + 3$, j, k), ($i + 4$, j, k), (i, $j - 4$, k), (i, $j - 3$, k), (i, $j - 2$, k), (i, $j - 1$, k), (i, $j + 1$, k), (i, $j + 2$, k), (i, $j + 3$, k), (i, $j + 4$, k), (i, j, $k - 4$), (i, j, $k - 3$), (i, j, $k - 2$), (i, j, $k - 1$), (i, j, $k + 1$), (i, j, $k + 2$), (i, j, $k + 3$) and (i, j, $k + 4$). Since the data value of each grid point for the next time step is calculated on the basis of the current data values of 25 points (24 neighbors and itself), this is a 25-point stencil computation.

We assume that the system is modeled as a structured grid in which spacing between grid points is constant within each direction. This allows us to use a 3D array in which each element stores the state of a grid point, as we discussed in Chapter 8, Stencil. The physical distance between adjacent elements in each dimension can be represented by a grid spacing variable. Note that this grid data structure is similar to that used in the electrostatic potential calculation in Chapter 18, Electrostatic Potential Map. Fig. 20.3 illustrates a 3D array that represents a rectangular ventilation duct, with x and y dimensions as the cross sections of the duct and the z dimension the direction of the heat flow along the duct.

We assume that the data is placed in the memory space in the row-major layout, where x is the lowest dimension, y is the next, and z is the highest. That is, all elements with $y = 0$ and $z = 0$ will be placed in consecutive memory locations according to their x coordinate. Fig. 20.4 shows a small example of the

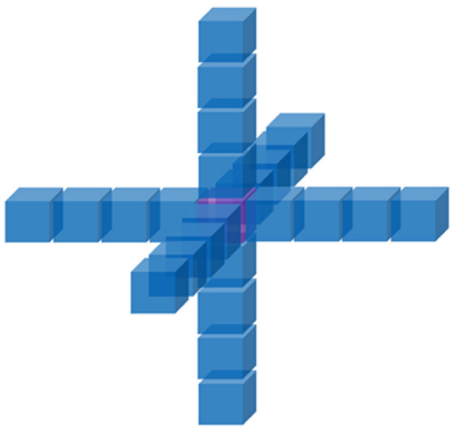


FIGURE 20.2

A 25-point stencil computation example, with four neighbors in each of the *x*, *y*, and *z* directions.

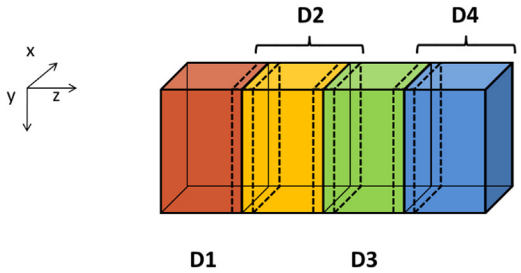


FIGURE 20.3

3D Grid array for the modeling heat transfer in a duct.

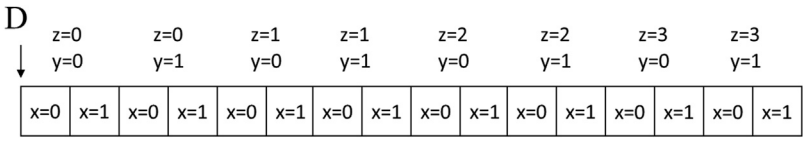


FIGURE 20.4

A small example of memory layout for the 3D grid.

grid data layout. This small example has only 16 data elements in the grid: two elements in the x dimension, two in the y dimension, and four in the z dimension. Both x elements with $y = 0$ and $z = 0$ are placed in memory first. They are followed by all elements with $y = 1$ and $z = 0$. The next group will be elements with $y = 0$ and $z = 1$.

When one uses a computing cluster, it is common to divide the input data into several partitions, called domain partitions, and assign each partition to a node in the cluster. In Fig. 20.3 we show that the 3D array is divided into four domain partitions: D0, D1, D2, and D3. Each of the partitions will be assigned to an MPI compute process.

The domain partitions can be further illustrated with Fig. 20.4. The first section, or slice, of four elements ($z = 0$) is in the first partition; the second section ($z = 1$) is in the second partition; the third section ($z = 2$) is in the third partition; and the fourth section ($z = 3$) is in the fourth partition. This is obviously a toy example. In a real application there are typically hundreds or even thousands of elements in each dimension. For the rest of this chapter it is useful to remember that all elements in a z slice are in consecutive memory locations.

20.3 Message passing interface basics

Like CUDA, MPI programs are based on the SPMD parallel programming model. All MPI processes execute the same program. The MPI system provides a set of

API functions to establish communication systems that allow the processes to communicate with each other. Fig. 20.5 shows five essential MPI functions that set up and tear down the communication system for an MPI application.

We will use a simple MPI program, shown in Fig. 20.6, to illustrate the usage the API functions. To launch an MPI application in a cluster, a user needs to supply the executable file of the program to the *mpirun* command or the *mpiexec* command in the login node of the cluster. Each process starts by initializing the MPI runtime with an `MPI_Init()` call (line 05). This initializes the communication system for all the processes that are running the application. Once the MPI runtime has been initialized, each process calls two functions to prepare for communication. The first function is `MPI_Comm_rank()` (line 06), which returns a unique number to each calling process, which is called the *MPI rank* or process

- `int MPI_Init(int *argc, char ***argv)`
  - Initialize MPI
- `int MPI_Comm_rank (MPI_Comm comm, int *rank)`
  - Rank of the calling process in group of comm
- `int MPI_Comm_size (MPI_Comm comm, int *size)`
  - Number of processes in the group of comm
- `int MPI_Comm_abort (MPI_Comm comm)`
  - Terminate MPI comminication connection with an error flag
- `int MPI_Finalize ()`
  - Ending an MPI application, close all resources

FIGURE 20.5

Basic MPI functions for establishing and closing a communication system.

```
01  #include "mpi.h"
02  int main(int argc, char *argv[]) {
03     int pad = 0, dimx  = 480+pad, dimy  = 480, dimz  = 400, nreps = 100;
04     int pid=-1, np=-1;
05     MPI_Init(&argc, &argv);
06     MPI Comm rank(MPI COMM WORLD, &pid);
07     MPI_Comm_size(MPI_COMM_WORLD, &np);
08     if(np < 3) {
09        if(0 == pid) printf("Needed 3 or more processes.\n");
10        MPI Abort( MPI COMM WORLD, 1 ); return 1;
11     }
12     if(pid < np - 1)
13        compute_process(dimx, dimy, dimz / (np - 1), nreps);
14     else
15        data server( dimx,dimy,dimz );
16     MPI_Finalize();
17     return 0;
18   }
```

FIGURE 20.6

A simple MPI main program.

id for the process. The numbers that are received by the processes vary from 0 to the number of processes minus 1. The MPI rank for a process is analogous to the expression `blockIdx.x*blockDim.x+threadIdx.x` for a CUDA thread. It uniquely identifies the process in a communication, which is also equivalent to the phone number in a telephone system. The main differences are that MPI ranks are one-dimensional.

The `MPI_Comm_rank()` function in line 06 of Fig. 20.6 takes two parameters. The first is an MPI built-in type `MPI_Comm` that specifies the scope of the request, that is, the collection of processes that form the group identified by a `MIP_Comm` variable. Each variable of the `MPI_comm` type is commonly referred to as a communicator. `MPI_Comm` and other MPI built-in types are defined in the "mpi.h" header file (line 01), which should be included in all C program files that use MPI. An MPI application can create one or more *communicators*, each of which is a group of MPI processes for the purpose of communication. `MPI_Comm_rank()` assigns a unique id to each process in a communicator. In Fig. 20.6 the parameter value that is passed is `MPI_COMM_WORLD`, which is used as a default and means that the communicator includes all MPI processes that are running the application.[1]

The second parameter of the `MPI_Comm_rank()` function is a pointer to an integer variable into which the function will deposit the returned rank value. In Fig. 20.6 a variable pid is declared for this purpose. After the `MPI_Comm_rank()` has returned, the pid variable will contain the unique id for the calling process.

The second API function is `MPI_Comm_size()` (line 07), which returns the total number of MPI processes running in the communicator. The `MPI_Comm_size()` function takes two parameters. The first one is of MPI_Comm type that gives the scope of the request. In Fig. 20.6 the parameter value that is passed in is `MPI_COMM_WORLD`, which means that the scope of the `MPI_Comm_size()` is all the processes in the application. Since the scope is all MPI processes, the returned value is the total number of MPI processes that are running the application. This value is configured by the user when the application is executed by using the mpirun command or the mpiexec command. However, the user may not have requested a sufficient number of processes. Also, the system may or may not be able to create all the processes that the user requested. Therefore it is a good practice for an MPI application program to check the actual number of processes that are running.

The second parameter is a pointer to an integer variable into which the `MPI_Comm_size()` function will deposit the return value. In Fig. 20.6 a variable np is declared for this purpose. After the function returns, the variable np contains the number of MPI processes that are running the application. In Fig. 20.6 we assume that the application requires at least 3 MPI processes. Therefore it checks

[1] Interested readers should refer to the MPI reference manual (Gropp et al., 1999) for details on creating and using multiple communicators in an application, in particular the definition and use intracommunicators and intercommunicators.

whether the number of processes is at least 3 (line 08). If not, it calls `MPI_Comm_abort()` function to terminate the communication connections and return with an error flag value 1 (line 10).

Fig. 20.6 also shows a common pattern for reporting errors or other chores. There are multiple MPI processes, but we need to report the error only once. The application code designates the process with `pid = 0` to do the reporting (line 09). This is similar to the pattern in CUDA kernels in which some tasks need to be done by only one of the threads in a thread-block.

As is shown in Fig. 20.5, the `MPI_Comm_abort()` function takes two parameters (line 10). The first sets the scope of the request. In Fig. 20.6 the scope is set as `MPI_COMM_WORLD`, which means all MPI processes that are running the application. The second parameter is a code for the type of error that caused the abort. Any number other than 0 indicates that an error has happened.

If the number of processes satisfies the requirement, the application program goes on to perform the calculation. In Fig. 20.6, the application uses np − 1 processes (`pid` from 0 to np − 2) to perform the calculation (lines 12−13) and one process (the last one whose pid is np − 1) to perform I/O service for the other processes (lines 14−15). We will refer to the process that performs the I/O services as the data server and the processes that perform the calculation as compute processes. In Fig. 20.6, if the pid of a process is within the range from 0 to np − 2, it is a compute process and calls the `compute_process()` function (line 13). If the process pid is np − 1, it is the data server and calls the `data_server()` function (line 15). This is similar to the pattern in which threads perform different actions according to their thread ids.

After the application has completed its computation, it notifies the MPI runtime with a call to `MPI_Finalize()`, which frees all MPI communication resources that are allocated to the application (line 16). The application can then exit with a return value 0, which indicates that no error has occurred (line 17).

20.4 Message passing interface point-to-point communication

MPI supports two major types of communication. The first is the point-to-point type, which involves one source process and one destination process. The source process calls the `MPI_Send()` function, and the destination process calls the `MPI_Recv()` function. This is analogous to a caller dialing a call and a receiver answering a call in a telephone system.

Fig. 20.7 shows the syntax for using the `MPI_Send()` function. The first parameter is a pointer to the starting location of the memory area where the data to be sent can be found. The second parameter is an integer that gives that number of data elements to be sent. The third parameter is of the MPI built-in type `MPI_Datatype`. It specifies the type of each data element that is being sent as far

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Buf: starting address of send buffer (pointer)
  - Count: Number of elements in send buffer (nonnegative integer)
  - Datatype: Datatype of each send buffer element (MPI_Datatype)
  - Dest: Rank of destination (integer)
  - Tag: Message tag (integer)
  - Comm: Communicator (handle)

FIGURE 20.7

Syntax for the MPI_Send() function.

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - buf: starting address of receive buffer (pointer)
  - Count: Maximum number of elements in receive buffer (integer)
  - Datatype: Datatype of each receive buffer element (MPI_Datatype)
  - Source: Rank of source (integer)
  - Tag: Message tag (integer)
  - Comm: Communicator (handle)
  - Status: Status object (Status)

FIGURE 20.8

Syntax for the MPI_Recv() function.

as the MPI library implementation is concerned. The values that can be held by a variable or argument of the `MPI_Datatype` are defined in mpi.h and include `MPI_DOUBLE` (double-precision floating-point), `MPI_FLOAT` (single-precision floating-point), `MPI_INT` (integer), and `MPI_CHAR` (character). The exact sizes of these types depend on the size of the corresponding C types in the host processor. See the MPI reference manual for more sophisticated use of MPI types (Gropp et al., 1999).

The fourth parameter for `MPI_Send()` is an integer that gives the MPI rank of the destination process. The fifth parameter gives a tag that can be used to classify the messages that are sent by the same process. The sixth parameter is a communicator that specifies the context in which the destination MPI rank is defined.

Fig. 20.8 shows the syntax for using the `MPI_Recv()` function. The first parameter is a pointer to the area in memory where the received data should be deposited. The second parameter is an integer that gives the maximum number of elements that the `MPI_Recv()` function is allowed to receive. The third parameter is an `MPI_Datatype` that specifies the type of each element to be received. The

fourth parameter is an integer that gives the process id of the source of the message. The fifth parameter is an integer that specifies the tag value that is expected by the destination process. If the destination process does not want to be limited to a particular tag value, it can use `MPI_ANY_TAG`, which means that the receiver is willing to accept messages of any tag value from the source.

We will first use the data server to illustrate the use of point-to-point communication. In a real application the data server process would typically perform data input and output operations for the compute processes. However, input and output have too much system-dependent complexity. Since I/O is not the focus of our discussion, we will avoid the complexity of I/O operations in a cluster environment. That is, instead of reading data from a file system, we will just have the data server initialize the data with random numbers and distribute the data to the compute processes. The first part of the data server code is shown in Fig. 20.9.

The data server function takes four parameters. The first three parameters specify the size of the 3D grid: the number of elements in the *x* dimension, dimx; the number of elements in the *y* dimension, dimy; and the number of elements in

```
01  void data_server(int dimx, int dimy, int dimz, int nreps) {
02     int np;
       /* Set MPI Communication Size */
03     MPI Comm size(MPI COMM WORLD, &np);
04     unsigned int num_comp_nodes = np - 1, first_node = 0, last_node = np - 2;
05     unsigned int num_points = dimx * dimy * dimz;
06     unsigned int num_bytes  = num_points * sizeof(float);
07     float *input=0, *output=0;
       /* Allocate input data */
08     input = (float *)malloc(num_bytes);
09     output = (float *)malloc(num_bytes);
10     if(input == NULL || output == NULL) {
11           printf("server couldn't allocate memory\n");
12           MPI_Abort( MPI_COMM_WORLD, 1 );
13     }
       /* Initialize input data */
14     random_data(input, dimx, dimy ,dimz , 1, 10);
       /* Calculate number of shared points */
15     int edge_num_points = dimx * dimy * ((dimz / num_comp_nodes) + 4);
16     int int_num_points  = dimx * dimy * ((dimz / num_comp_nodes) + 8);
17     float *send_address = input;
       /* Send data to the first compute node */
18     MPI_Send(send_address, edge_num_points, MPI_FLOAT, first_node,
                    0, MPI_COMM_WORLD );
19     send_address += dimx * dimy * ((dimz / num_comp_nodes) - 4);
       /* Send data to "internal" compute nodes */
20     for(int process = 1; process < last node; process++) {
21           MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
                          0, MPI_COMM_WORLD);
22           send_address += dimx * dimy * (dimz / num_comp_nodes);
       }
       /* Send data to the last compute node */
23     MPI_Send(send_address, edge_num_points, MPI_FLOAT, last_node,
                    0, MPI_COMM_WORLD);
```

FIGURE 20.9

Data server process code (part 1).

the z dimension, dimz. The fourth parameter specifies the number of iterations that need to be done for all the data points in the grid.

In Fig. 20.9, line 02 declares variable np that will contain the number of processes running the application. Line 03 calls `MPI_Comm_size()`, which will deposit the number of processes running the application into np. Line 04 declares and initializes several helper variables. The variable `num_comp_procs` contains the number of compute processes. Since we are reserving one process as the data server, there are np − 1 compute processes. The variable `first_proc` gives the process id of the first compute process, which is 0. The variable `last_proc` gives the process id of the last compute process, which is np − 2. That is, line 04 designates the first np − 1 processes, 0 through np − 2, as compute processes. The process with the largest rank serves as the data server. This reflects the design decision, and this decision will also be reflected in the compute process code.

Line 05 declares and initializes the `num_points` variable that gives the total number of grid data points to be processed, which is simply the product of the number of elements in each dimension, or `dimx * dimy * dimz`. Line 06 declares and initializes the `num_bytes` variable, which gives the total number of bytes needed to store all the grid data points. Since each grid data point is a float, this value is `num_points * sizeof(float)`.

Line 07 declares two pointer variables: input and output. These two pointers will point to the input data buffer and the output data buffer. Lines 08 and 09 allocate memory for the input and output buffers and assign their addresses to their respective pointers. Line 10 checks whether the memory allocations were successful. If either of the memory allocation fails, the corresponding pointer will have received a NULL pointer from the `malloc()` function. In this case, the code aborts the application and reports an error (lines 11−12).

Lines 15 and 16 calculate the number of grid point array elements that should be sent to each compute process. As is shown in Fig. 20.3, there are two types of compute processes: edge processes and internal processes. The first process (process 0, which computes D1) and the last process (process 3, which computes D4) compute an edge partition that has neighbors only on one side. Partition D1, which is assigned to process 0, has a neighbor only on the right side (D2). Partition D4, which is assigned to the last compute process, has a neighbor only on the left side (D3). We call the compute processes that compute edge partitions the *edge processes*. Each of the other processes computes an internal partition that has neighbors on both sizes. For example, process 1 computes partition D2, which has a left neighbor (D1) and a right neighbor (D3). We call the processes that compute internal partitions *internal processes*.

Recall that in the Jacobi iterative method, each calculation step for a grid point needs the values of its immediate neighbors from the previous step. This creates a need for halo cells for grid points at the left and right boundaries of a partition, which are shown as slices defined by dashed lines at the edge of each partition in Fig. 20.3. Note that these halo slices are similar to those in the stencil pattern that was presented in Chapter 8, Stencil. Since we are computing a

25-point stencil with four elements in each direction, each process needs to receive four slices of halo cells that contain all neighbors for each side of the boundary grid points of its partition. For example, in Fig. 20.3, partition D2 needs four halo slices from D1 and four halo slices from D3. Note that a halo slice for D2 is a boundary slice for D1 or D3.

Recall that the total number of grid points is `dimx*dimy*dimz`. Since we are partitioning the grid along the z dimension, the number of grid points in each partition should be `dimx*dimy*(dimz/num_comp_procs)`. Recall that we will need four neighbor slices in each direction in order to calculate values within each partition. Therefore the number of grid points that should be sent to each internal process is `dimx*dimy*((dimz/num_comp_procs) + 8)`. As for an edge process, there is only one neighbor. As in the case of convolution, we assume that zero values will be used for the ghost cells and that no input data needs to be sent for them. For example, partition D1 needs only the four neighbor slices from D2 on the right side. Therefore the number of grid points to be sent to an edge process is `dimx*dimy*((dimz/num_comp_procs) + 4)`. That is, each process receives four slices of halo grid points from the neighbor partition on each side.

Line 17 of Fig. 20.9 sets the `send_address` pointer to point to the beginning of the input grid point array. To send the appropriate partition to each process, we will need to add the appropriate offset to this beginning address for each `MPI_Send()`. We will come back to this point later.

We are now ready to complete the code for the data server. Line 18 sends process 0 its partition. Since this is the first partition, its starting address is also the starting address of the entire grid, which was set up in line 17. Process 0 is an edge process, and it does not have a left neighbor. Therefore the number of grid points to be sent is the value `edge_num_points`, that is, `dimx*dimy*((dimz/num_comp_procs) + 4)`. The third parameter specifies that the type of each element is an `MPI_FLOAT` which is C float (single-precision, 4 bytes). The fourth parameter specifies that the value of `first_node`, that is, 0, is the MPI rank of the destination process. The fifth parameter specifies 0 for the MPI tag. This is because we are not using tags to distinguish between messages sent from the data server. The sixth parameter specifies that the communicator to be used for interpreting the sender and receiver rank values for the message should be all MPI processes for the current application.

Line 19 of Fig. 20.9 advances the `send_address` pointer to the beginning of the data partition for process 1. From Fig. 20.3 there are `dimx*dimy*(dimz/num_comp_procs)` elements in partition D1, which means that D2 starts at the location that is `dimx*dimy*(dimz/num_comp_procs)` elements from the starting location of input. Recall that we also need to send the halo cells from D1. Therefore we adjust the starting address for the `MPI_Send()` back by four slices, which results in the expression for advancing the `send_address` pointer in line 19: `dimx*dimy*((dimz/num_comp_procs) - 4)`.

Line 20 is a loop that sends out the MPI messages to the internal processes (process 1 through process np − 3). In our small example for four compute

processes, np is 5, so the loop sends the MPI messages to process 1 and process 2. These internal processes need to receive halo grid points for neighbors on both sides. Therefore the second parameter of the `MPI_Send()` in line 21 uses `int_num_nodes`, that is, `dimx*dimy*((dimz/num_comp_procs) + 8)`. The rest of the parameters are similar to that for the `MPI_Send()` in line 18 with the obvious exception that the destination process is specified by the loop variable process, which is incremented from 1 to np − 3 (the `last_node` is np − 2).

Line 22 advances the send address for each internal process by the number of grid points in each partition: `dimx*dimy*dimz/num_comp_nodes`. Note that the starting locations of the halo grid points for internal processes are `dimx*dimy*dimz/num_comp_procs` points apart. Although we need to pull back the starting address by four slices to accommodate halo grid points, we do so for every internal process, so the net distance between the starting locations remains the number of grid points in each partition.

Line 23 sends the data to the process np − 2, the last compute process that has only one neighbor on the left. The reader should be able to reason through all the parameter values that are used. Note that we are not quite done with the data server code. We will come back later for the final part of the data server that collects the output values from all compute processes.

We now turn our attention to the compute processes that receive the input from the data server process. In Fig. 20.10, lines 03−04 establish the process id for the process and the total number of processes for the application. Line 05 establishes that the data server is process np − 1. Lines 06−07 calculate the number of grid points and the number of bytes that should be processed by each internal process. Lines 08−09 calculate the number of grid points and the number of bytes in each halo (4 slices).

