GP-GPU and High Performances Computing

Lecture 12 – Graph

December 18, 2023

Previously

- Sparse data.
- Histograms.
- Shared and private memory.
- Atomic operations.

An other sorting

Example 3: Bitonic Sort

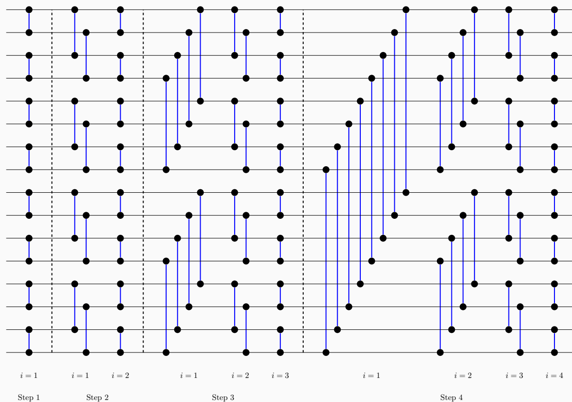
- A bitonic sequence is a sequence of numbers $a_0, a_1, \dots, a_{n-1}$ which monotonically increases in value, reaches a single maximum, and then monotonically decreases in value.

  $$a_0 < a_1 < \dots < a_{i-1} < a_i > a_{i+1} > \dots > a_{n-2} > a_{n-1}$$

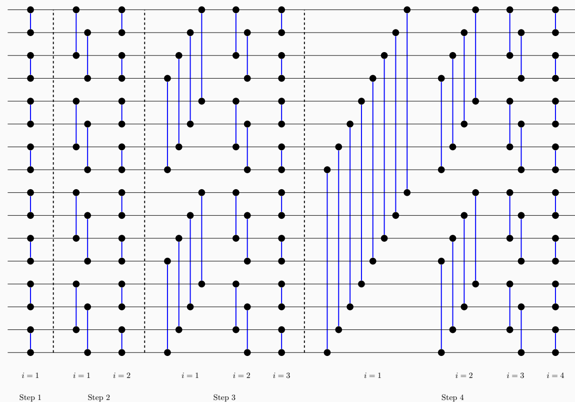
  for some value of i. A sequence is also considered to be bitonic if the relation above can be achieved by shifting the numbers cyclically.
- Every 2 element sequence is bitonic
- 4 element: (a_0, a_1, a_2, a_3)
  - Sort (a_0, a_1) such that $a_0 \leq a_1$
  - Sort (a_2, a_3) such that $a_2 \geq a_3$
- 8 element: $(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$
  - Make (a_0, a_1, a_2, a_3) and (a_4, a_5, a_6, a_7) bitonic
  - Use bitonic split to make (a_0, a_1, a_2, a_3) increasing
  - Use bitonic split to make (a_4, a_5, a_6, a_7) decreasing

Example 3: Sequential Idea

- For list length $l = 2, 4, 8, \dots 2^m = n$
- Use bitonic sort to make successive groups of l elements alternatively increasing and decreasing



Example 3: Parallel Idea



Graph traversal computation

Objectives

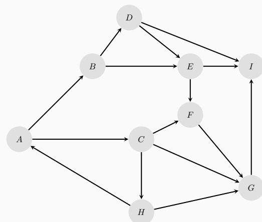
- Study graph search as a prototypical graph-based algorithm
- Learn techniques to mitigate the memory-bandwidth-centric nature of graph-based algorithms
- Introduce work queues and see how they fit into a massively parallel programming framework

Application of graphs

- Social media connection graphs
- Driving directions
- Telecommunication networks
- Manufacturing process dependencies
- Computation graph
- 3D Meshes
- Graphical models

Massive graphs tend to be sparse!

Example



	A	B	C	D	E	F	G	H	I
A		1	1						
B				1	1				
C						1	1		
D					1				1
E						1			1
F							1		
G									1
H	1						1		
I									

Adjacency matrix might be store using CSR format.

