ECE 508
Manycore Parallel Algorithms

Lecture 11: Parallel Ordered Merge
Background

- We started with easy parallelism,
  - used atomics to coordinate and
  - optimized the access patterns.
- Next, we looked at reorganizing data.
- With graphs, we looked at
  - finding the parallelism from step to step and
  - Using hierarchical kernels and dynamic parallelism to leverage the parallelism.
- But some algorithms may seem inherently sequential.
Objective

• to learn techniques for high-performance parallel merge sort
  – input identification
  – tiling for coalescing
  – circular buffering for data reuse

• to learn to hide complexities from library users
Sorting is an Important Problem

- **Sorting is a fundamental operation** in computing.
- Covered early, with many algorithms.
- Sort has long been a *challenge for parallel systems*.
- In my first parallel programming class,
  - we had a sorting competition.
  - Each person got a random algorithm and a random machine.
  - I got bitonic sort ($O(N^2)$) on a Cray,
  - so I had to argue that my constant was smallest!
Architecture Matters to the Algorithm

A few weeks ago, I mentioned NOWSort.

• On a cluster of $N$ workstations, one…
  – oversamples to pick $N$ splitters,
  – broadcasts the splitters,
  – bins data on each machine (based on the splitters),
  – sends the bins (all-to-all communication), and
  – performs the final sort locally.

• But those are CPUs—we need a good GPU sort for the last step.
We Focus on Parallel Merge Sort

• Let’s look at **merge sort**: sort chunks in parallel, then merge the chunks.
• Merge sort is **also a building block** for other sorting algorithms.
• We need to be careful about **complexity**; avoid adding too much extra work.
Merge by Repeatedly Choosing the Smaller Element

Choose smaller element from unused part of A and B. If equal, choose from A to support stable sorts (in which elements of equal value remain in the same order).
Implementation of Sequential Merge

```c
void merge_sequential (int* A, int m, int* B, int n, int *C)
{
    int i = 0;  //index into A
    int j = 0;  //index into B
    int k = 0;  //index into C
}
```

index variables into arrays

length of A

length of B

© Wen-mei W. Hwu, David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021
Copy Until One List is Empty

```c
void merge_sequential (int* A, int m, int* B, int n, int *C) {
    int i = 0;  //index into A
    int j = 0;  //index into B
    int k = 0;  //index into C
    while (i < m && j < n) {
        if (A[i] <= B[j]) {
            C[k++] = A[i++];
        } else {
            C[k++] = B[j++];
        }
    }
    ...  //Copy remainder of one list here.
}
```

both arrays still have elements
Copy an element from A to C.
Or copy an element from B to C.
Copy remainder of one list here.
Then Copy Array Remainder to Result

```c
if (i == m) {
    while (j < n) {
        C[k++] = B[j++];
    }
} else {
    while (j < n) {
        C[k++] = A[i++];
    }
}
```

Copy remainder of B to C.

Or copy remainder of A to C.
Can We Find Parallelism?

So … what can we parallelize?
• Each position depends on all previous choices.
• But not really on the details of those choices.
• We’ve seen this problem before, actually.
Pick a Splitter and Use it to Split!

Remember dynamic parallelism with neighbor lists?

Pick the middle element of A. Say it has value X.

Binary (N-ary) search for the first element Y of B such that Y >= X.
Sections Can be Merged in Parallel

Can **merge yellow and blue** regions **in parallel**!

Array A may contain more X values—that’s ok.

All values in this section are < X.
Parallelize Splitting

Divide and conquer?
No.
Parallelize!

Only total size (in both arrays) matters for load balance; can do hierarchically and use dynamic parallelism.
A “Scatter” Approach?

- In 2019, Wen-mei claimed that
  - no one had implemented a scatter approach:
  - each thread takes a section of input A and B values and delivers them to the final location.
- The approach just outlined (split, scan, merge sections) occurred to me immediately (on the objective slide).
Let’s Flip Around the Splitter Idea

• Maybe no one has gotten it to go fast?

• Try it if you’d like—maybe it’s a paper.

• Hard to believe no one has tried that approach, though.

• Especially given that we’re now going to use the same idea in reverse…
Name the Number of Elements per Array

- **Pick** some number \( i \) of elements from start of \( A \).
- These elements **join with** some number \( j \) of elements from start of \( B \) (find \( j \) as described, if desired).
- Together, they **become first** \( k = i + j \) elements of \( C \).
Co-Rank of an Output Prefix String

In this context, the tuple \((i,j)\) is the **co-rank of A and B for the prefix of k elements of C**.