Lines 10−12 allocate the host memory and device memory for the input data. Although the edge processes need less halo data, they still allocate the same

```
01  void compute node stencil(int dimx, int dimy, int dimz, int nreps ) {
02     int np, pid;
03     MPI_Comm_rank(MPI_COMM_WORLD, &pid);
04     MPI_Comm_size(MPI_COMM_WORLD, &np);
05     int server process = np - 1;
06     unsigned int num_points       = dimx * dimy * (dimz + 8);
07     unsigned int num_bytes        = num_points * sizeof(float);
08     unsigned int num_halo_points = 4 * dimx * dimy;
09     unsigned int num_halo_bytes  = num_halo_points * sizeof(float);
       /* Allocate host memory */
10     float *h_input  = (float *)malloc(num_bytes);
       /* Allocate device memory for input and output data */
11     float *d_input = NULL;
12     cudaMalloc((void **)&d input,  num bytes );
13     float *rcv_address = h_input + ((0 == pid) ? num_halo_points : 0);
14     MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
                  MPI_ANY_TAG, MPI_COMM_WORLD, &status );
15     cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);
```

FIGURE 20.10

Compute process code (part 1).

amount of memory for simplicity; part of the allocated memory will not be used by the edge processes. Line 13 sets the starting address of the host memory for receiving the input data from the data server. For all compute processes except process 0, the starting receiving location is simply the starting location of the allocated memory for the input data. However, for process 0, we adjust the receiving location by four slices. This is because for simplicity we assume that the host memory for receiving the input data is arranged in the same way for all compute processes: four slices of halo elements from the left neighbor followed by the partition, followed by four slices of halo elements from the right neighbor. However, as we showed in line 15 of Fig. 20.9, the data server will not send any halo data from the left neighbor to process 0. That is, for process 0, the MPI message from the data server contains only the partition and the halo from the right neighbor. Therefore line 13 adjusts the starting host memory location by four slices so that process 0 will correctly interpret the input data from the data server.

Line 14 receives the MPI message from the data server. Most of the parameters should be familiar. The last parameter reflects any error condition that has occurred when the data are received. The second parameter specifies that all compute processes will receive the full amount of data from the data server. However, the data server will send less data to process 0 and process np − 2. This is not reflected in the code because `MPI_Recv()` allows the second parameter to specify a larger number of data points than are actually received and will place only the actual number of bytes received from the sender into the receiving memory. In the case of process 0, the input data from the data server contains only the partition and the halo from the right neighbor. The received input will be placed by skipping the first four slices of the allocated memory, which correspond to the halo for the (nonexistent) left neighbor. This effect is achieved with the term `((0==pid)? num_halo_points: 0)` on line 13. In the case of process np − 2, the input data contains the halo from the left neighbor and the partition. The received input will be placed from the beginning of the allocated memory, leaving the last four slices of the allocated memory unused.

Line 15 copies the received input data to the device memory. In the case of process 0, the left halo points are not valid. In the case of process np − 2, the right halo points are not valid. However, for simplicity, all compute nodes send the full size to the device memory. The assumption is that the kernels will be launched in such a way that these invalid portions will be correctly ignored. After line 15, all the input data are in the device memory.

Fig. 20.11 shows part 2 of the compute process code. Lines 16–18 allocate host memory and device memory for the output data. The output data buffer in the device memory will be used with the input data buffer in a double-buffering scheme. That is, they will switch roles in each iteration. We will return to this point and cover the rest of the code in Fig. 20.11 later. We are now ready to present the code that performs computation steps on the grid points.

```
16     float *h output = NULL, *d output = NULL, *d vsq = NULL;
17     float *h_output = (float *)malloc(num_bytes);
18     cudaMalloc((void **)&d_output, num_bytes );
19     float *h_left_boundary = NULL, *h_right_boundary = NULL;
20     float *h left halo = NULL, *h right halo = NULL;
       /* Allocate host memory for halo data */
21     cudaHostAlloc((void **)&h_left_boundary,  num_halo_bytes,
                     cudaHostAllocDefault);
22     cudaHostAlloc((void **)&h_right_boundary, num_halo_bytes,
                     cudaHostAllocDefault);
23     cudaHostAlloc((void **)&h_left_halo,      num_halo_bytes,
                     cudaHostAllocDefault);
24     cudaHostAlloc((void **)&h_right_halo,     num_halo_bytes,
                     cudaHostAllocDefault);
    /* Create streams used for stencil computation */
25     cudaStream_t stream0, stream1;
26     cudaStreamCreate(&stream0);
27     cudaStreamCreate(&stream1);
```

FIGURE 20.11

Compute process code (part 2).

20.5 Overlapping computation and communication

A simple way to perform the computation steps is for each compute process to perform a computation step on its entire partition, exchange halo data with the left and right neighbors, and repeat. While this is a very simple strategy, it is not very effective. The reason is that this strategy forces the system to be in one of the two modes. In the first mode, all compute processes are performing computation steps. During this time, the communication network is not used. In the second mode, all compute processes are exchanging halo data with their left and right neighbors. During this time, the computation hardware is not well utilized. Ideally, we would like to achieve better performance by utilizing both the communication network and the computation hardware all the time. This can be achieved by dividing the computation tasks of each compute process into two stages, as illustrated in Fig. 20.12.

During the first stage (stage 1), each compute process calculates its boundary slices that will be needed as halo cells by its neighbors in the next iteration. Let's continue to assume that we use four slices of halo data. Fig. 20.12 shows the four halo slices as a dashed transparent piece and the four boundary slices as a colored piece. Note that the colored piece of process i will be copied into the dashed piece of process I + 1 and the colored piece of process I + 1 will be copied into the dashed piece of process i during the next communication. For process 0, the first phase calculates the right four slices of boundary data. For an internal node it calculates the left four slices and the right four slices of its boundary data. For process $n - 2$, it calculates the left four pieces of its boundary data. The rationale is that these boundary slices are needed by their neighbors for the next iteration. If these boundary slices are calculated first, the data can be communicated to the neighbors while the compute processes calculate the rest of their internal grid points.

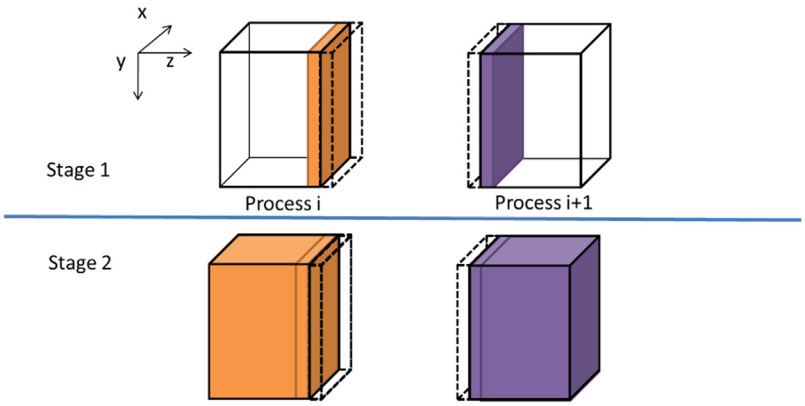


FIGURE 20.12

A two-stage strategy for overlapping computation with communication.

During the second stage (stage 2), each compute process performs two activities in parallel. The first is to communicate its new boundary values to its neighbor processes. This is done by first copying the data from the device memory into the host memory, followed by sending MPI messages to the neighbors. As we will discuss later, we need to be careful that the data that is received from the neighbors is used in the next iteration, not the current iteration. The second activity is to calculate the rest of the data in the partition. If the communication activity takes a shorter amount of time than the calculation activity, we can hide the communication delay and fully utilize the computing hardware all the time. This is usually achieved by having enough slices in the internal part of each partition to allow each compute process to perform enough computation to hide the communication.

To support the parallel activities in stage 2, we need to use two advanced features of the CUDA programming model: *pinned memory allocation* and *streams*. A pinned memory allocation requests that the memory that is allocated not be paged out by the operating system. This is done with the `cudaHostAlloc()` API call. Lines 21–24 in Fig. 20.11 allocate pinned memory buffers for the left and right boundary slices and the left and right halo slices. The left and right boundary slices need to be sent from the device memory to the left and right neighbor processes. The buffers are used as a host memory staging area for the device to copy data into and are then used as the source buffer for `MPI_Send()` to neighbor processes. The left and right halo slices need to be received from neighbor processes. The buffers are used as a host memory staging area for `MPI_Recv()` to use as destination buffers and then to copy data from them to the device memory. These buffers are sometimes referred to as *bounce buffers*, as their main role is to serve as temporary buffers that allow the data to be bounced from the device memory to the remote MPI process and vice versa.

Note that the host memory allocation for the bounce buffers is done with `cudaHostAlloc()` function rather than the standard `malloc()` function. The difference is that the `cudaHostAlloc()` function allocates a *pinned memory* buffer, sometimes also referred to as *page locked memory* buffer. We need to know a little more background on the memory management in operating systems in order to fully understand the concept of pinned memory buffers.

In a modern computer system the operating system manages virtual memory spaces for applications. Each application has access to a large, consecutive address space. In reality the system has a limited amount of physical memory that needs to be shared among all running applications. This sharing is performed by partitioning the virtual memory space into pages and mapping only the actively used pages into physical memory. When there is much demand for memory, the operating system needs to page out some of the pages from the physical memory to mass storage such as disks. Therefore an application may have its data paged out at any time during its execution.

The implementation of `cudaMemcpy()` uses a type of hardware called a direct memory access (DMA) device. When a `cudaMemcpy()` function is called to copy between the host and device memories, its implementation uses a DMA operation to complete the task. On the host memory side, the DMA hardware operates on physical addresses; that is, the operating system needs to give a translated physical address to the DMA device. However, there is a chance that the data may be paged out before the DMA operation is complete. The physical memory locations for the data may be reassigned to other data corresponding to a different virtual memory location. In this case, the DMA operation can potentially be corrupted, since its data can be overwritten by the paging activity.

A common solution to this data corruption problem is for the CUDA runtime to perform the copy operation in two steps. For a host-to-device copy, the CUDA runtime first copies the source host memory data into a pinned memory buffer, which means that the memory locations are marked so that the paging mechanism will not page out the data. It then uses the DMA device to copy the data from the pinned memory buffer to the device memory. For a device-to-host copy, the CUDA runtime first uses a DMA device to copy the data from the device memory into a pinned memory buffer. It then copies the data from the pinned memory buffer to the destination host memory location. By using an extra pinned memory buffer, the DMA copy will be safe from any paging activities.

There are two problems with this approach. One is that the extra copy adds delay to the `cudaMemcpy()` operation. The second is that the extra complexity involved leads to a synchronous implementation of the `cudaMemcpy()` function. That is, the host program cannot continue to execute until the `cudaMemcpy()` function has completed its operation and returned. This serializes all copy operations. To support fast copies with more parallelism, CUDA provides a `cudaMemcpyAsync()` function.

The `cudaMemcpyAsync()` function requires that the host memory buffer be allocated as a pinned memory buffer. This is done in lines 21–24 in Fig. 20.11 for

the host memory buffers of the left boundary, right boundary, left halo, and right halo slices. These buffers are allocated with the `cudaHostAlloc()` function, which ensures that the allocated memory is pinned or page locked from paging activities. Note that the `cudaHostAlloc()` function takes three parameters. The first two are the same as `cudaMalloc()`. The third specifies some options for more advanced usage. For most basic use cases, we can simply use the default value `cudaHostAllocDefault`.

The second advanced CUDA feature that is used for overlapping communication with computation is *streams*, a feature that supports managed concurrent execution of CUDA API functions. A stream is an ordered sequence of operations. When the host code calls a `cudaMemcpyAsync()` function or launches a kernel, it can specify a stream as one of its parameters. By specifying a stream in a `cudaMemcpyAsync()` call, the copy operation is placed into a stream. All operations in the same stream will be done sequentially according to the order in which they are placed into that stream. However, operations in different streams can be executed in parallel without any ordering constraint.

Line 25 of Fig. 20.11 declares two variables that are of CUDA built-in type `cudaStream_t`. These variables are then used in calling the `cudaStreamCreate()` function. Each call to `cudaStreamCreate()` creates a new stream and deposits an identifier of the stream into its parameter. After the calls in lines 26 and 27, the host code can use either stream 0 or stream 1 in subsequent `cudaMemcpyAsync()` calls and kernel calls.

Fig. 20.13 shows part 3 of the compute process. Line 28 declares an MPI status variable that will be used for MPI send and receive. Lines 29–30 calculate the process id of the left and right neighbors of the compute process. The `left_neighbor` and `right_neighbor` variables will be used by compute processes as

```
28     MPI_Status status;
29     int left_neighbor  = (pid > 0)      ? (pid - 1) : MPI_PROC_NULL;
30     int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;

       /* Upload stencil cofficients */
31     upload_coefficients(coeff, 5);
32     int left_halo_offset    = 0;
33     int right_halo_offset   = dimx * dimy * (4 + dimz);
34     int left_stage1_offset  = 0;
35     int right_stage1_offset = dimx * dimy * (dimz - 4);
36     int stage2 offset       = num halo points;
37     MPI_Barrier( MPI_COMM_WORLD );
38     for(int i=0; I < nreps; i++) {
             /* Compute boundary values needed by other nodes first */
39           call stencil kernel(d output + left stage1 offset,
                    d_input + left_stage1_offset, dimx, dimy, 12, stream0);
40           call_stencil_kernel(d_output + right_stage1_offset,
                    d_input + right_stage1_offset, dimx, dimy, 12, stream0);
             /* Compute the remaining points */
41           call_stencil_kernel(d_output + stage2_offset, d_input +
                        stage2_offset, dimx, dimy, dimz, stream1);
```

FIGURE 20.13

Compute process code (part 3).

parameters when they send messages to and receive messages from their neighbors. For process 0, there is no left neighbor, so line 29 assigns an MPI constant `MPI_PROC_NULL` to `left_neighbor` to note this fact. For process np − 2, there is no right neighbor, so line 30 assigns `MPI_PROC_NULL` to right_neighbor. For all the internal processes, lines 29−30 assign pid − 1 to `left_neighbor` and pid + 1 to `right_neighbor`.

Line 31 in Fig. 20.13 calls a function to copy the stencil coefficients to the GPU constant memory. The details will not be shown here because the reader should already be familiar with constant memory. Lines 32−36 set up several offsets that will be used to call kernels and exchange data so that the computation and communication can be overlapped. These offsets define the regions of grid points that will need to be calculated at each stage of Fig. 20.13. They are also visualized in Fig. 20.12.

Note that the total number of slices in each device memory is four slices of left halo points (dashed white) plus four slices of left boundary points plus `dimx*dimy*(dimz − 8)` internal points plus four slices of right boundary points plus four slices of right halo points (dashed white). As illustrated in Fig. 20.14, variable `left_stage1_offset` defines the starting point of the slices that are needed in order to calculate the left boundary slices. This includes 12 slices of data: four slices of left neighbor halo points, four slices of boundary points, and four slices of internal points. These slices are the leftmost in the partition, so the offset value is set to 0 by line 34. Variable `right_stage2_offset` defines the starting point of the slices that are needed for calculating the right boundary slices. This also includes 12 slices: four slices of internal points, four slices of right boundary points, and four slices of right halo cells. The beginning point of these 12 slices can be derived by subtracting 12 from the total number of slices dimz + 8. Therefore the starting offset for these 12 slices is set to `dimx*dimy*(dimz − 4)` by line 35 (Fig. 20.12).

Line 37 in Fig. 20.13 is an MPI barrier synchronization, which is similar to the `CUDA __syncthreads()` across threads in a block. MPI barrier forces all MPI processes that are specified by the input argument to wait for each other. None of the

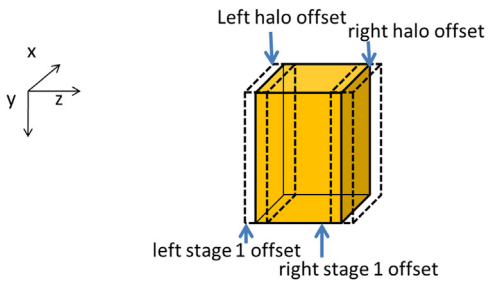


FIGURE 20.14

Device memory offsets used for data exchange with neighbor processes.

processes can continue their execution beyond this point until all have reached this point. The reason why we want the barrier synchronization here is to ensure that all compute nodes have received their input data and are ready to perform the computation steps. Since they will be exchanging data with each other, we would like to make them all start at about the same time. Thus we will not be in a situation in which a few tardy processes delay all other processes during the data exchange. `MPI_Barrier()` is a *collective communication* function. We will discuss collective communication API functions in more detail in the next section.

Line 38 starts a loop that performs the computation steps. For each iteration, each compute process will perform one cycle of the two-stage process shown in Fig. 20.12. Line 39 calls a function that will generate the four slices of the left boundary points in stage 1. We assume that the function will set up the grid configuration and call the stencil kernel that performs one computation step on a region of grid points as described in Chapter 8, Stencil. The `call_stencil_kernel()` function takes several parameters. The first parameter is a pointer to the output data area for the kernel. The second parameter is a pointer to the input data area. In both cases, we add the `left_stage1_offset` to the input and output data in the device memory. The next three parameters specify the dimensions of the portion of the grid to be processed. Note that we need to have four slices on each side in order to correctly perform the computation for all the points in the four left boundary slices. Line 40 does the same for the right boundary points in stage 1. Note that these kernels will be called within stream 0 and will be executed sequentially.

Line 41 calls the `call_stencil_kernel()` function, which will call a stencil kernel function to generate the `dimx*dimy*(dimz − 8)` internal points in stage 2. Note that this also requires four slices of input boundary values on each side, so the total number of input slices is `dimx*dimy*dimz`. The kernel is called in stream 1 and will be executed in parallel with those called by lines 39 and 40.

Fig. 20.15 shows part 4 of the compute process code. Line 42 copies the four slices of left boundary points to the host memory in preparation for data exchange with the left neighbor process. Line 43 copies the four slices of the right boundary points to the host memory in preparation for data exchange with the right neighbor process. Both are asynchronous copies in stream 0 and will wait for the two kernels in stream 0 to complete before they copy data. Line 44 is a synchronization that forces the process to wait for all operations in stream 0 to complete before it can continue. This ensures that the left and right boundary points are in the host memory before the process proceeds with data exchange.

During the data exchange phase, we will have all MPI processes send their boundary slices to their left neighbors. That is, all processes will have their right neighbors sending data to them. It is therefore convenient to have an MPI function that sends data to a destination and receives data from a source. This reduces the number of MPI function calls. The `MPI_Sendrecv()` function in Fig. 20.16 is such a function. It is a combination of `MPI_Send()` and `MPI_Recv()` so we will not elaborate further on the meaning of the parameters.

```
            /* Copy the data needed by other nodes to the host */
42          cudaMemcpyAsync(h_left_boundary, d_output + num_halo_points,
                        num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );
43          cudaMemcpyAsync(h_right_boundary,
                        d_output + right_stage1_offset + num_halo_points,
                        num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );
44          cudaStreamSynchronize(stream0);
            /* Send data to left, get data from right */
45          MPI_Sendrecv(h_left_boundary, num_halo_points, MPI_FLOAT,
                        left neighbor,  i, h right halo,  num halo points,
                        MPI_FLOAT, right_neighbor, i, MPI_COMM_WORLD, &status );
            /* Send data to right, get data from left */
46          MPI_Sendrecv(h_right_boundary, num_halo_points, MPI_FLOAT,
                        right_neighbor, i, h_left_halo, num_halo_points,
                        MPI_FLOAT, left_neighbor,  i, MPI_COMM_WORLD, &status );

47          cudaMemcpyAsync(d_output+left_halo_offset,  h_left_halo,
                        num_halo_bytes, cudaMemcpyHostToDevice, stream0);
48          cudaMemcpyAsync(d_output+right_halo_offset, h_right_halo,
                        num_halo_bytes, cudaMemcpyHostToDevice, stream0 );
49          cudaDeviceSynchronize();

50          float *temp = d_output;
51          d output = d input; d input = temp;
52      }
```

FIGURE 20.15

Compute process code (part 4).

- `int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`
  - `Sendbuf`: Initial address of send buffer (choice)
  - `Sendcount`: Number of elements in send buffer (integer)
  - `Sendtype`: Type of elements in send buffer (handle)
  - `Dest`: Rank of destination (integer)
  - `Sendtag`: Send tag (integer)
  - `Recvcount`: Number of elements in receive buffer (integer)
  - `Recvtype`: Type of elements in receive buffer (handle)
  - `Source`: Rank of source (integer)
  - `Recvtag`: Receive tag (integer)
  - `Comm`: Communicator (handle)
  - `Recvbuf`: Initial address of receive buffer (choice)
  - `Status`: Status object (Status). This refers to the receive operation.

FIGURE 20.16

Syntax for the MPI_Sendrecv() function.

Line 45 sends four slices of left boundary points to the left neighbor and receives four slices of right halo points from the right neighbors. Line 46 sends four slices of right boundary points to the right neighbor and receives four slices of left halo points from the left neighbor. In the case of process 0, its `left_neighbor` has been set to `MPI_PROC_NULL` in line 27, so the MPI runtime

will not send out the message in line 45 or receive the message in line 46 for process 0. Likewise, the MPI runtime will not receive the message in line 45 or send out the message in line 46 for process np − 2. Therefore the conditional assignments in lines 29 and 30 of Fig. 20.13 eliminate the need for special if-then-else statements in lines 45 and 46.

After the MPI messages have been sent and received, lines 47 and 48 transfer the newly received halo points to the `d_output` buffer of device memory. These copies are done in stream 0, so they will execute in parallel with the kernel launched in stream 1 on line 38.

Line 49 is a synchronization operation for all device activities. This call forces the process to wait for all device activities, including kernels and data copies, to complete. When the `cudaDeviceSynchronize()` function returns, all `d_output` data from the current computation step are in place: left halo data from the left neighbor process, boundary data from the kernel launched in line 36, internal data from the kernel launched in line 38, right boundary data from the kernel launched in line 37, and right halo data from the right neighbor.

Lines 50 and 51 swap the `d_input` and `d_output` pointers. This changes the output of the `d_ouput` data of the current computation step into the `d_input` data of the next computation step. The execution then proceeds to the next computation step by going to the next iteration of the loop of line 35. This will continue until all compute processes have completed the number of computations specified by the parameter nreps.

Fig. 20.17 shows part 5, the final part of the compute process code. Line 53 is a barrier synchronization that forces all processes to wait for each other to finish their computation steps. Lines 54−56 swap `d_output` with `d_input`. This is because lines 50 and 51 swapped `d_output` with `d_input` in preparation for the

```
      /* Wait for previous communications */
53    MPI_Barrier(MPI_COMM_WORLD);

54    float *temp = d_output;
55    d_output = d_input;
56    d_input = temp;

      /* Send the output, skipping halo points */
57    cudaMemcpy(h_output, d_output, num_bytes, cudaMemcpyDeviceToHost);
      float *send_address = h_output + num_ghost_points;
58    MPI_Send(send_address, dimx * dimy * dimz, MPI_REAL,
                  server_process, DATA_COLLECT, MPI_COMM_WORLD);
59    MPI_Barrier(MPI_COMM_WORLD);

      /* Release resources */
60    free(h_input); free(h_output);
61    cudaFreeHost(h_left_ghost_own); cudaFreeHost(h_right_ghost_own);
62    cudaFreeHost(h_left_ghost); cudaFreeHost(h_right_ghost);
63    cudaFree( d_input ); cudaFree( d_output );
64  }
```

FIGURE 20.17

Compute process code (part 5).

next computation step. However, this is unnecessary for the last computation step, so we use lines 54–56 to undo the swap. Line 57 copies the final output to the host memory. Line 58 sends the output to the data server. Line 59 waits for all processes to complete. Lines 60–63 free all the resources before returning to the main program.

Fig. 20.18 shows part 2, the final part of the data server code, which continues from Fig. 20.9. Line 24 is a barrier synchronization that waits for all compute nodes to complete their computation steps and send their outputs. This barrier corresponds to the barrier at line 59 of the compute process. Lines 26 and 27 receive the output data from all the compute processes. Line 28 stores the output into an external storage. Finally, lines 29 and 30 free resources before returning to the main program.

