Lecture 12 - GP-GPU and High Performances Computing

Graph traversal computation and parallel algorithms

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 $a_0, a_1, \ldots, a_{n-1}$ which monotonically increases in value, reaches a single maximum, and then monotonically decreases in value.

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

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.

Key properties:

- 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 $a_2 \geq a_3$

Bitonic Sort - 8 Elements

8 element: $(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$

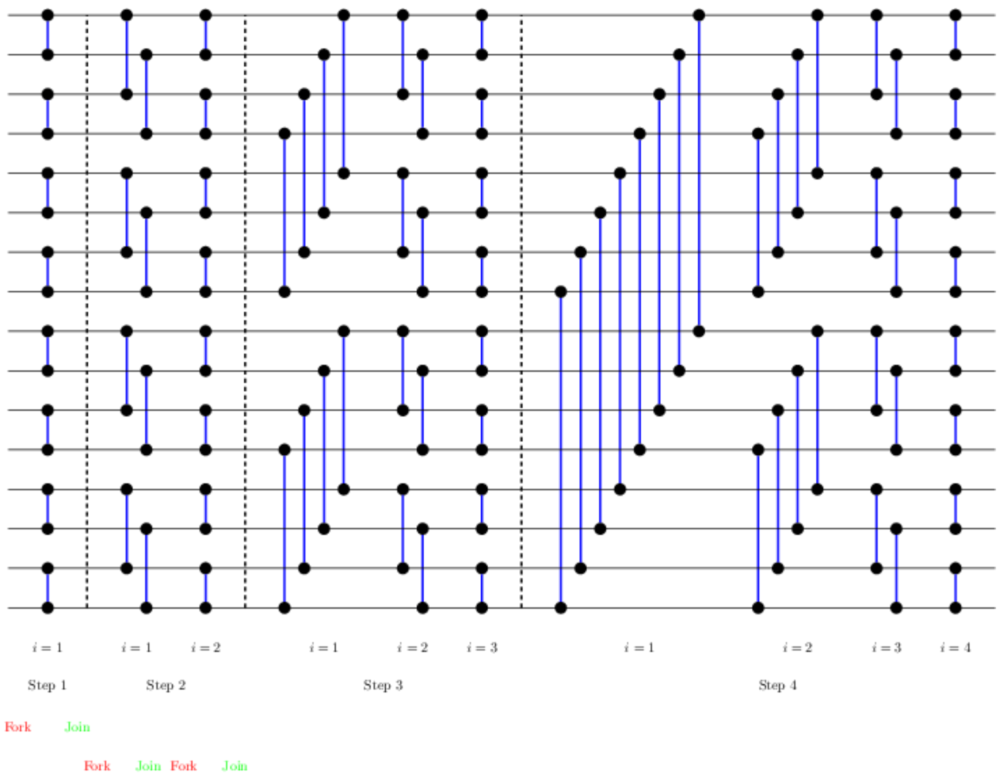
Steps:

1. Make (a_0, a_1, a_2, a_3) and (a_4, a_5, a_6, a_7) bitonic
2. Use bitonic split to make (a_0, a_1, a_2, a_3) increasing
3. Use bitonic split to make (a_4, a_5, a_6, a_7) decreasing

Algorithm:

- 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

Common applications:

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

⚠️ Massive graphs tend to be sparse!

Example - Graph Representation : Adjacency Matrix

	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

Adjacency matrix using CSR

Adjacency matrix stored using CSR (Compressed Sparse Row) 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}
```

CSR Format:

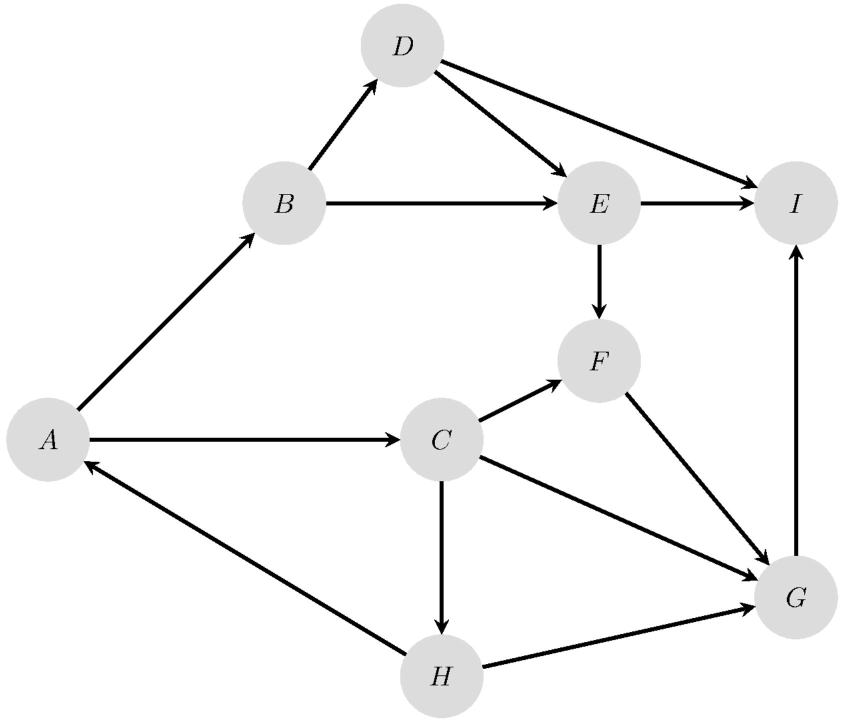
- `AA` : Non-zero values
- `JA` : Column indices
- `IA` : Row pointers

Graph search: BFS

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

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

Example graph



Sequential code

