In Memory of Nacho Navarro

PUMPS+AI 2022

Lecture 2: Input Regularization Techniques for Gather Parallelization

Wen-mei Hwu, Mateo Valero, Toni Pena, Juan Gomez-Luna, Marc Jorda, Leonidas Kosmidis, Bernat Font
Challenges in Arranging Input for Gather Parallelization

• Input data distribution in space may be far from uniform
  – Quad/Oct Trees

• One may not be able to pre-determine the input elements needed for calculating output elements
  – Dynamic Search

• …
2D Spatial Sorting Using - Quadtree

- Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold.

![Quadtree Diagram](image)
Quadtrees – 1st recursion

- Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold.

**Depth = 1**

**Threshold = 2**
Quadtrees – 2nd recursion

- Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold.

Depth = 2

Threshold = 2
Quadtrees – 3rd recursion

- Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold.

Depth = 3

Threshold = 2
Recall: CUDA Dynamic Parallelism

- CPU-GPU without and with dynamic parallelism

(a) Without Dynamic Parallelism
(b) With Dynamic Parallelism
Quadtree Implementation using Dynamic Parallelism (I)

- 1 thread block is launched from host

Outline of recursive kernel:

1. **Assign block to node**
2. **points > min_points && depth < max_depth**
3. **Y**
   - **Compute center of bounding box**
   - **Count points in children**
   - **Scan for offsets**
   - **Reorder points**
   - **Launch 4 children**
4. **N**
   - Exit

Depth = 0

Buffer 0:
```
| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u |
```

Buffer 1:
```
| b | c | d | e | f | g | j | m | o | a | d | h | i | k | l | p | q | r | s | t | u |
```
Quadtree Implementation using Dynamic Parallelism (II)

- Each block launches 1 child grid of 4 blocks
Quadtrees Implementation using Dynamic Parallelism (III)

- Each block launches 1 child grid of 4 blocks
Quadtree Implementation using Dynamic Parallelism (IV)

- Each block launches 1 child grid of 4 blocks

Depth = 3

Buffer 1: b c e f m n g j o a d h k p q i l r t s u

Buffer 0: b c e f m n g j o a d h k p q i l r t s u
Quadtree Implementation using Dynamic Parallelism (V)

- Points in the same quadrant are grouped together

Each thread block may need to move its part to Buffer 0 before retiring, since there is where we expect the final output.
Quad Tree Construction: Summary

• The execution starts with host launching one thread block
  – At each recursion, if the number of atoms in the quadrant is less than or equal to the threshold, the thread block exits

• At each recursion, threads in each thread block that do not exit will collaboratively
  – Determine the number of atoms that belong in each quadrant
  – Perform a scan to determine the starting point of each quadrant
  – Reorder the atoms so that all atoms in the same quadrant are placed consecutively
  – One representative thread launches a kernel with 4 child blocks

• Oct Tree is for 3D space
  – A 3D space is divided into 8 Octants
  – Each block that does not exit launches 1 child grid of 8 blocks
Example of Sorted vs. Unsorted Lists

Sorted (ordered):

1 7 7 8 9 10 10 10 12

Not sorted (unordered):

12 7 8 10 7 10 9 1 10
Merging Two Sorted Lists into One

- There is an order relation (e.g., less than or equal to) in the sorted lists and in the merged list.
- We focus on stable sort.
- Whenever A and B have elements of the same value, the element from A goes first.

Diagram:

- List A: 1 7 8 9 10
- List B: 7 10 10 12
- Merged List C: 1 7 7 8 9 10 10 10 12
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

    // Handle the comparison of A[] and B[]
    while ((i < m) && (j < n)) {
        if (A[i] <= B[j]) {
            C[k++] = A[i++];
        } else {
            C[k++] = B[j++];
        }
    }
}
A Sequential Ordered Merge Function (II)

... 

```c
if (i == m) {
    // Done with A[] handle remaining B[]
    for (; j < n; j++) {
        C[k++] = B[j];
    }
} else {
    // Done with B[], handle remaining A[]
    for (; i < m; i++) {
        C[k++] = A[i];
    }
}
```

Sequential algorithm complexity
$O(m + n)$
Parallel Merge

- Scatter parallelization?
  - Each thread would take a section of list A and a section of list B, and would find the element locations in list C.
  - The destination of an element of A or an element of B depends on the elements of the other list.
  - This includes the elements of A and B assigned to other threads.
  - No systematic solution yet.
Parallel Merge: Gather Parallelization (I)

- We partition the output list equally among threads.
- Each thread collects input elements for its section of the output.