20.6 Message passing interface collective communication

We saw an example of the MPI collective communication API in the previous section: `MPI_Barrier`. The other commonly used group collective communication types are broadcast, reduce, gather, and scatter (Gropp et al., 1999). Barrier synchronization `MPI_Barrier()` is perhaps the most commonly used collective communication function. As we saw in the stencil example, barriers are used to ensure that all MPI processes are ready before they begin to interact with each other. We will not elaborate on the other types of MPI collective communication functions, but we encourage the reader to read the details of these functions. In general, collective communication functions are highly optimized by the MPI runtime developers and system vendors. Using them usually leads to better performance as well as readability and productivity than trying to achieve the same functionality with combinations of send and receive calls.

```
      /* Wait for nodes to compute */
24    MPI_Barrier(MPI_COMM_WORLD);
      /* Collect output data */
25    MPI_Status status;
26    for(int process = 0; process < num comp nodes; process++)
27          MPI_Recv(output + process * num_points / num_comp_nodes,
                num_points / num_comp_nodes, MPI_REAL, process,
                DATA_COLLECT, MPI_COMM_WORLD, &status );

      /* Store output data */
28    store_output(output, dimx, dimy, dimz);
      /* Release resources */
29    free(input);
30    free(output);
}
```

FIGURE 20.18

Data server code (part 2).

20.7 CUDA aware message passing interface

Modern MPI implementations are aware of the CUDA programming model and are designed to minimize the communication latency between GPUs. Currently, direct interaction between CUDA and MPI is supported by MVAPICH2, IBM Platform MPI, and `OpenMPI`.

CUDA-aware MPI implementations are capable of sending messages from the GPU memory in one node to the GPU memory in a different node. This effectively removes the need for device-to-host data transfers before sending MPI messages and host-to-device data transfers after receiving an MPI message. This has the potential to simplify the host code and memory data layout. In our stencil example, if we used a CUDA-aware MPI implementation, we will no longer need host-pinned memory allocations and asynchronous memory copies.

The first simplification is that we no longer need to allocate host-pinned memory buffers to transfer the halo points to the host memory to prepare for `MPI_Send()`. This means that we can safely remove lines 21–24 in Fig. 20.11. However, we still need to use CUDA streams and two separate GPU kernels to start communicating across nodes as soon as the halo elements have been computed.

The second simplification is that we no longer need to asynchronously copy the halo data from the host to the device memory after `MPI_Recv()`. As a result, we can also remove lines 42 and 43 in Fig. 20.15. Since the MPI calls now accept device memory addresses, we need to modify the calls to `MPI_SendRecv` to use them. Note that these memory addresses actually correspond to the device addresses of the asynchronous memory copies in the previous versions. Since the CUDA-aware MPI implementations will directly update the contents of the GPU memory, we also remove lines 47 and 48 in Fig. 20.15. Fig. 20.19 shows the modifications to he MPI_SendRecv statements in lines 45 and 46 in Fig. 20.15, so that they directly read from and write to the device memory.

Besides removing the data transfers during the halo exchange using `MPI_SendRecv()`, it would also be possible to remove the initial and final memory copies by receiving/sending the input/output directly from the GPU memory.

```
MPI_SendRecv(d_output + num_halo_points, num_halo_points, MPI_FLOAT,
        left_neighbor, i, d_output + left_halo_offset, num_halo_points,
        MPI_FLOAT, right_neighbor, i, MPI_COMM_WORLD, &status);
MPI_SendRecv(d_output + right_stage1_offset, num_halo_points,
        num_halo_points, MPI_FLOAT, right_neighbor, i,
        d_output + right_halo_offset, num_halo_points,
        MPI_FLOAT, left_neighbor, i, MPI_COMM_WORLD, &status);
```

FIGURE 20.19

Revised MPI SendRec calls in using CUDA-aware MPI.

20.8 Summary

In this chapter we covered basic patterns of joint CUDA/MPI programming for HPC clusters with heterogeneous computing nodes. All processes in an MPI application run the same program. However, each process can follow different control flow and function call paths to specialize their roles, as illustrated by the data server and the compute processes in our example. We also used the stencil pattern to show how compute processes exchange data. We presented the use of CUDA streams and asynchronous data transfers to enable the overlap of computation and communication. We also showed how to use the MPI barrier to ensure that all processes are ready to exchange data with each other. Finally, we briefly outlined the use of CUDA-aware MPI to simplify the exchange of data in the device memory. We would like to point out that while MPI is a very different programming system, all major MPI concepts that we covered in this chapter, namely, SPMD, MPI ranks, and barriers, have counterparts in the CUDA programming model. This confirms our belief that by teaching parallel programming well with one model, our students can quickly and easily pick up other programming models. We would like to encourage the reader to build on the foundation provided by this chapter and study more advanced MPI features and other important patterns.

Exercises

1. Assume that the 25-point stencil computation in this chapter is applied to a grid whose size is 64 grid points in the x dimension, 64 in the y dimension, and 2048 in the z dimension. The computation is divided across 17 MPI ranks, of which 16 ranks are compute processes and 1 rank is the data server process.
   - **a.** How many output grid point values are computed by each compute process?
   - **b.** How many halo grid points are needed:
     - **i.** By each internal compute process?
     - **ii.** By each edge compute process?
   - **c.** How many boundary grid points are computed in stage 1 of Fig. 20.12:
     - **i.** By each internal compute process?
     - **ii.** By each edge compute process?
   - **d.** How many internal grid points are computed in stage 2 of Fig. 20.12:
     - **i.** By each internal compute process?
     - **ii.** By each edge compute process?
   - **e.** How many bytes are sent in stage 2 of Fig. 20.12:
     - **i.** By each internal compute process?
     - **ii.** By each edge compute process?

2. If the MPI call `MPI_Send`(ptr_a, 1000, `MPI_FLOAT`, 2000, 4, `MPI_COMM_WORLD`) results in a data transfer of 4000 bytes, what is the size of each data element being sent?
   a. 1 byte
   b. 2 bytes
   c. 4 bytes
   d. 8 bytes
3. Which of the following statements is true?
   a. `MPI_Send()` is blocking by default.
   b. `MPI_Recv()` is blocking by default.
   c. MPI messages must be at least 128 bytes.
   d. MPI processes can access the same variable through shared memory.
4. Modify the example code to remove the calls to `cudaMemcpyAsync()` from the compute processes' code by using GPU memory addresses on `MPI_Send` and `MPI_Recv`.

References

Gropp, W., Lusk, E., Skjellum, A., 1999. Using MPI, Portable Parallel Programming with the Message Passing Interface, 2nd Ed. MIT Press, Cambridge, MA, Scientific and Engineering Computation Series. ISBN 978-0-262−57132-6.

CHAPTER

CUDA dynamic parallelism 21

With special contributions from Juan Gómez-Luna

Chapter Outline

CUDA dynamic parallelism is an extension to the CUDA programming model that enables a kernel to call other kernels, thereby allowing threads executing on the device to launch new grids of threads. In early versions of CUDA, grids could be launched only from the host code. Algorithms that involved recursion, irregular loop structures, time-space variation, or other constructs that do not fit a flat and single level of parallelism needed to be implemented with multiple kernel calls from the host, which increases the burden on the host, the amount of host-device communication, and the total execution time. In some cases, programmers resort to loop serialization and other awkward techniques to support these algorithmic needs at the cost of software maintainability. The support for dynamic parallelism allows algorithms that dynamically discover new work to prepare and launch new grids without burdening the host or impacting the software maintainability. This chapter describes the extended capabilities of CUDA that enable dynamic parallelism, including the modifications and additions to the CUDA programming interface, as well as guidelines and best practices for exploiting this added capacity.

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00001-X

21.1 Background

Many real-world applications employ algorithms that have either a variation of work across space or a dynamically varying amount of work performed over time. For example, Fig. 21.1 shows a turbulence simulation example in which the level of required modeling details varies across both space and time. As the combustion flow moves from left to right, the range of activities and intensity increases. The level of details required to model the right side of the model is much higher than that for the left side of the model. On one hand, using a fixed fine grid would incur too much work for no gain for the left side of the model. On the other hand, using a fixed coarse grid would sacrifice too much accuracy for the right side of the model. Ideally, one should use fine grids for the parts of the model that require more details and coarse grids for the parts that do not.

Thus far, we have assumed that all kernels are called from the host code. The amount of work done by a thread grid is predetermined in calling the kernel function. With the single-program, multiple-data programming style for the kernel code, it is tedious if not extremely difficult to have thread blocks use different grid spacing. As a result, this limitation favors the use of a fixed and uniform (regular) grid system. To achieve the desired accuracy, such a fixed grid

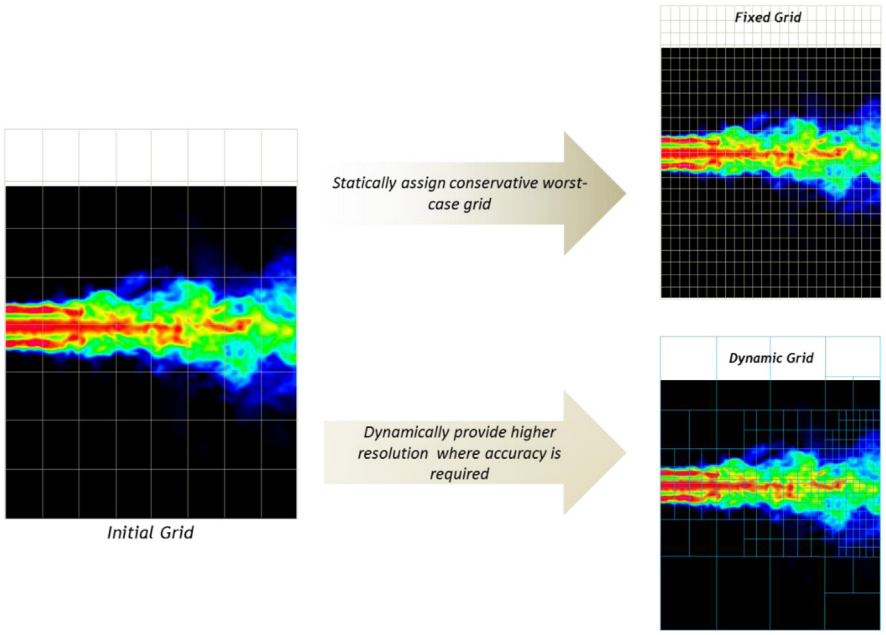


FIGURE 21.1

Fixed versus dynamic grids for a turbulence simulation model.

approach, as illustrated in the upper right portion of Fig. 21.1, typically needs to accommodate the most demanding parts of the model and maintain unnecessary extra data as well as perform unnecessary extra work in parts that do not require as much detail.

A more desirable approach is shown as the dynamic grid in the lower right portion of Fig. 21.1. As the simulation algorithm detects fast-changing simulation quantities in some areas of the model, it refines the grid in those areas to achieve the desired level of accuracy. Such refinement does not need to be done for the areas that do not exhibit such intensive activity. Thus the algorithm can dynamically direct more computational resources to the areas of the model that benefit from additional work.

Fig. 21.2 shows a conceptual comparison of behavior between a system without dynamic parallelism and another one with dynamic parallelism with respect to the simulation model in Fig. 21.1. Without dynamic parallelism the host thread must launch all grids. If new work is discovered, such as refining an area in the model during the execution of a grid, the grid needs to terminate, report back to the host, and have the host launch new grids. This is illustrated in Fig. 21.2A, in which the host launches a grid, receives information from this grid after its termination, and launches subsequent grids for any new work that is discovered by the completed grid. The figure depicts the subsequent grids as being launched one after the other; however, sophisticated optimizations may be applied, such as

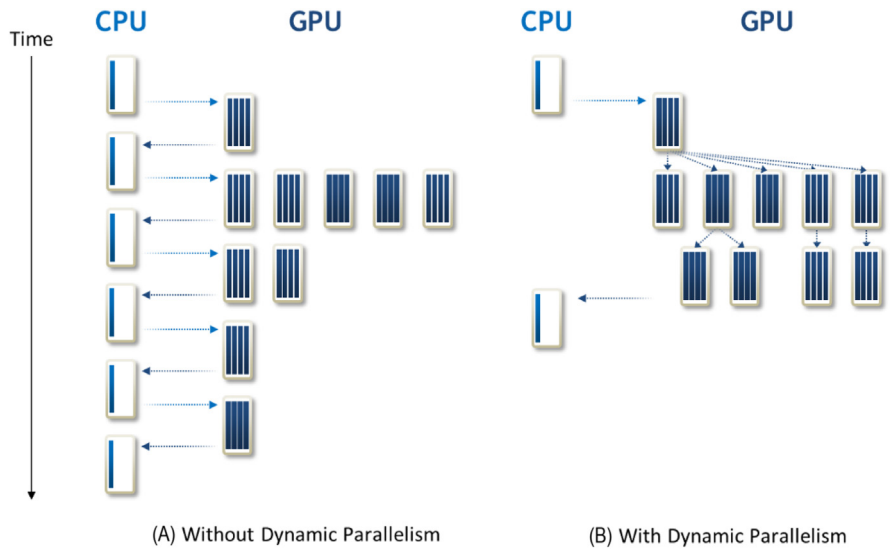


FIGURE 21.2

Grid launch patterns for algorithms with dynamic work variation, (A) without and (B) with dynamic parallelism.

launching independent grids in different streams or combining them so that they can run in parallel.

Fig. 21.2B shows that with dynamic parallelism the threads that discover new work can just go ahead and launch grids to do the work. In our example, when a thread discovers that an area in the model needs to be refined, it can launch a new grid to perform the computation on the refined area without the overhead of terminating the grid, reporting back to the host, and having the host launch the new grid.

21.2 Dynamic parallelism overview

From the perspective of programmers, dynamic parallelism means that they can write a statement within a kernel that calls another kernel function. In Fig. 21.3 the main function (host code) launches three kernels: A, B, and C. These are kernel calls in the host code, as we have been assuming throughout this book. What is new in Fig. 21.3 is that one of the kernels, B, calls three kernels X, Y, and Z. This would have been illegal in early CUDA systems that do not support dynamic parallelism.

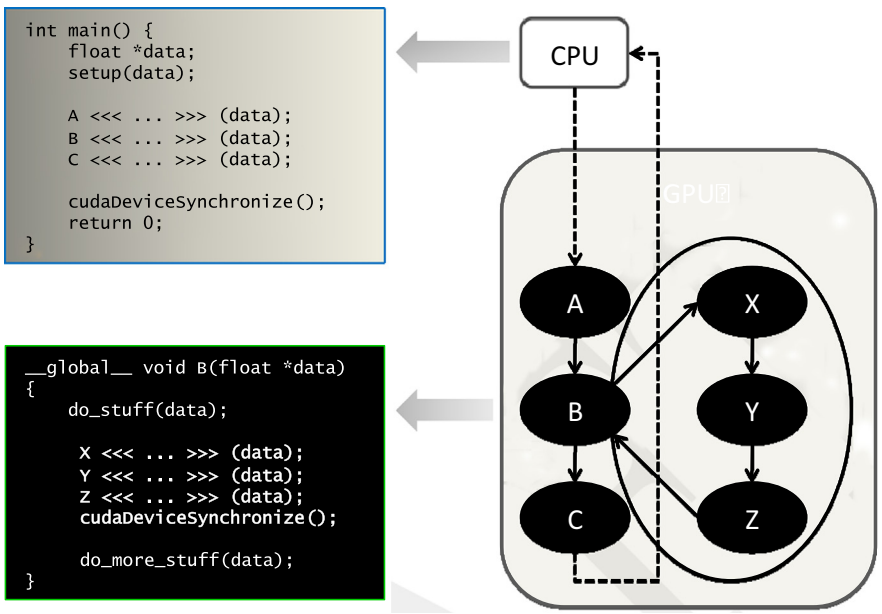


FIGURE 21.3

A simple example of a kernel (B) launching three kernels (X, Y, and Z).

The syntax for calling a kernel from inside of a kernel is the same as that for calling a kernel from the host code:

```
kernel_name<<< Dg, Db, Ns, S >>>([kernel arguments])
```

- `Dg` is of type `dim3` and specifies the dimensions and size of the grid.
- `Db` is of type `dim3` and specifies the dimensions and size of each thread block.
- `Ns` is of type `size_t` and specifies the number of bytes of shared memory that are dynamically allocated per-thread block for this call, which is in addition to the statically allocated shared memory. `Ns` is an optional argument that defaults to 0.
- `S` is of type `cudaStream_t` and specifies the stream associated with this call. The stream must have been allocated in the same thread block in which the call is being made. `S` is an optional argument that defaults to 0. Streams were discussed in Chapter 20, Programming a Heterogeneous Computing Cluster.

To demonstrate how dynamic parallelism can be used, we provide simple examples of kernels without and with dynamic parallelism. The examples are based on a hypothetical parallel algorithm that does not compute useful results but provides a conceptually simple computational pattern that recurs in many applications. It serves to illustrate the difference between the two approaches and how one can use the dynamic parallelism to extract more parallelism while reducing control flow divergence when the amount of work done by each thread in an algorithm can vary dynamically.

Fig. 21.4 shows a simple example kernel coded without dynamic parallelism. In this example, each thread of the kernel performs some computation (line 05), then loops over a list of data elements for which it is responsible (line 07), and performs another computation for each data element (line 08).

This computation pattern recurs frequently in many applications. For example, in graph search, each thread could visit a vertex and then loop over a list of

```
01    __global__ void kernel(unsigned int* start, unsigned int* end,
02        float* someData, float* moreData) {
03
04        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
05        doSomeWork_findMoreWork(someData[i]);
06
07        for(unsigned int j = start[i]; j < end[i]; ++j) {
08            doMoreWork(moreData[j]);
09        }
10
11    }
```

FIGURE 21.4

A simple example of a hypothetical parallel algorithm coded in CUDA without dynamic parallelism.

neighboring vertices. The reader should find this kernel structure similar to that of the vertex-centric BFS kernel in Figure 15.14. For another example, in sparse matrix computations, each thread could first identify the starting location of a row of nonzero elements and loop over the nonzero values. In simulations such as the example at the beginning of this chapter, each thread could first process a coarse grid element and then loop over finer grid elements if there is a need for refining the grid.

There are two main problems with writing applications in the style shown in Fig. 21.4. First, if the work in the loop (lines 07–09) can be profitably performed in parallel, then we have missed out on an opportunity to extract more parallelism from the application. Second, if the number of iterations in the loop varies significantly between threads in the same warp, then the resulting control divergence can degrade the performance of the program.

Fig. 21.5 shows a version of the same program that uses dynamic parallelism. In this version the original kernel is separated into two: a parent kernel and a child kernel. The parent kernel starts off the same as the original kernel, executed by a grid of threads that are referred to as the parent grid. Instead of looping, the parent kernel calls a child kernel to continue the work (lines 07–08). The child kernel is executed by grids of threads called the child grids, which perform the work (line 18, Fig. 21.5) that was originally performed inside the loop body (lines 07–09, Fig. 21.4).

Writing the program in this way addresses both problems that were mentioned about the original code. First, the loop iterations are now executed in parallel by the child kernel threads instead of serially by the original kernel thread. Thus we have extracted more parallelism from the program. Second, each thread now executes a single loop iteration, which results in better load balance and eliminates

```
01    __global__ void kernel_parent(unsigned int* start, unsigned int* end,
02        float* someData, float* moreData) {
03
04        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
05        doSomeWork(someData[i]);
06
07        kernel_child <<< ceil((end[i]-start[i])/256.0), 256 >>>
08            (start[i], end[i], moreData);
09
10    }
11
12    __global__ void kernel_child(unsigned int start, unsigned int end,
13        float* moreData) {
14
15        unsigned int j = start + blockIdx.x*blockDim.x + threadIdx.x;
16
17        if(j < end) {
18            doMoreWork(moreData[j]);
19        }
20
21    }
```

FIGURE 21.5

A revised example using CUDA dynamic parallelism.

control divergence. Although these two goals could have been achieved by the programmer rewriting the kernels differently, for example, by using an edge-centric breadth-first implementation, such transformations for some applications can be awkward, complicated, and error prone. Dynamic parallelism provides an easy way to express such computational patterns.

21.3 An example: Bezier curves

We now show an example that is a more interesting and useful case: the adaptive subdivision of spline curves. This example illustrates the case of having a variable amount of child grid launches, according to the workload. The example is to calculate Bezier curves (Bezier Curves), which are frequently used in computer graphics to draw smooth, intuitive curves that are defined by a set of *control points*, which are typically defined by a user.

Mathematically, a Bezier curve is defined by a set of control points $\mathbf{P}_0$ through $\mathbf{P}_n$, where n is called the control point's *order* (n=1 for linear, 2 for quadratic, 3 for cubic, etc.). The first and last control points are always the end points of the curve; however, the intermediate control points (if any) generally do not lie on the curve.

Linear Bezier curves

Given two control points $\mathbf{P}_0$ and $\mathbf{P}_1$, a linear Bezier curve is simply a straight line connecting these two points. The coordinates of the points on the curve are given by the following linear interpolation formula:

$$\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1, \quad t \in [0, \quad 1]$$

Quadratic Bezier curves

A quadratic Bezier curve is defined by three control points $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$. The points on a quadratic curve are defined as a linear interpolation of corresponding points on the linear Bezier curves from $\mathbf{P}_0$ to $\mathbf{P}_1$ and from $\mathbf{P}_1$ to $\mathbf{P}_2$. The calculation of the coordinates of points on the curve is expressed by the following formula:

$$\mathbf{B}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_0 + t\mathbf{P}_2], \quad t \in [0, \quad 1],$$

which can be simplified into the following formula:

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2, \quad t \in [0, \quad 1].$$

Bezier curve calculation (without dynamic parallelism)

Fig. 21.6 shows part of a CUDA C program that calculates the coordinates of points on a Bezier curve. The `computeBezierLines` kernel starting at line 13 is

```
01  #include <stdio.h>
02  #include <cuda.h>
03
04  #define MAX_TESS_POINTS 32
05
06  // A structure containing all parameters needed to tessellate a Bezier line
07  struct BezierLine {
08      float2 CP[3];                       //Control points for the line
09      float2 vertexPos[MAX_TESS_POINTS]; //Vertex position array to tessellate into
10      int nVertices;                      //Number of tessellated vertices
11  };
12
13  __global__ void computeBezierLines(BezierLine *bLines, int nLines) {
14      int bidx = blockIdx.x;
15      if(bidx < nLines){
16          //Compute the curvature of the line
17          float curvature = computeCurvature(bLines);
18
19          //From the curvature, compute the number of tessellation points
20          int nTessPoints = min(max((int)(curvature*16.0f),4),32);
21          bLines[bidx].nVertices = nTessPoints;
22
23          //Loop through vertices to be tessellated, incrementing by blockDim.x
24          for(int inc = 0; inc < nTessPoints; inc += blockDim.x){
25              int idx = inc + threadIdx.x;  //Compute a unique index for this point
26              if(idx < nTessPoints){
27                  float u = (float)idx/(float)(nTessPoints-1);  //Compute u from idx
28                  float omu = 1.0f - u;   //pre-compute one minus u
29                  float B3u[3]; //Compute quadratic Bezier coefficients
30                  B3u[0] = omu*omu;
31                  B3u[1] = 2.0f*u*omu;
32                  B3u[2] = u*u;
33                  float2 position = {0,0};  //Set position to zero
34                  for(int i = 0; i < 3; i++){
35                      //Add the contribution of the i'th control point to position
36                      position = position + B3u[i] * bLines[bidx].CP[i];
37                  }
38                  //Assign value of vertex position to the correct array element
39                  bLines[bidx].vertexPos[idx] = position;
40              }
41          }
42      }
43  }
```

FIGURE 21.6

Bezier curve calculation without dynamic parallelism.

designed to use a block to calculate the curve points for a set of three control points (of the quadratic Bezier formula). Each thread block first computes a measure of the curvature of the curve defined by the three control points. Intuitively, the larger the curvature, the more points it takes to draw a smooth quadratic Bezier curve for the three control points. This defines the amount of work to be done by each block. This is reflected in lines 20 and 21, in which the total number of points to be calculated by the current thread block is proportional to the curvature value.