Given A, B, and a value k, can we compute \((i,j)\)?

- **Of course!**
- First, we know that \(j = k - i\), so **computing i suffices**.
- Also, the **value of i is unique** (given A, B, and k).
- Let’s look at the arrays again…
First Constraint Generalizes Splitter Search

First, we know that

- the **element at the end of the yellow region in** A—X
- **must be** sorted **before** the element just after the **yellow region in** B—Y.
- So **X ≤ Y**. That was our splitter search condition.
- Let’s generalize to (j = n) OR (A[i – 1] ≤ B[j]).
Second Constraint Arises from Swapping Arrays

Now do the same with the arrays reversed:

- the element at the end of the yellow region in B
- must be sorted before the element
  just after the yellow region in A.

- That gives \((i = m)\ OR (A[i] > B[j – 1])\).
- (We know \(A[i] \geq X > B[j – 1]\) in the splitter case.)
Find Initial Lower Bound for Binary Search

But now we can find i using binary (N-ary) search!

What is the minimum value of i? 0?

What if k > n (n is the length of B)?

Even if all elements of B are first in C, C must include some of A.

So the smallest possible i is max (0, k – n).
Find Initial Upper Bound for Binary Search

And the largest $i$? $m$?

What if $k < m$?
$i$ cannot be greater than $k$, either.

So the largest possible $i$ is min $(k, m)$.

Now we can simply search…
Computing the Co-Rank

```c
int co_rank (int k, int* A, int m, int* B, int n) {
    int low = (k > n ? k - n : 0);
    int high = (k < m ? k : m);
    while (low < high) {
        ...
    }
    return low;
}
```

- **Compute initial bounds.**
- **Search until found or only one choice remains (next slide).**
- **Remaining choice must be correct.**
Compute $i$ and $j$.

```c
int i = low + (high - low) / 2;
int j = k - i;
if (j < n && A[i - 1] > B[j]) {
    high = i - 1;
} else if (i < m && A[i] <= B[j - 1]) {
    low = i + 1;
} else {
    return i;
}
```

Need more from B.

Need more from A.

Both conditions met? We’re done!
int co_rank (int k, int* A, int m, int* B, int n) {
    int low = (k > n ? k - n : 0);
    int high = (k < m ? k : m);
    while (low < high) {
        int i = low + (high - low) / 2;
        int j = k - i;
        if (j < n && A[i - 1] > B[j]) {
            high = i - 1;
        } else if (i < m && A[i] <= B[j - 1]) {
            low = i + 1;
        } else {
            return i;
        }
    }
    return low;
}
1 int co_rank(int k, int* A, int m, int* B, int n) {
2     int i = k < m ? k : m;  // i = min(k,m)
3     int j = k - i;
4     int i_low = 0 > (k - n) ? 0 : k - n;  // i_low = max(0, k-n)
5     int j_low = 0 > (k - m) ? 0 : k - m; // i_low = max(0,k-m)
6     int delta;
7     bool active = true;
8     while (active) {
9         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        }
14     }
15 }
Wen-mei’s Version (part 2 of 2)

```c
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 }
```
Gather Approach Assigns Segment of C per Thread

So … now what?

Gather!

Assign a segment of C to each thread.

Three threads, for example…
Co-Rank Provides Bounds in A and B

- Each thread uses co-rank twice
  - to obtain starting points \((i_{\text{start}}, j_{\text{start}})\) and
  - to obtain ending points \((i_{\text{end}}, j_{\text{end}})\).
- Then performs a sequential merge.
Co-Rank Results Specify A and B Segments

- Thread 1, for example…
  - Co-rank 3 gives \((i_{\text{start}}, j_{\text{start}}) = (2,1)\).
  - Co-rank 6 gives \((i_{\text{end}}, j_{\text{end}}) = (5,1)\).
- Thread 1’s subset of B is empty. That’s ok.

© Wen-mei W. Hwu, David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021 30
Some Load Imbalance

• Work necessary for co-rank calls is imbalanced.

• Higher-indexed threads have a bigger search space.

