CSE 4373/5373 - General Purpose GPU Programming

GPU Pattern: Merge

Alex Dillhoff

University of Texas at Arlington

Merge

- The **merge** operation takes two sorted subarrays and combines them into a single sorted array.
- You may be familiar with this approach from studying Divide and Conquer algorithms.
- Parallelizing the merge operation is a non-trivial task and will require the use of a few new techniques.

Merge

In the context of GPU programming, we will learn about the following techniques:

- Dynamic input data identification
- Data locality
- Buffer management schemes

Merge

- Using merge from ”Perfectly load-balanced, optimal, stable, parallel merge” (Siebert et al., 2013).
- Compute which values are needed in each merge step,
- Use a parallel kernel to compute the merge.
- These steps can be computed by each thread independently.

Co-rank Function

The key to this implementation is the **co-ranking function**.

- Compute the range of indices needed from two input values to produce a given output value.
- Avoids the need to merge the two input arrays explicitly

Co-rank Function

When merging two sorted arrays, we can observe that the output index $0 \leq k < m + n$ comes from either

1. $0 \leq i < m$ from input A or
2. $0 \leq j < n$ from input B.

Co-rank Function

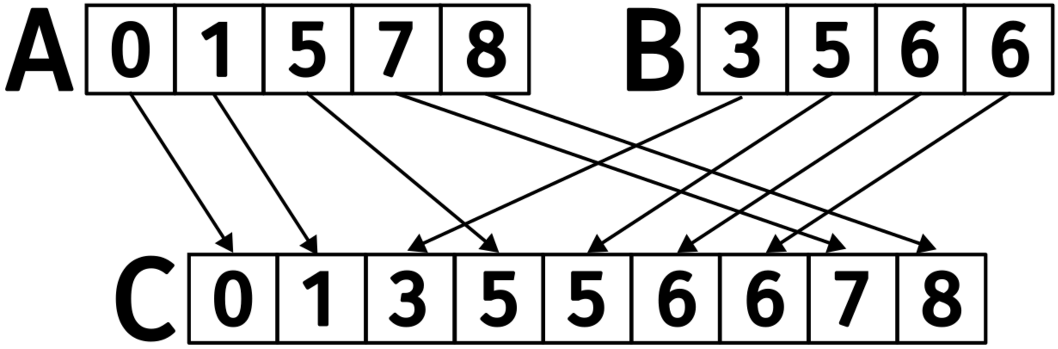

Figure 1: Merging two sorted arrays.

Co-rank Function

The element at $k = 3$ comes from $A[2]$, so $i = 2$.

It must be that $k = 3$ is the result of merging the first $i = 2$ elements of A with the first $j = k - i$ elements of B.

Co-rank Function

This works both ways: for $k = 6$, the value is taken from $B[3]$, so $j = 3$, and the result is the merge of the first $i = k - j$ elements of A with the first $j = 3$ elements of B.

Co-rank Function

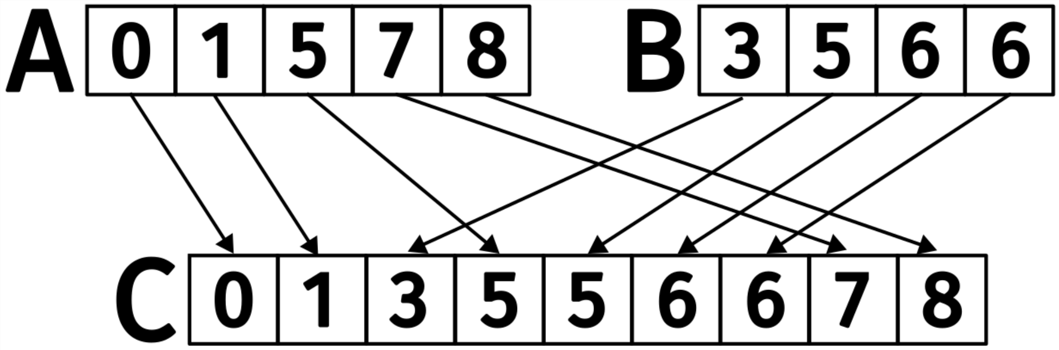

Figure 2: Merging two sorted arrays.