In the for-loop in line 24, all threads calculate a consecutive set of Bezier curve points in each iteration. The detailed calculation in the loop body is based on the formula that we presented earlier. The key point is that the number of iterations taken by threads in a block can be very different from the number taken by threads in another block. Depending on the scheduling policy, such variation of the amount of work done by each thread block can result in decreased utilization of streaming multiprocessors (SMs) and thus reduced performance.

Bezier curve calculation (with dynamic parallelism)

Fig. 21.7 shows a Bezier curve calculation code using dynamic parallelism. It breaks the `computeBezierLines` kernel in Fig. 21.6 into two kernels. The first part, `computeBezierLines_parent`, discovers the amount of work to be done for each control point. The second part, `computeBezierLines_child`, performs the calculation.

With the new organization the amount of work that is done for each set of control points by the `computeBezierLines_parent` kernel is much smaller than that for the original `computeBezierLines` kernel. Therefore we use one thread to do this work in `computeBezierLines_parent` rather than using one block in `computeBezierLines`. In line 58, we need to launch only one thread per set of control points. This is reflected by dividing the `N_LINES` by `BLOCK_DIM` to form the number of blocks in the kernel launch configuration.

```
01  struct BezierLine {
02      float2 CP[3];      //Control points for the line
03      float2 *vertexPos; //Vertex position array to tessellate into
04      int nVertices;     //Number of tessellated vertices
05  };
06  __global__ void computeBezierLines_parent(BezierLine *bLines, int nLines) {
07      //Compute a unique index for each Bezier line
08      int lidx = threadIdx.x + blockDim.x*blockIdx.x;
09      if(lidx < nLines){
10          //Compute the curvature of the line
11          float curvature = computeCurvature(bLines);
12
13          //From the curvature, compute the number of tessellation points
14          bLines[lidx].nVertices = min(max((int)(curvature*16.0f),4),MAX_TESS_POINTS);
15          cudaMalloc((void**)&bLines[lidx].vertexPos,
16              bLines[lidx].nVertices*sizeof(float2));
17
18          //Call the child kernel to compute the tessellated points for each line
19          computeBezierLine_child<<<ceil((float)bLines[lidx].nVertices/32.0f), 32>>>
20              (lidx, bLines, bLines[lidx].nVertices);
21      }
22  }
23  __global__ void computeBezierLine_child(int lidx, BezierLine* bLines,
24      int nTessPoints) {
25      int idx = threadIdx.x + blockDim.x*blockIdx.x;//Compute idx unique to this vertex
26      if(idx < nTessPoints){
27          float u = (float)idx/(float)(nTessPoints-1);  //Compute u from idx
28          float omu = 1.0f - u;   //Pre-compute one minus u
29          float B3u[3];   //Compute quadratic Bezier coefficients
30          B3u[0] = omu*omu;
31          B3u[1] = 2.0f*u*omu;
32          B3u[2] = u*u;
33          float2 position = {0,0};  //Set position to zero
34          for(int i = 0; i < 3; i++) {
35              //Add the contribution of the i'th control point to position
36              position = position + B3u[i] * bLines[lidx].CP[i];
37          }
38          //Assign the value of the vertex position to the correct array element
39          bLines[lidx].vertexPos[idx] = position;
40      }
41  }
42    __global__ void freeVertexMem(BezierLine *bLines, int nLines) {
43      //Compute a unique index for each Bezier line
44      int lidx = threadIdx.x + blockDim.x*blockIdx.x;
45      if(lidx < nLines)
46          cudaFree(bLines[lidx].vertexPos);   //Free the vertex memory for this line
47  }
```

FIGURE 21.7

Bezier calculation with dynamic parallelism.

There are two key differences between the computeBezierLines_parent kernel and the computeBezierLines kernel. First, the index that is used to access the control points is formed on a thread basis (line 08 in Fig. 21.7) rather than a block basis (line 14 in Fig. 21.6). This is because the work for each control point is done by a thread rather than a block, as we mentioned earlier in the chapter. Second, the memory for storing the calculated Bezier curve points is dynamically determined and allocated in line 15 in Fig. 21.7. This allows the code to assign just enough memory to each set of control points to be placed in the bLines variable. Note that in Fig. 21.6, each BezierLine element is declared with the maximum possible number of points (line 09). On the other hand, the declaration in Fig. 21.7 has only a pointer to a dynamically allocated storage. Allowing a kernel to call the cudaMalloc() function can lead to substantial reduction of memory usage for situations in which the curvature of control points varies significantly.

Once a thread of the computeBezierLines_parent kernel has determined the amount of work needed by its set of control points, it calls the computeBezierLines_child kernel and launches a child grid to do the work (line 19 in Fig. 21.7). In our example, every thread from the parent grid creates a new grid for its assigned set of control points. This way, the work that is done by each thread block is balanced. The amount of work that is done by each child grid varies.

Dynamic parallelism kernel calls have the same asynchronous semantics as kernel calls from the host. That is, once the child kernel has been called, the parent thread that called it may proceed to execute the subsequent code without waiting for the child grid to complete. If the parent thread wishes to wait for its child grid to complete, it must perform an explicit synchronization similar to what a host thread would do. If no explicit synchronization is made, then the thread can finish executing, but there is an implicit synchronization at the end. The implicit synchronization ensures that all child grids have terminated before the parent grid terminates. We refer the reader to the CUDA C++ Programming Guide for more details about the semantics of parent-child grid synchronization (NVIDIA Corporation, 2021).

After computeBezierLines_parent terminates, the memory allocated inside the kernel using cudaMalloc still needs to be freed. For this reason, an additional kernel freeVertexMem is implemented (lines 42–47) to be called by the host. This kernel frees all storage allocated to the vertices in the bLines_d data structure in parallel (line 61). This kernel is necessary because vertex storage allocated on the device by a device kernel has to be freed by a device kernel. We refer the reader to the CUDA C++ Programming Guide for more details about the use of cudaMalloc and cudaFree on the device (NVIDIA Corporation, 2021).

21.4 A recursive example: quadtrees

Dynamic parallelism also allows programmers to implement recursive algorithms. In this section we illustrate the use of dynamic parallelism for implementing recursion with a *quadtree* (Finkel and Bentley, 1974). Quadtrees partition a two-dimensional space by recursively subdividing it into four equally sized *quadrants*. Each quadrant

is considered to be a *node* of the quadtree and contains a number of points. If the number of points in a quadrant is greater than a fixed minimum, the quadrant will be recursively subdivided into four more quadrants, that is, four child nodes.

Fig. 21.8 illustrates the construction of a quadtree with dynamic parallelism. In this implementation, one node (quadrant) is assigned to one block. Initially (depth=0), one block (block 0) is assigned the entire two-dimensional space (root node of the

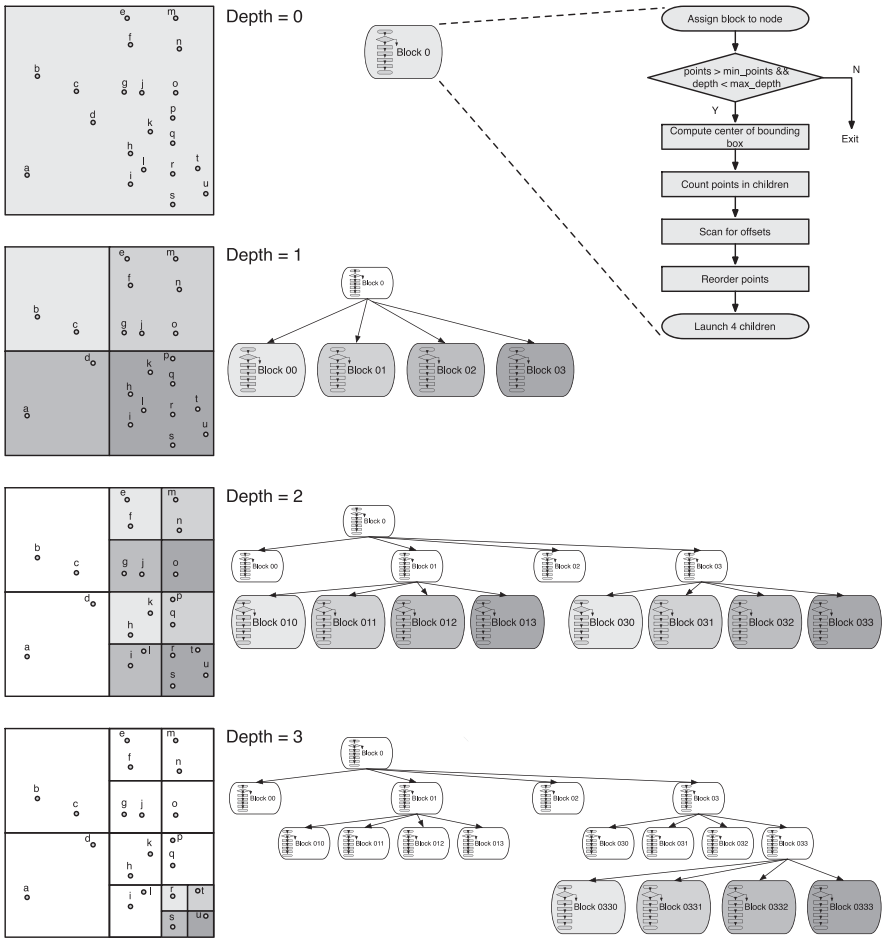


FIGURE 21.8

Quadtree example. Each block is assigned to one quadrant. If the number of points in a quadrant is more than two, the block launches four child blocks. Shadowed blocks are active blocks in each level of depth.

quadtree), which contains all points. It divides the space into four quadrants and launches one block for each quadrant (depth=1). These child blocks (blocks 00 through 03) will again subdivide their quadrants if they contain more points than a fixed minimum. In this example we assume that the minimum is two; thus blocks 00 and 02 do not launch children. Blocks 01 and 03 launch a grid with four blocks each.

As the flow graph in the right-hand side of Fig. 21.8 shows, a block first checks whether the number of points in its quadrant is greater than the minimum required for further division and whether the maximum depth has not been reached. If either of the conditions fails, the work for the quadrant is complete, and the block returns. Otherwise, the block computes the center of the bounding box that surrounds its quadrant. The center is in the middle of four new quadrants. The number of points in each of them is counted. A four-element scan operation is used to compute the offsets to the locations where the points will be stored. Then the points are reordered so that those points in the same quadrant are grouped together and placed into their section of the point storage. Finally, the block launches a child grid with four blocks, one for each of the four new quadrants.

Fig. 21.9 continues the small example in Fig. 21.8 and illustrates in detail how the points are reordered at each level of depth. For the example we assume that each quadrant must have more than two points to be further divided. The algorithm uses two buffers to store the points and reorder them. The points should be in buffer 0 at the end of the algorithm. Thus it might be necessary to swap the buffer contents before leaving if the points are in buffer 1 when the terminating condition is met.

In the initial kernel call from the host code (for depth=0), block 0 is assigned all the points that reside in buffer 0, shown in Fig. 21.9A. Block 0 further divides

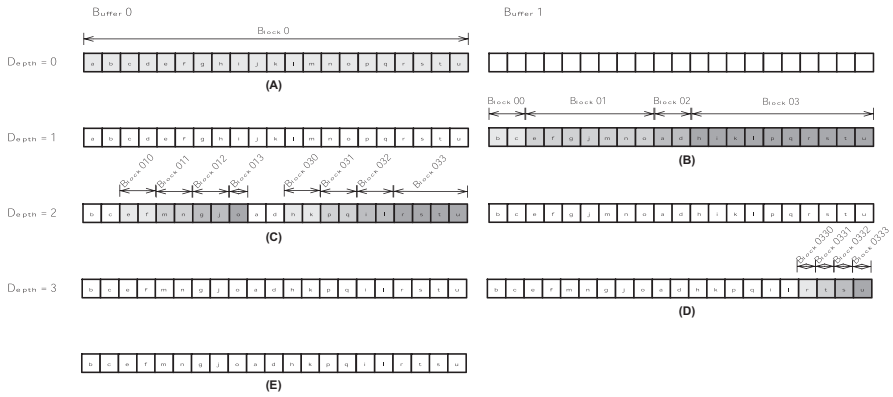


FIGURE 21.9

Quadtree example. At each level of depth, a block groups all points in the same quadrant together. (A) shows the initial input list in Buffer 0, (B) The list after being rearranged into four sublists that correspond to the four quadrants, (C) shows the list after being rearranged to reflect the second-level quadrants, (D) shows the list after being rearranged to reflect the third-level quadrant, (E) the final list is copied into Buffer 0 for return to the caller.

the quadrant into four child quadrants, groups together all points in the same child quadrant, and stores the points according to the quadrants into buffer 1, as shown in Fig. 21.9B. Its four children, block 00 to block 03, are assigned each of the four new quadrants, shown as marked ranges in Fig. 21.9B. Blocks 00 and 02 will not launch children, since the number of points in their respective assigned quadrant is only 2. Blocks 01 and 03 reorder their points to group those in the same quadrant and launch four child blocks each, as shown in Fig. 21.9C. Blocks 010, 011, 012, 013, 030, 031, and 032 do not launch children (they have two or fewer points) and do not need to swap points (they are already in buffer 0). Only block 033 reorders its points and launches four blocks, as shown in Fig. 21.9D. Blocks 0330 to 0333 will exit after swapping their points to buffer 0, which can be seen in Fig. 21.9E.

The kernel code in Fig. 21.10 implements the flow graph from Fig. 21.8. The quadtree is implemented with a node array, in which each element contains all the

```
01    __global__ void build_quadtree_kernel
02                  (Quadtree_node *nodes, Points *points, Parameters params) {
03        __shared__ int smem[8]; // To store the number of points in each quadrant
04
05        // The current node in the quadtree
06        Quadtree_node &node = nodes[blockIdx.x];
07        node.set_id(node.id() + blockIdx.x);
08        int num_points = node.num_points(); // The number of points in the node
09
10        // Check the number of points and its depth
11        bool exit = check_num_points_and_depth(node, points, num_points, params);
12        if(exit) return;
13
14        // Compute the center of the bounding box of the points
15        const Bounding_box &bbox = node.bounding_box();
16        float2 center;
17        bbox.compute_center(center);
18
19        // Range of points
20        int range_begin = node.points_begin();
21        int range_end   = node.points_end();
22        const Points &in_points = points[params.point_selector]; // Input points
23        Points &out_points = points[(params.point_selector+1) % 2]; // Output points
24
25        // Count the number of points in each child
26        count_points_in_children(in_points, smem, range_begin, range_end, center);
27
28        // Scan the quadrants' results to know the reordering offset
29        scan_for_offsets(node.points_begin(), smem);
30
31        // Move points
32        reorder_points(out_points, in_points, smem, range_begin, range_end, center);
33
34        // Launch new blocks
35        if (threadIdx.x == blockDim.x-1) {
36            // The children
37            Quadtree_node *children = &nodes[params.num_nodes_at_this_level];
38
39            // Prepare children launch
40            prepare_children(children, node, bbox, smem);
41
42            // Launch 4 children.
43            build_quadtree_kernel<<<4, blockDim.x, 8 *sizeof(int)>>>
44                                (children, points, Parameters(params, true));
45        }
46    }
```

FIGURE 21.10

Quadtree with dynamic parallelism: recursive kernel (support code in Appendix A210.1).

pertinent information for one node of the quadtree (definition given in Appendix A21.1). As the quadtree is constructed, new nodes will be created and placed into the array during the execution of the kernels. The kernel code assumes that the `node` parameter points to the next available location in the node array.

Every block starts by checking the number of points in its node (quadrant). A point is a pair of floats representing `x` and `y` coordinates (definition in Appendix A21.1). If the number of points is less than or equal to the minimum or if the maximum depth is reached (line 11), the block will exit. The maximum depth may be specified on the basis of application requirements or hardware constraints (see Section 12.5). Before exiting, the block may need to write its points from buffer 1

```
001 // Check the number of points and its depth
002 __device__ bool check_num_points_and_depth(Quadtree_node &node, Points *points,
003                                             int num_points, Parameters params){
004     if(params.depth >= params.max_depth || num_points <= params.min_points_per_node) {
005     // Stop the recursion here. Make sure points[0] contains all the points
006       if(params.point_selector == 1) {
007         int it = node.points_begin(), end = node.points_end();
008         for (it += threadIdx.x ; it < end ; it += blockDim.x)
009           if(it < end)
010             points[0].set_point(it, points[1].get_point(it));
011       }
012       return true;
013     }
014     return false;
015 }
016
017 // Count the number of points in each quadrant
018 __device__ void count_points_in_children(const Points &in_points, int* smem,
019     int range_begin, int range_end, float2 center) {
020     // Initizalize shared memory
021     if(threadIdx.x < 4) smem[threadIdx.x] = 0;
022       syncthreads();
023     // Compute the number of points
024     for(int iter=range begin+threadIdx.x; iter<range end; iter+=blockDim.x){
025         float2 p = in points.get point(iter); // Load the coordinates of the point
026         if(p.x < center.x && p.y >= center.y)
027             atomicAdd(&smem[0], 1); // Top-left point?
028         if(p.x >= center.x && p.y >= center.y)
029             atomicAdd(&smem[1], 1); // Top-right point?
030         if(p.x < center.x && p.y < center.y)
031             atomicAdd(&smem[2], 1); // Bottom-left point?
032         if(p.x >= center.x && p.y < center.y)
033             atomicAdd(&smem[3], 1); // Bottom-right point?
034     }
035     __syncthreads();
036 }
037
038 // Scan quadrants' results to obtain reordering offset
039 __device__ void scan_for_offsets(int node_points_begin, int* smem){
040     int* smem2 = &smem[4];
041     if(threadIdx.x == 0){
042     for(int i = 0; i < 4; i++)
043         smem2[i] = i==0 ? 0 : smem2[i-1] + smem[i-1]; // Sequential scan
044     for(int i = 0; i < 4; i++)
045         smem2[i] += node_points_begin;  // Global offset
046     }
047     __syncthreads();
048 }
049
050 // Reorder points in order to group the points in each quadrant
051 __device__ void reorder_points(
```

FIGURE 21.11

Quadtree with dynamic parallelism: device functions (support code in Appendix A21.1).

```
052                 Points& out_points, const Points &in_points, int* smem,
053                 int range_begin, int range_end, float2 center){
054     int* smem2 = &smem[4];
055     // Reorder points
056     for(int iter=range_begin+threadIdx.x; iter<range_end; iter+=blockDim.x){
057         int dest;
058         float2 p = in_points.get_point(iter); // Load the coordinates of the point
059         if(p.x<center.x && p.y>=center.y)
060             dest=atomicAdd(&smem2[0],1); // Top-left point?
061         if(p.x>=center.x && p.y>=center.y)
062             dest=atomicAdd(&smem2[1],1); // Top-right point?
063         if(p.x<center.x && p.y<center.y)
064             dest=atomicAdd(&smem2[2],1); // Bottom-left point?
065         if(p.x>=center.x && p.y<center.y)
066             dest=atomicAdd(&smem2[3],1); // Bottom-right point?
067         // Move point
068         out_points.set_point(dest, p);
069     }
070     __syncthreads();
071 }
072
073 // Prepare children launch
074 __device__ void prepare_children(Quadtree_node *children, Quadtree_node &node,
075                                  const Bounding_box &bbox, int *smem){
076     int child_offset = 4*node.id(); // The offsets of the children at their level
077
078     // Set IDs
079     children[child_offset+0].set_id(4*node.id()+ 0);
080     children[child_offset+1].set_id(4*node.id()+ 4);
081     children[child_offset+2].set_id(4*node.id()+ 8);
082     children[child_offset+3].set_id(4*node.id()+12);
083
084     // Points of the bounding-box
085     const float2 &p_min = bbox.get_min();
086     const float2 &p_max = bbox.get_max();
087
088     // Set the bounding boxes of the children
089     children[child_offset+0].set_bounding_box(
090         p_min.x , center.y, center.x, p_max.y);    // Top-left
091     children[child_offset+1].set_bounding_box(
092         center.x, center.y, p_max.x , p_max.y);    // Top-right
093     children[child_offset+2].set_bounding_box(
094         p_min.x , p_min.y , center.x, center.y);   // Bottom-left
095     children[child_offset+3].set_bounding_box(
096         center.x, p_min.y , p_max.x , center.y);   // Bottom-right
097
098     // Set the ranges of the children.
099     children[child_offset+0].set_range(node.points_begin(),   smem[4 + 0]);
100     children[child_offset+1].set_range(smem[4 + 0], smem[4 + 1]);
101     children[child_offset+2].set_range(smem[4 + 1], smem[4 + 2]);
102     children[child_offset+3].set_range(smem[4 + 2], smem[4 + 3]);
103 }
```

FIGURE 21.11

(Continued)

to buffer 0 if necessary. This is because the output is expected in buffer 0 after the quadtree is complete. The transfer of points from buffer 1 to buffer 0 is done in the device function `check_num_points_and_depth()` shown in Fig. 21.11.

Next, the center of the bounding box is computed (line 17). A bounding box is defined by its top-left and bottom-right corners. The coordinates of the top-left and the bottom-right corner points of the bounding box for the current node are given as part of the node data by the caller. The coordinates of the center are computed as the coordinates of the middle point between these two corner points. The definition of a bounding box (including function `compute_center()`) is given in Appendix A21.1.

As the center separates the four quadrants, one can determine which quadrant each point in the current node belongs to by comparing it with the center point. The number of points in each quadrant are counted (line 26). The device function `count_points_in_children()` can be found in Fig. 21.11. These are simplified for clarity reasons. The threads of the block collaboratively go through the range of points and use atomic operations to update the counters in shared memory for each quadrant.

The device function `scan_for_offsets()` is called then (line 29). As can be seen in Fig. 21.11, it performs a scan on the four counters in shared memory. Then it adds the global offset of the parent quadrant to these values to derive the starting offset for each quadrant's group in the buffer.

Using the quadrants' offsets, the points are reordered with `reorder_points()` (line 32). For simplicity this device function (Fig. 21.11) uses an atomic operation on one of the four quadrant counters to derive the location for placing each point.

Finally, the last thread of the block (line 35) determines the next available location in the node array for children nodes (line 37), prepares the new node contents for the child quadrants (line 40), and launches one child kernel with four thread blocks (line 43). The device function `prepare_children()` prepares the new node contents for the children by setting the limits of the children's bounding boxes and the range of points in each quadrant. The `prepare_children()` function can be found in Figure 13.14 (line 75).

The rest of the definitions can be found in Appendix A21.1.

21.5 Important considerations

In this section we will briefly explain some important considerations for the execution behavior of programs that use dynamic parallelism. It is important for a programmer to understand these considerations well in order to use dynamic parallelism confidently.

