Lec9: Merge

Introduction: Merge

- Given two sorted array A, and B, merge them into a **sorted** array C.
- Useful in say merge sort
- New: in parallelization, the inputs to each thread is dynamically determined because they are data dependent.
  Unlikely all previous examples, which are all statically determined and only dependent on compute structure but not on data.

Merge operation: Sequential Algorithm

Sequential merge:
C[k] = min(A[i],B[j])

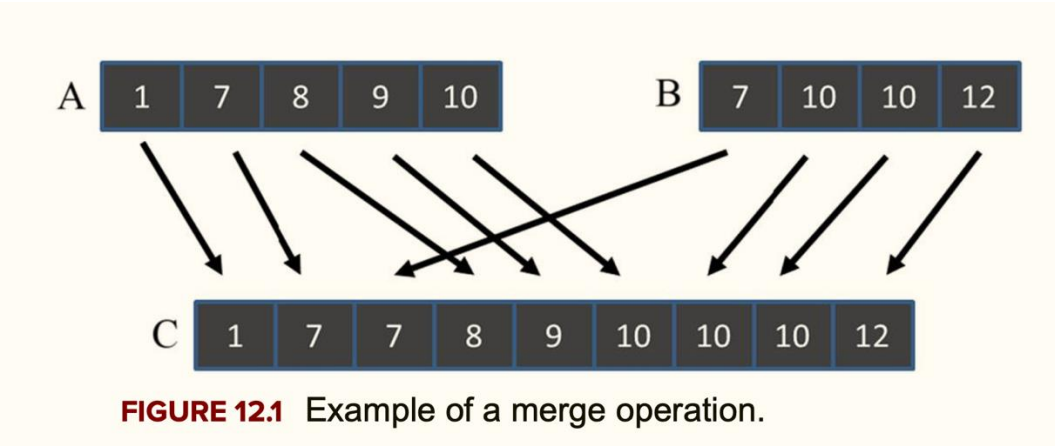

FIGURE 12.1 Example of a merge operation.

```
// merge sorted arrays A[0:m] and B[0:n] into C[0:m+n]
void merge_sequential(int* A, int m, int* B, int n, int* C) {
    int i = 0, j = 0, k = 0;
    while ( i < m && j < n) {
        if (A[i] > B[j])
            C[k++] = B[j++];
        else
            C[k++] = A[i++];
    }
    while (i < m) {
        C[k++] = A[i++];
    }
    while (j < n) {
        C[k++] = B[j++];
    }

}
```

Merge: Parallelization

- How to parallelize?
- Recall previous examples:
  - VectorAdd: input decomposition
  - Histogram: input decomposition
  - Scan/Reductin: input decomposition
  - MatrixMultiplication: output decomposition + owner computes rule
  - ImageBlur/Convolution: output decomposition + owner compute rule
- What about merge?
  - Input decomposition: seems difficult: which chunk of A and which chunk of B to merge?
  - Output decomposition?: possible, if noting
    any chunk of C is a merge of **a** chunk of A and **a** chunk of B.
    Q: which chunk of A and which chunk of B?

- any chunk of C is a merge of **a** chunk of A and **a** chunk of B.
  Q: which chunk of A and which chunk of B?
- Or put it more formally:
  for any C[k1:k2] in the merged result, there must exist i1,i2 and j1,j2 such that
  C[k1:k2] = merge(A[i1:i2], B[j1:j2])
- If we can compute mapping (k1,k2) -> (i1, i2, j1, j2)
  then we have an **output** decomposition:
  Give a C chunk to a thread/block, and owner computes.

Co-rank

- for any C[k1:k2] in the merged result, there must exist i1,i2 and j1,j2 such that
  C[k1:k2] = merge(A[i1:i2], B[j1:j2])
- Co-rank is a function that is
  co_rank(k, A, m, B, n) -> i
  where C[0:k] = merge(A[0:i], B[0:k-i])
- Once we have co_rank() function, for any chunk of C, C[k1:k2] we can compute its merge inputs A[i1:i2], B[j1:j2] as
  co_rank(k1, A, m, B, n) -> i1
  co_rank(k2, A, m, B, n) -> i2
  j1 = k1 – i1, j2 = k2 – i2

co_rank() example:

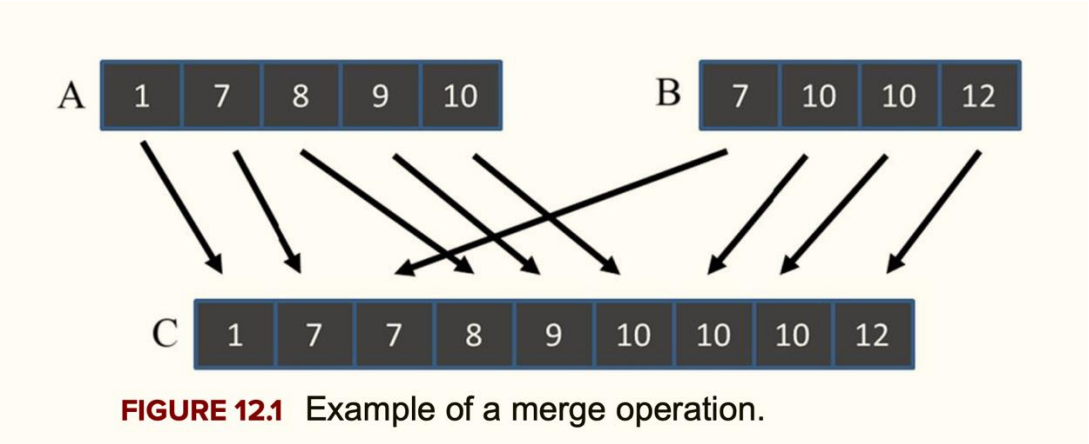

FIGURE 12.1 Example of a merge operation.

k=0 i=0

k=1 i=1

k=2 i=2

k=3 i=2

k=4 i=3

k=5 i=4

k=6 i=5

k=7 i=5

k=8 i=5

Co_rank() implementation

- Definition: co_rank(k, A, m, B, n) -> i
  where C[0:k] = merge(A[0:i], B[0:k-i])
- How to compute co_rank()?
  Consider a simpler question, given an ii can we test whether ii is the i that we want, given the parameters k, A, m, B, n?
- Sufficient and necessary condition for ii = i:
  let jj = k - ii
  A[ii] >= B[jj-1]
  B[jj] >= A[ii-1]
- There exists at least one such ii
- Now we know how to test ii, next Q: how to find ii?

Co_rank(k, A, m, B, n)

- How to find i, given that we can test ii for the solution?
- Naïve method: iterate through all possible i, and test it.
  This is very expensive, O(n) time complexity.
- A better way is to do **binary search**
  take an interval for search [a,b], and pick any point in it (say mid-point m = (a+b)/2)
  if A[m] < B[k-m-1], then m is too small, search in [m,b] next
  if B[k-m] < A[m-1], then m is too large, search in [a,m] next
- Each above condition test reduce the search space **by half.**
- After O(log(n)) rounds of search we must be able to find the i.

Co-rank() implementation

```
// merge two sorted array A & B into C;
// C[0:k] is a merge of A[0:i] and B[0:j]
// given k, compute and return i.
// j can be derived by k-i
__host__ __device__ int co_rank(int k, int *A, int m, int *B, int n)
{
    // starting position of binary search
    int i = k<m ? k:m;
    int j = k - i;
    // i, j cannoot be below i0, j0
    int i0 = k < n ? 0 : k - n;
    int j0 = k < m ? 0 : k - m;
    int delta;
    while(1) {
        if (i > 0 && j < n && A[i-1] > B[j]) { // i overshoot, need to reduce it
            delta = (i-i0+1) / 2; // CEIL((i-i0)/2)
            j0 = j;
            j = j+delta;
            i = i - delta;
        } else if (j > 0 && i < m && B[j-1] > A[i]) { // i undershoot, need to increase it
            delta = (j-j0+1) / 2;
            i0 = i;
            i = i + delta;
            j = j - delta;
        } else {
            return i;
        }
    }
}
```

Kernel1:

- Output decomposition:
  each thread gets CHUNK sized chunk in C, and is responsible to compute it.
- How? Say thread tid gets chunk C[k1:k2], it computes A[i1:i2], B[j1:j2] using co_rank(), and then do a sequential merge: C[k1:k2] = merge(A[i1:i2],B[j1:j2])

Kernel1:

```
// Kernel 1: Each thread processes a chunk of the output array
template<int CHUNK_SIZE>
__global__ void merge_kernel1(int* A, int m, int* B, int n, int* C) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int c_start = tid * CHUNK_SIZE;
    int c_end = min(c_start + CHUNK_SIZE, m + n);
    if (c_start >= c_end) return;

    int a_start = co_rank(c_start, A, m, B, n);
    int b_start = c_start - a_start;

    int a_end = co_rank(c_end, A, m, B, n);
    int b_end = c_end - a_end;

    int i = a_start, j = b_start, k = c_start;
    merge_sequential(A + i, a_end - i, B + j, b_end - j, C + k);
}
```

Kernel1

- Performance expectation?
- Merge must be memory bandwidth bound, so optimizing and minimizing global memory access is crucial.
  - Memory access coalesced?
  - How many times are global memory accessed?
  - What’s the expected performance in GB/s?