```
void BFS_sequential(int source, const int * row_ptr, const int * dest, int * dist) {
  int queue[2][MAX_QUEUE_SIZE];
  int * currentQueue= &queue[0];
  int * previousQueue = &queue[1];
  int currentQueueSize= 0, previousQueueSize = 0;
  insertIntoQueue(source, previousQueue, &previousQueueSize);
  dist[source] = 0;
  while (previousQueueSize > 0) { // visit all vertices on the previous Queue
    for (int f = 0; f < previousQueueSize; f++) {
      const int currentVertex = previousQueue[f];
      // check all outgoing edges
      for (int i = row_ptr[currentVertex];
           i < row_ptr[currentVertex+1]; ++i) {
        if (dist[dest[i]] == -1) { // this vertex has not been visited yet
          insertIntoQueue(dest[i], currentQueue, &currentQueueSize);
          dist[dest[i]] = dist[currentVertex] + 1;
        }
      }
    }
    swap(currentQueue, previousQueue);
    previousQueueSize = currentQueueSize;
    currentQueueSize = 0;
  }
}
```

Parallel BFS: basic approach

Strategy:

- 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

Issues:

- ❌ Not very work efficient: O(VL) for V = number of vertices, L = length of longest path
- ❌ Difficult to detect stopping criterion

Parallel BFS: Improved approach

Better strategy:

- ✅ 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

```
void BFS_host(int source, const int * row_ptr, const int * dest, int * dist) {
  int dQueue[2][MAX_Queue_SIZE];
  int * dCurrentQueueSize;
  int * dPreviousQueueSize; int hPreviousQueueSize;
  int * dVisited;
  int * dCurrentQueue = &Queue[0];
  int * dPreviousQueue = &Queue[1];
  // allocate device memory, copy memory from device to host,
  // initialize Queue sizes, etc.
  ...
  hPreviousQueueSize = 1;
  while (hPreviousQueueSize > 0) {
    int numBlocks = (hPreviousQueueSize-1) / BLOCK_SIZE + 1;
    BFS_Bqueue_kernel<<<numBlocks, BLOCK_SIZE>>>( dPreviousQueue, dPreviousQueueSize, dCurrentQueue,
      dCurrentQueueSize, drow_ptr, dDestinations, dDistances, dVisited);
    swap(dCurrentQueue,dPreviousQueue);