Memory and data visibility

When a parent thread passes a memory pointer to a child grid, it must ensure that the memory being pointed to is accessible to the child grid so that the child grid does not attempt to access invalid memory. The memory that can be accessed by both parent threads and their child grids includes global memory, constant memory, and texture memory. A parent thread should not pass pointers to local memory or shared memory to their child grids because local memory and shared memory are private to the thread and the thread block, respectively.

Besides ensuring that the memory that is passed by the parent thread to the child grid is accessible to the child grid, programmers must also be aware of when the data written to that memory by a parent thread will be visible to the

child grid and vice versa. There are two points in the execution at which the parent thread and the child grid have the same view of memory: when the parent thread launches the child grid and when the child grid completes as signaled by the completion of a synchronization API call by the parent thread. In other words, when a parent thread launches a child grid, any updates to memory prior to that launch will be seen by the child grid, but there is no such guarantee for updates after that launch. Also, there is no guarantee that any updates to memory made by the child grid will be visible to the parent thread until after the parent thread has synchronized on the completion of the child grid.

We refer the reader to the CUDA C++ Programming Guide for more details about memory and data visibility between parent threads and child grids (NVIDIA Corporation, 2021).

Pending launch pool configuration

The pending launch pool is a buffer that tracks the kernels that are executing or waiting to be executed. This pool is allocated a fixed amount of space, thereby supporting a fixed number of pending kernel calls (2048 by default). If this number is exceeded, a virtualized pool is used, leading to a significant slowdown, which can be an order of magnitude or more. To avoid this slowdown, the programmer can increase the size of the fixed pool by executing the `cudaDeviceSetLimit()` API call from the host function to set the `cudaLimitDevRuntimePendingLaunchCount` configuration.

For example, in the Bezier curve calculation in Section 21.3, if `N_LINES` is set to 4096, then 4096 child grids will be launched, so half of the launches will use the virtualized pool. This will incur a significant performance penalty. However, if the fixed-size pool is set to 4096, the execution time will be reduced substantially.

As a general recommendation, the size of the fixed-size pool should be set to the expected number of launched grids (if it exceeds the default size). In the Bezier curves example we would call `cudaDeviceSetLimit (cudaLimitDevRuntimePendingLaunchCount, N_LINES)` before calling the `computeBezierLines_parent()` kernel.

Streams

Just as host code can use streams to execute kernels concurrently, device threads can use streams in launching grids with dynamic parallelism. The scope of a stream is private to the block in which the stream was created. When a stream is not specified in a kernel call, the default NULL stream in the block is used by all threads. This means that all grids that are launched in the same block will be serialized even if they were launched by different threads. However, it is often the case that grids launched by different threads in a block are independent and can be executed concurrently. Therefore programmers must be careful to explicitly use different streams in each thread if they wish to avoid the performance penalty from serialization.

```
// Create non-blocking stream
cudaStream_t stream;
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);

//Call the child kernel to compute the tessellated points for each line
computeBezierLine_child<<<ceil((float)bLines[lidx].nVertices/32.0f), 32, 0, stream>>>
    (lidx, bLines, bLines[lidx].nVertices);

// Destroy stream
cudaStreamDestroy(stream);
```

FIGURE 21.12

Child kernel launch with named streams.

The Bezier curves example in Section 21.3 launches as many child grids as there are parent threads in the `computeBezierLines_parent()` kernel (line 19 in Fig. 21.7). If the default NULL stream is used, all the grids that are launched by the same parent block will be serialized. Thus using the default NULL stream in launching the `computeBezierLines_child()` kernel can result in a drastic reduction in parallelism compared to the original kernel without dynamic parallelism.

If more concurrency is desired, named streams must be created and used in each thread. Fig. 21.12 shows the code that should replace line 19 in Fig. 21.7. With this code, kernels that are launched from the same thread block will be placed in different streams and can run concurrently. This would better utilize all SMs, leading to a considerable reduction in the execution time.

Nesting depth

Kernels that are launched with dynamic parallelism may themselves call other kernels, which may in turn call other kernels, and so on. We saw an example of such a kernel in the quadtree application in Section 12.4. Each subordinate launch is considered a new *nesting level*, and the total number of levels that are reached is called the *nesting depth*. The maximum nesting depth supported by current hardware is 24 levels. For this reason, kernels such as the one in the quadtree example should check this limit before deciding whether to make a dynamic launch.

In the presence of parent-child synchronization there are additional constraints on the nesting depth due to the amount of memory required by the system to store the state of the parent grid. This constraint is referred to as the *synchronization depth*. We refer the reader to the CUDA C++ Programming Guide for more details about the nesting depth and synchronization depth (NVIDIA Corporation, 2021).

21.6 Summary

CUDA dynamic parallelism extends the CUDA programming model to allow kernels to call other kernels. This allows each thread to dynamically discover work and launch new grids according to the amount of work that is newly discovered.

It also supports dynamic allocation of device memory by threads. As we showed in the Bezier curve calculation example, these extensions can lead to better work balance across threads and blocks as well as more efficient memory usage. CUDA Dynamic Parallelism also helps programmers to implement recursive algorithms, as the quadtree example shows.

Besides ensuring better work balance, dynamic parallelism offers many advantages in terms of programmability. However, it is important to keep in mind that launching grids with a small number of threads could lead to severe underutilization of the GPU resources. A general recommendation is launching child grids with many blocks, or at least blocks with many threads if the number of blocks is small.

Similarly, nested parallelism, which can be seen as a form of tree processing, provides higher performance when tree nodes are thick (that is, each node deploys many threads) and/or when the branch degree is large (i.e., each parent node has many children). As the nesting depth is limited in hardware, only relatively shallow trees can be implemented efficiently.

To use dynamic parallelism effectively, programmers need to understand details such as visibility of memory contents, pending launch count, and streams. Careful use of memory and streams between parents and children in dynamic parallelism is critical for the correct execution and achieving the expected level of parallelism in launching child grids.

Exercises

1. Which of the following statements are true for the Bezier curves example?
   a. If `N_LINES`=1024 and `BLOCK_DIM`=64, the number of child kernels that are launched will be 16.
   b. If `N_LINES`=1024, the fixed-size pool should be reduced from 2048 (the default) to 1024 to get the best performance.
   c. If `N_LINES`=1024 and `BLOCK_DIM`=64 and per-thread streams are used, a total of 16 streams will be deployed.
2. Consider a two-dimensional organization of 64 equidistant points. It is classified with a quadtree. What will be the maximum depth of the quadtree (including the root node)?
   a. 21
   b. 4
   c. 64
   d. 16
3. For the same quadtree, what will be the total number of child kernel launches?
   a. 21
   b. 4

c. 64
d. 16

4. **True or False:** Parent kernels can define new `__constant__` variables that will be inherited by child kernels.
5. **True or False:** Child kernels can access their parents' shared and local memories.
6. Six blocks of 256 threads run the following parent kernel:

```
__global__ void parent_kernel(int *output, int *input, int *size) {
  // Thread index
  int idx = threadIdx.x + blockDim.x*blockIdx.x;

  // Number of child blocks
  int numBlocks = size[idx] / blockDim.x;

  // Launch child
  child_kernel<<< numBlocks, blockDim.x >>>(output, input, size);

}
```

How many child kernels could run concurrently?
a. 1536
b. 256
c. 6
d. 1

A21.1 Support code for quadtree example

```
001    // A structure of 2D points
002    class Points {
003        float *m x;
004        float *m y;
005
006        public:
007        // Constructor
008          host     device   Points() : m x(NULL), m y(NULL) {}
009
010        // Constructor
011          host     device   Points(float *x, float *y) : m x(x), m y(y) {}
012
013        // Get a point
014          host     device     forceinline   float2 get point(int idx) const {
015            return make float2(m x[idx], m y[idx]);
016        }
017
018        // Set a point
019        __host__ __device__ __forceinline__ void set_point(int idx, const float2 &p) {
020            m_x[idx] = p.x;
021            m_y[idx] = p.y;
022        }
023
024        // Set the pointers
025        __host__ __device__ __forceinline__ void set(float *x, float *y) {
026            m_x = x;
027            m_y = y;
028        }
029    };
030
031    // A 2D bounding box
032    class Bounding box {
033        // Extreme points of the bounding box
034        float2 m p min;
035        float2 m p max;
036
037        public:
038        // Constructor. Create a unit box
039          host     device   Bounding box(){
040            m p min = make float2(0.0f, 0.0f);
041            m p max = make float2(1.0f, 1.0f);
042        }
043
044        // Compute the center of the bounding-box
045        __host__ __device__ void compute_center(float2 &center) const {
046            center.x = 0.5f * (m_p_min.x + m_p_max.x);
047            center.y = 0.5f * (m_p_min.y + m_p_max.y);
048        }
049
050        // The points of the box
051        __host__ __device__ __forceinline__ const float2 &get_max() const {
052            return m_p_max;
053        }
054
055        __host__ __device__ __forceinline__ const float2 &get_min() const {
056            return m_p_min;
057        }
058
059        // Does a box contain a point
060          host     device   bool contains(const float2 &p) const {
061            return p.x>=m p min.x && p.x<m p max.x && p.y>=m p min.y && p.y<m p max.y;
062        }
063
```

```
064         // Define the bounding box
065               host     device   void set(float min x, float min y, float max x, float
max y){
066             m p min.x = min x;
067             m p min.y = min y;
068             m p max.x = max x;
069             m p max.y = max y;
070         }
071     };
072
073     // A node of a quadree
074     class Quadtree_node {
075         // The identifier of the node
076         int m_id;
077         // The bounding box of the tree
078         Bounding_box m_bounding_box;
079         // The range of points
080         int m_begin, m_end;
081
082         public:
083         // Constructor
084         __host__ __device__ Quadtree_node() : m_id(0), m_begin(0), m_end(0) {}
085
086         // The ID of a node at its level
087         __host__ __device__ int id() const {
088             return m_id;
089         }
090
091         // The ID of a node at its level
092           host     device   void set id(int new id) {
093             m id = new id;
094         }
095
096         // The bounding box
097           host     device     forceinline   const Bounding box &bounding box() const {
098             return m bounding box;
099         }
100
101         // Set the bounding box
102         __host__ __device__ __forceinline__ void set_bounding_box(float min_x,
103             float min_y, float max_x, float max_y) {
104             m_bounding_box.set(min_x, min_y, max_x, max_y);
105         }
106
107         // The number of points in the tree
108         __host__ __device__ __forceinline__ int num_points() const {
109             return m_end - m_begin;
110         }
111
112         // The range of points in the tree
113         __host__ __device__ __forceinline__ int points_begin() const {
114             return m_begin;
115         }
116
117         __host__ __device__ __forceinline__ int points_end() const {
118             return m_end;
119         }
120
121         // Define the range for that node
122           host     device     forceinline   void set range(int begin, int end) {
123             m begin = begin;
124             m end = end;
125         }
126     };
127
128     // Algorithm parameters
129     struct Parameters {
```

```
130         // Choose the right set of points to use as in/out
131         int point selector;
132         // The number of nodes at a given level (2^k for level k)
133         int num nodes at this level;
134         // The recursion depth
135         int depth;
136         // The max value for depth
137         const int max depth;
138         // The minimum number of points in a node to stop recursion
139         const int min points per node;
140
141         // Constructor set to default values.
142         __host__ __device__ Parameters(int max_depth, int min_points_per_node) :
143             point_selector(0),
144             num_nodes_at_this_level(1),
145             depth(0),
146             max_depth(max_depth),
147             min_points_per_node(min_points_per_node) {}
148
149         // Copy constructor. Changes the values for next iteration
150         __host__ __device__ Parameters(const Parameters &params, bool) :
151             point_selector((params.point_selector+1) % 2),
152             num_nodes_at_this_level(4*params.num_nodes_at_this_level),
153             depth(params.depth+1),
154             max_depth(params.max_depth),
155             min_points_per_node(params.min_points_per_node) {}
156     };
```

References

Bezier Curves, http://en.wikipedia.org/wiki/B%C3%A9zier_curve.

Finkel, R.A., Bentley, J.L., 1974. Quad trees: a data structure for retrieval on composite keys. Acta Inform. 4 (1), 1–9.

NVIDIA Corporation, 2021. CUDA C++ programming guide, March.

CHAPTER

Advanced practices and future evolution

22

With special contributions from Isaac Gelado and Mark Harris

Chapter Outline

Our focus throughout this book has been on scalable parallel programming. CUDA C and the GPU hardware have mostly played the role of the programming platform for our examples and exercises. However, the parallel programming concepts and skills that are learned on the basis of CUDA C can be easily adapted for other parallel programming platforms. For example, as we saw in Chapter 20, Programming a Heterogeneous Computing Cluster, Programming a Heterogeneous Computing Cluster: An Introduction to CUDA Streams, most key concepts of Message Passing Interface (MPI), such as processes, rank, and barriers, have counterparts in CUDA C. Furthermore, as we also discussed in Chapter 20, Programming a Heterogeneous Computing Cluster, CUDA-enabled GPUs have become widely available in high-performance computing (HPC) systems. For many readers, CUDA C will likely be an important application development and deployment platform rather than just a learning vehicle. For this reason, it is important for the reader to understand the advanced CUDA C features and practices that are designed to support high-performance programming at the application level. For example, as we saw in Chapter 20, Programming a Heterogeneous Computing Cluster, CUDA streams enable an MPI HPC application to overlap communication with the computation. Such capability is especially important for achieving whole-application performance goals. With this in mind, this chapter will provide the reader with an overview of the advanced features of CUDA C and GPU computing hardware that will be important in achieving high performance and maintainability in your applications. For each feature we will present the basic concepts as well as a brief history of its

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00013-6

evolution through different generations of GPU computing. A good understanding of the concepts and the evolution history will help to clear up some common confusion about these features. The goal is to help the reader to establish a conceptual framework for more detailed studies of these features.

22.1 Model of host/device interaction

So far, we have assumed a fairly simple model of interaction between host and device in a heterogeneous computing system. In this simple model, as presented in Chapter 2, Heterogeneous Data Parallel Computing, Heterogeneous Data Parallel Computing, each device has a device memory (CUDA global memory) that is separate from the host memory, or the system memory. The data to be processed by a kernel running on a device needs to be transferred from the host memory to the device memory by calling the `cudaMemcpy()` function. The data that is produced by the device also needs to be transferred from the device memory to the host by calling the `cudaMemcpy()` function before it can be utilized by the host. While the model is simple and easy to understand, it results in several problems at the application level.

First, I/O devices such as disk controllers and network interface cards are designed to operate efficiently on the host memory. Since the device memory is separate from the host memory, input data need to be transferred from the host memory to the device memory, and output data need to be transferred from the device memory to the host memory to be used in I/O operations. Such additional transfers increase the I/O latency and reduce the achievable I/O throughput. For many applications the ability of I/O devices to operate directly on the device memory would improve overall application performance and simplify the application code.

Second, the host memory is where the traditional programming systems place their application data structures. Some of the data structures are large. The device memory in early generations of CUDA-enabled GPUs were small in comparison to the host memory, which forces application developers to partition their large data structures into chunks that fit into the device memory. For example, in Chapter 18, Electrostatic Potential Map, the three-dimensional (3D) electrostatic energy grid array was partitioned into two-dimensional slices that are transferred between the host memory and the device memory. For many applications it would be much better if the entire data structures could reside in the device memory. For some applications there may not even be a good way to partition the data structure into smaller chunks. For these applications it would be best if the GPU could directly access the data in the host memory or have the CUDA runtime system software migrate the data that is used during kernel execution.

These limitations of the host/device interaction model were rooted in the limitations of the memory architecture of early generations of CUDA-enabled

GPUs. In these early devices, the only viable host/device interaction model for applications was the simple model that we assumed in the previous chapters. As more applications adopted GPU computing, their needs motivated CUDA system software developers and GPU hardware designers to provide a better solution. Researchers have been aware of these needs and have proposed solutions since the early days of CUDA (Gelado et al., 2010). The rest of this section will go over a brief history of advancements that address these limitations.

Zero-copy memory and unified virtual address space

In 2009, CUDA 2.2 introduced zero-copy access to system memory. This enables the host code to supply a special device data pointer to host memory to a kernel. The code that is running on the device can use this pointer to directly access the host memory through the system interconnect, such as the PCIe bus, without calling to `cudaMemcpy()`. Zero-copy memory is pinned host memory (Chapter 20, Programming a Heterogeneous Computing Cluster,) and is allocated by calling `cudaHostAlloc()` with `cudaHostAllocMapped` as the value of the flag argument. We mentioned that the other values of the flag argument are for more advanced usage. The data pointer that is returned by `cudaHostAlloc()` cannot be directly passed to the kernel; the host code has to obtain a valid device data pointer, using `cudaHostGetDevicePointer()` first, and pass the device data pointer that is returned by this function to the kernel. This means that different data pointers for host and device codes are used to access the same physical memory.

As we explained in Chapter 20, Programming a Heterogeneous Computing Cluster, the host memory pages must be pinned to prevent the operating system from accidentally paging out the data while the GPU is accessing them. Obviously, the access will suffer from the long latency and limited bandwidth of the system interconnect. The bandwidth of the system interconnect is typically less than 10% of the global memory bandwidth. As we learned in Chapter 5, Memory Architecture and Data Locality, a kernel's performance is typically limited by the global memory bandwidth unless we use tiling techniques to drastically reduce the number of global memory accesses per floating-point operation performed. If most of the memory accesses of a kernel are to zero-copy memory, the kernel's execution speed can be even more severely limited by the bandwidth of the system interconnect. Therefore one should use zero-copy memory only for application data structures that are occasionally and sparsely accessed by a kernel running on a GPU.

In 2011, CUDA 4 introduced the Unified Virtual Addressing. Until this CUDA release, the host and the device had their own virtual address spaces, each of them mapping host or device data pointers to physical host or device memory locations. These disjoint virtual address spaces imply that the same physical memory location could be accessed by different virtual addresses in the host and the device, which effectively happens in using zero-copy memory. The Unified Virtual Address Space (UVAS), first introduced by the GMAC library (Gelado et al., 2010) and adopted in

CUDA 4, uses a single virtual address space that is shared by the host and the device. The UVAS guarantees that each physical memory address is mapped to only one virtual memory location. This enables the CUDA runtime to determine whether a data pointer is referencing host or device memory by just inspecting its virtual memory address. This feature removes the need to specify the data copy direction on `cudaMemcpy()` calls.

It is important to note that the UVAS in CUDA 4 does not guarantee the accessibility of the data that is referenced by a pointer. For example, the host code cannot use a device pointer that is returned by `cudaMalloc()` to directly access the device memory and vice versa. Zero-copy memory is the exception: The host code can directly pass a pointer to zero-copy memory as a kernel launch parameter to the device. When the kernel code dereferences this zero-copy pointer, the pointer value is translated to a physical system memory location and is accessed directly through the PCIe bus. Note that this approach does not necessarily allow the kernel code to dereference a pointer value that is read from a memory location, such as following a chain of pointers while traversing a linked data structure, unless all memory has been allocated by using `cudaHostAlloc()`.

The limitations in both the types of data structures that can be supported and the bandwidth of data accesses of zero-copy memory motivate further improvements in the memory model of GPU architectures beyond UVAS.

Large virtual and physical address spaces

One fundamental limitation of early CUDA-enabled GPUs is the size of their virtual and physical addresses. These early devices support 32-bit virtual addresses and up to 32-bit physical addresses. For these devices the size of the device memory is limited to 4 gigabytes, the maximum amount of memory that can be addressed with 32 physical address bits. Furthermore, CUDA kernels can operate only on datasets whose sizes are less than 4 gigabytes, the maximum number of virtual memory locations that can be accessed through 32-bit pointers, regardless of whether the dataset resides in the host memory or the device memory. Furthermore, modern CPUs are based on 64-bit virtual addresses with 48 bits actually utilized. These host virtual addresses cannot be accommodated by the 32-bit virtual addresses that are used by GPUs, which contributed to the limitation of the types of data structures supported by zero-copy memory.

To remove this limitation, GPU generations starting with the Kepler GPU architecture introduced in 2013 have adopted modern virtual memory architecture with 64-bit virtual addresses and physical addresses of at least 40 bits. The obvious benefits are that these GPUs can incorporate more than 4 gigabytes of DRAM and that CUDA kernels can now operate on large datasets. While the enlarged virtual and physical address spaces obviously enable the use of large device memories, they also open the door for much better host/device interaction models. For example, the host and the device can now use exactly the same

pointer value to access a piece of data, regardless of whether it is in the host memory or the device memory.

The large GPU physical address space also allows the CUDA system software to place device memory of different GPUs in the system into a unified physical address space. The benefit is that one GPU can directly access the memory of any other GPU that is attached to the same PCIe bus by simply dereferencing a data pointer that is mapped to the physical address of such a GPU. Prior to the Kepler GPU architecture, communication among different GPUs (e.g., halo exchange in the stencil example in Chapter 20, Programming a Heterogeneous Computing Cluster) was possible only through device-to-device memory copies triggered by the host code. This resulted in extra memory consumed to store the data being copied from other GPUs and extra performance overheads due to the memory copy operations. Direct access to other device memory in the system enables just passing the device pointer to the other GPU in the kernel call and using it to load and/or store the data that need to be communicated.

Unified memory

In 2013, CUDA 6 introduced unified memory, which creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. Variables in the managed memory can reside in the CPU physical memory, the GPU physical memory, or even both. The CUDA runtime software and hardware implement data migration and coherence support, such as the GMAC system (Gelado et al., 2010). The net effect is that the managed memory looks like CPU memory to code running on the CPU and like GPU memory to code running on the GPU. Of course, the application must perform appropriate synchronization operations such as barriers or atomic operations to coordinate any concurrent accesses to the managed memory locations. A shared global virtual address space allows all variables in an application to have unique addresses. Such memory architecture, when exposed by programming tools and runtime systems to applications, can result in several major benefits.

One such benefit is the reduced amount of effort that is required for porting CPU code to CUDA. In Fig. 22.1 we show a simple CPU code example on the left side. With unified memory, the code can be ported to CUDA with two simple changes. The first change is to use `cudaMallocManaged()` and `cudaFree()` in place of `malloc()` and `free()`. The second change is to launch a kernel and perform device synchronization rather than calling the `qsort()` function. Obviously, one still needs to write or have access to a parallel `qsort` kernel. What we are showing is that the change to the host code is straightforward and easy to maintain.

The performance of the CUDA 6 unified memory was limited by the hardware capabilities of the Kepler and Maxwell GPU architectures. The contents of all managed memory locations that had been modified by the CPU had to be flushed out to the GPU device memory before any grid launch. The CPU and GPU could not simultaneously

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

FIGURE 22.1

Unified memory simplifies porting of CPU code (left) to CUDA code (right).

access a managed memory allocation, and the unified memory address space was limited to the size of the GPU physical memory. These limitations are due to the fact that these GPU architectures lacked the ability to support coherence between the host and device memories and that the data migration was mostly performed by software.

In 2016 the Pascal GPU architecture added features to further simplify programming and sharing of memory between CPU and GPU, and further reduce the effort required to use GPUs for significant speedups. Two main hardware features enable these improvements: support for large address spaces and page fault handling capability.