The range of input elements to be used by each thread is a function of the input elements.
Parallel Merge: Gather Parallelization (II)

- **Observation**
  - Each output section receives its elements from a continuous (possibly empty) section of A and a continuous (possibly empty) section of B.
  - For any \( k \) such that \( 0 \leq k < m + n \), we can find \( i \) and \( j \) such that \( k = i + j \), \( 0 \leq i < m \) and \( 0 \leq j < n \).

For an element \( C[k] \), \( k \) is referred to as its **rank** and \( i \) and \( j \) are referred to as its **co-ranks**.
Parallel Merge: Gather Parallelization (III)

• Each thread gathers the elements of a continuous section of the output
  – Coarsened from one element per thread
• All threads identify the starting and ending locations of the continuous sections of the inputs (A and B) that they will use
  – Input identification and partition by finding co-ranks
• All threads perform merge for their sections in parallel
  – Each thread executes a sequential merge for its own section
Co-Rank Value Calculation (I)

- The co-rank values of a prefix string leading to the $k^{th}$ output element is a pair of $i$ and $j$ values that specify the rank values of the prefix strings of $A$ and $B$ that are used in forming the output prefix string leading to that $k^{th}$ output element.

- For a given output prefix string of rank $k$, its co-rank values can be found by searching for $i$ and $j$ such that $k = i + j$ and:
  - $A[i-1] \leq B[j]$
Co-Rank Value Calculation (II)

- Finding co-rank values for different threads is not balanced
- The search range for a prefix string is a subset of that of a higher ranked one
Co-Rank Function (I)

- Use binary search (or n-ary search) for $i$ and $j$ values to minimize the effect of increasing search ranges
  - Reduces the computational complexity from $O(N)$ to $O(\log N)$
- Co-rank function
  - $\text{int co_rank}(\text{int } k, \text{int } *A, \text{int } m, \text{int } *B, \text{int } n)$
  - Returns $i$, and $j$ is derived as $k - i$
Co-Rank Function (II)

```c
int co_rank(int k, int* A, int m, int* B, int n) {
    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; // j_low = max(0, k-m)
    int delta;
    bool active = true;
    while (active) {
        if (i > 0 && j < n && A[i - 1] > B[j]) {
            delta = ((i - i_low + 1) >> 1); // ceil(i-i_low)/2
            j_low = j;
            j = j + delta;
            i = i - delta;
        } else if (j > 0 && i < m && B[j - 1] >= A[i]) {
            delta = ((j - j_low + 1) >> 1);
            i_low = i;
            i = i + delta;
            j = j - delta;
        } else {
            active = false;
        }
    }
    return i;
}
```

Example:
```c
int i = co_rank(3, A, 5, B, 4);
```
Co-Rank Function: Iteration 0

```c
int co_rank(int k, int* A, int m, int* B, int n) {
    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; // j_low = max(0, k-m)
    int delta;
    bool active = true;
    while(active) {
        if (i > 0 && j < n && A[i-1] > B[j]) {
            delta = ((i - i_low +1) >> 1); // ceil(i-i_low)/2
            j_low = j;
            j = j + delta;
            i = i - delta;
        } else if (j > 0 && i < m && B[j-1] >= A[i]) {
            delta = ((j - j_low +1) >> 1);
            i_low = i;
            i = i + delta;
            j = j - delta;
        } else {
            active = false;
        }
    }
    return i;
}
```
int co_rank(int k, int* A, int m, int* B, int n) {
    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; // j_low = max(0, k-m)
    int delta;
    bool active = true;
    while(active) {
        if (i > 0 && j < n && A[i-1] > B[j]) {
            delta = ((i - i_low +1) >> 1); // ceil(i-i_low)/2
            j_low = j;
            j = j + delta;
            i = i - delta;
        } else if (j > 0 && i < m && B[j-1] >= A[i]) {
            delta = ((j - j_low +1) >> 1);
            i_low = i;
            i = i + delta;
            j = j - delta;
        } else {
            active = false;
        }
    }
    return i;
}
int co_rank(int k, int* A, int m, int* B, int n) {
    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; // j_low = max(0, k-m)
    int delta;
    bool active = true;
    while(active) {
        if (i > 0 && j < n && A[i-1] > B[j]) {
            delta = ((i - i_low +1) >> 1); // ceil(i-i_low)/2
            j_low = j;
            j = j + delta;
            i = i - delta;
        } else if (j > 0 && i < m && B[j-1] >= A[i]) {
            delta = ((j - j_low +1) >> 1);
            i_low = i;
            i = i + delta;
            j = j - delta;
        } else {
            active = false;
        }
    }
    return i;
}
Basic Parallel Merge (I)