Co-rank Function

Lemma 1

For any $k, 0 \leq k < m + n$, there exists a unique $i, 0 \leq i \leq m$, and a unique $j, 0 \leq j \leq n$, with $i + j = k$ such that

1. $i = 0$ or $A[i - 1] \leq B[j]$ and
2. $j = 0$ or $B[j - 1] < A[i]$.

Co-rank Function

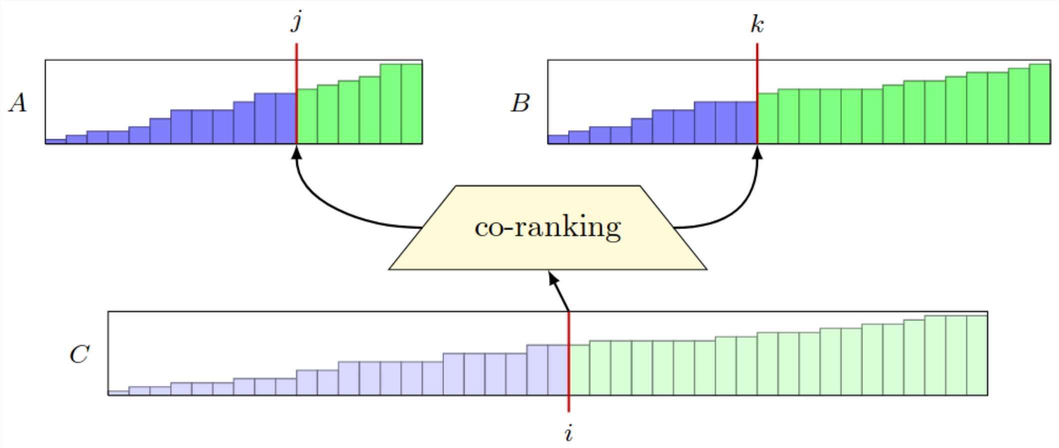

Figure 3: Visualization of the co-rank function.

Implementation

Implementation of Co-rank

Given the rank k of an element in an output array C and two input arrays A and B...

the co-rank function f returns the co-rank value for the corresponding element in A and B.

Implementation of Co-rank

How would the co-rank function be used in the example above?

Implementation of Co-rank

How would the co-rank function be used in the example above?

Given two threads, let thread 1 compute the co-rank for $k = 4$.

This would return $i = 3$ and $j = 1$.

Implementation of Co-rank

We can verify this result using Lemma 1.

$$A[2] = 5 \leq B[1] = 5 \text{ and } B[0] = 3 < A[3] = 7.$$

Implementation of Co-rank

Initialization

```
int co_rank(int k, int *A, int m, int *B, int n) {
    int i = min(k, m);
    int j = k - i;
    int i_low = max(0, k-n);
    int j_low = max(0, k-m);
    int delta;
    bool active = true;
```

Implementation of Co-rank

```
while (active) {
    if (i > 0 && j < n && A[i-1] > B[j]) {
        delta = (i - i_low + 1) / 2;
        j_low = j;
        i -= delta;
        j += delta;
    } else if (j > 0 && i < m && B[j-1] >= A[i]) {
        delta = (j - j_low + 1) / 2;
        i_low = i;
        j -= delta;
        i += delta;
    } else {
        active = false;
    }
}
```

Co-rank Example

Consider running a merge kernel across 3 threads where each thread takes 3 sequential output values.

Use the co-rank function to compute the co-rank values for $k = 3$ and $k = 6$, simulating the tasks for the second and third threads.