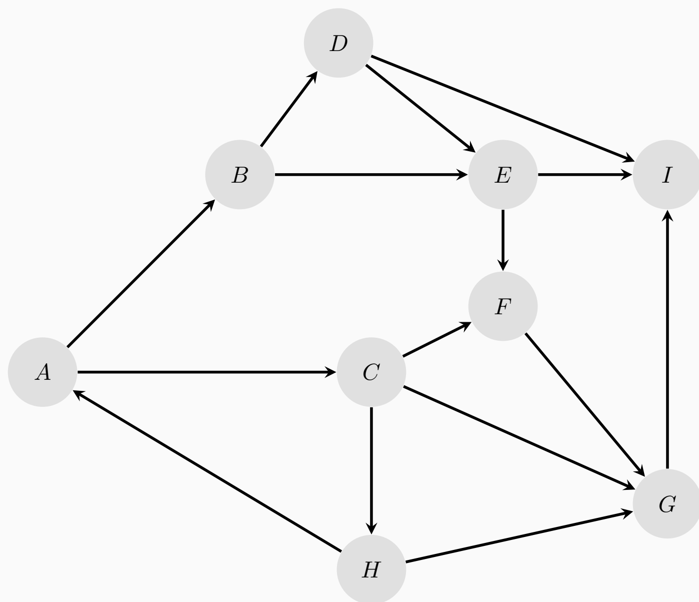
```
AA[15] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0 }
JA[15] = {1, 2, 3, 4, 5, 6, 7, 4, 8, 5, 8, 6, 8, 0, 6}
IA[9] = { 0, 2, 4, 7, 9, 11, 12, 13, 15, 15}
```

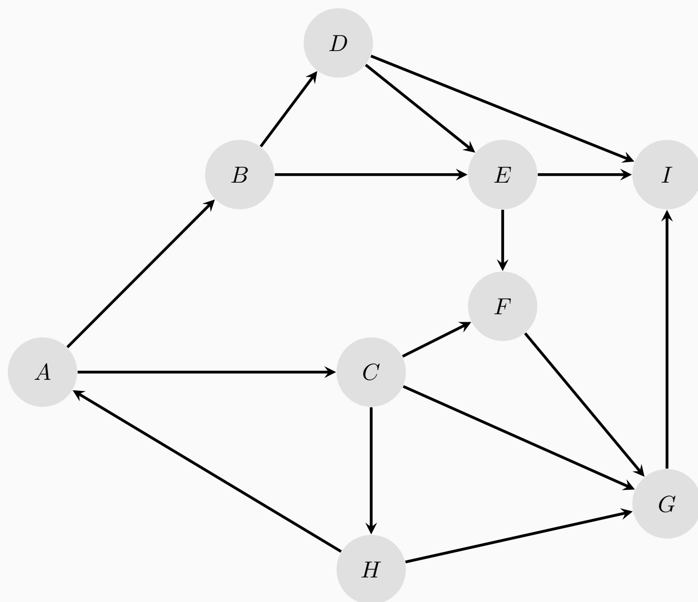
Graph search: BFS

Given a source node S, find the number of steps required to reach each node N in the graph.

Given this labelling of the graph, one can easily find a shortest path from S to a destination T.

Example graph




D
B
E
I
F
A
C
G
H

Sequential code

```
1   void BFS_sequential(int source, const int * row_ptr, const int * dest, int * dist) {
2     int queue[2][MAX_QUEUE_SIZE];
3     int * currentQueue= &queue[0];
4     int * previousQueue = &queue[1];
5     int currentQueueSize= 0, previousQueueSize = 0;
6     insertIntoQueue(source, previousQueue, &previousQueueSize);
7     dist[source] = 0;
8     while (previousQueueSize > 0) {
9       // visit all vertices on the previous Queue
10      for (int f = 0; f < previousQueueSize; f++) {
11        const int currentVertex = previousQueue[f];
12        // check all outgoing edges
13        for (int i = row_ptr[currentVertex]; i < row_ptr[currentVertex+1]; ++i) {
14          if (dist[dest[i]] == -1) {
15            // this vertex has not been visited yet
16            insertIntoQueue(dest[i], currentQueue, &currentQueueSize);
17            dist[dest[i]] = dist[currentVertex] + 1;
18          }
19        }
20      }
21      swap(currentQueue, previousQueue);
22      previousQueueSize = currentQueueSize;
23      currentQueueSize = 0;
24    }
25  }
```

Parallel BFS: basic approach

- Assign one thread per vertex
- For each iteration, check all incoming edges to see if the source vertex was just visited in the last iteration; if so, mark as visited in this iteration
- Not very work efficient; $O(VL)$ for V = number of vertices, L = length of longest path
- Difficult to detect stopping criterion