- Each thread is in charge of a continuous section of the output
  - $k_{\text{curr}}$ and $k_{\text{next}}$ define the output range for a thread $C[k_{\text{curr}}]$ to $C[k_{\text{next}}]$
- Each thread calls the co-rank function to get $i_{\text{curr}}$, $i_{\text{next}}$, $j_{\text{curr}}$, and $j_{\text{next}}$ values
__global__ void merge_basic_kernel(int* A, int m, int* B, int n, int* C) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;

    // Start index of output section
    int k_curr = tid * ceil((m+n)/(blockDim.x*gridDim.x));
    // End index of output section
    int k_next = min((tid+1) * ceil((m+n)/(blockDim.x*gridDim.x)), m+n);

    // Co-rank values
    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]);
}
Basic Parallel Merge (III)

- **Problem**: Uncoalesced memory accesses
  - E.g., first read of thread 0 to A[0], thread 1 to A[2], thread 2 to B[1]; first write of thread 0 to C[0], thread 1 to C[3], thread 2 to C[6]

- **Solution**: Collaborative loading of sections of A and B into the shared memory – **tiled merge kernel**
Tiled Merge Kernel (II)

- Assign output sections to thread blocks
  - Each section should have at least a few thousand elements
- Leader thread of each block performs binary (n-ary) search to identify input sections
- Each block iteratively generates its output section
- In each iteration:
  - Threads of a block collaboratively load a tile of A and a tile of B into shared memory
    - Each tile should have at least a few hundred elements
  - Divide the output tile into subsections and assign them to threads
    - Each subsection should have ten or more of elements
  - All threads perform parallel merge on input and output tiles
Tiled Merge Kernel: Partitioning among Thread Blocks

- **A**: input subarrays for block 0
- **B**: input subarrays for block 1
- **C**: input subarrays for block 2

**Global Memory**

- **A_S**: output subarray for block 0
- **B_S**: output subarray for block 1
- **C**: output subarray for block 2

**Shared Memory**
Tiled Merge Kernel: Co-Rank Values

```c
__global__ void merge_tiled_kernel(int* A, int m, int* B, int n, int* C, int tile_size) {
    // Shared memory allocation
    extern __shared__ int shareAB[];
    int* A_S = &shareAB[0];  // A_S is the first half of shareAB
    int* B_S = &shareAB[tile_size];  // B_S is the second half of ShareAB

    // Starting point for current block
    int C_curr = blockIdx.x * ceil((m+n)/gridDim.x);
    // Starting point for next block
    int C_next = min((blockIdx.x+1) * ceil((m+n)/gridDim.x), (m+n));

    if (threadIdx.x == 0) {
        A_S[0] = co_rank(C_curr, A, m, B, n);  // Make the block-level co-rank values visible to
        A_S[1] = co_rank(C_next, A, m, B, n);  // other threads in the block
    }
    __syncthreads();

    int A_curr = A_S[0]; int B_curr = C_curr - A_curr;
    int A_next = A_S[1]; int B_next = C_next - A_next;

    __syncthreads();

    ...
```
Tiled Merge Kernel: Loading Tiles (Iteration 0)

- **input subarrays for block 0**
- **input subarrays for block 1**
- **input subarrays for block 2**