Co-rank Example

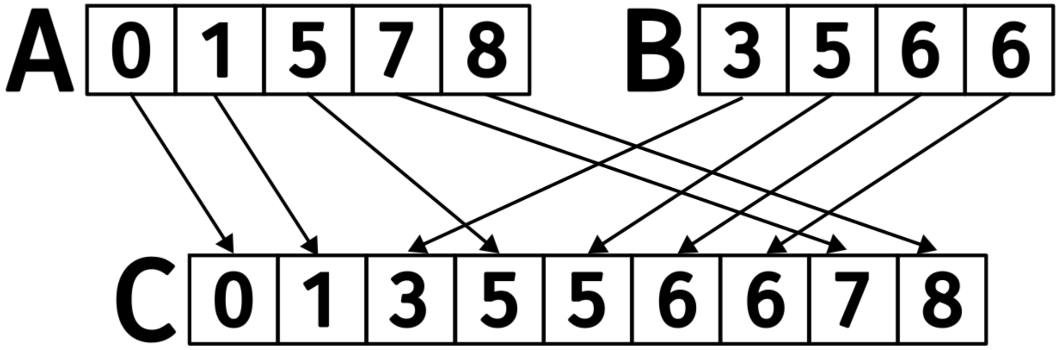

Figure 4: Merging two sorted arrays.

Co-rank Example

The values for $k = 3$ should be $i = 2$ and $j = 1$, for reference.

All values below these indices would be used by the first thread.

Parallel Kernel

We can now implement a basic parallel merge kernel.

- Each thread is responsible for determining how many elements it will be responsible for merging.
- The range of input values is determined via two calls to `co_rank`, one for the starting and ending point.

Parallel Kernel

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

Parallel Kernel

Bottlenecks?

Parallel Kernel

Bottlenecks?

1. Memory accesses to input arrays are not coalesced.
2. Binary search in the co-rank function is not coalesced.

Tiled Merge

Tiled Merge

- Improve memory accesses by having the threads transfer data from global memory to shared memory in a coalesced manner.
- The higher latency operation will be coalesced.
- The data in shared memory may be accessed out of order, but the latency is much lower.

Tiled Merge

- The subarrays from A and B that are used by adjacent threads are also adjacent in memory.
- By considering block-level subarrays, we can ensure that the data is coalesced.
- **This is the idea behind the tiled merge algorithm.**

Tiled Merge

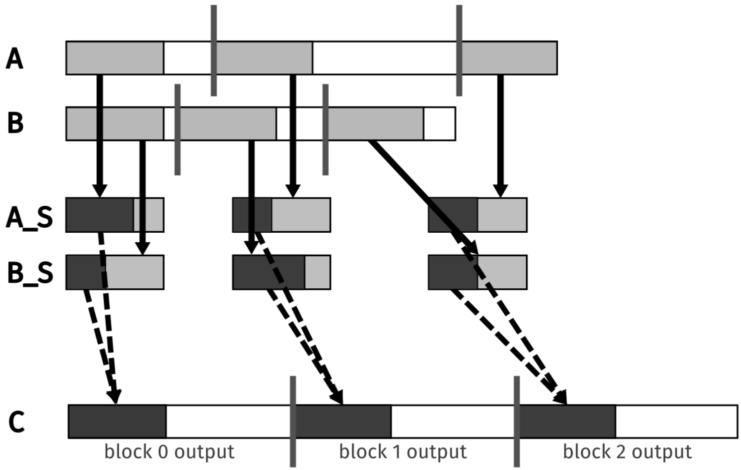

Figure 5: Visualization of the tiled merge algorithm.

Tiled Merge

- The shared memory blocks cannot store the entire range of data needed.

Tiled Merge

- The shared memory blocks cannot store the entire range of data needed.
- In each iteration, the threads in a block will load a new set of data from global memory to shared memory.

Tiled Merge

- The shared memory blocks cannot store the entire range of data needed.
- In each iteration, the threads in a block will load a new set of data from global memory to shared memory.
- If they collectively load $2n$ elements, only n elements will be used in the merge operation.

Tiled Merge

- The shared memory blocks cannot store the entire range of data needed.
- In each iteration, the threads in a block will load a new set of data from global memory to shared memory.
- If they collectively load $2n$ elements, only n elements will be used in the merge operation.
- This is because in the worst case, all elements going to the output array will come from one of the two input arrays.

Tiled Merge

Each block will use a portion of both A_s and B_s to compute the merge.

