PARALLEL PATTERNS: MERGE SORT

Data Parallelism / Data-Dependent Execution

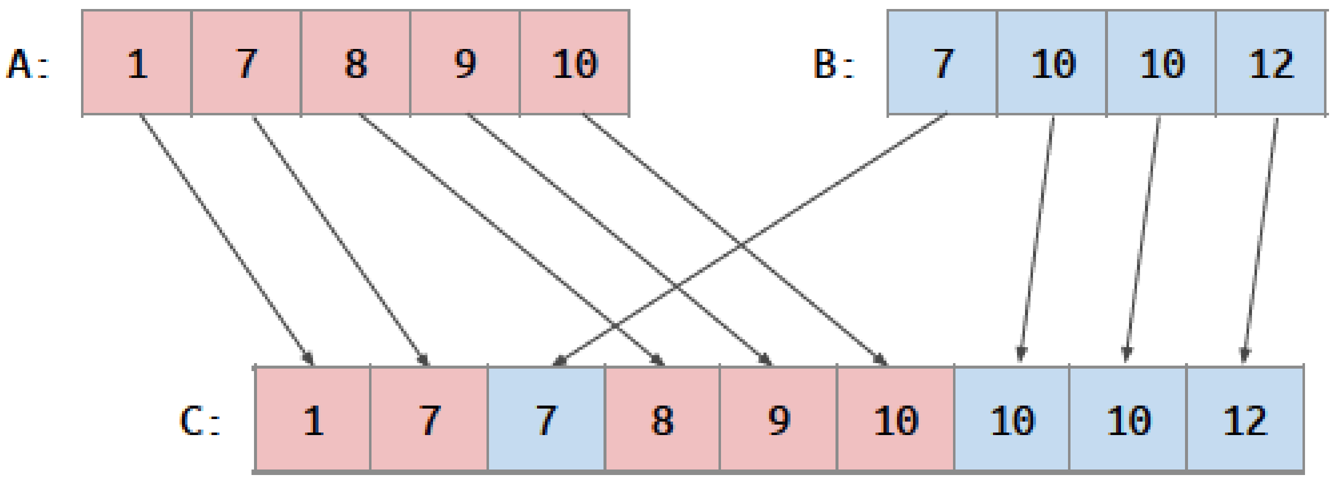
	Data-Independent	Data-Dependent
Data Parallel	Stencil Histogram	SpMV
Not Data Parallel	Prefix Scan	**Merge**

Objective

- Study increasingly sophisticated parallel merge kernels
- Observe the combined effects of data - dependent execution and a lack of data parallelism on GPU algorithm design

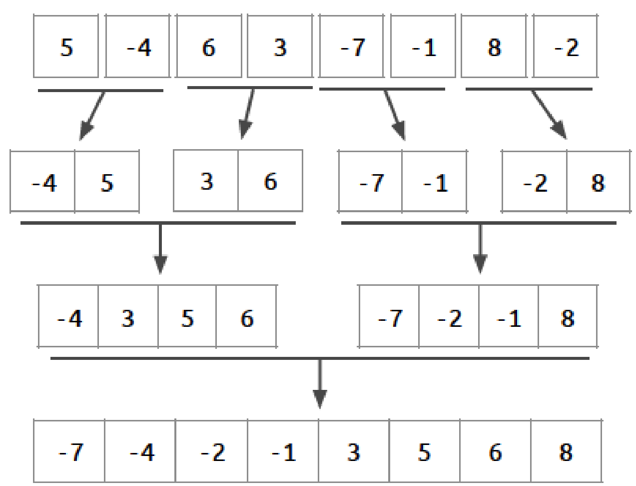
Merge Sort

- **Input: two sorted arrays**
- **Output: the (sorted) union of the input**



Merge Sort

- A bottom-up divide-and-conquer sorting algorithm
- O(n log n) average - (and worst -) case performance
- O(n) additional space requirement
- Merging two arrays is the core computation



Other Uses for Merge

- **Taking the union of two (non-overlapping) sparse matrices represented in the CSR format**
  - Each row is merged
  - col_indices are the keys
- **In MapReduce, when Map produces sorted key-value pairs and Reduce must**
  - maintain sorting

Sequential Merge

```
void merge(const int * A, int m, const int * B, int n, int * C) {
   int i = 0; // Index into A
   int j = 0; // Index into B
   int k = 0; // Index into C
           // merge the initial overlapping sections of A and B
   while ((i < m) && (j < n)) {
      if (A[i] <= B[j]) {
         C[k++] = A[i++];
      } else {
         C[k++] = B[j++];
      }
   }
   if (i == m) {
         // done with A, place the rest of B
      for ( ; j < n; j++) {
         C[k++] = B[j];
      }
   } else {
         // done with B, place the rest of A
      for ( ; i < m; i++) {
         C[k++] = A[i];
      }
   }
}
```

k increases by one for every iteration of the loops

In any given iteration (other than the first), the values of i and j are data-dependent

Sequential Merge Parallelization Challenges

- **We could assign one thread to write each output element**
- **However, given a particular output location, the input element that belongs**
  - there is data-dependent
- **The sequential merge is O(n) in the length of the output array**
  - so we must be work-efficient

Observations about Merge

- **1. For any k s.t. $0 \le k < m + n$, there is either:**
  - a. an i s.t. $0 \le i < m$ and $C[k] \Leftarrow A[i]$
  - b. a j s.t. $0 \le j < n$ and $C[k] \Leftarrow B[j]$
- **2. For any k s.t. $0 \le k < m + n$, there is an *i* and a *j* s.t. :**
  - a. $i + j = k$
  - b. $0 \le i \le m$
  - c. $0 \le j \le n$
  - d. The subarray C[0 : k-1] is the result of merging A[0 : i-1] and B[0 : j-1]