• But use of binary search in co-rank reduces imbalance.
Structure of Basic Merge Kernel

Basic merge kernel is then pretty simple:
• Assign ceil (size of $C$ / # of threads) elements per thread
• Find thread’s bounds in $C$.
• Use `co_rank` to find input bounds.
• Use `sequential_merge` to produce thread’s output.
Find Thread Index and Elements per Thread

```c
__global__ void merge_basic_kernel
  (int* A, int m, int* B, int n, int* C)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int elt = ceil ((m+n)*1.0f/(blockDim.x*gridDim.x));

    // Code here...
}
```

- **linearized thread index**
- **elements per thread**
Find Start and End Indices in Output Array C

```c
__global__ void merge_basic_kernel
    (int* A, int m, int* B, int n, int* C)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    int elt = ceil ((m+n)*1.0f/(blockDim.x*gridDim.x));

    int k_curr = tid * elt;
    if (m + n < k_curr) { k_curr = m + n; }

    int k_next = k_curr + elt;
    if (m + n < k_next) { k_next = m + n; }
```

start index in C

end index in C
Co-Rank, then Merge

```c
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]);
```

co_rank gives indices in A

j = k - i

indices define sequential merge of segments
Basic Merge Kernel Performs Poorly

- Global **memory accesses not coalesced**:  
  - binary search (**co_rank**) on A/B, and  
  - sequential merge reads and writes.

- Also **lots of** localized **control divergence**:  
  - **co_rank** search direction and depth, and  
  - sequential merge A/B select, final list copy.
Solution: Aggregate, Collaborate, Tile

Consider A and B segments for threads in a block.

- Only need aggregate bounds to allow collaborative load/store to/from shared memory.
- Choose one thread per block to find bounds, so reduce pressure on global memory.
- Can tile segment loads to fit shared memory.
- Can determine per-thread bounds using co_rank on shared memory data.
Representative Thread(s) Find(s) Bounds

thread block 0’s outputs

thread 0 in block 0 calls \texttt{co\_rank}

thread 0 (or 1) in block 0 calls \texttt{co\_rank}

Share A and B bounds with all threads.
Operate on Tiles in Shared Memory

Read tiles collaboratively into shared memory.

All threads co_rank and merge into shared tile.
How Much Can We Merge?

A question for you:
What is the relationship between
the sizes of tiles for A, B, and C?

Hint: how much data can we safely write into C?

Say we use all data from tile A. What comes next:
• something from tile B?
• Or something not yet in shared memory (from A)?

So size of tile C ≤ min (size of tile A, size of tile B).

We’ll set all three to be equal size.
Write Back to C Collaboratively

We use half of the data from tiles A and B.

Tile C is then written back to C collaboratively.

All threads co_rank and merge into shared tile.
Discard Remaining Data and Load Next Tile

Then what?  
Start over! Flush and load next tile.  
(2× bandwidth loss—we’ll come back later)

Next tile loads start after consumed data.
Oops! B has too little data left to fill a tile!
That’s ok: we know B is out of data, not just tile B—just need to use that difference in the code!
Performance Hints for Lab 8

Some performance guidelines…

• **Thread block output** sections should have at least *a few thousand* elements.

• **Tiles** should have at least *a few hundred* elements.

• **Each thread** should be responsible for *tens of outputs per tile*.

Now, let’s look at some code!
Tile Size Passed as Parameter

__global__ void merge_tiled_kernel
    (int* A, int m, int* B, int n,
     int* C, __int tile_size)
{

    extern __shared__ int shareAB[];

    new parameter: tile size

    syntax for dynamic shared memory size (set by kernel launch)
Tiles Split Shared Memory

```c
__global__ void merge_tiled_kernel
    (int* A, int m, int* B, int n,
     int* C, int tile_size)
{

    extern __shared__ int shareAB[];

    int* tileA = &shareAB[0];

    int* tileB = &shareAB[tile_size];

    // Your version needs another block for tileC.
}
```

- **tileA** occupies the first half of shared memory.
- **tileB** occupies the second half.

Your version needs another block for tileC.
All Threads Find Output Bounds

int elt = ceil ((m + n) * 1.0f / gridDim.x);

int blk_C_curr = blockIdx.x * elt;

block's ending output bound

block's starting output bound (assumes 1+ elts/block)

int blk_C_next = blk_C_curr + elt;

if (m + n < blk_C_next) { blk_C_next = m + n; }

output elements per thread block
Representative Thread(s) Find Input Bounds

```c
if (threadIdx.x == 0) {
    tileA[0] = co_rank (blk_C_curr, A, m, B, n);
    tileA[1] = co_rank (blk_C_next, A, m, B, n);
}
__syncthreads();
```

Be sure that other threads see the values.