This is shown with dotted lines going from the shared memory to the output array.

Tiled Merge Kernel: Part 1

Tiled Merge: Part 1

Establish shared memroy and output boundaries.

```
__global__ void merge_tiled_kernel(int *A, int m, int n,
                                   int *C, int tile_size) {
    extern __shared__ int shareAB[];
    int *A_s = &shareAB[0];
    int *B_s = &shareAB[tile_size];
    int C_curr = blockIdx.x * ceil((m+n)/gridDim.x);
    int C_next = min((blockIdx.x+1) * ceil((m+n)/gridDim.x), m+n);
```

Tiled Merge: Part 1

Only the first thread needs to compute the co-rank values.

```
if (threadIdx.x == 0) {
    // Block-level co-rank values will be
    // available to all threads in the block
    A_s[0] = co_rank(C_curr, A, m, B, n);
    A_s[1] = co_rank(C_next, A, m, B, n);
}
__syncthreads();
```

Tiled Merge: Part 1

Compute *j* values based on *i* values.

```
int A_curr = A_s[0];
int A_next = A_s[1];
int B_curr = C_curr - A_curr;
int B_next = C_next - A_next;
__syncthreads();
```

Tiled Merge Kernel: Part 2

Tiled Merge: Part 2

The second part of the kernel is responsible for loading the input data into shared memory.

This is done in a coalesced manner, as the threads in a block will load a contiguous section of the input arrays.

Tiled Merge: Part 2

Compute lengths of each section and number of iterations for coarsening.

```
int counter = 0;
int C_length = C_next - C_curr;
int A_length = A_next - A_curr;
int B_length = B_next - B_curr;
int total_iteration = ceil(C_length / tile_size);
int C_completed = 0;
int A_consumed = 0;
int B_consumed = 0;
```

Tiled Merge: Part 2

Copy data from global memory to shared memory.

```
while (counter < total_iteration) {
    for (int i = 0; i < tile_size; i += blockDim.x) {
        if (i + threadIdx.x < A_length - A_consumed) {
            A_s[i + threadIdx.x] = A[A_curr + A_consumed + i + threadIdx.x];
        }
        if (i + threadIdx.x < B_length - B_consumed) {
            B_s[i + threadIdx.x] = B[B_curr + B_consumed + i + threadIdx.x];
        }
    }
    __syncthreads();
```

Tiled Merge Kernel: Part 3

Tiled Merge: Part 3

With the input in shared memory, each thread will divide up this input and merge their respective sections in parallel.

This is done by calculating the `c_curr` and `c_next` first, which is the output section of the thread.

Using those boundaries, two calls to `co_rank` will determine the input sections the thread.

Tiled Merge: Part 3

Compute output section for the current thread.

```
int c_curr = threadIdx.x * (tile_size / blockDim.x);
int c_next = (threadIdx.x + 1) * (tile_size / blockDim.x);
c_curr = (c_curr <= C_length - C_completed) ?
          c_curr : C_length - C_completed;
c_next = (c_next <= C_length - C_completed) ?
          c_next : C_length - C_completed;
```

Tiled Merge: Part 3

Compute co-rank values for the current section.

```
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));
int b_curr = c_curr - a_curr;
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));
int b_next = c_next - a_next;
```

Tiled Merge: Part 3

Merge the data and compute how much data was used.

```
        merge_sequential(&A_s[a_curr], a_next - a_curr,
                         &B_s[b_curr], b_next - b_curr,
                         &C[C_urr + C_completed + c_curr]);
        counter++;
        C_completed += tile_size;
        A_consumed += co_rank(tile_size A_s, tile_size, B_s, tile_size);
        B_consumed = C_completed - A_consumed;
        __syncthreads();
    }
}
```

Kernel Walkthrough

We have two input arrays $A = [1, 3, 5, 7, 9]$ and $B = [2, 4, 6, 8, 10]$.

- The output array C will have 10 elements.
- Use 2 blocks and 4 threads per block.
- The tile size is 4.
- With 10 elements and 2 blocks, each block is responsible for 5 elements.