Indices *i* and *j* are referred to as co-ranks

A Merge Parallelization Approach

- Assume a co-rank function of the form:

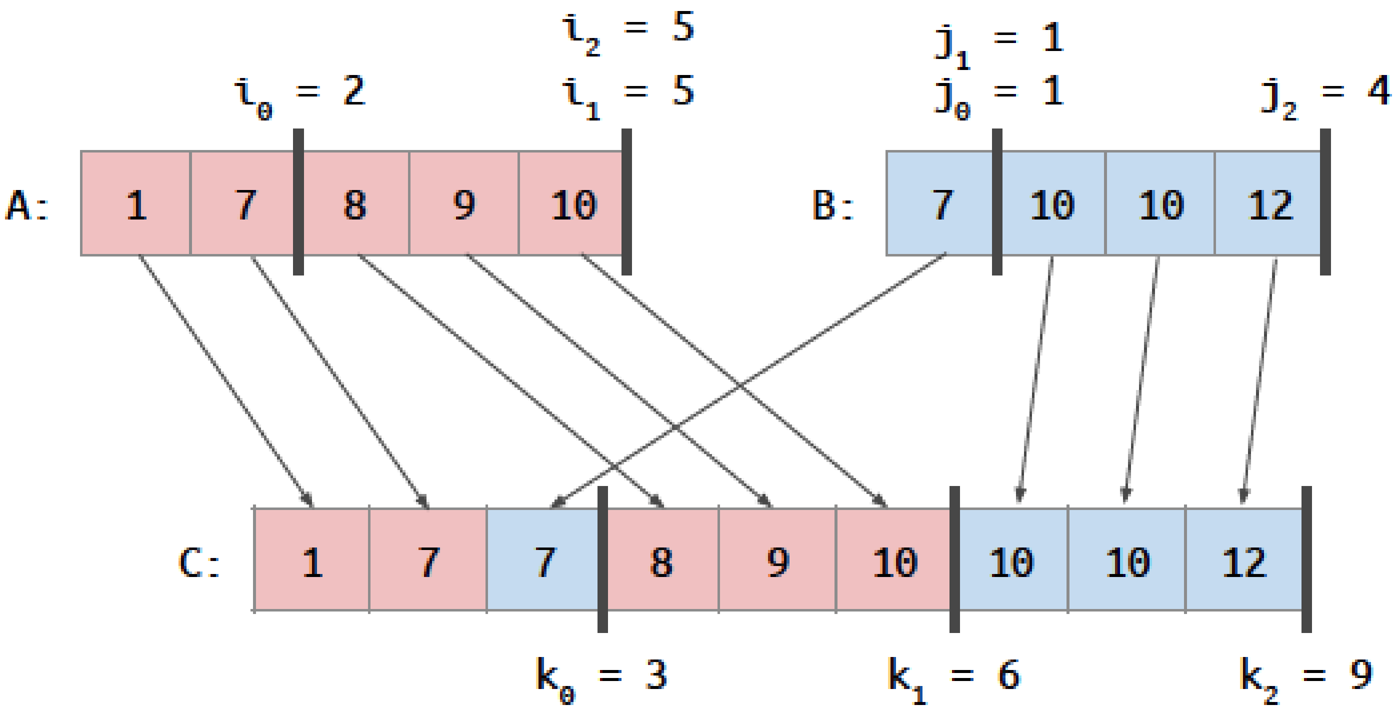
  co-rank(k,A,B) = i

  j = k - i

- We can use the co-rank function to map a range of output values to a range of input values
- We'll need to compute co-rank efficiently for a work-efficient merge