    cudaMemcpy(dPreviousQueueSize, dCurrentQueueSize, sizeof(int), cudaMemcpyDeviceToDevice);
    cudaMemset(dCurrentQueueSize, 0, sizeof(int));
    cudaMemcpy(&hPreviousQueueSize, dPreviousQueueSize, sizeof(int), cudaMemcpyDeviceToHost);
  }
}
```

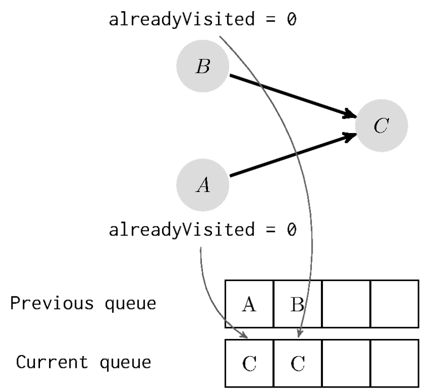
Output Interference

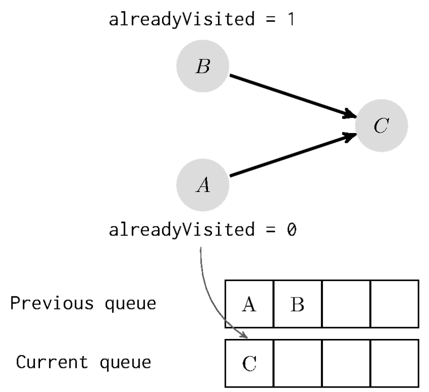
Problem:

- A flag marks whether or not a vertex has been visited
- Output interference on flags can be ignored from a correctness perspective
- BUT: it will lead to additional/replicated work

Solution:

- Use `atomicExch` to handle concurrent access





BFS kernel - Basic

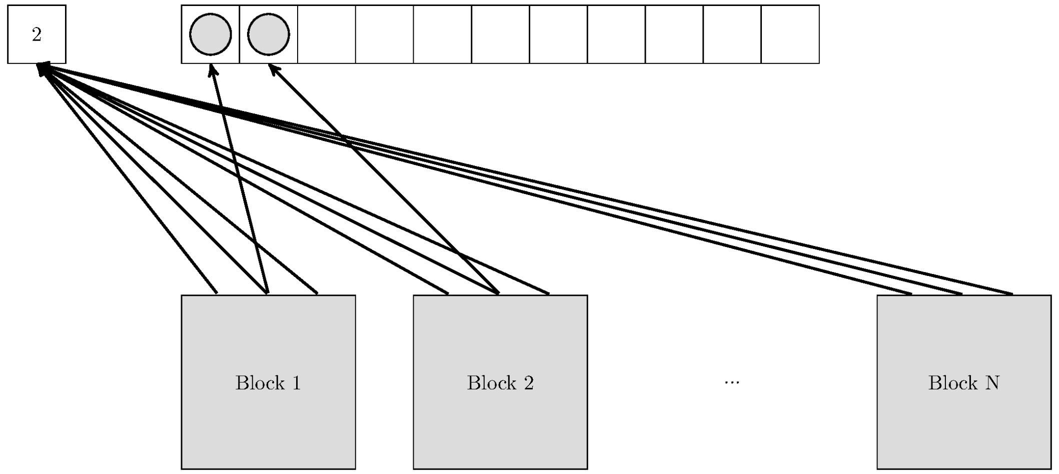
```
__global__ void BFS_Bqueue_kernel(
    const int * previousQueue, const int * previousQueueSize,
    int * currentQueue, int * currentQueueSize, const int * row_ptr,
    const int * destinations, int * distances, int * visited) {
  const int t = threadIdx.x + blockDim.x * blockIdx.x;
  if (t < *previousQueueSize) {
    const int vertex = previousQueue[t];
    for (int i = row_ptr[vertex]; i < row_ptr[vertex+1]; ++i) {
      // check visitation atomically, avoiding redundant expansion
      const int alreadyVisited =
        atomicExch(&(visited[destinations[i]]), 1);
      if (!alreadyVisited) {
        // we're visiting a new vertex: get a spot in line atomically
        const int queueIndex = atomicAdd(currentQueueSize, 1);
