CHAPTER 12

Merge

An introduction to dynamic input data identification

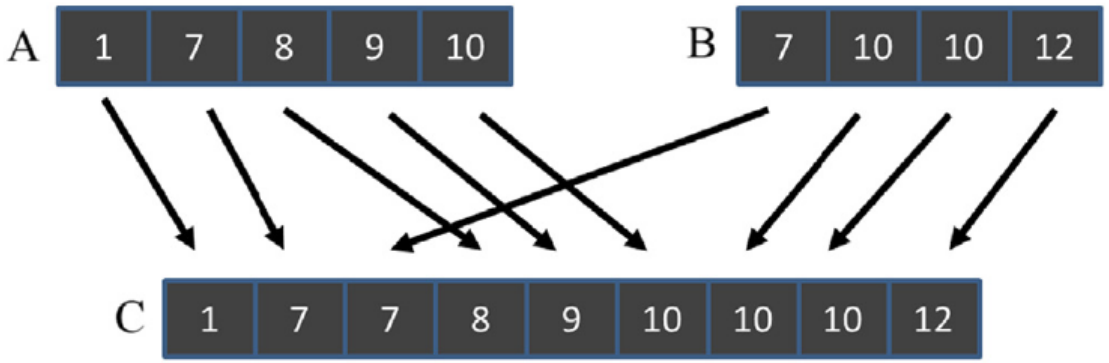

FIGURE 12.1

Example of a merge operation.

```
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.