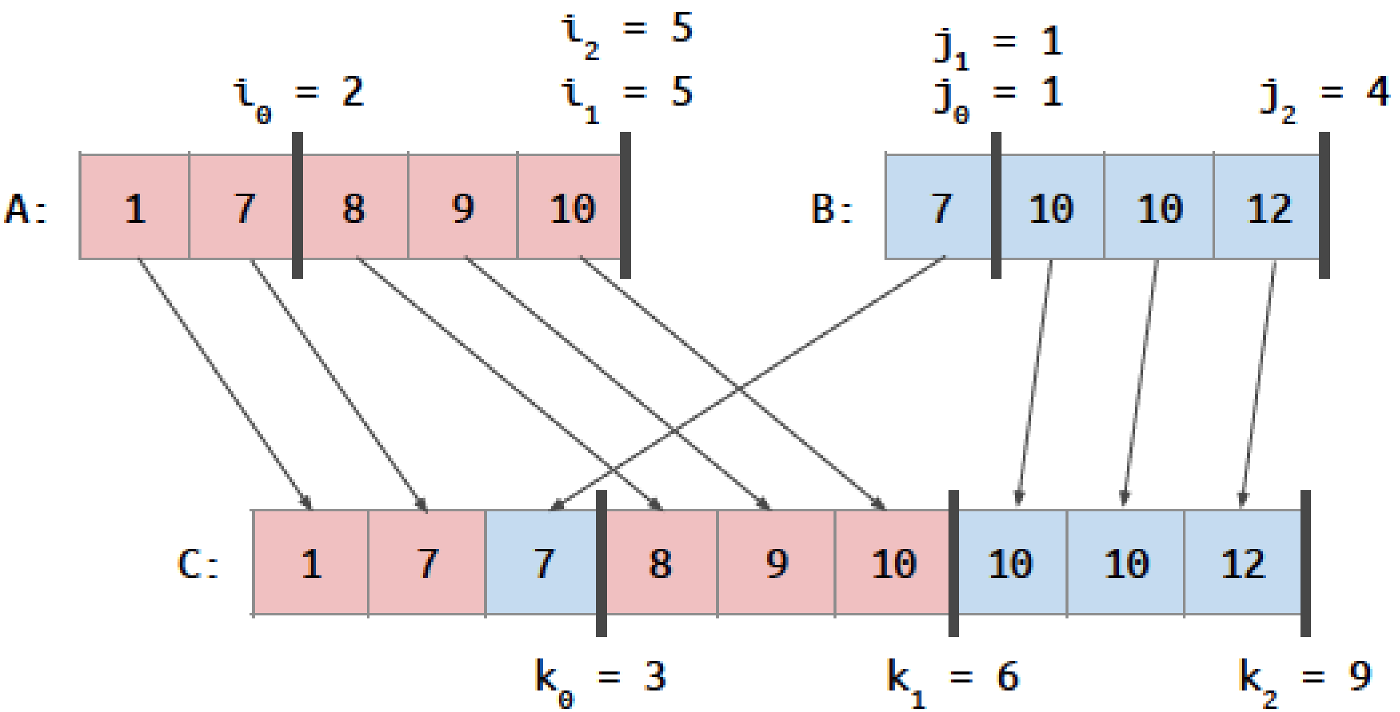
Merge with Co-rank

- **Divide the output into regular intervals**
- **Compute the co-ranks for each output interval endpoint**



Merge with Co-rank

- Thread 0: C[0 : k0] ⇐ merge(A[0 : i0], B[0 : j0])
- Thread 1: C[k0 : k1] ⇐ merge(A[i0 : i1], B[j0 : j1])
- Thread 2: C[k1 : k2] ⇐ merge(A[i1 : i2], B[j1 : j2])



Sequential Co-rank Implementation

```
int co_rank(int k, int * A, int m, int * B, int n) {
  int i = k < m ? k : m; // initial guess for i
  int j = k - i; // corresponding j
  int i_low = 0 > (k-n) ? 0 : k-n; // lower bound on i
  int j_low = 0 > (k-m) ? 0 : k-m; // lower bound on j
  int delta;
  while(true) {
    if (i > 0 && j < n && A[i-1] > B[j]) {
        // first excluded B comes before last included A
      delta = ((i - i_low + 1) >> 1);
      j_low = j;
      j = j + delta;
      i = i - delta;
    } else if (j > 0 && i < m && B[j-1] >= A[i]) {
        // first excluded A comes before last included B
      delta = ((j - j_low + 1)>>1);
          i_low = i;
          i = i + delta;
          j = j - delta;
          } else {
        break;
      }
  }
  return i;
}
```

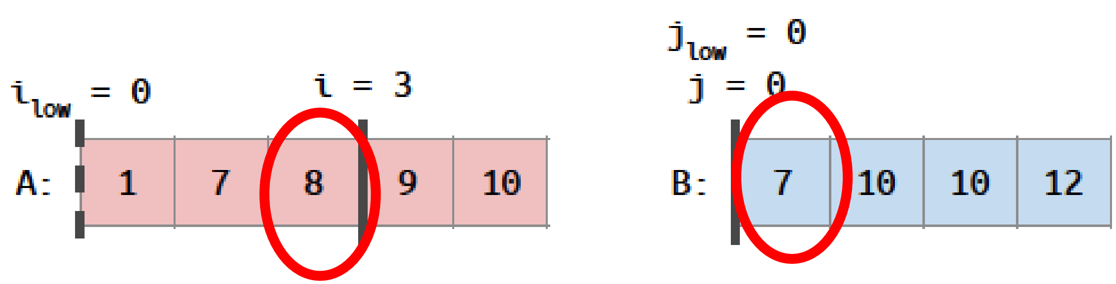
Sequential Co-rank Implementation

```
int co_rank(int k, int * A, int m, int * B, int n) {
      int i = k < m ? k : m; // initial guess for i
      int j = k - i;         // corresponding j
      int i_low = 0 > (k-n) ? 0 : k-n; // lower bound on i
      int j_low = 0 > (k-m) ? 0 : k-m; // lower bound on j
      int delta;
      while(true) {
            if (i > 0 && j < n && A[i-1] > B[j]) {
                  // first excluded B comes before last included A
                  delta = ((i - i_low + 1) >> 1);
                  j_low = j;
                  j = j + delta;
                  i = i - delta;
            } else if (j > 0 && i < m && B[j-1] >= A[i]) {
                  // first excluded A comes before last included B
                  delta = ((j - j_low + 1)>>1);
                  i_low = i;
                  i = i + delta;
                  j = j - delta;
            } else {
                  break;
            }
      }
      return i;
}
```

This case is true

delta = 2

k = 3



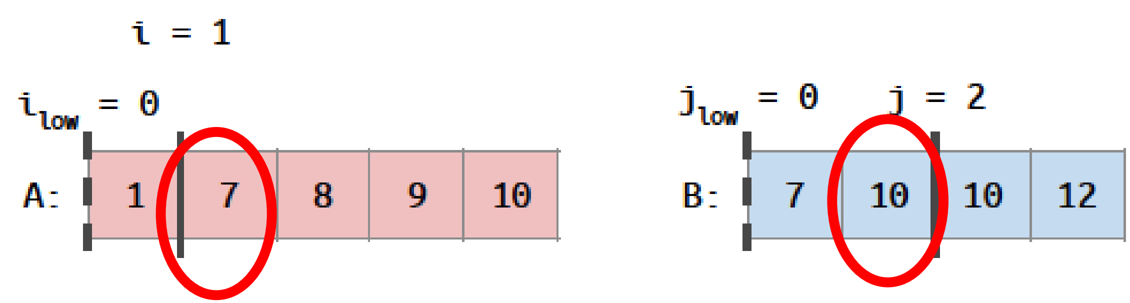
Sequential Co-rank Implementation

```
int co_rank(int k, int * A, int m, int * B, int n) {
      int i = k < m ? k : m; // initial guess for i
      int j = k - i;         // corresponding j
      int i_low = 0 > (k-n) ? 0 : k-n; // lower bound on i
      int j_low = 0 > (k-m) ? 0 : k-m; // lower bound on j
      int delta;
      while(true) {
            if (i > 0 && j < n && A[i-1] > B[j]) {
                  // first excluded B comes before last included A
                  delta = ((i - i_low + 1) >> 1);
                  j_low = j;
                  j = j + delta;
                  i = i - delta;
            } else if (j > 0 && i < m && B[j-1] >= A[i]) {
                  // first excluded A comes before last included B
                  delta = ((j - j_low + 1)>>1);
                  i_low = i;
                  i = i + delta;
                  j = j - delta;
            } else {
                  break;
            }
      }
      return i;
}
```