Parallel BFS: Improve

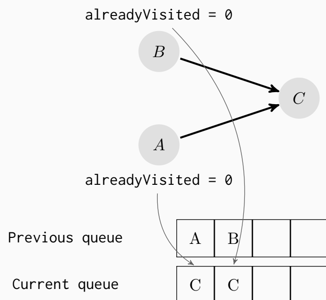
- Parallelize each individual iteration of the while loop in the sequential BFS code
- Assign a section of the vertices in the previous Queue to each thread
- Introduce a synchronization point at the end of each iteration

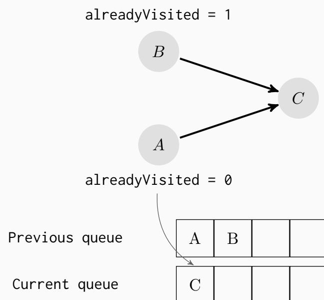
BFS host

```
1     void BFS_host(int source, const int * row_ptr, const int * dest, int * dist) {
2       int dQueue[2][MAX_Queue_SIZE];
3       int * dCurrentQueueSize;
4       int * dPreviousQueueSize; int hPreviousQueueSize;
5       int * dVisited;
6       int * dCurrentQueue = &Queue[0];
7       int * dPreviousQueue = &Queue[1];
8       // allocate device memory, copy memory from device to host, initialize Queue sizes, etc.
9   ...
10      hPreviousQueueSize = 1;
11      while (hPreviousQueueSize > 0) {
12        int numBlocks = (hPreviousQueueSize-1) / BLOCK_SIZE + 1;
13        BFS_Bqueue_kernel<<<numBlocks, BLOCK_SIZE>>>(dPreviousQueue, dPreviousQueueSize, dCurre
14        drow_ptr, dDestinations, dDistances, dVisited);
15        swap(dCurrentQueue,dPreviousQueue);
16
17        cudaMemcpy(dPreviousQueueSize, dCurrentQueueSize, sizeof(int), cudaMemcpyDeviceToDevice
18        cudaMemset(dCurrentQueueSize, 0, sizeof(int));
19        cudaMemcpy(&hPreviousQueueSize, dPreviousQueueSize, sizeof(int), cudaMemcpyDeviceToHost
20      }
21    }
```

Output Interference

- A flag marks whether or not a vertex has been visited.
- From a correctness perspective, output interference on flags can be ignored, but it will lead to additional/replicated work.
- We will be using `atomicExch`.





BFS kernel - Basic

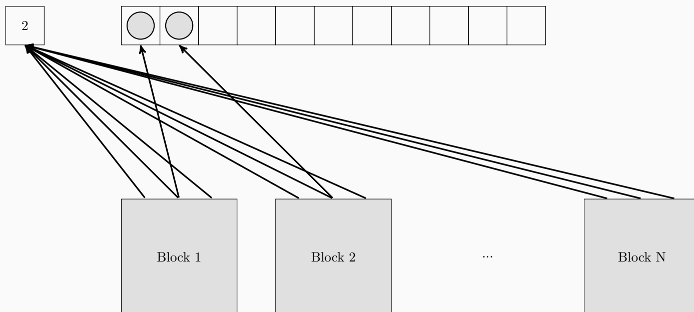
```
1   __global__ void BFS_Bqueue_kernel(const int * previousQueue, const int * pre
2     int * currentQueue, int * currentQueueSize, const int * row_ptr,
3     const int * destinations, int * distances, int * visited) {
4     const int t = threadIdx.x + blockDim.x * blockIdx.x;
5     if (t < *previousQueueSize) {
6       const int vertex = previousQueue[t];
7       for (int i = row_ptr[vertex]; i < row_ptr[vertex+1]; ++i) {
8         // check visitation atomically, avoiding redundant expansion
9         const int alreadyVisited = atomicExch(&(visited[destinations[i]]),1);
10        if (!alreadyVisited) {
11          // we're visiting a new vertex: get a spot in line atomically
12          const int queueIndex = atomicAdd(currentQueueSize, 1);
13          // place the vertex in line
14          currentQueue[queueIndex] = destinations[i];
15        }
16      }
17    }
18    __syncthreads();
19  }
```

Inference (2)

Inference occurred when writing when placing vertices in the queue (line 14 of kernel code).