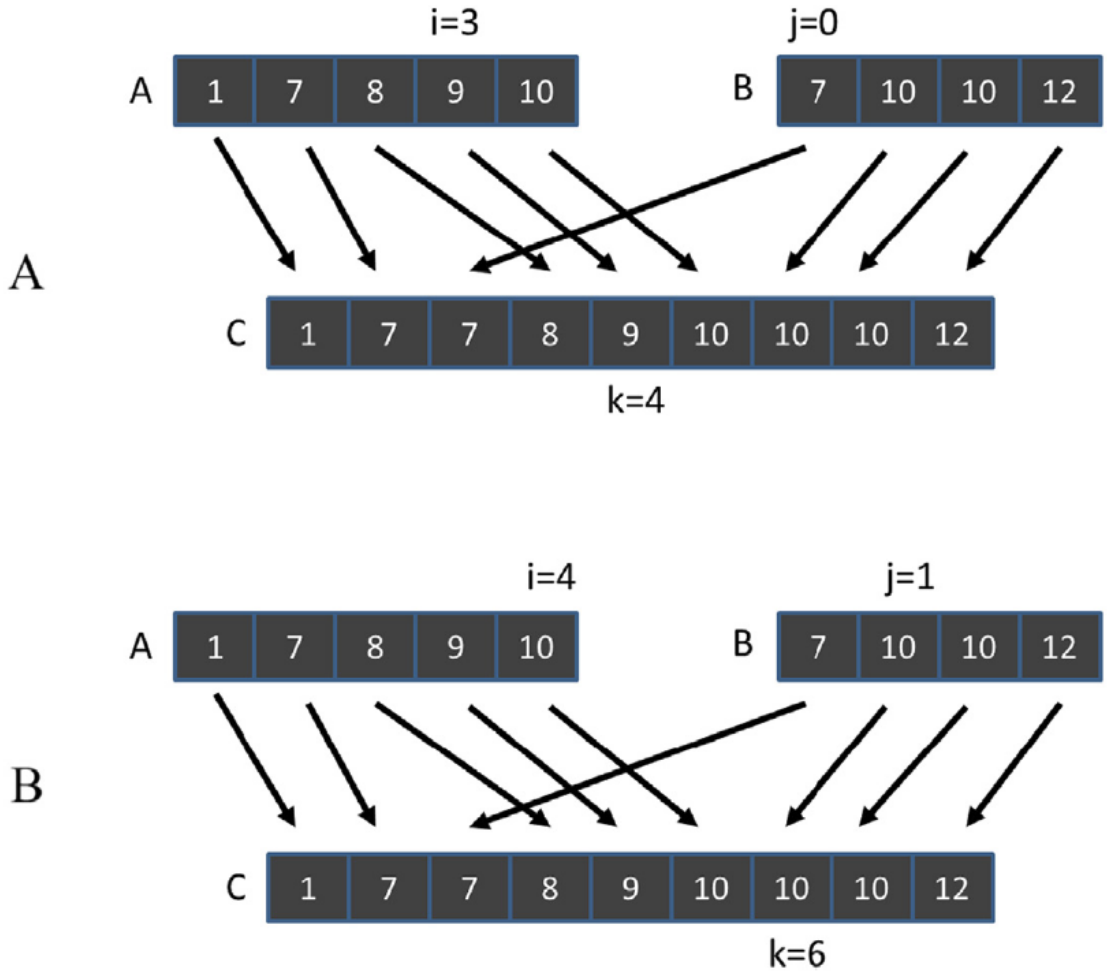

FIGURE 12.3

Examples of observation 1.

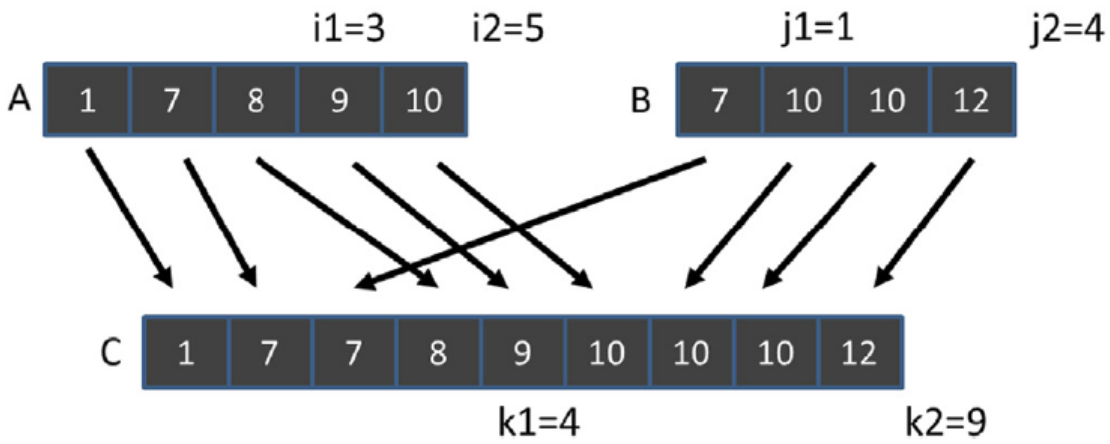

FIGURE 12.4

Example of co-rank function execution.

```
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.