This case is true

delta = 1

k = 3

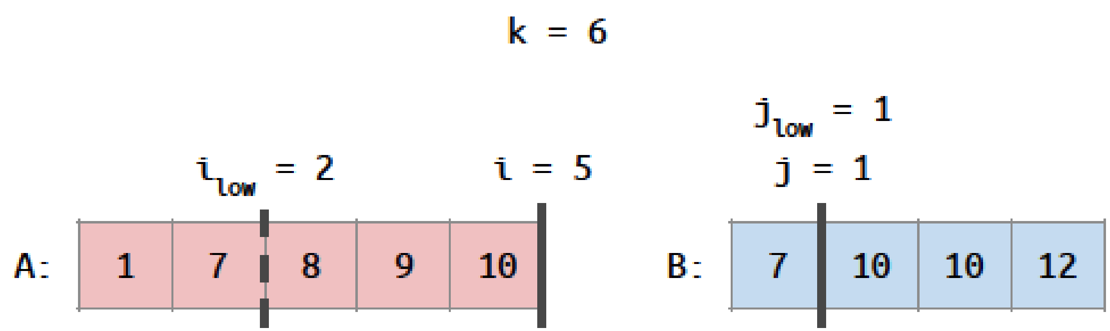


Sequential Co-rank Implementation

```c
int co_rank(int k, int * A, int m, int * B, int n) {
      int i = k < m ? k : m; // initial guess for i
      int j = k - i;         // corresponding j
      int i_low = 0 > (k-n) ? 0 : k-n; // lower bound on i
      int j_low = 0 > (k-m) ? 0 : k-m; // lower bound on j
      int delta;
      while(true) {
            if (i > 0 && j < n && A[i-1] > B[j]) {
                  // first excluded B comes before last included A
                  delta = ((i - i_low + 1) >> 1);
                  j_low = j;
                  j = j + delta;
                  i = i - delta;
            } else if (j > 0 && i < m && B[j-1] >= A[i]) {
                  // first excluded A comes before last included B
                  delta = ((j - j_low + 1)>>1;
                  i_low = i;
                  i = i + delta;
                  j = j - delta;
            } else {
                  break;
            }
      }
      return i;
}
```

This case is false

This case is also false



Co-rank Computational Efficiency

- The search range is cut down by a factor of 2 in every iteration
- Thus worst-case run-time is O(log n) in the length of the output sequence
- Co-rank towards the ends is more efficient as the initial search range is smaller

Basic Parallel Merge Kernel

```
__global__ void basicMergeKernel(const int * A, int m, const int * B, int n, int * C) {
   int tid = blockIdx.x * blockDim.x * threadIdx.x;
   int sectionSize = ((m + n - 1) / ( blockDim.x * gridDim.x) + 1; // number of outputs per thread
  int thisK = tid * sectionSize; // starting index of this thread’s output
  int nextK = min((tid + 1) * sectionSize, m + n); // final index of this thread’s output
  int thisI = co_rank(thisK, A, m, B, n);
  int nextI = co_rank(nextK, A, m, B, n);
  int thisJ = thisK - thisI;
  int nextJ = nextK - nextI;
  // run sequential merge with all threads in parallel
  merge(&A[thisI], nextI - thisI, &B[thisJ], nextJ - thisJ, &C[thisK]);
}
```

Sequential Co-rank Implementation (CUDA version)

```
__device__
int co_rank(int k, int * A, int m, int * B, int n) {
   int i = k < m ? k : m; // initial guess for i
   int j = k - i; // corresponding j
   int i_low = 0 > (k-n) ? 0 : k-n; // lower bound on i
   int j_low = 0 > (k-m) ? 0 : k-m; // lower bound on j
   int delta;
   while(true) {
      if (i > 0 && j < n && A[i-1] > B[j]) {
            // first excluded B comes before last included A
         delta = ((i - i_low + 1) >> 1);
         j_low = j;
         j = j + delta;
         i = i - delta;
      } else if (j > 0 && i < m && B[j-1] >= A[i]) {
            // first excluded A comes before last included B
         delta = ((j - j_low + 1)>>1);
          i_low = i;
          i = i + delta;
          j = j - delta;
       } else {
            break;
      }
   }
   return i;
}
```

Sequential Merge (CUDA version)

```
__device__
void merge(const int * A, int m, const int * B, int n, int * C) {
   int i = 0; // Index into A
   int j = 0; // Index into B
   int k = 0; // Index into C
            // merge the initial overlapping sections of A and B
   while ((i < m) && (j < n)) {
      if (A[i] <= B[j]) {
         C[k++] = A[i++];
      } else {
         C[k++] = B[j++];
      }
   }
   if (i == m) {
         // done with A, place the rest of B
      for ( ; j < n; j++) {
         C[k++] = B[j];
      }
   } else {
         // done with B, place the rest of A
      for ( ; i < m; i++) {
         C[k++] = A[i];
      }
   }
}
```