Pass to other threads.

Compute input bounds (representative threads only).
All Threads Compute Bounds for B

```c
if (threadIdx.x == 0) {
    tileA[0] = co_rank (blk_C_curr, A, m, B, n);
    tileA[1] = co_rank (blk_C_next, A, m, B, n);
}
__syncthreads();
int blk_A_curr = tileA[0];
int blk_A_next = tileA[1];
int blk_B_curr = blk_C_curr - blk_A_curr;
int blk_B_next = blk_C_next - blk_A_next;
__syncthreads();
```

All threads read and compute input bounds.

Finish reads before loading first tile.
Representative Thread(s) Find(s) Bounds

Share A and B bounds with all threads.

thread block 0’s outputs
thread 0 in block 0 calls co_rank
thread 0 (or 1) in block 0 calls co_rank

Now we’re done with this part and ready to load a tile.

Share A and B bounds with all threads.
Compute Lengths and Number of Tiles

Compute block’s segment lengths.

```c
int C_length = blk_C_next - blk_C_curr;
int A_length = blk_A_next - blk_A_curr;
int B_length = blk_B_next - blk_B_curr;
```

Number of tiles needed

```c
int num_tiles =
    ceil (C_length * 1.0f / tile_size);
```

Data consumed / produced already

```c
int C_produced = 0;
int A_consumed = 0;
int B_consumed = 0;
```
Tile Loop Contains Three Steps

```c
for (int counter = 0; num_tiles > counter; counter++) {
    // load tile
    // process tile
    // advance variables for next tile
}
```
Use a Loop to Load Tiles to Shared Memory

```
for (int i = 0; i < tile_size; i += blockDim.x) {
    if (i + threadIdx.x < A_length - A_consumed) {
        tileA[i + threadIdx.x] =
        A[blk_A_curr + A_consumed + i + threadIdx.x];
    }
}
```

Read remaining data (up to a tile) for block into `tileA`. 
Load Tile from Both A and B

```c
for (int i = 0; i < tile_size; i += blockDim.x) {
    if (i + threadIdx.x < A_length - A_consumed) {
        tileA[i + threadIdx.x] = 
        A[blk_A_curr + A_consumed + i + threadIdx.x];
    }
    if (i + threadIdx.x < B_length - B_consumed) {
        tileB[i + threadIdx.x] = 
        B[blk_B_curr + B_consumed + i + threadIdx.x];
    }
}
__syncthreads();

Wait for tile loads to complete.
```

Do the same for `tileB`.

Write Back to C Collaboratively

We use half of the data from tiles A and B.

Tile C is then written back to C collaboratively.

We have a tile—time to do some work! We’ll just merge directly into C in this code.
Compute per-thread output bounds.

```c
int per_thread = tile_size / blockDim.x;
int thr_C_curr = threadIdx.x * per_thread;
int thr_C_next = thr_C_curr + per_thread;
```

This ratio should be integral.
Do Not Produce More Output than Needed

```c
int per_thread = tile_size / blockDim.x;
int thr_C_curr = threadIdx.x * per_thread;
int thr_C_next = thr_C_curr + per_thread;

int C_remaining = C_length - C_produced;
if (C_remaining < thr_C_curr) {
    thr_C_curr = C_remaining;
}
if (C_remaining < thr_C_next) {
    thr_C_next = C_remaining;
}
```

Limit to remaining output needed.
Compute Data Actually in Tiles A and B

```c
int A_in_tile = A_length - A_consumed;
if (tile_size < A_in_tile) { A_in_tile = tile_size; }
int B_in_tile = B_length - B_consumed;
if (tile_size < B_in_tile) { B_in_tile = tile_size; }
```

Compute amount in tiles.
Find Per-Thread Input Bounds for A

```c
int A_in_tile = A_length - A_consumed;
if (tile_size < A_in_tile) { A_in_tile = tile_size; }
int B_in_tile = B_length - B_consumed;
if (tile_size < B_in_tile) { B_in_tile = tile_size; }
```

```c
int thr_A_curr = co_rank
  (thr_C_curr, tileA, A_in_tile, tileB, B_in_tile);
int thr_A_next = co_rank
  (thr_C_next, tileA, A_in_tile, tileB, B_in_tile);
```