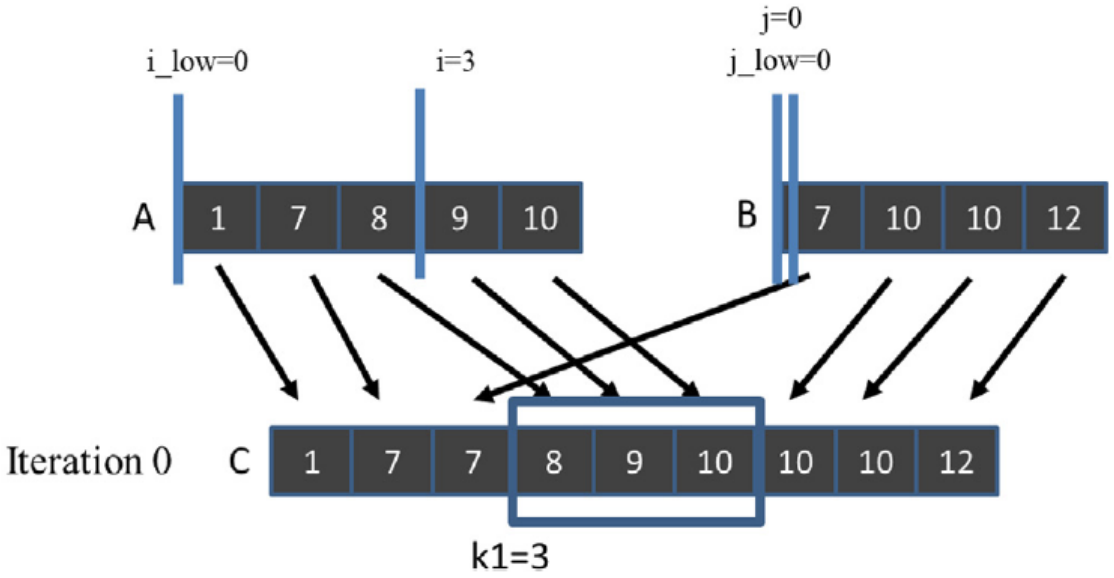

FIGURE 12.6

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

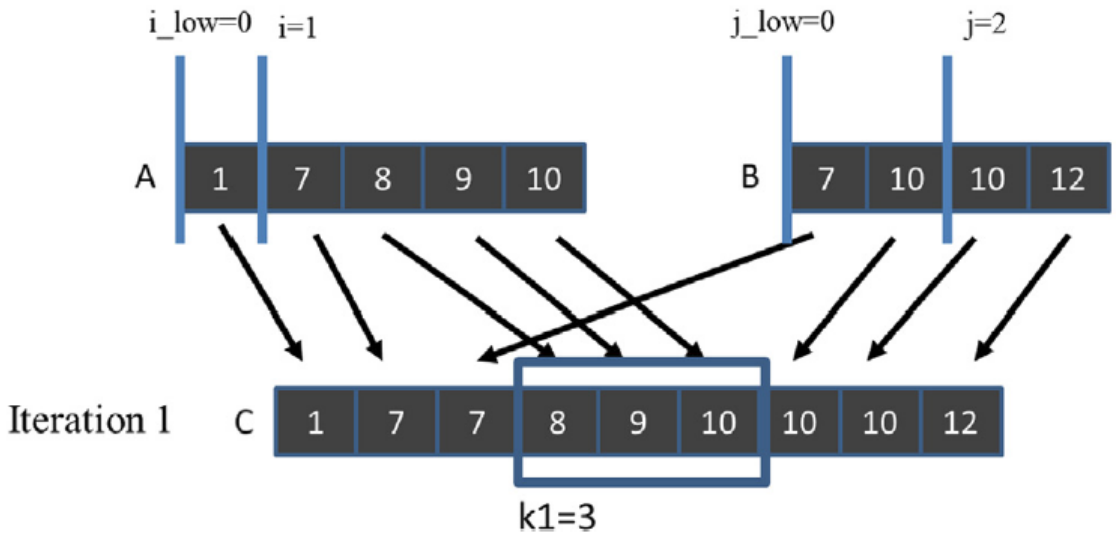

FIGURE 12.7

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