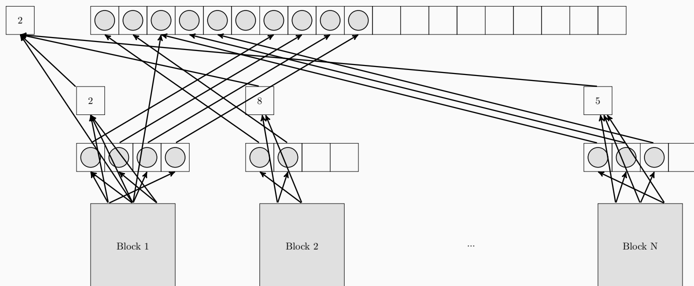
To obtain a correct output, threads need to synchronize with an atomic operation.

Main bottleneck of the basic kernel.



Privatization of the Queue

- We can make a private, block-level copy of the queue.
- Once complete, the private queues are combined to form the global queue.



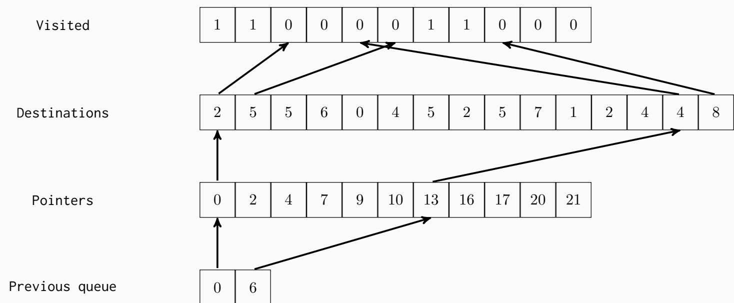
Kernel

```
1         __global__ void BFS_Bqueue_kernel(const int * previousQueue, const int * previousQueueSize,
2          int * currentQueue, int * currentQueueSize, const int * row_ptr, const int * destinations,
3          int * distances, int * visited)
4         {
5           __shared__ int sharedCurrentQueue[BLOCK_QUEUE_SIZE];
6           __shared__ int sharedCurrentQueueSize, blockGlobalQueueIndex;
7
8           if (threadIdx.x == 0) sharedCurrentQueueSize = 0;
9           __syncthreads();
10
11          const int t = threadIdx.x + blockDim.x * blockIdx.x;
12          if (t < *previousQueueSize) {
13            const int vertex = previousQueue[t];
14            for (int i = row_ptr[vertex]; i < row_ptr[vertex+1]; ++i) {
15              const int alreadyVisited = atomicExch(&(visited[destinations[i]]),1);
16              if (!alreadyVisited) {
17                distances[destinations[i]] = distances[i] + 1;
18                const int sharedQueueIndex = atomicAdd(&sharedCurrentQueueSize,1);
19                if (sharedQueueIndex < BLOCK_QUEUE_SIZE) { // there is space in the   local queue
20                  sharedCurrentQueue[sharedQueueIndex] = destinations[i];
21                } else { // go directly to the global queue
22                  sharedCurrentQueueSize = BLOCK_QUEUE_SIZE;
23                  const int globalQueueIndex = atomicAdd(currentQueueSize, 1);
24                  currentQueue[globalQueueIndex] = destinations[i];
25                }
26              }
27            }
28          }
29          __syncthreads();
30
31          if (threadIdx.x == 0) blockGlobalQueueIndex = atomicAdd(currentQueueSize, sharedCurrentQueueSize);
32          __syncthreads();
33
34          for (int i = threadIdx.x; i < sharedCurrentQueueSize; i += blockDim.x) {
35            currentQueue[blockGlobalQueueIndex + i] = sharedCurrentQueue[i];
36          }
37      }
```

Challenges

- Irregular global memory access: Access patterns depend on graph structure and is unpredictable
- Kernel launch overhead: There is little parallel work in iterations with narrow Queues
- Block-level queue length counter contention: Better than before, but there will still be many serialized atomic operations
- Load imbalance: Vertices can have vastly different numbers of outgoing edges

Highly Irregular Memory Access



Texture memory - quick and dirty

- Texture memory is another form of global memory
- Like constant memory, it is aggressively cached for read-only access
- Originally developed and optimized for storing and reading textures for graphics applications
  - Has hardware-level support for 1-,2-, or 3-D layouts and interpolated reads
  - The texture cache is also spatial layout-aware
- Can be useful for irregular access patterns with un-coalesced reads