Find tile A input bounds for thread.
Compute Per-Thread Input Bounds for B

```c
int A_in_tile = A_length - A_consumed;
if (tile_size < A_in_tile) { A_in_tile = tile_size; }
int B_in_tile = B_length - B_consumed;
if (tile_size < B_in_tile) { B_in_tile = tile_size; }

int thr_A_curr = co_rank
    (thr_C_curr, tileA, A_in_tile, tileB, B_in_tile);
int thr_A_next = co_rank
    (thr_C_next, tileA, A_in_tile, tileB, B_in_tile);

int thr_B_curr = thr_C_curr - thr_A_curr;
int thr_B_next = thr_C_next - thr_A_next;
```

Compute tile B input bounds for thread.
Merge Each Thread’s Shared Memory Segments

```c
merge_sequential
    (tileA + thr_A_curr, thr_A_next - thr_A_curr,
     tileB + thr_B_curr, thr_B_next - thr_B_curr,
     C + blk_C_curr + C_produced + thr_C_curr);
```

Merge each thread’s segment in tiles A and B into output C.

Remember that your version should merge into a shared memory tile and then write back collaboratively to C.
Variable Updates Left for You in Lab 8

for (int counter = 0; num_tiles > counter; counter++) {
    // load tile
    // process tile
    // advance variables for next tile
}
Advantages of the Tiled Merge Kernel

• **Reduced global memory traffic** for `co_rank`.
• **Coalesced loads** from `A` and `B`.
• Thread-level **`co_rank` calls**
  – **use shared memory** and
  – **reduced load imbalance** by limiting range to within a tile.
• **Coalesced stores** to `C`.
Remaining Problem with Tiled Merge Kernel

But we still have an obvious inefficiency: only half of the data loaded in each tile iteration are actually used!

How can we fix this problem?

• Copy unused data to the start of each tile.
• Probably need to add double-buffering ... right?
• Or use cyclic / circular buffers. A bit tricky.
Cyclic Buffers Common in Systems Apps

- Cyclic/circular buffering **fairly common in systems applications**.
- examples:
  - fixed hardware resources
  - avoid dynamic allocation overhead for high-performance software (in OS, for example)
  - avoid copying / allocation in high-performance software
Count States for a Small Buffer

There are a couple of tricky aspects.

Consider a 256-entry buffer.

• How many entries in the buffer are valid?
  • 0 to 256. That’s 257 possible answers.

• Where does the data start?
  • Index 0 to 255. That’s 256 possible answers.
Too Few Bits Means Disallowing States

If there's no data,
- the starting point doesn't matter.
- So we have 65,537 \(2^{16} + 1\) possible states.

If we use two 8-bit indices (start and end)
- to record the state of the buffer,
- we have an issue.

Such a design must guarantee that the buffer is either never full or never empty.
Larger Indices Allows Use of All States

Alternatively, we can use bigger indices. Consider 16-bit indices for our 256-entry buffer.

- Start + 256 == End means full.
- Start == End means empty.

These conditions are the same mod 256 (when mapped to actual locations in buffer).

The extra index bits differentiate full from empty.
Usually, Choose Power of 2 Sizes

In software, extra index bits are cheap, hence typical.

**Index wrap can also lead to problems:**

- integer indices wrap at $2^m$.
- If buffer length does not divide $2^m$ evenly,
- index **wrapping shifts position** in buffer!

So we usually **choose power of 2 sizes** for buffers.
With Proper Design, Not Too Hard to Use

Once we define a cyclic buffer using these rules—
- **power of 2 length** \((2^k)\) and
- **indices with extra bits**—

using such a buffer is fairly easy:
- **indices virtualize physical buffer** as many virtual copies lined up one after another.
- On **each access**, transform “virtual” **index** into a real index **using mod \(2^k\).**

Higher-level **software can sometimes be oblivious** to the circular nature of arrays (in the buffer).
Example of Tile Load with Cyclic Buffer

For example, \texttt{A\_consumed}

- plays role of virtual index into \texttt{tileA}
- (instead of resetting to 0 for each tile).

\begin{verbatim}
if (i + threadIdx.x < A\_length - A\_consumed) {
    tileA[i + threadIdx.x] = 
    A[blk\_A\_curr + A\_consumed + i + threadIdx.x];
}
\end{verbatim}