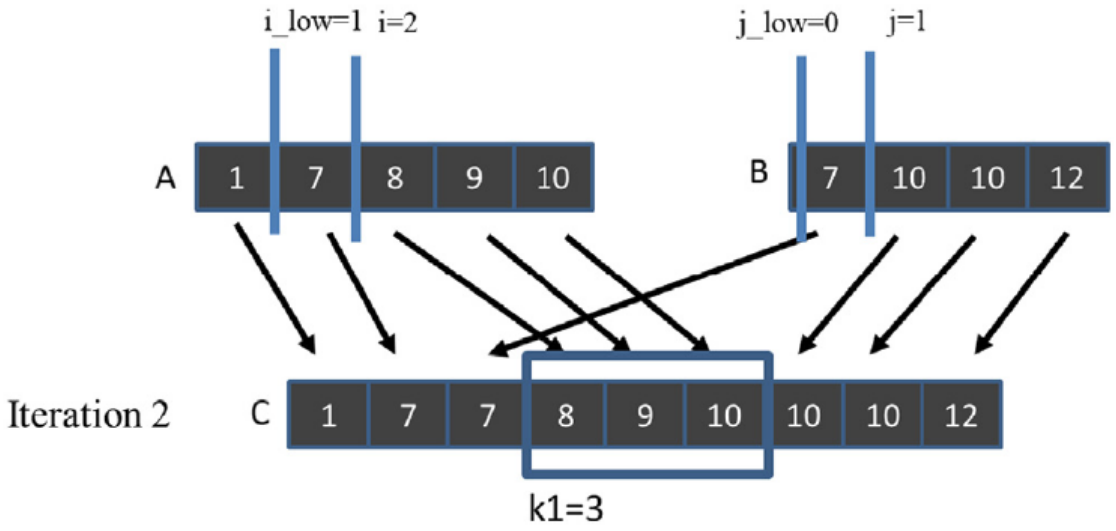

FIGURE 12.8

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

```
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.

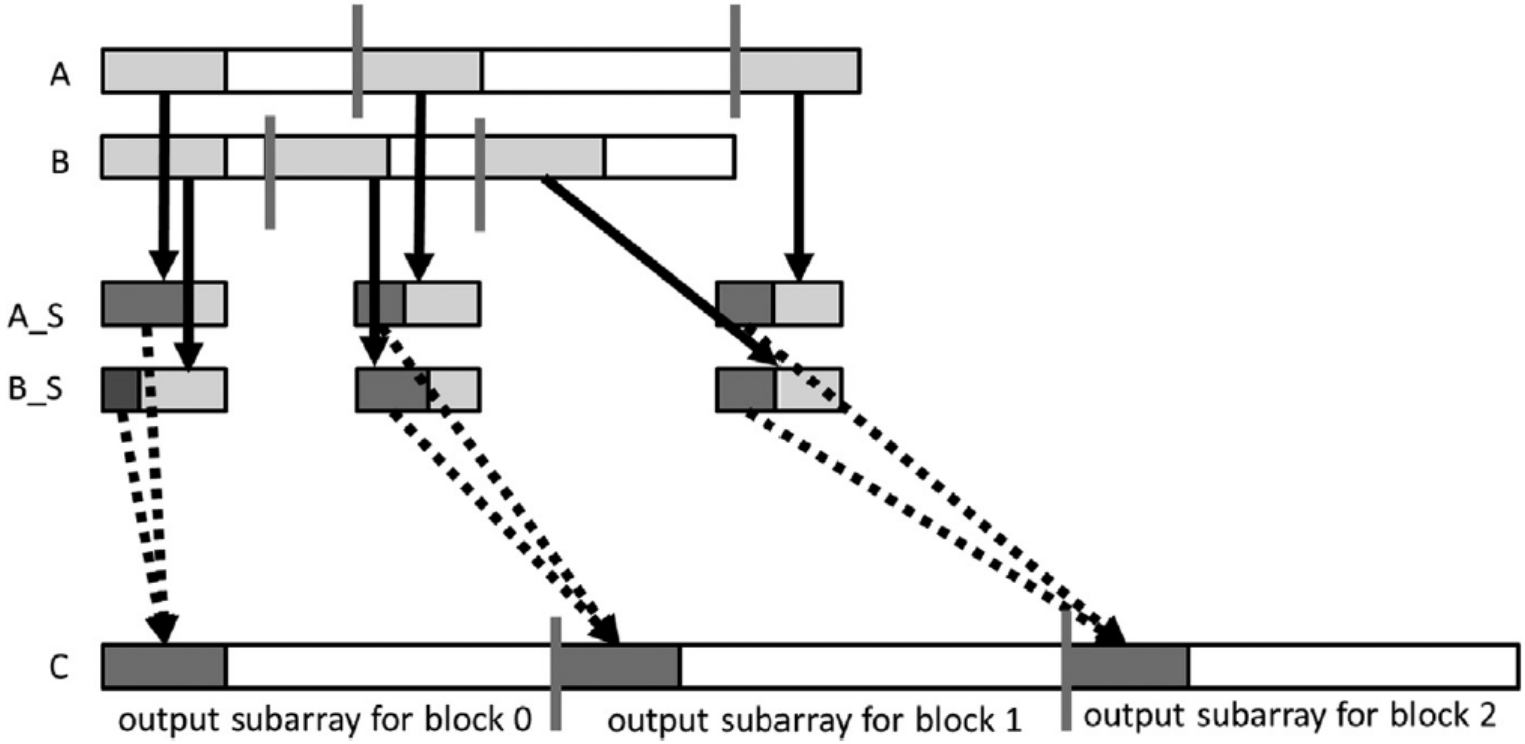

FIGURE 12.10

Design of a tiled merge kernel.