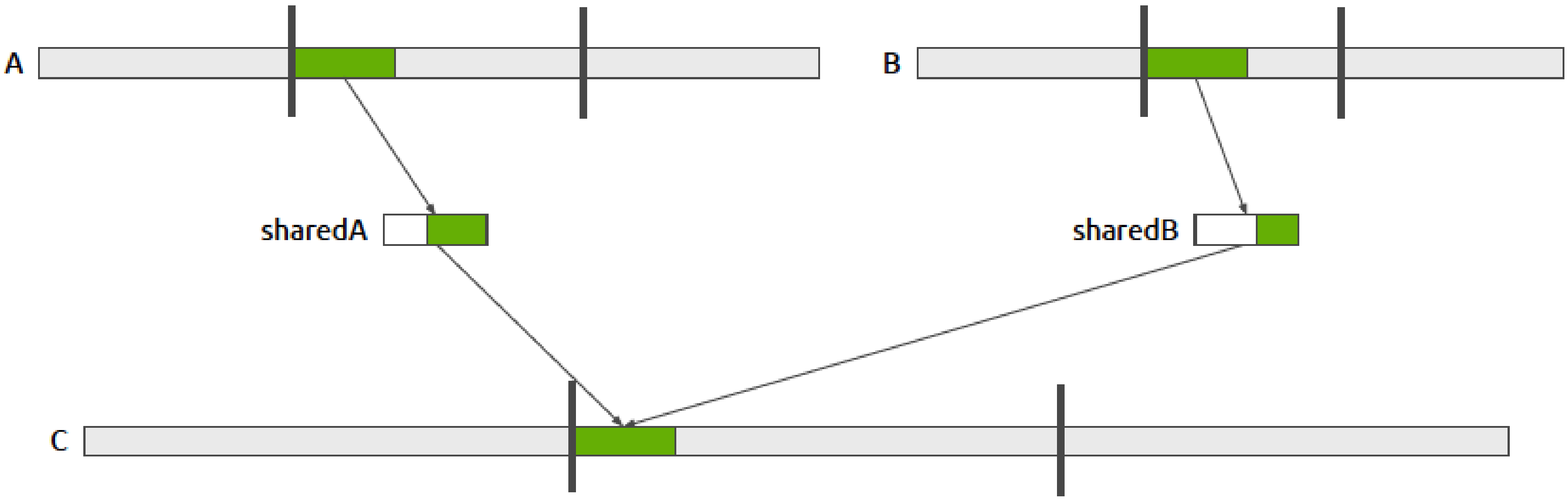
Basic Parallel Merge Kernel Shortcomings

- **Non-coalesced memory access!**
  - A single thread processes neighboring input / output values
  - The co-rank function operates on global memory and has a highly irregular memory access pattern

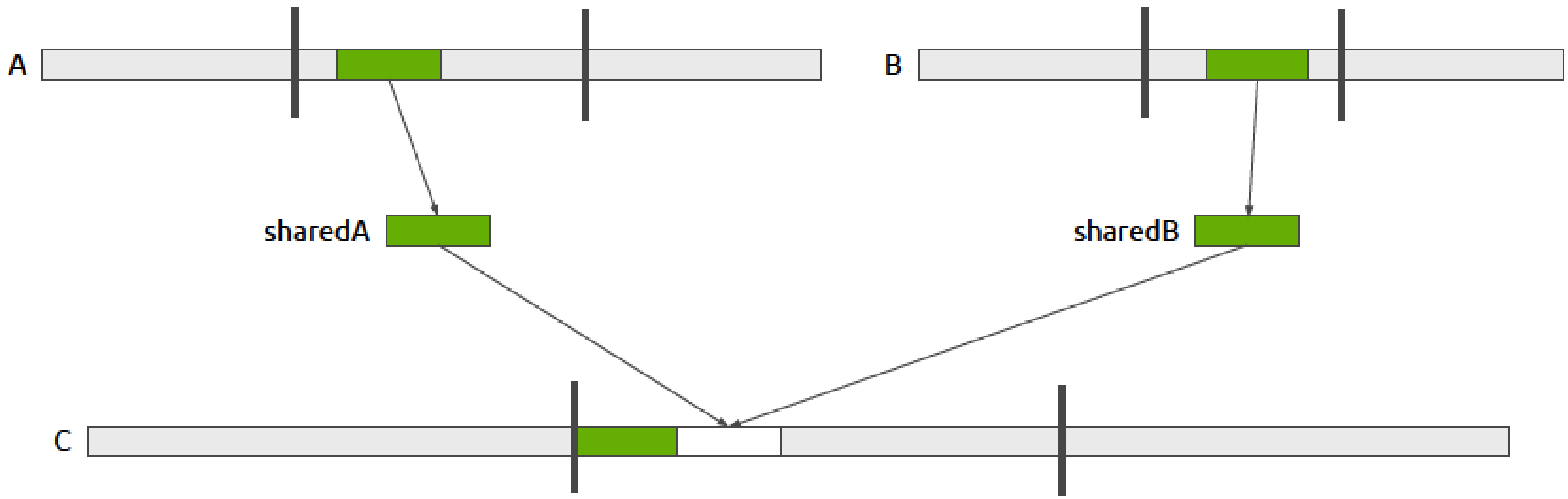
Tiled Parallel Merge Kernel

- **Blocks of threads collaboratively produce a contiguous chunk of output using a contiguous chunk of each input**
- **The location and size of the output chunk is known given the block index**
- **Both the location and the size of the input chunks are data dependent**
- **The co-rank function can also be used to find the size and locations of the input chunks**