Global Memory

- **output subarray for block 0**
- **output subarray for block 1**
- **output subarray for block 2**

Shared Memory
Tiled Merge Kernel: Loading Tiles

```c
int counter = 0;  // Iteration counter
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);  // Total iterations
int C_completed = 0;  int A_consumed = 0;  int B_consumed = 0;

while(counter < total_iteration)
{
  // Loading tile-size A and B elements into shared memory
  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];
    }
  }

  for(int i=0; i<tile_size; i+=blockDim.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: Generating Output Tiles (Iteration 0)

- Input subarrays for block 0, 1, and 2
- Output subarrays for block 0, 1, and 2
- Global Memory: A_consumed, B_consumed
- Shared Memory: A_S, B_S
- C: Combined output subarray for all blocks

Diagram showing the flow of data from input subarrays to output subarrays, with arrows indicating the movement and consumption of data in global and shared memory.
... 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(c_curr, A_S, min(tile_size, A_length-A_consume),
                       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_consume),
                       B_S, min(tile_size, B_length-B_consumed));
  int b_next = c_next – a_next;

...
Each Thread Generates a Subsection of the Output Tile

...  

// All threads call sequential merge
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 A and B elements that have been consumed thus far
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();
}  
}

Performance can be further improved by generating output to shared memory first
Tiled Merge Kernel: Loading Tiles (Iteration 1)

input subarrays for block 0
input subarrays for block 1
input subarrays for block 2

Global Memory

A_consumed
B_consumed

Shared Memory

output subarray for block 0
output subarray for block 1
output subarray for block 2
Tiled Merge Kernel: Pros & Cons

• **Pros**
  – Coalesced loads of the input elements
  – Reduced global memory traffic for co-rank functions
    • Thread-level co-rank functions are done in shared memory
  – Coalesced stores if the output tiles are generated in shared memory

• **Cons**
  – Only half of the input elements loaded into shared memory are used (in the worst case)
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];
        }
    }

    ...
Simplified Model for Co-Rank Values

- Tiles wrap around in the circular buffer
  - This makes the handling of co-rank values more complex

(a) Reality

\[
\begin{align*}
A_S & \quad A_S_{\text{start}} \\
B_S & \quad B_S_{\text{start}}
\end{align*}
\]

\[
\begin{align*}
\text{a}_\text{curr} & \quad \text{a}_\text{next} \\
\text{a}_\text{next} - \text{a}_\text{curr} & \\
\text{b}_\text{curr} & \quad \text{b}_\text{next} \\
\text{b}_\text{next} - \text{b}_\text{curr} + \text{tile}_\text{size}
\end{align*}
\]

(b) Simplified

\[
\begin{align*}
A_S & \quad A_S_{\text{start}} \\
B_S & \quad B_S_{\text{start}}
\end{align*}
\]

\[
\begin{align*}
\text{a}_\text{curr} & \quad \text{a}_\text{next} \\
\text{a}_\text{next} - \text{a}_\text{curr} & \\
\text{b}_\text{curr} & \quad \text{b}_\text{next} \\
\text{b}_\text{next} - \text{b}_\text{curr}
\end{align*}
\]
Circular-Buffer Merge Kernel (I)

...\n
```c
int c_curr = threadIdx.x * (tile_size/blockDim.x);
int c_next = (threadIdx.x+1) * (tile_size/blockDim.x);

if (c_curr <= C_length-C_completed) c_curr = C_length-C_completed;
if (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);
```

...
Circular-Buffer Merge Kernel (II)

...  

// 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();
}

Sequential merge and co-rank functions are the only change:
A well-designed library interface limits the impact on user code.
Co-Rank Function with Circular Buffer

```c
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 + j - 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;
}
```
Sequential Merge with Circular Buffers

```c
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];
        }
    }
}
```
Merge Sort: Summary

- Parallelization of merge sort requires each thread to dynamically identify its input ranges
  - Input ranges are data dependent
  - Challenges when using tiling

- Circular buffers to make full use of the data loaded into shared memory
  - Simplified buffer access model to not increase code complexity
ANY FURTHER QUESTIONS?