```
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.

```
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.

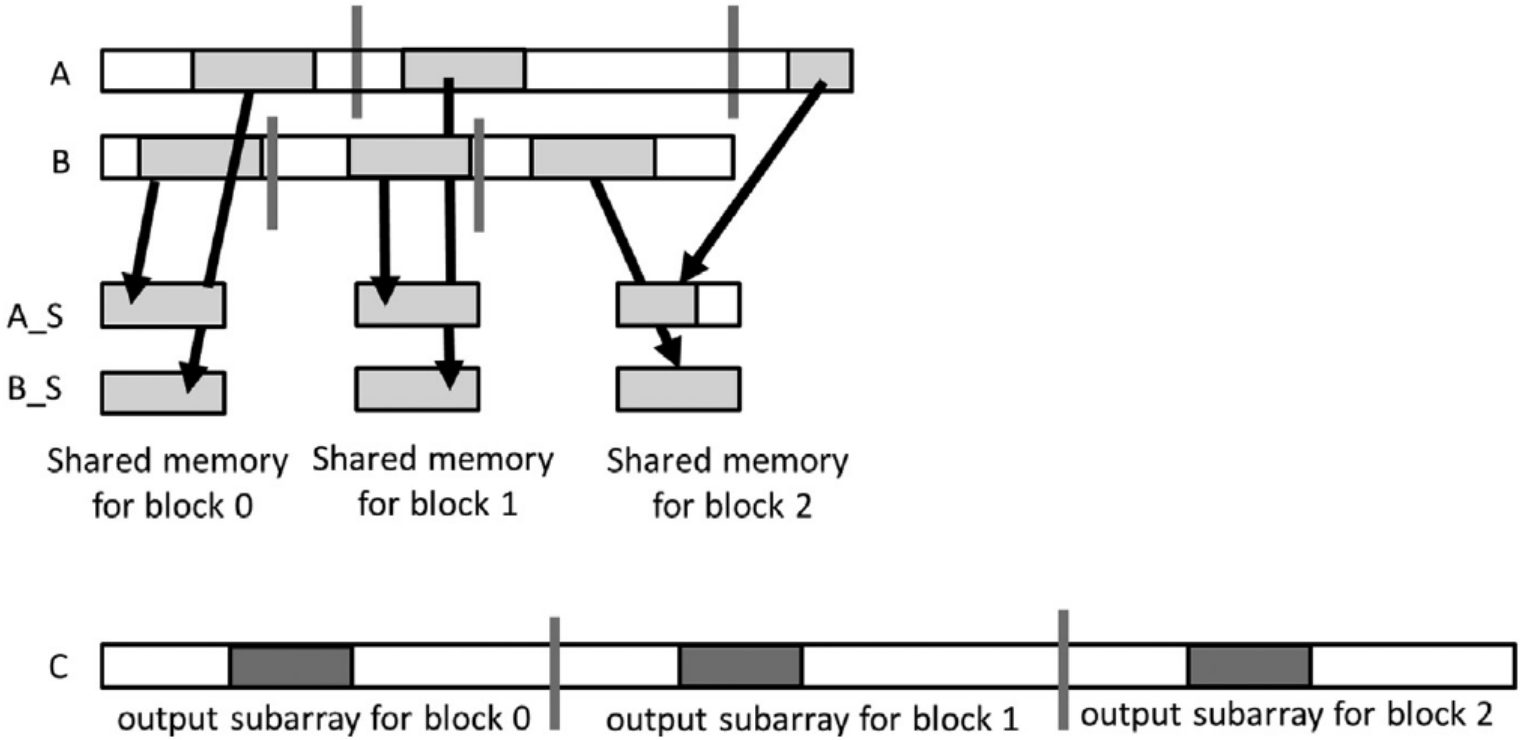

FIGURE 12.14