Tiled Parallel Merge Kernel (visually)



Tiled Parallel Merge Kernel (visually)



Example Grid / Tile Sizing

- **Assume output C will have 65,536 values**
- **We can use 16 blocks such that each block processes 4096 output elements**
- **We can use a tile size of 1024 elements from each input array**
- **This will require 4 tile loading / computation phases**
- **We can use blocks of 128 threads**
- **Each thread thus loads 8 values from each input and computes 8 output values in each phase**

Notes on Tiled Parallel Merge Kernel

- **Unlike in tiled matrix multiplication, tile locations (other than the first) depend on input values**
- **The amount of data that has been "consumed" from each input array must be tracked by the kernel**

Tiled Parallel Merge Kernel (Part 1)

```
__global__ void tiledMergeKernel(const int * A, int m, const int * B, int n, int * C) {

      extern __shared__ int sharedAB[];
      int * sharedA = &sharedAB[0];
      int * sharedB = &sharedAB[TILE_SIZE];

      int k_blockStart = blockIdx.x * ((m + n - 1) / gridDim.x) + 1;           // start point for block
      int k_blockEnd = min(blockIdx.x * ((m + n - 1) / gridDim.x) + 1, m + n); // end point for block

      if (threadIdx.x == 0) {
            sharedA[0] = co_rank(k_blockStart, A, m, B, n); // Compute block-level co-rank values
            sharedA[1] = co_rank(k_blockEnd, A, m, B, n);   // with a single thread, make it shared
      }
      __syncthreads();

      int i_blockStart = A_S[0];                          // fetch block bounds out of shared memory
      int i_blockEnd = A_S[1];                            // save in local registers

      int j_blockStart = k_blockStart - i_blockStart;
      int j_blockEnd = k_blockEnd - i_blockEnd;

      __syncthreads();
```

Tiled Parallel Merge Kernel (Part 2)

```
int counter = 0;

int A_chunkSize = i_blockEnd - iBlockStart;
int B_chunkSize = j_blockEnd - jBlockStart;
int C_chunkSize = k_blockEnd - kBlockStart;

int numPhases = (C_chunkSize - 1) / TILE_SIZE + 1;

int A_consumed = 0;
int B_consumed = 0;
int C_completed = 0;

for (int phase = 0; phase < numPhases; ++phase) {

      // load A and B values into shared memory
      for (int i = 0; i < TILE_SIZE; i += blockDim.x) {

            if (i + threadIdx.x < A_chunkSize - A_consumed) {
                  sharedA[i + threadIdx.x] = A[i_blockStart + A_consumed + i + threadIdx.x];
            }
            if (i + threadIdx.x < B_chunkSize - B_consumed) {
                  sharedB[i + threadIdx.x] = B[j_blockStart + B_consumed + i + threadIdx.x];
            }

      }

      __syncthreads();
```

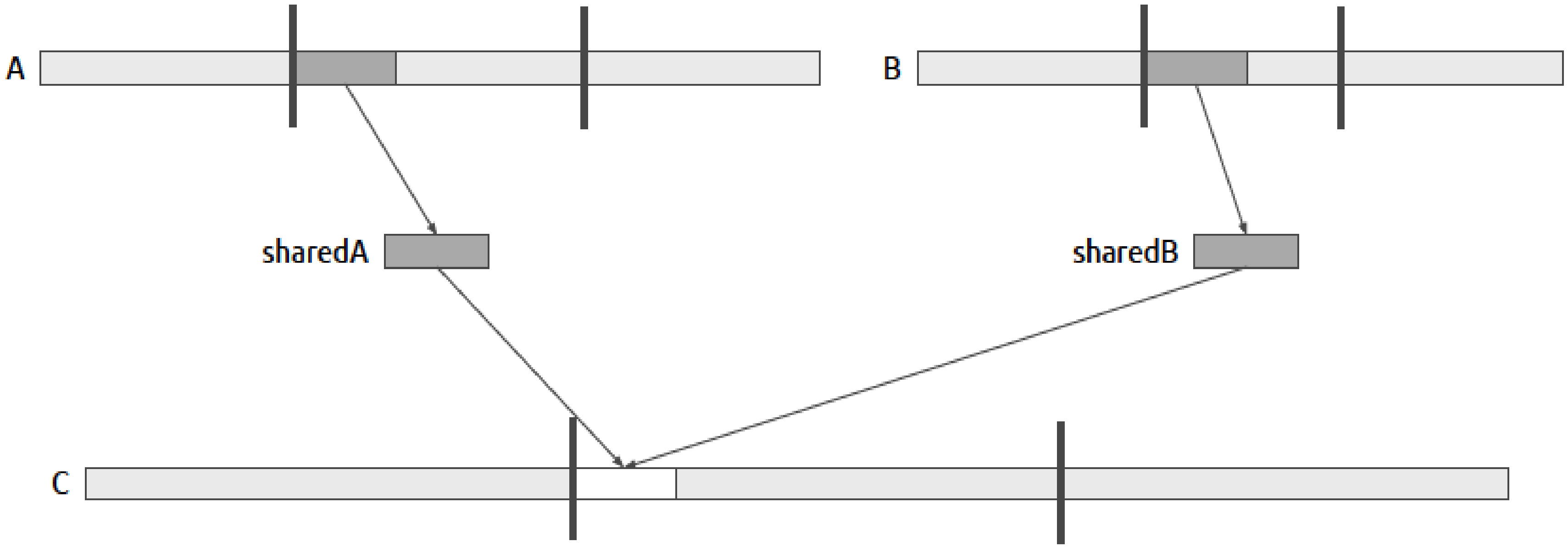
Tiled Parallel Merge Kernel (Part 3)

```
        int k_threadStart = min(threadIdx.x * (TILE_SIZE / blockDim.x), C_chunkSize - C_completed);
        int k_threadEnd = min((threadIdx.x+1) * (TILE_SIZE / blockDim.x), C_chunkSize - C_completed);

        // compute thread-level co-rank
        int i_threadStart = co_rank(k_threadStart, sharedA, min(TILE_SIZE, A_chunkSize - A_consumed),
                                    sharedB, min(TILE_SIZE, B_chunkSize - B_consumed);
        int j_threadStart = k_threadStart - i_threadStart;
        int i_threadEnd = co_rank(k_threadEnd, sharedA, min(TILE_SIZE, A_chunkSize - A_consumed),
                                  sharedB, min(TILE_SIZE, B_chunkSize - B_consumed);
        int j_threadEnd = k_threadEnd - i_threadEnd;

        // all threads do sequential merge
        merge_sequential(sharedA + i_threadStart, i_threadEnd - i_threadStart,
                         sharedB + j_threadStart, j_threadEnd - j_threadStart,
                         C + k_blockStart + C_completed + k_threadStart);

        // track the amount of A and B that have been "consumed" so far
        C_completed += TILE_SIZE;
        A_consumed += co_rank(TILE_SIZE, sharedA, TILE_SIZE, sharedB, TILE_SIZE);
        B_consumed = C_completed - A_consumed;
        __syncthreads();

    }

}
```