Kernel Walkthrough

The main `while` loop will need to iterate twice to cover the entire output array.

- The first iteration will load the first 4 elements of A and B into shared memory.

Kernel Walkthrough

The main `while` loop will need to iterate twice to cover the entire output array.

- The first iteration will load the first 4 elements of A and B into shared memory.
- Next, each thread divides the input tiles by running the co-rank function on the data that is in shared memory.
- The computed indices are the boundaries between each thread.

Kernel Walkthrough

- In each iteration, a block is responsible for 4 elements.
- Given that we have 4 threads per block, each thread will be responsible for 1 output element per iteration.

Kernel Walkthrough

- In each iteration, a block is responsible for 4 elements.
- Given that we have 4 threads per block, each thread will be responsible for 1 output element per iteration.
- **Thread 0:** `c_curr = 0` and `c_next = 2`.
- This results in `a_curr = 0`, `b_curr = 0`, `a_next = 1`, and `b_next = 1`.
- **The merge operation will then be performed on the first element of *A* and *B*.**

Kernel Analysis

Analysis

The tiled kernel is an improvement due to...

1. coalescing of global memory accesses
2. Shared memory use for reduced latency

Analysis

What is the bottleneck of this kernel?

Analysis

What is the bottleneck of this kernel?

Only half the data loaded into shared memory is used, leading to wasted memory bandwidth.

Analysis

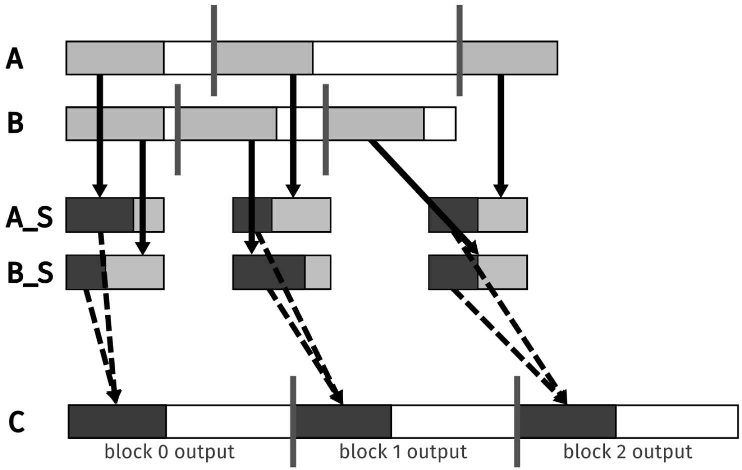

Figure 6: Visualization of the tiled merge algorithm.

Circular Buffers

Circular Buffers

The tiled version leaves half of the data unused every iteration.

A **circular buffer** can be implemented to reuse the data that was loaded into shared memory.

Circular Buffers

Instead of writing over the shared memory values each iteration, the data stays in memory.

A portion of new data is loaded into shared memory, reducing the number of reads from global memory.

Circular Buffers

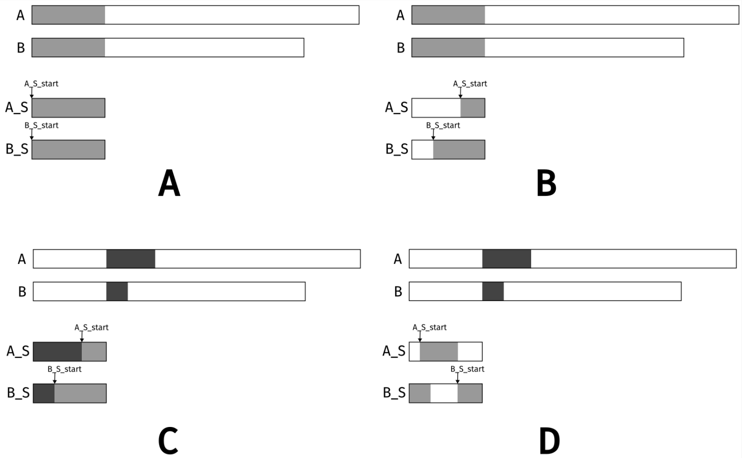

Figure 7: Visualization of the circular buffer.