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

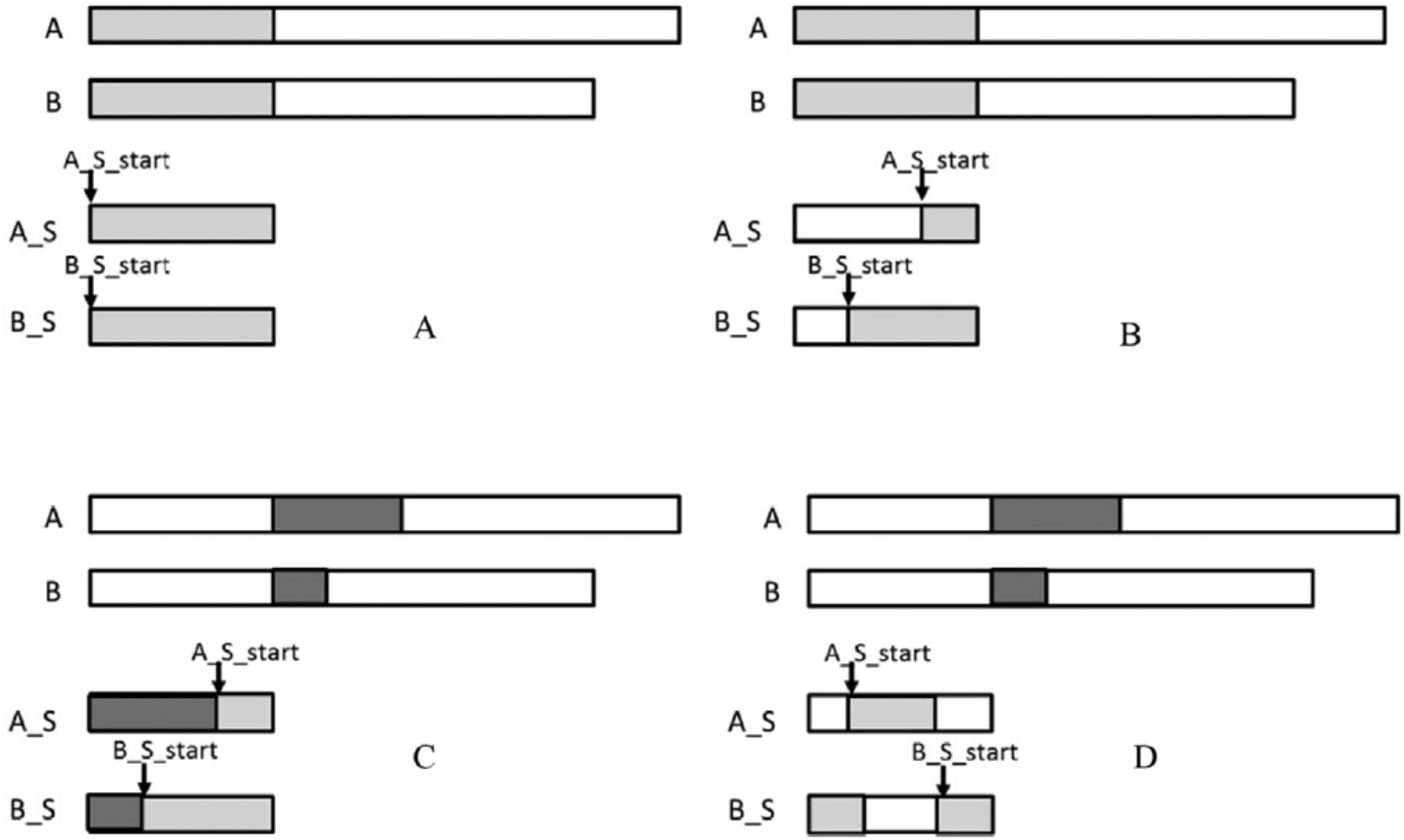

FIGURE 12.15

A circular buffer scheme for managing the shared memory tiles.