Kernel2: coalescing via shared memory

- To improve global memory access pattern and also make sure that co_rank() does not hit global memory,
- We can decompose C into bigger chunks and assign chunk to thread block
- And each thread block loads the A chunk and B chunk into shared memory collaboratively (coalesced)

Kernel2: #1

```cpp
// Kernel 2: Coalesced access with proper merge in shared memory
template<int TILE_SIZE>
__global__ void merge_kernel2(int* A, int a_len, int* B, int b_len, int* C) {
    extern __shared__ int shared_mem[];
    int* sA = shared_mem;
    int* sB = shared_mem + TILE_SIZE;
    int tile_idx = blockIdx.x;
    int tile_start = tile_idx * TILE_SIZE;
    int tile_end   = min(tile_start + TILE_SIZE, a_len + b_len);
    if (tile_start >= tile_end) return;

    // --- 1) Global split for this entire tile ---
    if (threadIdx.x == 0) {
        sA[0] = co_rank(tile_start, A, a_len, B, b_len);
        sA[1]   = co_rank(tile_end,   A, a_len, B, b_len);
    }
    __syncthreads();
    int a_start = sA[0];
    int a_end   = sA[1];
    int b_start = tile_start - a_start;
    int b_end   = tile_end - a_end;

    // Load A/B segments into shared memory
    int a_load = a_end - a_start;
    int b_load = b_end - b_start;

    for (int i = threadIdx.x; i < a_load; i += blockDim.x) {
        sA[i] = (a_start + i < a_len) ? A[a_start + i] : INT_MAX;
    }
    for (int i = threadIdx.x; i < b_load; i += blockDim.x) {
        sB[i] = (b_start + i < b_len) ? B[b_start + i] : INT_MAX;
    }
    __syncthreads();
```

Kernel2: #2

```
// --- 2) Each thread merges a sub-chunk of the tile ---
int tid = threadIdx.x;
// how many elements in this tile
int tile_size = a_load + b_load;
// chunk_size for each thread
int chunk_size = (tile_size + blockDim.x - 1) / blockDim.x;

// c_start/c_end for *this thread* in global space:
int c_start = tile_start + tid * chunk_size;
int c_end   = min(c_start + chunk_size, tile_start + tile_size);

// "Empty" chunk if c_start >= c_end
if (c_start >= c_end) return;

// local_start/local_end within shared arrays
int local_start = c_start - tile_start;  // how far into the tile
int local_end   = c_end   - tile_start;

// --- 3) Do two splits inside the tile: one for local_start, one for local_end ---
int a_split_start = tile_find_split(sA, a_load, sB, b_load, local_start);
int b_split_start = local_start - a_split_start;
int a_split_end   = tile_find_split(sA, a_load, sB, b_load, local_end);
int b_split_end   = local_end - a_split_end;

// Merge sA[a_split_start .. a_split_end] & sB[b_split_start .. b_split_end]
int out_pos = c_start; // global output index
// int out_pos = 0;
int i = a_split_start;
int j = b_split_start;


merge_sequential(sA + i, a_split_end - i, sB + j, b_split_end - j, C + out_pos);
```

Benchmarks

- A, B of size 16Million int

```
Kernel 1 valid
Kernel 2 valid
CUB Merge valid
C1[100] = 318, C2[100] = 318, C3[100] = 318
C1[1000] = 2947, C2[1000] = 2947, C3[1000] = 2947
C1[10000] = 29743, C2[10000] = 29743, C3[10000] = 29743
Kernel 1: 10.7196 ms, 25.0414 GB/s
Kernel 2: 1.51219 ms, 177.514 GB/s
CUB Merge: 0.445248 ms, 602.89 GB/s
```

```
// Kernel 1 execution
const int CHUNK_SIZE = 32;
dim3 block1(256);
dim3 grid1((c_len + CHUNK_SIZE * block1.x - 1) / (CHUNK_SIZE * block1.x));
merge_kernel1<CHUNK_SIZE><<<grid1, block1>>>(d_A, a_len, d_B, b_len, d_C1);
// Kernel 2
const int TILE_SIZE = 1024;
dim3 block2(128);
dim3 grid2((c_len + TILE_SIZE - 1) / TILE_SIZE);
size_t shared_mem =  TILE_SIZE * sizeof(int);
merge_kernel2<TILE_SIZE><<<grid2, block2, shared_mem>>>(d_A, a_len, d_B, b_len, d_C2);
// CUB
DeviceMerge::MergeKeys(d_temp_storage, temp_storage_bytes,
                                    d_A, a_len,
                                    d_B, b_len,
                                    d_C3
                                    ));
```

How did CUB get so fast?

- GPU Merge Path – https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=ff958b54a930bda2ca902f9abd4f0aaf55b21862
  Oded Green, Robert McColl, and David A. Bader. 2012. GPU merge path: a GPU merging algorithm. In Proceedings of the 26th ACM international conference on Supercomputing (ICS '12).