The Pascal GPU architecture extends GPU addressing capabilities to 49-bit virtual addressing. This extension is large enough to cover the 48-bit virtual address spaces of modern CPUs as well as the GPU's own memory. This allows unified memory programs to access the full address spaces of all CPUs and GPUs in the system as a single virtual address space rather than being limited by the amount of data that can be copied to the device memory. As a result, the CPUs and GPUs can truly share the pointer values, enabling the GPUs to traverse pointer-based data structures in the host memory.

Memory page fault handling support in the Pascal GPU architecture is a crucial new feature that provides more seamless unified memory functionality. Combined with the system-wide virtual address space, the ability to handle page faults eliminates the need for the CUDA system software to synchronize (flush) all managed memory contents to the GPU before each grid launch. The CUDA runtime can implement a coherence mechanism by allowing the host and the device to invalidate each other's copy when they modify a variable in the managed memory. The invalidation can be done by using the page mapping and protection mechanisms. When launching a grid, the CUDA system software no longer has to bring all GPU copies of the managed memory data up to date. If the grid accesses a piece of data whose copy in the device memory has been invalidated by the host, the GPU will handle a page fault to bring the data up to date and resume execution.

If a grid running on the GPU accesses a page that is not resident in its device memory, it also will take a page fault, allowing the page to be automatically

migrated to the GPU memory on demand. Alternatively, the page may be mapped into the GPU address space for access over the system interconnects (mapping on access can sometimes be faster than migration) if the data is expected to be accessed only occasionally. Note that unified memory is system-wide: GPUs (and CPUs) can fault and migrate memory pages either from CPU memory or from the memory of other GPUs in the system. If a CPU function dereferences a pointer and accesses a variable that is mapped to the GPU physical memory, the data access would still be serviced but perhaps at a longer latency. Such capability allows the CUDA programs to more easily call legacy libraries that have not been ported to GPUs. In the prior CUDA memory architecture the developer must manually transfer data from the device memory to the host memory in order to use legacy library functions to process them on the CPU.

The unified memory with page fault handling capability enables a much more general CPU/GPU interaction mechanism than zero-copy memory. It allows the GPU to traverse large data structures in the host memory. Starting with the Pascal architecture, a GPU device can traverse a linked data structure even if the data structure does not reside in zero-copy memory. This is because the same pointer value is used in the host code and the device code to refer to the same variable. Thus the embedded pointer values of a linked data structure built by the host can be traversed by the device and vice versa. In some application areas, such as CAD, the host physical memory system may have hundreds of gigabytes of capacity. These physical memory systems are needed because the applications require the entire dataset to be "in core." With the ability to directly access very large CPU physical memories, it becomes feasible for GPUs to accelerate these applications.

Virtual address space control

CUDA 11 introduced a set of low-level APIs to give programmers more flexibility regarding memory allocation. The new API allows reserving a range of the virtual address space using `cuMemAddressReserve()`. Later on, the programmer can allocate physical memory on any device using `cuMemCreate()` and map it to any position in the reserved range using `cuMemMap()`. These APIs enable building custom layouts of data structures across multiple devices. For instance, it would be possible to allocate a 3D volume across multiple devices while using a single pointer to reference it.

22.2 Kernel execution control

Function calls within kernel functions

Early CUDA versions did not allow function calls during kernel execution. Although the source code of kernel functions can appear to have function calls, the compiler must be able to inline all function bodies into the kernel object so that there are no function calls in the kernel function at runtime. Although this

model works reasonably well for performance-critical portions of many applications, it does not support the common software engineering practices in more sophisticated applications. In particular, it does not support system calls, dynamically linked library calls, recursive function calls, and virtual functions in object-oriented languages such as C++.

Later device architectures, such as Kepler, supported function calls in kernel functions at runtime. This feature is supported in CUDA 5 and beyond. The compiler is no longer required to inline the function bodies. It can still do so as a performance optimization. This capability is partly enabled by a cached and fast implementation of massively parallel call frame stacks for CUDA threads. It makes CUDA device code much more "composable" by allowing different authors to write different CUDA kernel components and assemble them all together without heavy redesign costs. It also allows software vendors to release device libraries without source code for intellectual property protection. However, some limitations still exist. For example, support for virtual functions is limited to objects constructed by device code, and dynamic libraries for device code are not supported.

Support for function calls at runtime also enables recursion and significantly eases the burden on programmers as they transition from legacy CPU-oriented algorithms toward GPU-tuned code. In some cases, developers will be able to "cut and paste" CPU code into a CUDA kernel and obtain a reasonably performing kernel, although continued performance tuning would still add benefit.

With the function call support, kernels can now call standard library functions such as `printf()` and `malloc()`. In our experience, the ability to call `printf()` in a kernel provides a subtle but important aid in debugging and supporting kernels in production software. Many end users are nontechnical and cannot be easily trained to run debuggers that will provide developers with more details on what happened before a crash. The ability to call `printf()` in the kernel allows developers to add a mode to the application to dump internal state so that the end users can submit meaningful bug reports.

CUDA 8 added support for C++11 and, with it, another form of function calls: lambdas. When coupled with metaprogramming techniques, device lambdas enable the development of high-performance reusable code. CUDA also supports passing lambda functions as parameters to CUDA kernels. This feature can be used to write generic kernels (e.g., a sorting kernel), in which the comparison function is just an input parameter to the kernel. CUDA also added experimental support for "extended lambdas," which are enabled with the `–extended-lambda` compiler flag. This feature allows programmers to annotate C++ lambdas with the `__host__ __device__` modifiers, further simplifying the task of writing reusable code.

Exception handling in kernel functions

Early CUDA systems did not support exception handling in kernel code. While not a significant limitation for performance-critical portions of many high-performance applications, it often incurs software engineering cost in production quality

applications that rely on exceptions to detect and handle rare conditions without executing code to explicitly test for such conditions.

With the availability of limited exception handling support, CUDA debuggers allow a user to perform step-by-step execution, to set breakpoints, and or to run a kernel until an invalid memory access happens. In each case, the user can inspect the values of the kernel's local and global variables when the execution is suspended. In our experience, the CUDA debugger is a very helpful tool to detect out-of-bounds memory accesses and potential race conditions.

Simultaneous execution of multiple grids

The earliest CUDA systems allowed only one grid to execute on each GPU device at any point in time. Multiple grids could be submitted for execution using CUDA streams, but they were buffered in a queue that released the next grid after the current one had completed execution. The Fermi GPU architecture and its successors allowed multiple grids from the same application to be executed simultaneously, which reduces the pressure for the application developer to "batch" multiple kernels into a larger kernel in order to more fully utilize a device. Also, it is sometimes beneficial to partition work into chunks that can execute with different levels of priority.

A typical example of the benefit of executing multiple grids simultaneously is for parallel cluster applications that segment work into "local" and "remote" partitions, in which remote work is involved in interactions with other nodes and resides on the critical path of global progress (Chapter 20, Programming a Heterogeneous Computing Cluster). In previous CUDA systems, grids needed to perform a lot of work to utilize the device efficiently, and one had to be careful not to launch local work such that global work could be blocked. This meant choosing between underutilizing the device while waiting for remote work to arrive or eagerly starting on local work to keep the device productive at the cost of increased latency for completing remote work units (Phillips and Stone, 2009). With multiple grid execution the application can use much smaller grid sizes for launching work, and as a result, when high-priority remote work arrives, it can start running with low latency instead of being stuck behind a large grid of local computation.

Hardware queues and dynamic parallelism

In the Kepler architecture and CUDA 5, the multiple grid launch facility was extended by the addition of multiple hardware queues, which allow much more efficient scheduling of thread blocks from multiple grids from multiple streams. In addition, CUDA dynamic parallelism, which was covered in Chapter 21, CUDA Dynamic Parallelism, allows GPU work creation: GPU grids can launch child grids asynchronously, dynamically, and in a data-dependent or compute load−dependent fashion. This reduces CPU-GPU interaction and synchronization, since the GPU can now manage more complex workloads independently. The CPU is in turn free to perform other useful computation.

Interruptable grids

The Fermi GPU architecture allowed a running grid to be "canceled," enabling the creation of CUDA-accelerated apps that allow the user to abort a long-running calculation at any time without requiring significant design effort on the part of the programmer. This enables implementation of user-level task scheduling systems that can better perform load balance between GPU nodes of a computing system and allows more graceful handling of cases in which one GPU is heavily loaded and may be running more slowly than its peers (Stone & Hwu, 2009).

Cooperative kernels

GPU kernels working on irregular data often suffer from load imbalance. CUDA 11 introduced the cooperative kernels to alleviate this problem. A cooperative kernel can execute up to a maximum number of thread blocks that would completely fill the GPU, and the CUDA runtime guarantees that all the thread blocks will execute concurrently. This concurrency guarantee enables thread blocks to safely cooperate without deadlock over shared mutual exclusion mechanisms (e.g., a mutex) that can be used to protect shared data structures (e.g., work queues). Cooperative kernels require a special API, `cudaLaunchCooperativeKernel()`, and provide a device API to identify and partition groups of threads. Cooperative kernels are not limited to a single device but can be called to run in multiple devices using `cudaLaunchCooperativeKernelMultiDevice()`.

22.3 Memory bandwidth and compute throughput

Double-precision speed

Early devices performed double-precision floating-point arithmetic with significant speed reduction (around eight times slower) compared to single-precision. The floating-point arithmetic units of Fermi and its successors were significantly strengthened to perform double-precision arithmetic at about half the speed of single-precision. Applications that are intensive in double-precision floating-point arithmetic benefited tremendously.

In practice, the most significant benefit was obtained by developers who were porting CPU-based numerical applications to GPUs. With the improved double-precision speed, they had little incentive to spend the effort to evaluate whether their applications or portions of their applications could fit into single-precision. This significantly reduced the development cost for porting CPU applications to GPUs and addressed a major criticism of GPUs in their earliest days by the HPC community.

Some applications that were operating on smaller input data types (8-bit, 16-bit, or single-precision floating-point) continued to benefit from using single-precision arithmetic, owing to the reduced bandwidth of using 32-bit versus

64-bit data. Applications such as medical imaging, remote sensing, radio astronomy, seismic analysis, and other natural data frequently fit into this category. The Pascal GPU architecture introduced new hardware support for computing with 16-bit half-precision numbers to further improve the performance and energy efficiency of applications whose data can fit into the half-precision representation. For example, in A100 based on the Ampere architecture, the 16-bit half-precision arithmetic throughput using tensor cores is 156 TFLOPS, a dramatic improvement compared to its 19.5 TFLOPS single-precision throughput.

Better control flow efficiency

Starting with the Fermi GPU architecture, CUDA systems adopted a general compiler-driven predication technique (Mahlke et al., 1995) that can more effectively handle control flow than previous CUDA systems. While this technique was moderately successful in VLIW systems, it provided even more dramatic speed improvements in GPU warp-style SIMD execution systems. This capability broadens the range of applications that can take advantage of GPUs. In particular, major performance benefits can potentially be realized for applications that are very data-driven, such as ray tracing, quantum chemistry visualization, and cellular automata simulation.

Configurable caching and scratchpad

The shared memory in early CUDA systems served as a programmer-managed scratchpad memory and increased the speed of applications in which key data structures have localized and predictable access patterns. Starting with the Fermi GPU architecture, the shared memory was enhanced to a larger on-chip memory that can be configured to be partially cache memory and partially shared memory, which allows coverage of both predictable and less predictable access patterns to benefit from on-chip memory. This configurability allows programmers to apportion the resources according to the best fit for their application.

Applications in an early design stage that are ported directly from CPU code will benefit greatly from caching as the dominant part of the on-chip memory. This further smoothes the performance-tuning process by increasing the level of "easy performance" when a developer ports a CPU application to GPU.

Existing CUDA applications and those that have predictable access patterns have the ability to increase their use of fast shared memory while retaining the same device "occupancy" they had on previous-generation devices. For CUDA applications whose performance or capabilities are limited by the size of the shared memory, the increase in size will be a welcome improvement. For example, in stencil computation (Chapter 8, Stencil, and Chapter 20, Programming a Heterogeneous Computing Cluster), such as finite difference methods for computational fluid dynamics, the increased shared memory capacity improves the memory bandwidth efficiency and the performance of the application.

Enhanced atomic operations

The atomic operations in the Fermi GPU architecture were much faster than those in previous CUDA systems, and the atomic operations in Kepler were even faster. In addition, the Kepler atomic operations were more general. The atomic operations over shared memory variables in the Maxwell GPU architecture were further enhanced in their throughput. Atomic operations are frequently used in random scatter computation patterns, such as histograms (Chapter 9, Parallel Histogram: An Introduction to Atomic Operations and Privatization). Faster atomic operations reduce the need for algorithm transformations such as prefix sum (Chapter 11, Prefix Sum (Scan): An Introduction to Work Efficiency in Parallel Algorithms) (Sengupta et al., 2007) and sorting (Chapter 13, Sorting) (Satish et al., 2009) for implementing such random scattering computations. These transformations tend to increase the number of kernel invocations as well as the total number of operations that are needed to perform the target computation. Faster atomic operations can also reduce the need for involving the host CPU in algorithms that do collective operations or where multiple thread blocks update shared data structures, thus reducing the data transfer pressure between CPU and GPU.

Enhanced global memory access

The speed of random memory access is much faster in Fermi and Kepler than in earlier GPU architectures. Programmers can be less concerned about memory coalescing. This allows more CPU algorithms to be directly used in the GPU as an acceptable base, further smoothing the path of porting applications that access a diversity of data structures, such as ray tracing, and other applications that are heavily object-oriented and may be difficult to convert into perfectly tiled arrays.

The Pascal GPU architecture incorporated HBM2 (High-Bandwidth Memory version 2) 3D-stacked DRAM memory, which provided up to 3× the memory bandwidth of previous-generation NVIDIA Maxwell architecture GPUs. Pascal was also the first architecture to support the NVLink processor interconnect, which gave Tesla P100 up to 5× the GPU-GPU and GPU-CPU communication performance of PCI Express 3.0. This interconnect greatly improved the scalability of multi-GPU computation within a node as well as the efficiency of data sharing between GPUs and NVLink-capable CPUs.

22.4 Programming environment

Unified device memory space

In early CUDA devices, shared memory, local memory, and global memory formed their own separate address spaces. The developer could use pointers into the global memory but not others. Starting with the Fermi architecture that was introduced in 2009, these memories became parts of a unified address space. This unified address

space enabled a single set of load/store instructions and pointer addresses to access any of the GPU memory spaces (global, local, or shared memory) rather than different instructions and pointers for each. This made it easier to abstract away which memory contains a particular operand, allowing the programmer to deal with this only during allocation and making it simpler to pass CUDA data objects into other procedures and functions, irrespective of which memory area they come from.

This made CUDA code modules much more "composable." That is, a CUDA device function can accept a pointer that may point to any of these memories. For example, without unified GPU address space, a device function needs to have one implementation for each type of memory in which one of its arguments can reside. Unified GPU address space allows variables in all main types of GPU memories to be accessed in the same way, thus allowing one device function to accept arguments that can reside in different types of GPU memory. The code will run faster if a function argument pointer points to a shared memory location and more slowly if it points to a global memory location. The programmer can still perform manual data placement and transfers as a performance optimization. This capability has significantly reduced the cost of building production-quality CUDA libraries. This also enabled full C and C++ pointer support, which was a significant advancement at the time.

Future CUDA compilers will include enhanced support for C++ templates and virtual function calls in kernel functions. Although the hardware enhancements, such as runtime function calling capability, are in place, enhanced C++ language support in the compiler has been taking more time. With these enhancements, future CUDA compilers will support most mainstream C++ features. For example, using C++ features such as new, delete, constructors, and destructors in kernel functions is already supported.

New and evolved programming interfaces continue to improve the productivity of heterogeneous parallel programmers. OpenACC allows developers to annotate their sequential loops with compiler directives to enable a compiler to generate CUDA kernels. One can use the Thrust library of parallel type-generic functions, classes, and iterators to describe their computation and have the underlying mechanism generate and configure the kernels that implement the computation. CUDA FORTRAN allows FORTRAN programmers to develop CUDA kernels in their familiar language. In particular, CUDA FORTRAN offers strong support for indexing into multidimensional arrays. C++AMP allows developers to describe their kernels as parallel loops that operate on logical data structures, such as multidimensional arrays in a C++ application. We fully expect that new innovations will continue to arise to further boost the productivity of developers in this exciting area.

Profiling with critical path analysis

In heterogeneous applications that do significant computation on both CPUs and GPUs, it can be a challenge to locate the best place to spend optimization effort. Ideally, when optimizing code, one would like to target the locations in the application that will provide the highest speedup for the least effort. To this end, CUDA 7.5

introduced PC sampling, providing instruction-level profiling so that the user could pinpoint specific lines of code that are taking the most time in his or her application.

However, a challenge facing the users of such profilers is that the longest-running kernel in an application is not always the most critical optimization target. As Fig. 22.2 shows, Kernel X is the longer-running kernel. However, its execution time is fully overlapped with the CPU execution activity A. Any further improvement in the execution time of Kernel X without simultaneous improvement in the execution time of A is unlikely to improve the application performance. While the execution time of Kernel Y is not as long as that of kernel X, it is on the critical path of the application execution. The CPU is idling while waiting for the completion of Kernel Y. Speeding up Kernel Y will reduce the time the CPU spends waiting, so it is the best optimization target.

In 2016 the Visual Profiler in CUDA 8 provided a critical path analysis between GPU kernels and CPU CUDA API calls, enabling more precise targeting of optimization efforts. Fig. 22.3 shows critical path analysis in the CUDA 8

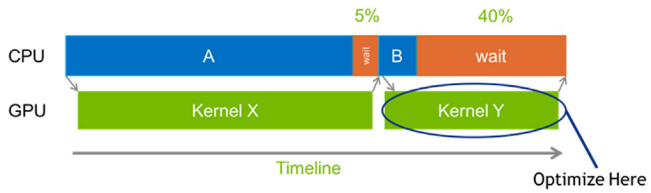


FIGURE 22.2

Importance of critical path analysis for identifying the key kernels to optimize.

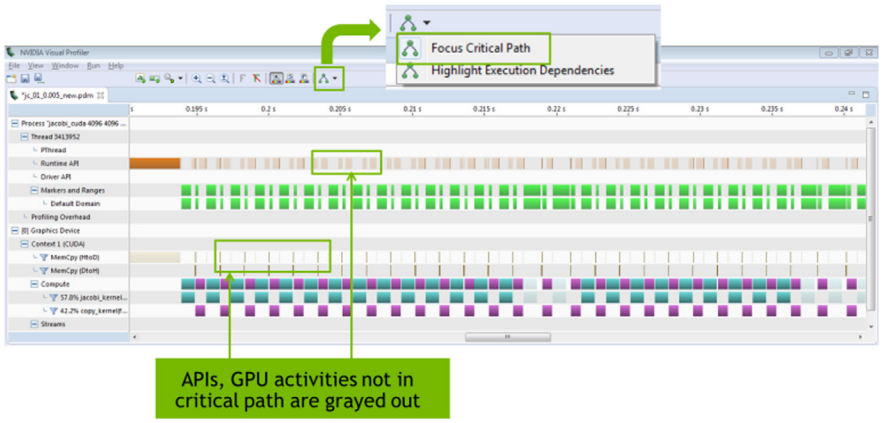


FIGURE 22.3

Application critical path analysis in CUDA 8 Visual Profiler.

Visual Profiler. GPU kernels, copies, and API calls that are not on the critical path are grayed out. Only the activities that are on the critical path of the application execution are highlighted in color. This allows the user to easily identify the kernels and other activities to target the user's optimization efforts.

22.5 Future outlook

The evolution of CUDA continues to increase its support for developer productivity and modern software engineering practices. With the new capabilities and programming language support, the range of applications that will be able to get reasonable performance at minimal development cost will expand significantly. Developers have experienced the reduction in application development, porting, and maintenance cost compared to previous CUDA systems. The existing applications that have been developed with Thrust and similar high-level tools that automatically generate CUDA code will also likely get an immediate boost in their performance. While the benefit of hardware enhancements in memory architecture, kernel execution control, and compute core performance will be visible in the associated SDK releases, the true potential of these enhancements may take years to be fully exploited in the SDKs and runtimes. We predict an exciting time for innovations from both industry and academia in programming tools and runtime environments for GPU computing in the next few years.

References

Gelado, I., Stone, J.E., Cabezas, J., Patel, S., Navarro, N., Hwu, W.W., 2010. An asymmetric distributed shared memory model for heterogeneous parallel systems. In: The ACM/IEEE 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10), March 2010. Pittsburgh, PA.

Mahlke, S.A., Hank, R.E., Mcormick, J.E., August, D.I., Hwu, W.W., 1995. A comparison of full and partial predicated execution support for ILP processors. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995, pp. 138–150.

Phillips, J., Stone, J., 2009. Probing biomolecular machines using graphics processors. Commun. ACM 52 (10).

Satish, N., Harris, M., Garland, M., 2009. Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of 23rd IEEE International Parallel & Distributed Processing Symposium, May 2009.

Sengupta, S., Harris, M., Zhang, Y., Owens, J.D., 2007. Scan primitives for GPU Computing. In: Proceedings of Graphics Hardware, August 2007, pp. 97–106.

Stone, J.E., Hwu, W.W., 2009. WorkForce: A Lightweight Framework for Managing Multi-GPU Computations, Technical Report, IMPACT Group, University of Illinois at Urbana-Champaign.

CHAPTER

Conclusion and outlook 23

Chapter Outline

You made it! We have arrived at the finishing line. In this final chapter we will briefly review the learning goals that you have achieved through this book. Instead of drawing a conclusion, we will offer our vision for the future of massively parallel computing and how its advancements will affect the future course of science and technology.

23.1 Goals revisited

As we stated in the Introduction, our primary goal is to teach you, the reader, how to program massively parallel processors. We promised that it would become easy once you develop the right intuition and go about it the right way. In particular, we promised to focus on computational thinking skills that would enable you to think about problems in ways that are amenable to parallel computing. We also promised to focus on the factors that limit the performance of parallel applications and provide a systematic approach for optimizing code to overcome these factors.

We delivered on these promises through four steps. In step one, Chapters 2 through 6 introduce the essential concepts of parallel computing and CUDA C and the key performance considerations in developing massively parallel code in CUDA. They also introduce the pertinent computer architecture concepts needed to understand the hardware limitations that must be addressed in high-performance parallel programming. With this knowledge, a developer can be confident in writing their parallel code and reasoning about the relative merit of alternative threading arrangements, loop structures, and coding styles. We conclude this part with a checklist of common optimization techniques that we apply throughout the rest of the book to optimize the performance of various parallel patterns and applications.

In the second step, we introduce six major parallel patterns (Chapters 7 through 12) that have been proven useful in introducing parallelism into

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00020-3

many applications. These chapters cover the concepts behind the most useful patterns of parallel computation. Each pattern is illustrated with concrete code examples. Each pattern is also used to introduce important techniques for overcoming frequently encountered parallelization and performance obstacles in parallel programming. We also use these patterns to showcase how the optimizations introduced in the first step can be applied in a wide variety of scenarios.

In the third step, we introduce additional advanced patterns and applications (Chapters 13–19) to reinforce the knowledge and skills gained in the previous step. While in this step, we continue to apply the optimizations that we practiced in the previous step, we put more emphasis on exploring alternative forms of problem decomposition to enable parallelization and analyze the tradeoffs between different decompositions and their associated data structures. The final chapter in this step is dedicated to recap the computational thinking skills (Chapter 19, Parallel Programming and Computational Thinking) that help the reader to generalize the concepts learnt in the previous chapters into the high-level thinking required to tackle a new problem. With these insights, high-performance parallel programming becomes a thought process, rather than a black art.