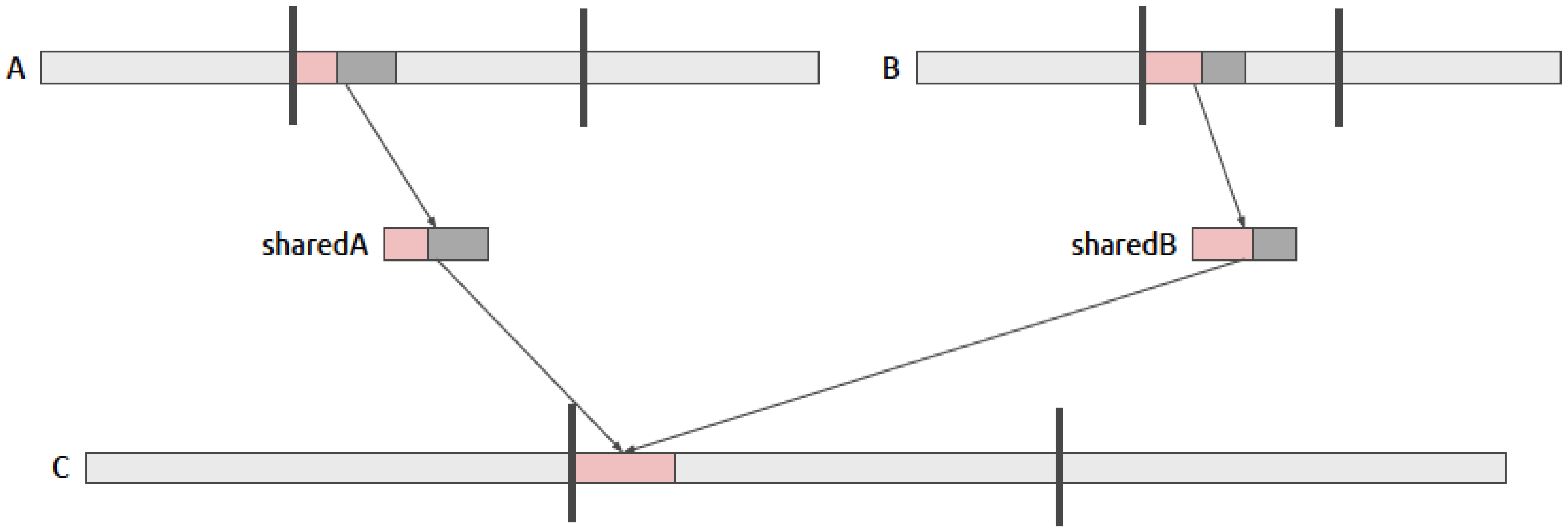
Tiled Parallel Merge Kernel Analysis

- **Pros:**
  - Global memory reads incurred by the co-rank have been greatly reduced
    - through the use of shared memory
  - Tiles are loaded in a coalesced pattern:
    - sharedA[i + threadIdx.x] = A[i_blockStart + A_consumed + i + threadIdx.x];
- **Cons:**
  - We load 2*TILE_SIZE values in every phase (TILE_SIZE each from A and B)
  - Only half of these values are actually written out to C, and the rest are thrown away.