```
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.

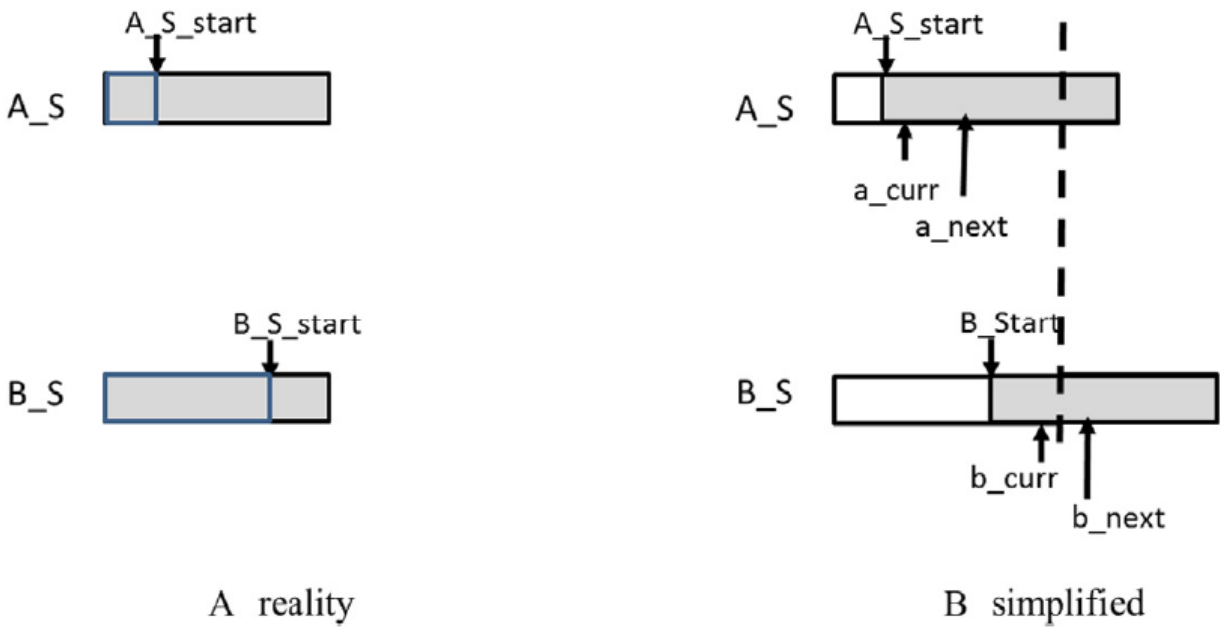

FIGURE 12.17

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

```
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.

```
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.

```
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.

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

In-text figure 1

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

In-text figure 2

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

In-text figure 3