The fourth step is to expose the reader to related advanced practices in parallel programming. Chapter 20, Programming a Heterogeneous Computing Cluster, presents the basic skills required to program an HPC cluster using MPI and CUDA C. Chapter 21, CUDA Dynamic Parallelism, presents an introduction to dynamic parallelism that helps parallel programmers to address more complex parallel algorithms with dynamically varying workload in many real-world applications. Chapter 22, Advanced Practices and Future Evolution, summarizes other advanced practices as well as expectations for the future evolution of massively parallel processors.

We hope that you have enjoyed the book and agree with us that you are now well equipped for programming massively parallel computing systems.

23.2 Future outlook

Since the introduction of the first CUDA-enabled GPU, the G80, in 2007, the capability of GPUs as massively computing devices has improved at an amazing 452× in computing throughput and 18× in memory bandwidth, as shown in Fig. 23.1. These advancements have stimulated tremendous progress in HPC, AI, and data analytics for both science and engineering, with significant impact on many vertical areas such as finance, manufacturing, and medicine. For example, as we have seen in Chapter 16, Deep Learning, GPUs have ignited a revolution in deep learning from very large datasets, with applications in image recognition, speech recognition, and video analytics.

	G80 (2007)	A100 (2020)	A100/G80 Ratio
Compute Throughput	0.345 F32 TFLOPS	156 TFLOPS	452x
Memory Bandwidth	86.4 GB/s	1,555 GB/s	18x
Memory Capacity	1.5GB	40 GB	27x
PCIe Bandwidth	8 GB/s (Gen2 x16, each way)	32 GB/s (Gen4 x16, each way)	4x

FIGURE 23.1

From G80 to A100, a 13-year comparison.

Since the first edition of this book in 2010, the field of parallel computing has also advanced at an amazing pace. The spectrum of problems that can be solved with scalable algorithms has broadened significantly. While the use of GPUs was initially concentrated on regular, dense matrix computation and Monte Carlo methods, their use has quickly expanded into sparse methods, graph computation, and adaptive refinement methods. In many areas, there has also been fast advancement in algorithms. Some of the algorithms presented in the parallel pattern, advanced patterns, and applications chapters represent significant recent advancements.

It is only natural for some of us to wonder if we have reached the end of the fast advancement in parallel computing. From all indications, the answer is definitely no. We are only at the beginning of the parallel computing revolution. The amazing advancement in computing in the past three decades has triggered a paradigm shift in industry. The major innovations used to be driven by physical instruments assisted by computing devices. They are now driven by computing assisted by physical instruments.

For example, two decades ago, GPS revolutionized the way we drive. GPS is primarily based on satellite signal sensing assisted by computing methods that determine the shortest path between two locations. More recently, peer reporting and computing through the map apps in the phones and data services in the cloud have further enabled drivers to change their routes based on the traffic condition. Today, the most exciting revolution in the automobile industry is self-driving cars, which is primarily based on machine-learning computing methods assisted by physical sensors.

For another example, MRI and PET revolutionized medicine in the past two decades. These technologies are primarily based on electromagnetic and light sensors assisted by computational image reconstruction methods. They allowed doctors to see the pathology inside human bodies without surgery. Today, the field of medicine is going through the revolution of individualized medicine, which is primarily driven by computational genomics methods assisted by sequencing sensors.

For yet another example, the semiconductor industry used to rely on advancement in physical light sources assisted by computing methods that enforce design rules in their push to reduce the device feature size in the manufacturing process. Today, the advancement in physical light sources has practically stopped. The advancement in feature size reduction is primarily driven by lithography masks that are computationally designed to orchestrate the interference of light waves to result in extremely precise etching patterns on the chips.

The same kind of paradigm shift has been taking place in many other areas. Computing has become the primary driving force for virtually all exciting innovations in our society. This has created an insatiable demand for faster computing systems. As we discussed in Chapter 1, Introduction, parallel computing is the only viable approach to the growth of computing performance. This powerful demand will continue to motivate the industry to innovate and create more powerful parallel computing devices. One of the highest potential areas for improvement is the level of parallelism in accessing storage data, which has been done with a very low level of parallelism in the past. Radical improvements in accessing massive storage data will likely inspire a whole generation of applications that we currently cannot even imagine.

In conclusion, we are at the dawn of a golden age of computing. The industry will continue to recruit and reward highly skilled parallel programmers. Your work will make a real difference in the field of your choice.

Enjoy the ride!

APPENDIX

Numerical considerations A

In the early days of computing, floating-point arithmetic capability was found only in mainframes and supercomputers. Although many microprocessors that were designed in the 1980s started to have floating-point coprocessors, their floating-point arithmetic speed was extremely slow, about three orders of magnitude slower than that of mainframes and supercomputers. With advances in microprocessor technology, many microprocessors that were designed in the 1990s, such as Intel Pentium III and AMD Athlon, started to have high-performance floating-point capabilities that rivaled those of supercomputers. High-speed floating-point arithmetic is a standard feature of microprocessors and GPUs today. Floating-point representation allows for a larger dynamic range of representable data values and more precise representation of tiny data values. These desirable properties make floating-point arithmetic the preferred data representative for modeling physical and artificial phenomena, such as combustion, aerodynamics, light illumination, and financial risks. Large-scale evaluation of these models has been driving the need for parallel computing. As a result, it is important for application programmers to understand the nature of floating-point arithmetic in developing their parallel applications. In particular, we will focus on the accuracy of floating-point arithmetic operations, the precision of floating-point number representation, the stability of numerical algorithms, and how they should be taken into consideration in parallel programming.

A.1 Floating-point data representation

The IEEE-754 Floating-Point Standard is an effort to ensure that computer manufacturers conform to a common representation and arithmetic behavior for floating-point data (IEEE Microprocessor Standards Committee, 2008). Most, if not all, computer manufacturers in the world have accepted this standard. In particular, virtually all microprocessors that are designed in the future will either fully conform to or almost fully conform to the IEEE-754 Floating-Point Standard and its more recent IEEE-754 2008 revision. Therefore it is important for application developers to understand the concepts and practical considerations of this standard.

A floating-point number system starts with the representation of a numerical value as bit patterns. In the IEEE Floating-Point Standard, a numerical value is

represented in three groups of bits: sign (S), exponent (E), and mantissa (M). With some exceptions that will be detailed later in this appendix, each (S, E, M) pattern uniquely identifies a numerical value according to the following formula:

$$\text{value} = (-1)^{S} * 1.M * \{2^{E\text{-bias}}\} \tag{A.1}$$

The interpretation of S is simple: S = 0 means a positive number, and S = 1 a negative number. Mathematically, any number, including −1, when raised to the power of 0, results in 1. Thus the value is positive. On the other hand, when −1 is raised to the power of 1, it is −1 itself. With a multiplication by −1, the value becomes negative. However, the interpretation of M and E bits is much more complex. We will use the following example to help explain the interpretation of M and E bits.

Assume for the sake of simplicity that each floating-point number consists of a 1-bit sign, a 3-bit exponent, and a 2-bit mantissa. We will use this hypothetical 6-bit format to illustrate the challenges that are involved in encoding E and M. As we discuss numerical values, we will sometimes need to express a number either in decimal place value or in binary place value. Numbers that are expressed in decimal place value will have a subscript $_{D}$, and those that are expressed in binary place value will have a subscript $_{B}$. For example, 0.5_{D} ($5 * 10^{-1}$, since the place to the right of the decimal point carries a weight of 10^{-1}) is the same as 0.1_{B} ($1 * 2^{-1}$, since the place to the right of the decimal point carries a weight of 2^{-1}).

A.1.1 Normalized representation of M

Eq. (A.1) requires that all values be derived by treating the mantissa value as 1.M, which makes the mantissa bit pattern for each floating-point number unique. For example, in this interpretation of the M bits, the only mantissa bit pattern that is allowed for 0.5_{D} is the one in which all bits that represent M are 0s:

$$0.5_{D} = 1.0_{B} * 2^{-1}$$

Other potential candidates would be $0.1_{B} * 2^{0}$ and $10.0_{B} * 2^{-2}$, but neither fits the form of 1.M. The numbers that satisfy this restriction will be referred to as normalized numbers. Because all mantissa values that satisfy the restriction are of the form 1.XX, we can omit the "1." part from the representation. Therefore the mantissa value of 0.5 in a 2-bit mantissa representation is 00, which is derived by omitting "1." from 1.00. This makes a 2-bit mantissa effectively a 3-bit mantissa. In general, in IEEE format, an m-bit mantissa is effectively an (m + 1)-bit mantissa.

A.1.2 Excess encoding of E

The number of bits that are used to represent E determines the range of numbers that can be represented. Large positive E values result in very large floating-point

absolute values. For example, if the value of E is 64, the floating-point number that is being represented is between 2^{64} ($>10^{18}$) and 2^{65}. You would be extremely happy if this was the balance of your savings account! Large negative E values result in very small floating-point values. For example, if the E value is -64, the number that is being represented is between 2^{-64} ($<10^{-18}$) and 2^{-63}. This is a very tiny fractional number. The E field allows a floating-point number format to represent a wider range of numbers than integer number formats can. We will come back to this point when we look at the representable numbers of a format.

The IEEE standard adopts an excess or biased encoding convention for E. If E bits are used to represent the exponent E, $(2^{e-1}-1)$ is added to the two's complement representation for the exponent to form its excess representation. A two's complement representation is a system in which the negative value of a number can be derived by first complementing every bit of the value and adding 1 to the result. In our 3-bit exponent representation, there are three bits in the exponent ($e=3$). Therefore the value $2^{3-1}-1=011$ will be added to the two's complement representation of the exponent value.

The advantage of excess representation is that an unsigned comparator can be used to compare signed numbers. As shown in Fig. A.1, in our 3-bit exponent representation, the excess-3 bit patterns increase monotonically from -3 to 3 when they are viewed as unsigned numbers. We will refer to each of these bit patterns as the code for the corresponding value. For example, the code for -3 is 000 and that for 3 is 110. Thus if one uses an unsigned number comparator to compare excess-3 code for any number from -3 to 3, the comparator gives the correct comparison result in terms of which number is larger, smaller, and so on. For another example, if one compares excess-3 codes 001 and 100 with an unsigned comparator, 001 is smaller than 100. This is the right conclusion, since the values that they represent, -2 and 1, have exactly the same relation. This is a desirable property for hardware implementation, since unsigned comparators are smaller and faster than signed comparators.

Fig. A.1 also shows that the pattern of all 1's in the excess representation is a reserved pattern. Note that a 0 value and an equal number of positive and

2's complement	Decimal value	Excess-3
101	–3	000
110	–2	001
111	–1	010
000	0	011
001	1	100
010	2	101
011	3	110
100	Reserved pattern	111

FIGURE A.1

Excess-3 encoding, sorted by excess-3 ordering.

negative values result in an odd number of patterns. Having the pattern 111 as either an even number or an odd number would result in an unbalanced number of positive and negative numbers. The IEEE standard uses this special bit pattern in special ways that will be discussed later in this appendix.

Now we are ready to represent 0.5_D with our 6-bit format:

$$0.5_D = 0\ 01000, \text{where } S = 0,\ E = 010, \text{and } M = (1.)\ 00$$

That is, the 6-bit representation for 0.5_D is 001000.

In general, with a normalized mantissa and an excess-coded exponent, the value of a number with an n-bit exponent is

$$\text{value} = (-1)^S * 1.M * 2^{(E-(2^{(n-1)}-1))} \tag{A.2}$$

A.2 Representable numbers

The representable numbers of a representation format are the numbers that can be exactly represented in the format. For example, if one uses a 3-bit unsigned integer format, the representable numbers are shown in Fig. A.2.

Neither −1 nor 9 can be represented in the format given above. We can draw a number line to identify all the representable numbers, as shown in Fig. A.3, in which all representable numbers of the 3-bit unsigned integer format are marked with stars.

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

FIGURE A.2

Representable numbers of a 3-bit unsigned integer format.

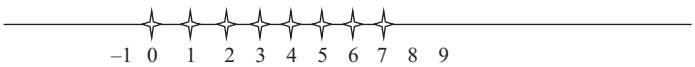


FIGURE A.3

Representable numbers of a 3-bit unsigned integer format.

The representable numbers of a floating-point format can be visualized in a similar manner. In Fig. A.4 we show all the representable numbers of what we have so far and two variations. We use a 5-bit format to keep the size of the table manageable. The format consists of 1-bit S, 2-bit E (excess-1 coded), and 2-bit M (with "1." part omitted). The no-zero column gives the representable numbers of the format we discussed thus far. The reader is encouraged to generate at least part of the no-zero column using Eq. (A.2). Note that with this format, 0 is not one of the representable numbers.

A quick look at how these representable numbers populate the number line, as shown in Fig. A.5, provides further insights into these representable numbers. In Fig. A.5 we show only the positive representable numbers. The negative numbers are symmetric to their positive counterparts on the other side of 0.

We can make five observations. First, the exponent bits define the major intervals of representable numbers. In Fig. A.5 there are three major intervals on each side of 0 because there are two exponent bits. Basically, the major intervals are between powers of 2. With two bits of exponents and one reserved bit pattern (11), there are three powers of 2 ($2^{-1} = 0.5_D$, $2^0 = 1.0_D$, $2^1 = 2.0_D$), each of which starts an interval of representable numbers. Keep in mind that there are also three powers of 2 ($-2^{-1} = -0.5_D$, $-2^0 = -1.0_D$, $-2^1 = -2.0_D$) on the left of zero that are not shown in Fig. A.5.

The second observation is that the mantissa bits define the number of representable numbers in each interval. With two mantissa bits we have four

		No-zero		Abrupt underflow		Denormalized	
E	M	S=0	S=1	S=0	S=1	S=0	S=1
00	00	2^{-1}	$-(2^{-1})$	0	0	0	0
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	0	0	$1*2^{-2}$	$-1*2^{-2}$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	0	0	$2*2^{-2}$	$-2*2^{-2}$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	0	0	$3*2^{-2}$	$-3*2^{-2}$
01	00	2^0	$-(2^0)$	2^0	$-(2^0)$	2^0	$-(2^0)$
	01	2^0+1*2^{-2}	$-(2^0+1*2^{-2})$	2^0+1*2^{-2}	$-(2^0+1*2^{-2})$	2^0+1*2^{-2}	$-(2^0+1*2^{-2})$
	10	2^0+2*2^{-2}	$-(2^0+2*2^{-2})$	2^0+2*2^{-2}	$-(2^0+2*2^{-2})$	2^0+2*2^{-2}	$-(2^0+2*2^{-2})$
	11	2^0+3*2^{-2}	$-(2^0+3*2^{-2})$	2^0+3*2^{-2}	$-(2^0+3*2^{-2})$	2^0+3*2^{-2}	$-(2^0+3*2^{-2})$
10	00	2^1	$-(2^1)$	2^1	$-(2^1)$	2^1	$-(2^1)$
	01	2^1+1*2^{-1}	$-(2^1+1*2^{-1})$	2^1+1*2^{-1}	$-(2^1+1*2^{-1})$	2^1+1*2^{-1}	$-(2^1+1*2^{-1})$
	10	2^1+2*2^{-1}	$-(2^1+2*2^{-1})$	2^1+2*2^{-1}	$-(2^1+2*2^{-1})$	2^1+2*2^{-1}	$-(2^1+2*2^{-1})$
	11	2^1+3*2^{-1}	$-(2^1+3*2^{-1})$	2^1+3*2^{-1}	$-(2^1+3*2^{-1})$	2^1+3*2^{-1}	$-(2^1+3*2^{-1})$
11	Reserved pattern						

FIGURE A.4

Representable numbers of no-zero, abrupt underflow, and denormalized formats.

FIGURE A.5

Representable numbers of the no-zero representation.

representable numbers in each interval. In general, with N mantissa bits, we have 2^N representable numbers in each interval. If a value to be represented falls within one of the intervals, it will be rounded to one of these representable numbers. Obviously, the larger the number of representable numbers in each interval, the more precisely we can represent a value in the region. Therefore the number of mantissa bits determines the *precision* of the representation.

The third observation is that 0 is not representable in this format. It is missing from the representable numbers in the no-zero columns of Fig. A.5. Because 0 is one of the most important numbers, not being able to represent 0 in a number representation system is a serious deficiency. We will address this deficiency soon.

The fourth observation is that the representable numbers become closer to each other toward the neighborhood of 0. Each interval is half the size of the previous interval as we move toward zero. In Fig. A.5 the rightmost interval is of width 2, the next interval is of width 1, and the next interval is of width 0.5. Although not shown in Fig. A.5, there are three intervals on the left of zero. They contain the representable negative numbers. The leftmost interval is of width 2, the next interval is of width 1, and the next interval is width 0.5. Since every interval has the same representable numbers, four in Fig. A.5, the representable numbers becomes closer to each other as we move toward zero. In other words, the representative numbers become closer as their absolute values become smaller. This is a desirable trend because as the absolute values of these numbers become smaller, it is more important to represent them more precisely. The distance between representable numbers determines the maximal rounding error for a value that falls into the interval. For example, if you have one billion dollars in your bank account, you might not even notice that there is a $1 rounding error in calculating your balance. However, if the total balance is $10, having a $1 rounding error would be much more noticeable!

The fifth observation is that, unfortunately, the trend of increasing density of representable numbers and thus the increasing precision of representing numbers in the intervals as we move toward 0 does not hold near 0. That is, there is a gap of representable numbers in the immediate vicinity of 0. This is because the range of normalized mantissa precludes 0. This is another serious deficiency. The representation introduces significantly larger (4×) errors in representing numbers between 0 and 0.5 compared to the errors for the larger numbers between 0.5 and 1.0. In general, with m bits in the mantissa, this style of representation would introduce 2^m times more error in the interval closest to zero than in the next interval. For numerical methods that rely on accurate detection of convergence conditions based on very small data values, such deficiency can cause instability in execution time and inaccuracy of results. Furthermore, some algorithms generate small numbers and eventually use them as denominators. The errors in representing these small numbers can be greatly magnified in the division process and cause numerical instability in these algorithms.

One method that can accommodate 0 into a normalized floating-point number system is the *abrupt underflow* convention, which is illustrated in the second set of

columns in Fig. A.4. Whenever E is 0, the number is interpreted as 0. In our 5-bit format, this method takes away eight representable numbers (four positive and four negative) in the vicinity of 0 (between -1.0 and $+1.0$) and makes them all 0. Because of its simplicity, some minicomputers in the 1980s used the abrupt underflow convention. Even to this day, some arithmetic units that need to operate at high speed still use the abrupt underflow convention. Although this method makes 0 a representable number, it creates an even larger gap between representable numbers in the vicinity of 0, as shown in Fig. A.6. It is obvious, when compared with Fig. A.5, that the gap of representable numbers has been enlarged significantly (by 2×) from 0.5 to 1.0. As we explained earlier, this is very problematic for many numerical algorithms whose correctness reply on accurate representation of small numbers near zero.

The actual method that was adopted by the IEEE standard is called denormalization. The method relaxes the normalization requirement for numbers very close to 0. As is shown in Fig. A.6, whenever $E = 0$, the mantissa is no longer assumed to be of the form 1.XX. Rather, it is assumed to be 0.XX. The value of the exponent is assumed to be the same as in the previous interval. For example, in Fig. A.4 the denormalized representation 00001 has exponent value 00 and mantissa value 01. The mantissa is assumed to be 0.01, and the exponent value is assumed to be the same as that of the previous interval: 0 rather than -1. That is, the value that 00001 represents is now $0.01 * 2^0 = 2^{-2}$. Fig. A.7 shows the representable numbers for the denormalized format. The representation now has uniformly spaced representable numbers in the close vicinity of 0. Intuitively, the denormalized convention takes the four numbers in the last interval of representable numbers of a no-zero representation and spreads them out to cover the gap area. This eliminates the undesirable gap in the previous two methods. Note that the distances between representable numbers in the last two intervals are actually identical. In general, if the n-bit exponent is 0, the value is

$$0.M * 2^{-2^{(n-1)}+2}$$

As we can see, the denormalization formula is quite complex. The hardware also needs to be able to detect whether a number falls into the denormalized

FIGURE A.6

Representable numbers of the abrupt underflow format.

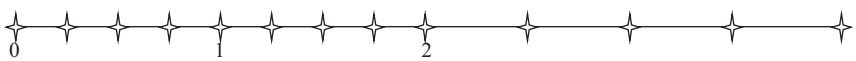


FIGURE A.7

Representable numbers of a denormalization format.

interval and choose the appropriate representation for that number. The amount of hardware required to implement denormalization at high speed is quite significant. Implementations that use a moderate amount of hardware often introduce thousands of clock cycles of delay whenever a denormalized number needs to be generated or used. This was the reason why early generations of CUDA devices did not support denormalization. However, virtually all recent generations of CUDA devices do support denormalization, thanks to the increasing number of available transistors of more recent fabrication processes. More specifically, all CUDA devices of compute capability 1.3 and up support denormalized double-precision operands, and all devices of compute capability 2.0 and up support denormalized single-precision operands.

In summary, the precision of a floating-point representation is measured by the maximal error that we can introduce to a floating-point number by representing that number as one of the representable numbers. The smaller the error, the higher is the precision. The precision of a floating-point representation can be improved by adding more bits to the mantissa. Adding one bit to the representation of the mantissa improves the precision by reducing the maximal error by half. Thus a number system has higher precision when it uses more bits for the mantissa. This is reflected in double-precision versus single-precision numbers in the IEEE standard.

A.3 Special bit patterns and precision in IEEE format

We now turn to more specific details of the actual IEEE format. When all exponent bits are 1s, the number that is represented is an infinity value if the mantissa is 0. It is not a number (NaN) if the mantissa is not 0. All special bit patterns of the IEEE floating-point format are described in Fig. A.8.

All other numbers are normalized floating-point numbers. Single-precision numbers have 1-bit S, 8-bit E, and 23-bit M. Double-precision numbers have 1-bit S, 11-bit E, and 52-bit M. Since a double-precision number has 29 more bits for the mantissa, the largest error for representing a number is reduced to $1/2^{29}$ of that of the single-precision format! With the additional 3 bits of exponent, the double-precision format also extends the number of intervals of representable numbers. This extends the range of representable numbers to very large as well as very small values.

All representable numbers fall between $-\infty$ (negative infinity) and $+\infty$ (positive infinity). An ∞ can be created by overflow, for example, a large number divided by a very small number. Any representable number divided by $+\infty$ or $-\infty$ results in 0.

NaN is generated by operations whose input values do not make sense, such as 0/0, $0 * \infty$, ∞/∞, $\infty - \infty$. NaN is also used for data that has not been properly initialized in a program. There are two types of NaN in the IEEE

exponent	mantissa	meaning
11…1	≠ 0	NaN
11…1	=0	$(-1)^S * \infty$
00…0	≠0	denormalized
00…0	=0	0

FIGURE A.8

Special bit patterns in the IEEE standard format.

standard: signaling and quiet. Signaling NaNs should be represented with the most significant mantissa bit cleared, whereas quiet NaNs are represented with the most significant mantissa bit set.