Circular Buffers

- Part A shows the initial layout.
- Part B shows a blank portion that depicts what was used, the light gray represents what remains.
- Part C shows the new data loaded into shared memory.
- Part D shows the next iteration.

Implementation

- `A_consumed` is used to keep track of how many new elements need to be read.
- The `co_rank` and `merge_sequential` functions need to be updated to work with circular buffers.
- It is easier to treat the shared memory as an extended array...
- this avoids situations where the `next` index is less than the `current` index.

Updating the Co-rank Function

```
int co_rank_circular(int k, int *A, int m,
                     int *B, int n,
                     int A_S_start, int B_S_start,
                     int tile_length) {
    int i = min(k, m);
    int j = k - i;
    int i_low = max(0, k-n);
    int j_low = max(0, k-m);
    int delta;
    bool active = true;
```

Updating the Co-rank Function

```
while (active) {
    int i_cir = (A_S_start + i) % tile_length;
    int j_cir = (B_S_start + j) % tile_length;
    int i_m_1_cir = (A_S_start + i - 1) % tile_length;
    int j_m_1_cir = (B_S_start + j - 1) % tile_length;
```

Updating the Co-rank Function

```
        if (i > 0 && j < n && A[i_m_1_cir] > B[j_cir]) {
            delta = ((i - i_low + 1) >> 1);
            j_low = j;
            i -= delta;
            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;
            j -= delta;
            i += delta;
        } else {
            active = false;
        }
    }
    return i;
```

Updating the Co-rank Function

In this updated version of the co-rank function...

the user only needs to provide the start indices for the shared memory arrays along with the tile size.

Updating the Merge Function

```
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;
    int j = 0;
    int k = 0;
```

Updating the Merge Function

```
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++;
    }
    k++;
}
```

Updating the Merge Function

```
if (i == m) {
    while (j < n) {
        int j_cir = (B_S_start + j) % tile_size;
        C[k] = B[j_cir];
        j++;
        k++;
    }
} else {
    while (i < m) {
        int i_cir = (A_S_start + i) % tile_size;
        C[k] = A[i_cir];
        i++;
        k++;
    }
}
```

Updating the Merge Function

This too acts as a drop-in replacement, only requiring the start indices of the shared memory arrays and the tile size.

Circular Buffer Kernel

```
int c_curr = threadIdx.x * (tile_size / blockDim.x);
int c_next = (threadIdx.x + 1) * (tile_size / blockDim.x);
c_curr = (c_curr <= C_length - C_completed) ?
            c_curr : C_length - C_completed;
c_next = (c_next <= C_length - C_completed) ?
            c_next : C_length - C_completed;
```

Circular Buffer Kernel

```
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_curr, B_curr, tile_size);
int b_curr = c_curr - a_curr;
int a_next = co_rank_circular(c_curr,
                              A_s, min(tile_size, A_length - A_consumed),
                              B_s, min(tile_size, B_length - B_consumed),
                              A_curr, B_curr, tile_size);
int b_next = c_next - a_next;
```

Circular Buffer Kernel

```
merge_sequential_circular(A_s, a_next - a_curr,
                          B_s, b_next - b_curr,
                          &C[C_urr + C_completed + c_curr],
                          A_S_start + A_curr, B_S_start + B_curr, tile_size);
```

Circular Buffer Kernel

```
// Compute the indices that were used
counter++;
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);
B_S_consumed = min(tile_size, C_length - C_completed) - A_S_consumed;
A_consumed += A_S_consumed;
C_completed += min(tile_size, C_length - C_completed);
B_consumed = C_completed - A_consumed;
```

Circular Buffer Kernel

```
// Update the start indices for the next iteration
A_S_start = (A_S_start + A_S_consumed) % tile_size;
B_S_start = (B_S_start + B_S_consumed) % tile_size;
__syncthreads();
```

Coarsening

Thread Coarsening

- The kernels presented already utilize thread coarsening.
- Each thread is responsible for a range of output values.
- Each thread was only responsible for a single output value.