Using texture

Declaration:

```
texture<int, 1, cudaReadModeElementType> row_ptrTexture;
```

Host side:

```
int * hrow_ptr;
int row_ptrLength;
cudaArray * texArray = 0;
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<int>();
cudaMallocArray(&texArray, &channelDesc, row_ptrLength);
cudaMemcpyToArray(texArray, 0, 0, hrow_ptr,
row_ptrLength*sizeof(int), cudaMemcpyHostToDevice);
cudaBindTextureToArray(row_ptrTexture, texArray);
```

Device side:

```
for (int i = tex1D(row_ptrTexture,vertex); i < tex1D(row_ptrTexture,vertex+1); ++i)
```

Kernel Launch Overhead

For some iterations of BFS (especially near the beginning), the Queue can be quite small

The benefits of parallelism only outweigh the kernel launch overhead when the Queue becomes large enough

Some options:

Use the CPU if the queue size is below some threshold

Create a single-block variant of the BFS kernel that iterates until its block-level queue is full before returning to the host

Small queue size

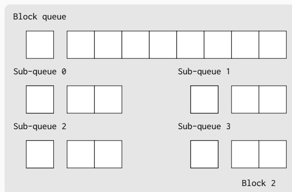
```
// is the most up-to-date Queue information on host or device?
bool currentDataOnDevice = false;
while (hPreviousQueueSize > 0) {
    int numBlocks = (hPreviousQueueSize-1) / BLOCK_SIZE + 1;
    if (numBlocks < NUM_BLOCKS_THRESHOLD) {
      if (currentDataOnDevice) {
      // copy data to host
      ...
      }
      BFS_iterate_sequential(hPreviousQueue, hPreviousQueueSize,
          hCurrentQueue, hCurrentQueueSize, row_ptr, destinations, distances);
      currentDataOnDevice = false;
    } else {
      if (!currentDataOnDevice) {
      // copy data to device
      ...
    }
    BFS_Bqueue_kernel<<<numBlocks, BLOCK_SIZE>>>(dPreviousQueue, dPreviousQueueSize,
                        dCurrentQueue, dCurrentQueueSize,
                        drow_ptr, dDestinations, dDistances, dVisited);
    currentDataOnDevice = true;
  }
}
```

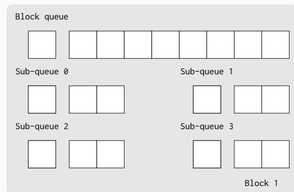
Contention at block level

- While the block-level queues reduced contention for global memory, the block-level counter is now the bottleneck
- We can extend the hierarchy by further splitting the block-level queue

Three-Level Queue Hierarchy

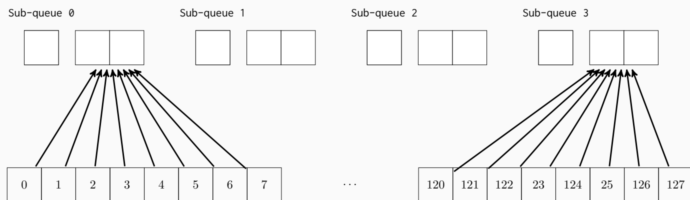
Global queue





Each thread may assign a element based on its index.

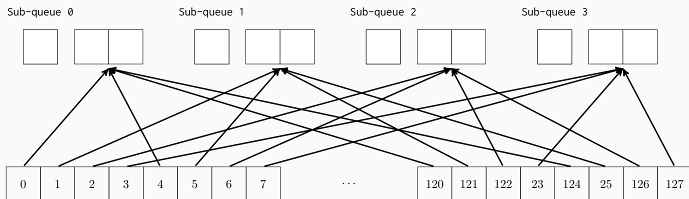
```
subQueueIndex = threadIdx.x / (blockDim.x / NUM_SUB_QUEUES);
```



Threads in the same warp will be using the same queue!

Avoid queue conflicts

```
subQueueIndex = threadIdx.x & (NUM_SUB_QUEUES-1);
```



Dealing with load imbalance

Load imbalance is caused by a data dependency and is thus tricky to avoid

Two potential strategies:

1. Delay the assignment of work to threads until after the total amount of work to be done is known
2. Spawn new threads when needed to account for additional work

Conclusion

- Graphs can be processed in parallel!
- Texture memory can help with large, read-only memory w/ irregular access
- Work queues can be used to track tasks of varying size
- Privatization (and multi-level privatization hierarchies) can be used to reduce contention for work queue insertion