Signaling NaNs cause an exception when used as input to arithmetic operations. For example, the operation (1.0 + signaling NaN) raises an exception signal to the operating system. Signaling NaNs are used in situations in which the programmer would like to make sure that the program execution is interrupted whenever any NaN values are used in floating-point computations. These situations usually mean that there is something wrong with the execution of the program. In mission critical applications the execution cannot continue until the validity of the execution can be verified by a separate means. For example, software engineers often mark all the uninitialized data as signaling NaN. This practice ensures the detection of using uninitialized data during program execution. The current generation of GPU hardware does not support signaling NaN, owing to the difficulty of supporting accurate signaling during massively parallel execution.

A quiet NaN generates another quiet NaN without causing an exception when it is used as input to an arithmetic operation. For example, the operation (1.0 + quiet NaN) generates a quiet NaN. Quiet NaNs are typically used in applications in which the user can review the output and decide whether the application should be rerun with a different input for more valid results. When the results are printed, quiet NaNs are printed as "NaN" so that the user can spot them in the output file easily.

A.4 Arithmetic accuracy and rounding

Now that we have a good understanding of the IEEE floating-point format, we are ready to discuss the concept of arithmetic accuracy. While the precision is determined by the number of mantissa bits that are used in a floating-point number format, the accuracy is determined by the operations that are performed on a floating-point number. The accuracy of a floating-point arithmetic operation is measured by the maximal error that is introduced by the operation. The smaller the error, the higher is the accuracy. The most common source of error in floating-point arithmetic is when the operation generates a result that cannot be exactly represented and thus requires rounding. Rounding occurs if the mantissa of the result value needs too many bits to be represented exactly. For example, a

multiplication might generate a product value that consists of twice the number of bits as either of the input values. For another example, adding two floating-point numbers can be done by adding their mantissa values together if the two floating-point values have identical exponents. When two input operands to a floating-point addition have different exponents, the mantissa of the one with the smaller exponent is repeatedly divided by 2 or right-shifted (that is, all the mantissa bits are shifted to the right by one bit position) until the exponents are equal. As a result, the final result can have more bits than the format can accommodate.

Alignment shifting of operands can be illustrated with a simple example based on the 5-bit representation in Fig. A.4. Assume that we need to add $1.00_B * 2^{-2}$(0, 00, 01) to $1.00 * 2^1{}_D$ (0, 10, 00), that is, we need to perform the operation $1.00_B * 2^1 + 1.00_B * 2^{-2}$. Owing to the difference in exponent values, the mantissa value of the second number needs to be right-shifted by 3 bit positions before it is added to the first mantissa value. That is, the addition becomes $1.00_B * 2^1 + 0.001_B * 2^1$. The addition can now be performed by adding the mantissa values together. The ideal result would be $1.001_B * 2^1$. However, we can see that this ideal result is not a representable number in a 5-bit representation. It would have required three bits of mantissa, and there are only two mantissa bits in the format. Therefore the best one can do is to generate one of the closest representable numbers, which is either $1.01_B * 2^1$ or $1.00_B * 2^1$. By doing so, we introduce an error, $0.001_B * 2^1$, which is half the place value of the least significant place. We refer to this as 0.5_D ULP (units in the last place). If the hardware is designed to perform arithmetic and rounding operations perfectly, the most error that one should introduce should be no more than 0.5_D ULP. To the best of our knowledge, this is the accuracy that is achieved by addition and subtraction operations in all CUDA devices today.

In practice, some of the more complex arithmetic hardware units, such as division and transcendental functions, are typically implemented with polynomial approximation algorithms. If the hardware does not use a sufficient number of terms in the approximation, the result may have an error that is larger than 0.5_D ULP. For example, if the ideal result of an inversion operation is $1.00_B * 2^1$ but the hardware generates a $1.10_B * 2^1$, owing to the use of an approximation algorithm, the error is 2_D ULP, since the error ($1.10_B - 1.00_B = 0.10_B$) is two times bigger than the units in the last place (0.01_B). In practice, the hardware inversion operations in some early devices introduce an error that is twice the place value of the least place of the mantissa, or 2 ULP. Thanks to the more abundant transistors in more recent generations of CUDA devices, their hardware arithmetic operations are much more accurate.

A.5 Algorithm considerations

Numerical algorithms often need to sum up a large number of values. For example, the dot product in matrix multiplication needs to sum up pairwise products of input

matrix elements. Ideally, the order of summing these values should not affect the final total, since addition is an associative operation. However, with finite precision the order of summing these values can affect the accuracy of the final result. For example, suppose we need to perform a sum reduction on four numbers in our 5-bit representation: $1.00_B * 2^0 + 1.00_B * 2^0 + 1.00_B * 2^{-2} + 1.00_B * 2^{-2}$. If we add up the numbers in strict sequential order, we have the following sequence of operations:

$$\begin{aligned}&1.00_B * 2^0 + 1.00_B * 2^0 + 1.00_B * 2^{-2} + 1.00_B * 2^{-2}\\ &= 1.00_B * 2^1 + 1.00_B * 2^{-2} + 1.00_B * 2^{-2}\\ &= 1.00_B * 2^1 + 1.00_B * 2^{-2} = 1.00_B * 2^1\end{aligned}$$

Note that in the second and third steps, the smaller operand simply disappears because it is too small in comparison to the larger operand.

Now let's consider a parallel algorithm in which the first two values are added and the second two operands are added in parallel. The algorithm then adds up the pairwise sum:

$$\left(1.00_B * 2^0 + 1.00_B * 2^0\right) + \left(1.00_B * 2^{-2} + 1.00_B * 2^{-2}\right) = 1.00_B * 2^1 + 1.00_B * 2^{-1} = 1.01_B * 2^1$$

The reader should recognize that this algorithm was is the reduction tree of Chapter 10, Reduction and Minimizing Divergence, and it assumes the associativity of the add operation. Note that the results are different from the sequential result. This is because the sum of the third and fourth values is large enough that it now affects the addition result. This discrepancy between sequential algorithms and parallel algorithms often surprises application developers who are not familiar with floating-point precision and accuracy considerations. Although we showed a scenario in which a parallel algorithm produced a more accurate result than a sequential algorithm, the reader should be able to come up with a slightly different scenario in which the parallel algorithm produces a less accurate result than a sequential algorithm would. That is, the associativity of the add operation is not strictly true for floating-point numbers. Experienced application developers either make sure that the variation in the final result can be tolerated or ensure that the data is sorted or grouped in such a way that the parallel algorithm results in the most accurate results.

A common technique to maximize floating-point arithmetic accuracy is to presort data before a reduction computation. In our sum reduction example, if we presort the data according to ascending numerical order, we will have the following:

$$1.00_B * 2^{-2} + 1.00_B * 2^{-2} + 1.00_B * 2^0 + 1.00_B * 2^0$$

When we divide up the numbers into groups in a parallel algorithm, say, the first pair in one group and the second pair in another group, numbers with numerical values that are close to each other are in the same group. Obviously, the sign of the numbers needs to be taken into account during the presorting process. Therefore when we perform addition in these groups, we will likely have accurate

results. Furthermore, some parallel algorithms use each thread to sequentially reduce values within each group. Having the numbers sorted in ascending order allows a sequential addition to get higher accuracy. This is a reason why sorting is frequently used in massively parallel numerical algorithms. Interested readers should study more advanced techniques, such as compensated summation algorithm, also known as the Kahan summation algorithm, for getting an even more robust approach to accurate summation of floating-point values (Kahan, 1965).

A.6 Linear solvers and numerical stability

While the order of operations may cause variation in the numerical outcome of reduction operations, it may have even more serious implications for some types of computation, such as solvers for linear systems of equations. In these solvers, different numerical values of input may require different ordering of operations in order to find a solution. If an algorithm fails to follow a desired order of operations for an input, it may fail to find a solution even though the solution exists. Algorithms that can always find an appropriate operation order and thus finding a solution to the problem as long as it exists for any given input values are called *numerically stable*. Algorithms that fall short are referred to as *numerically unstable*.

In some cases, numerical stability considerations can make it more difficult to find efficient parallel algorithms for a computational problem. We can illustrate this phenomenon with a solver that is based on Gaussian elimination. Consider the following system of linear equations:

$$3X + 5Y + 2Z = 19 \tag{A.3}$$

$$2X + 3Y + Z = 11 \tag{A.4}$$

$$X + 2Y + 2Z = 11 \tag{A.5}$$

As long as the three planes represented by these equations have an intersection point, we can use Gaussian elimination to derive the solution that gives the coordinate of the intersection point. We show the process of applying Gaussian elimination to this system in Fig. A.9, in which variables are systematically eliminated from lower positioned equations.

In the first step, all equations are divided by their coefficient for the X variable: 3 for Eq. (A.3), 2 for Eq. (A.4), and 1 for Eq. (A.5). This makes the coefficients for X in all equations the same. In step 2, Eq. (A.3) is subtracted from Eqs. (A.4) and (A.5). These subtractions eliminate variable X from Eqs. (A.4) and (A.5), as shown in Fig. A.9.

We can now treat Eqs. (A.4) and (A.5) as a smaller system of equations with one fewer variable than the original equation. Since they do not have variable X, they can be solved independently from Eq. (A.3). We can make more progress by eliminating variable Y from Eq. (A.5). This is done in step 3 by dividing Eqs. (A.4) and (A.5) by the coefficients for their Y variables: −1/6 for Eq. (A.4)

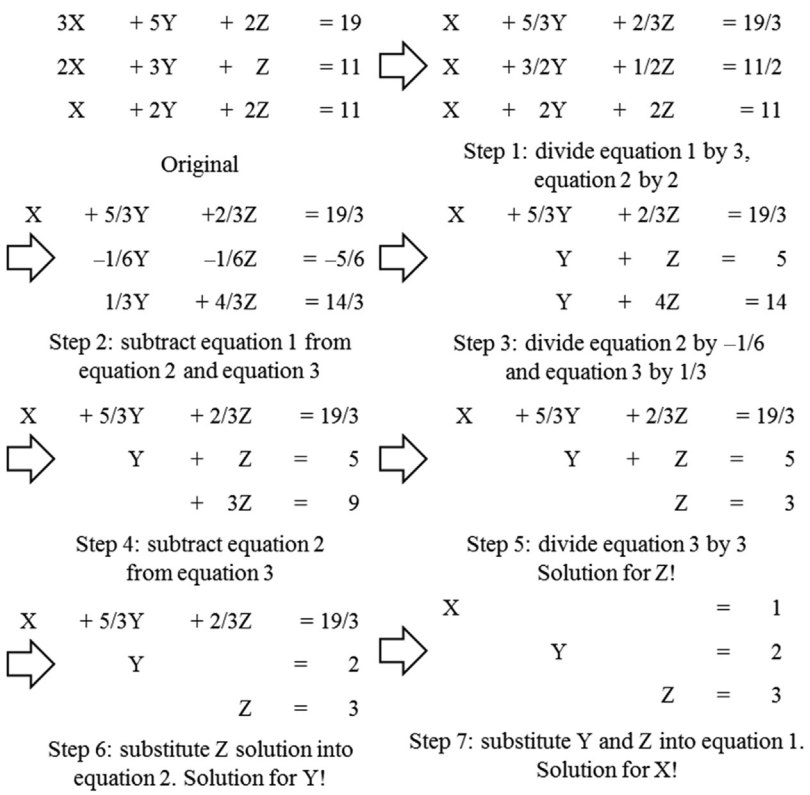


FIGURE A.9

Gaussian elimination and backward substitution for solving systems of linear equations.

and 1/3 for Eq. (A.5). This makes the coefficients for Y in both Eqs. (A.4) and (A.5) the same. In step 4, Eq. (A.4) is subtracted from Eq. (A.5), which eliminates variable Y from Eq. (A.5).

For systems with larger number of equations the process would be repeated more. However, since we have only three variables in this example, the third equation has only the Z variable. We simply need to divide Eq. (A.5) by the coefficient for variable Z. This conveniently gives us the solution $Z = 3$.

With the solution for the Z variable in hand, we can substitute the Z value into Eq. (A.4) to get the solution $Y = 2$. We can then substitute both $Z = 3$ and $Y = 2$ into Eq. (A.3) to get the solution $X = 1$. We now have the complete solution for the original system. It should be obvious why step 6 and step 7 form the second phase of the method called backward substitution. We go backwards from the last equation to the first equation to get solutions for more and more variables.

In general, the equations are stored in matrix forms in computers. Since all calculations involve only the coefficients and the right-hand-side values, we can

just store these coefficients and right-hand-side values in a matrix. Fig. A.10 shows the matrix view of the Gaussian elimination and backward substitution process. Each row of the matrix corresponds to an original equation. Operations on equations become operations on matrix rows.

After Gaussian elimination the matrix becomes a triangular matrix. This is a very popular type of matrix for various physics and mathematics reasons. We see that the end goal is to make the coefficient part of the matrix into a diagonal form, in which each row has only a value 1 on the diagonal line. This is called an identity matrix because the result of multiplying any matrix multiplied by an identity matrix is itself. This is also the reason why performing Gaussian elimination on a matrix is equivalent to multiplying the matrix by its inverse matrix.

In general, it is straightforward to design a parallel algorithm for the Gaussian elimination procedure that we illustrated in Fig. A.10. For example, we can write

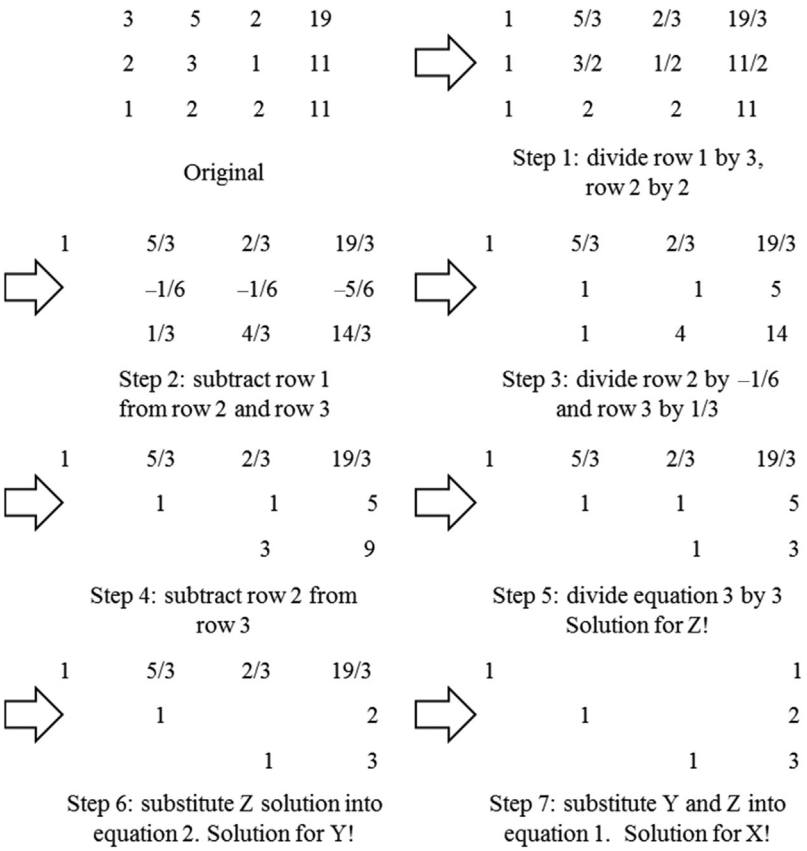


FIGURE A.10

Gaussian elimination and backward substitution in matrix view.

a CUDA kernel and designate each thread to perform all calculations to be done on a row of the matrix. For systems that can fit into shared memory, we can use a thread block to perform Gaussian elimination. All threads iterate through the steps. After each division step, all threads participate in barrier synchronization. They all then perform a subtraction step, after which one thread will stop its participation, since its designated row has no more work to do until the backward substitution phase. After the subtraction step, all threads need to perform barrier synchronization again to ensure that the next step will be done with the updated information. When we have systems of equations with many variables, we can expect a reasonable amount of speedup from the parallel execution.

Unfortunately, the simple Gaussian elimination algorithm that we have been using can suffer from numerical instability. This can be illustrated with the following example:

$$5Y + 2Z = 16 \tag{A.6}$$

$$2X + 3Y + Z = 11 \tag{A.7}$$

$$X + 2Y + 2Z = 11 \tag{A.8}$$

We will encounter a problem when we perform step 1 of the algorithm. The coefficient for the X variable in Eq. (A.6) is zero. We will not be able to divide Eq. (A.6) by the coefficient for variable X and eliminate the X variable from Eqs. (A.7) and (A.8) by subtracting Eq. (A.6) from Eqs. (A.7) and (A.8). The reader should verify that this system of equation is solvable and has the same solution: $X = 1$, $Y = 2$, and $Z = 3$. Therefore the algorithm is numerically unstable. It can fail to generate a solution for certain input values even though the solution exists.

This is a well-known problem with Gaussian elimination algorithms and can be addressed with a method that is commonly referred to as *pivoting*. The idea is to find one of the remaining equations whose coefficient for the lead variable is not zero. By swapping the current top equation with the identified equation, the algorithm can successfully eliminate the lead variable from the rest of the equations. If we apply pivoting to the three equations, we end up with the following set of equations:

$$2X + 3Y + Z = 11 \tag{A.9}$$

$$5Y + 2Z = 16 \tag{A.10}$$

$$X + 2Y + 2Z = 11 \tag{A.11}$$

Note that the coefficient for X in Eq. (A.9) is no longer zero. We can proceed with Gaussian elimination, as illustrated in Fig. A.11.

The reader should follow the steps in Fig. A.11. The most important additional insight is that some equations might not have the variable that the algorithm is eliminating at the current step (see row 2 of step 1 in Fig. A.11). The designated thread does not need to do the division on the equation.

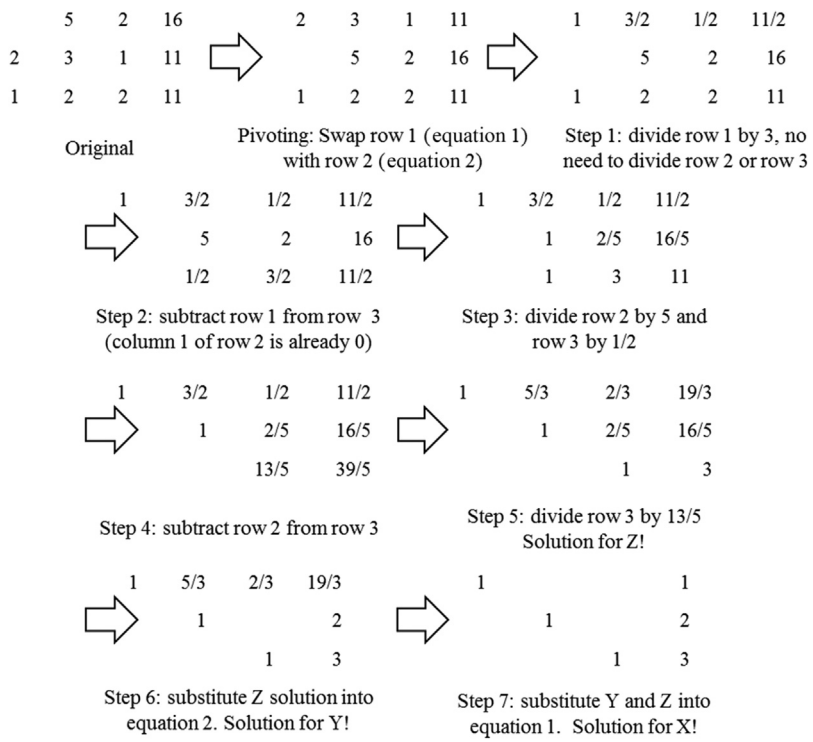


FIGURE A.11

Gaussian elimination with pivoting.

In general, the pivoting step should choose the equation with the largest absolute coefficient value among all the lead variables and swap its equation (row) with the current top equation as well as swap the variable (column) with the current variable. While pivoting is conceptually simple, it can incur significant implementation complexity and performance overhead. In the case of our simple CUDA kernel implementation, recall that each thread is assigned a row. Pivoting requires an inspection and perhaps swapping of coefficient data spread across these threads. This is not a big problem if all coefficients are in the shared memory. We can run a parallel max reduction using threads in the block as long as we control the level of control flow divergence within warps.

However, if the system of linear equations is being solved by multiple thread blocks or even multiple nodes of a compute cluster, the idea of inspecting data that is spread across multiple thread blocks or multiple compute cluster nodes can be an extremely expensive proposition. This is the main motivation for *communication-avoiding algorithms* that avoid a global inspection of data such as pivoting (Ballard et al., 2011). In general, there are two approaches to this problem. Partial pivoting restricts the candidates of the swap operation to come from a

localized set of equations so that the cost of global inspection is limited. However, this can slightly reduce the numerical accuracy of the solution. Researchers have also demonstrated that randomization tends to maintain a high level of numerical accuracy for the solution.

A.7 Summary

In this appendix we introduced the concepts of floating-point format and representable numbers, which are foundational to the understanding of precision. On the basis of these concepts, we explained the denormalized numbers and why they are important in many numerical applications. In early CUDA devices, denormalized numbers were not supported. However, more recent hardware generations support denormalized numbers. We also explained the concept of arithmetic accuracy of floating-point operations. It is important for CUDA programmers to understand the potential lower accuracy of fast arithmetic operations implemented in the special function units. More important, the reader should now have a good understanding of why parallel algorithms can often affect the accuracy of calculation results and how sorting and other techniques can be used to improve the accuracy of computation.

Exercises

1. Draw the equivalent of Fig. A.5 for a 6-bit format (1-bit sign, 3-bit mantissa, 2-bit exponent). Use your result to explain what each additional mantissa bit does to the set of representable numbers on the number line.
2. Draw the equivalent of Fig. A.5 for another 6-bit format (1-bit sign, 2-bit mantissa, 3-bit exponent). Use your result to explain what each additional exponent bit does to the set of representable numbers on the number line.
3. Assume that in a new processor design, owing to technical difficulty, the floating-point arithmetic unit that performs addition can only do "round to zero" (rounding by truncating the value toward 0). The hardware maintains a sufficient number of bits that the only error that is introduced is due to rounding. What is the maximal ULP error value for add operations on this machine?
4. A graduate student wrote a CUDA kernel to reduce a large floating-point array to the sum of all its elements. The array will always be sorted from the smallest values to the largest values. To avoid branch divergence, the student decided to implement the algorithm of Fig. A.4. Explain why this can reduce the accuracy of the results.
5. Assume that in an arithmetic unit design, the hardware implements an iterative approximation algorithm that generates two additional accurate

mantissa bits of the result for the sin() function in each clock cycle. The architect decided to allow the arithmetic function to iterate nine clock cycles. Assume that the hardware fills in all remaining mantissa bits as 0's. What would be the maximal ULP error of the hardware implementation of the sin() function in this design for IEEE single-precision numbers? Assume that the omitted "1." mantissa bit must also be generated by the arithmetic unit.

References

IEEE Microprocessor Standards Committee, 2008. Draft Standard for Floating-Point Arithmetic P754, Most recent revision January 2008.

Kahan, W., 1965. Further remarks on reducing truncation errors. Commun. ACM 8 (1), 40. Available from: https://doi.org/10.1145/363707.363723.

Ballard, G., Demmel, J., Holtz, O., Schwartz, O., 2011. Minimizing communication in numerical linear algebra. SIAM J. Matrix Anal. Appl. 32 (3), 866–901.

Index

Note: Page numbers followed by "*b*," "*f*," and "*t*" refer to boxes, figures, and tables, respectively.

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z