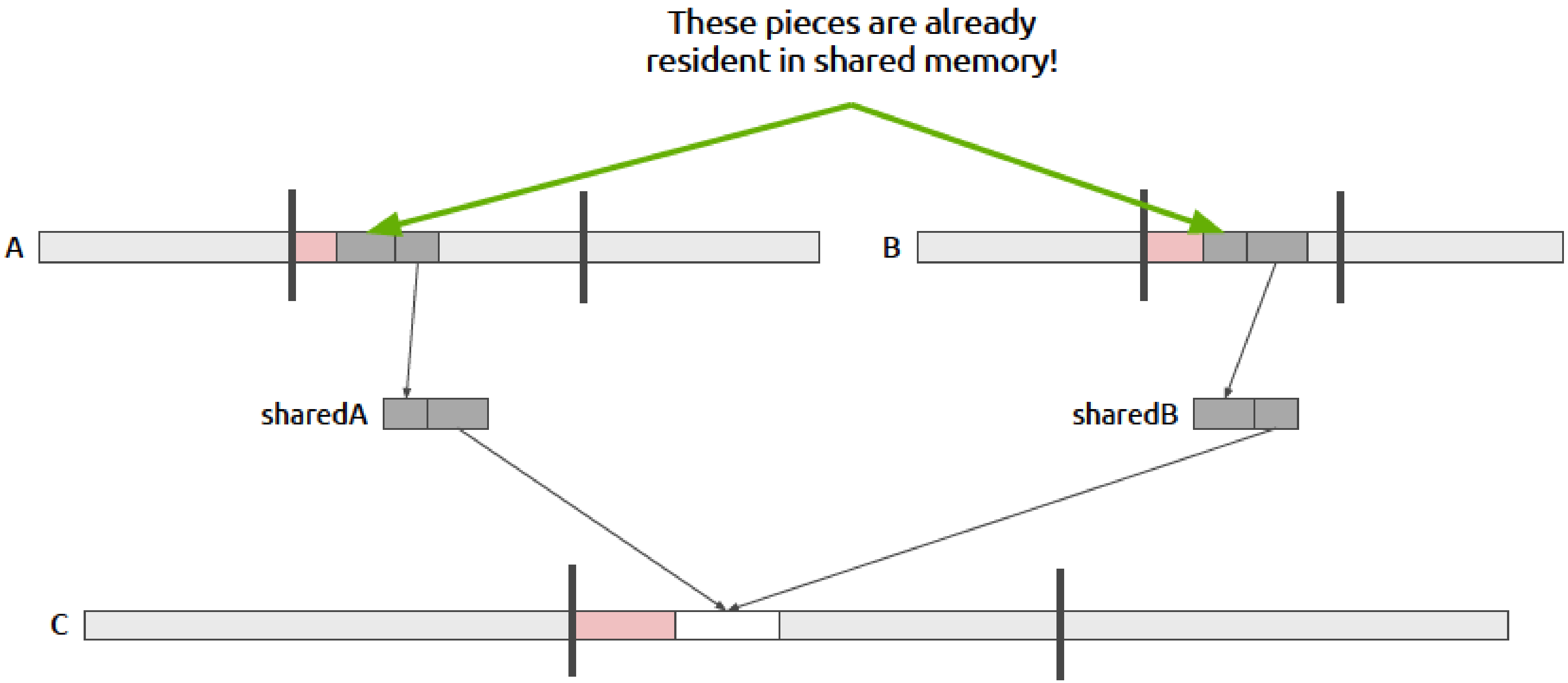
Circular Buffer Tiled Merge Kernel (visually)



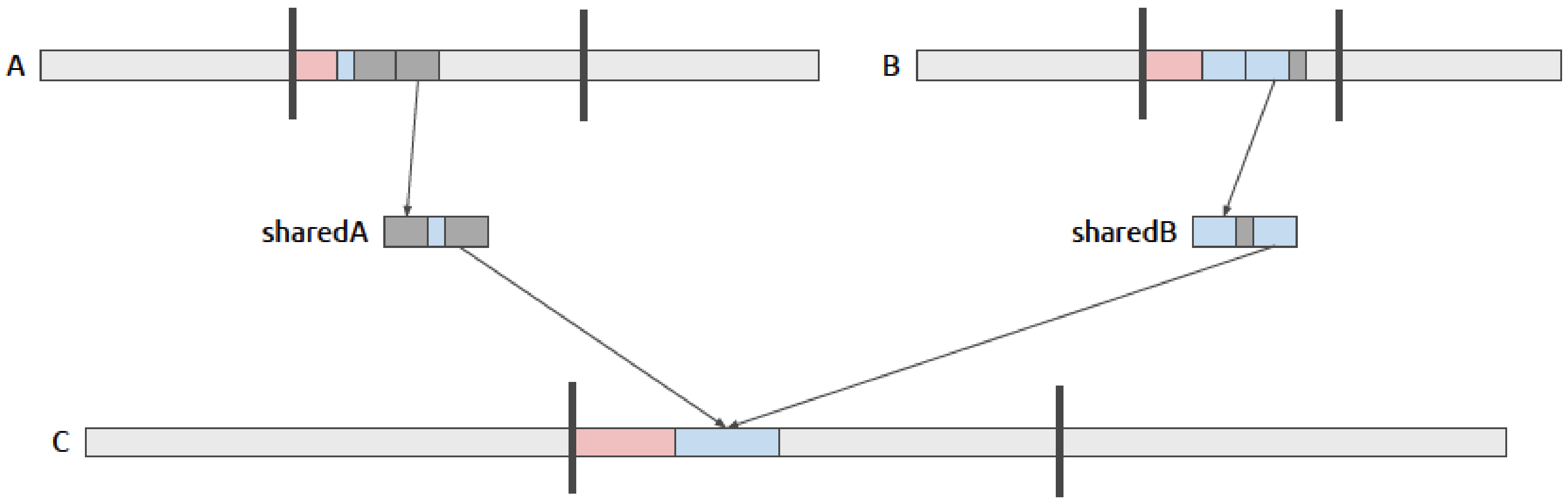
Circular Buffer Tiled Merge Kernel (visually)



Circular Buffer Tiled Merge Kernel (visually)



Circular Buffer Tiled Merge Kernel (visually)



Circular Buffer Tiled Merge Kernel Analysis

- **Pros:**
  - 2X reduction in global memory reads!
- **Cons:**
  - Significant increase in code complexity
    - Essentially everything needs to be re-indexed, including co_rank and merge_sequential
  - Increased register usage

Conclusion / Takeaways

- **Shared memory strategies get complicated when data usage depends on the data themselves**
- **Circular buffers are useful tools for getting the most out of values loaded into shared memory**
- **The GPU architecture is better suited to fixed computation graphs**
- **Sometimes small tweaks to an algorithm can make a variable computation graph become fixed**

ANY QUESTIONS?