Replace with \( (i + threadIdx.x + A\_consumed) \% tile\_size \).
Example of Tile Load with Cyclic Buffer

But **to avoid reloading** data,
- we **need a second virtual index** to track
- how much has been loaded, \texttt{A\_loaded}.

```c
if (i + threadIdx.x < A\_length - A\_consumed) {
    tileA[(i + threadIdx.x + A\_consumed) \% tile\_size] =
    A[blk\_A\_curr + A\_consumed + i + threadIdx.x];
}

Add condition \(i + threadIdx.x + A\_consumed \geq A\_loaded\).
Example of Tile Load with Cyclic Buffer

We could then **optimize by**

- **initializing i above 0** at the start of the loop
- (split the tile load loop into two loops for simplicity).

```c
if (i + threadIdx.x + A_consumed >= A_loaded &&
    i + threadIdx.x < A_length - A_consumed) {
    tileA[(i + threadIdx.x + A_consumed) % tile_size] =
        A[blk_A_curr + A_consumed + i + threadIdx.x];
}
```
Also See Code in the Text

More example code and explanations are available in the textbook.

But … Wen-mei’s style is pretty different.

I’ll leave his code in the printed slides, too.
Circular Buffering
Loading Circular Buffering Tiles

... int A_S_start = 0; int B_S_start = 0; int A_S_consumed = tile_size; // in the first iteration, fill the tile_size int B_S_consumed = tile_size; // in the first iteration, fill the tile_size

while(counter < total_iteration)
{
   /* loading (refilling) A_S_consumed elements into A_S */
   for(int i=0; i<A_S_consumed; i+=blockDim.x) {
       if( i + threadIdx.x < A_length - A_consumed && i + threadIdx.x < A_S_consumed) {
           A_S[(A_S_start + (tile_size-A_S_consumed) + i + threadIdx.x)%tile_size] =
           A[A_curr + A_consumed + i + threadIdx.x ];
       }
   }

   /* loading B_S_consumed elements into B_S */
   for(int i=0; i<B_S_consumed; i+=blockDim.x) {
       if(i + threadIdx.x < B_length - B_consumed && i + threadIdx.x < B_S_consumed) {
           B_S[(B_S_start + (tile_size-B_S_consumed) + i + threadIdx.x)%tile_size] =
           B[B_curr + B_consumed + i + threadIdx.x ];
       }
   }
}
Reality vs. Simplified View

(a) reality

(b) simplified
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;

/* find co-rank for c_curr and c_next */
int a_curr = co_rank_circular(c_curr,
   A_S, min(tile_size, A_length-A_completed),
   B_S, min(tile_size, B_length-B_completed),
   A_S_start, B_S_start, tile_size);
int b_curr = c_curr - a_curr;
int a_next = co_rank_circular(c_next,
   A_S, min(tile_size, A_length-A_completed),
   B_S, min(tile_size, B_length-B_completed),
   A_S_start, B_S_start, tile_size);
int b_next = c_next - a_next;

/* do merge in parallel */
merge_sequential_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 */
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;

A_S_start = A_S_start + A_S_consumed;
if (A_S_start >= tile_size) A_S_start = A_S_start - tile_size;

B_S_start = B_S_start + B_S_consumed;
if (B_S_start >= tile_size) B_S_start = B_S_start - tile_size;

__syncthreads();
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) ? A_S_start + i - tile_size : A_S_start + i;
        int j_cir = (B_S_start + j >= tile_size) ? B_S_start + j - tile_size : B_S_start + j;
        int i_m_1_cir = (A_S_start + i - 1 >= tile_size) ? A_S_start + i - 1 - tile_size : A_S_start + i - 1;
        int j_m_1_cir = (B_S_start + j - 1 >= tile_size) ? B_S_start + j - 1 - tile_size : B_S_start + j - 1;
        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;
}
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)?
            A_S_start+i-tile_size:  A_S_start+i;
        int j_cir = (B_S_start + j>= tile_size)?
            B_S_start+j-tile_size:  B_S_start+j;
        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)?
                B_S_start+j-tile_size:  B_S_start+j;
            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)?
                A_S_start+i-tile_size:  A_S_start+i;
            C[k++] = A[i_cir];
        }
    }
}

© Wen-mei W. Hwu, David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021 81
ANY QUESTIONS?