        // place the vertex in line
        currentQueue[queueIndex] = destinations[i];
      }
    }
  }
  __syncthreads();
}
```

Interference (2)

Problem: Interference occurs when placing vertices in the queue (line 14 of kernel code)

Solution: Threads need to synchronize with an atomic operation for correct output

Bottleneck: Main bottleneck of the basic kernel



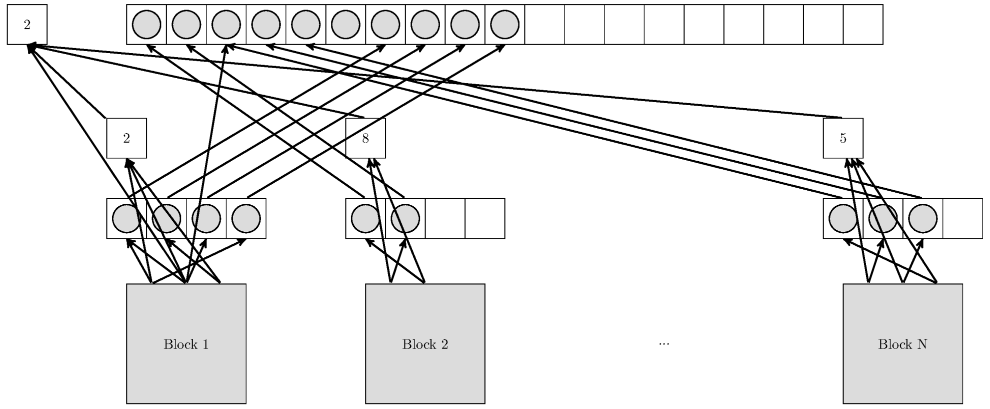
Privatization of the Queue

Strategy:

- ✅ Make a private, block-level copy of the queue
- ✅ Once complete, the private queues are combined to form the global queue

Benefits:

- Reduced contention on global queue
- Better performance through local operations



Kernel with Privatization

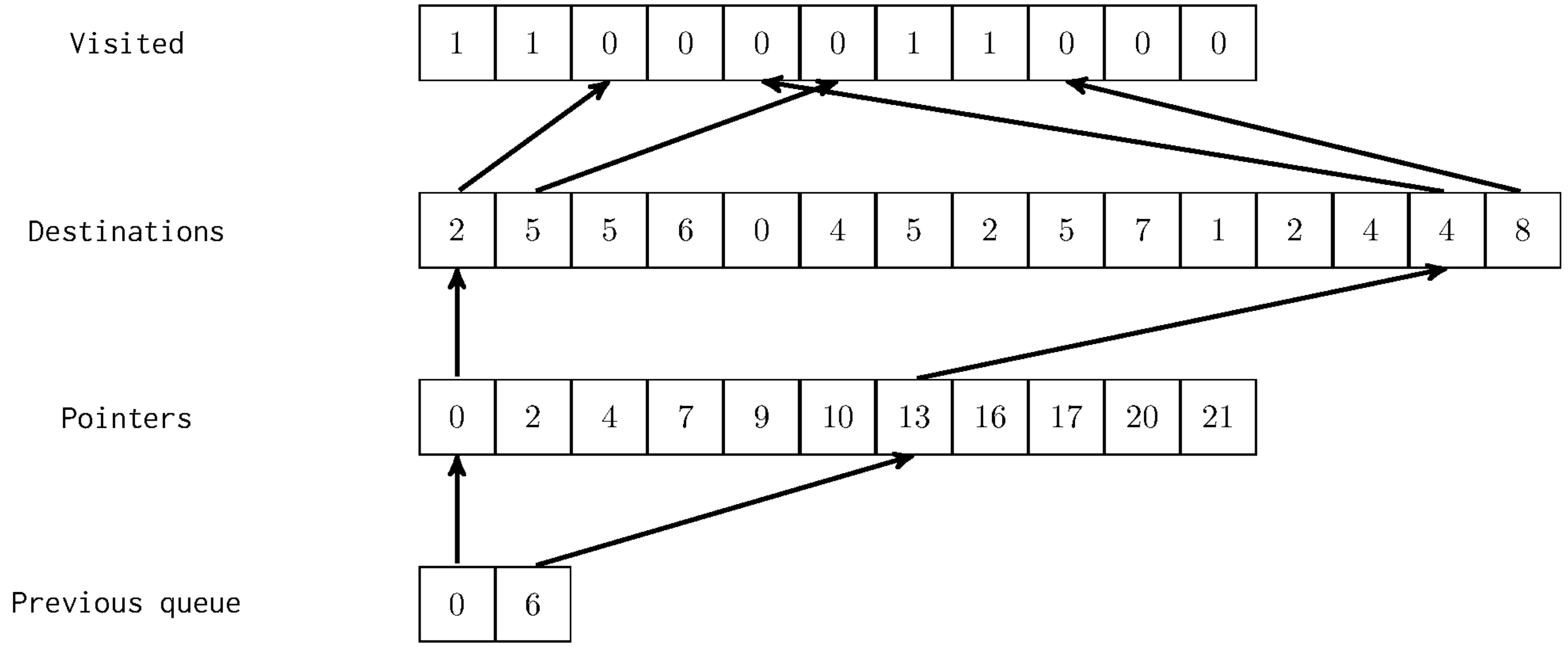
```
__global__ void BFS_Bqueue_kernel(
    const int * previousQueue, const int * previousQueueSize,
    int * currentQueue, int * currentQueueSize, const int * row_ptr,
    const int * destinations, int * distances, int * visited) {
  __shared__ int sharedCurrentQueue[BLOCK_QUEUE_SIZE];
  __shared__ int sharedCurrentQueueSize, blockGlobalQueueIndex;

  if (threadIdx.x == 0) sharedCurrentQueueSize = 0;
  __syncthreads();

  const int t = threadIdx.x + blockDim.x * blockIdx.x;
  if (t < *previousQueueSize) {
    const int vertex = previousQueue[t];
    for (int i = row_ptr[vertex]; i < row_ptr[vertex+1]; ++i) {
      const int alreadyVisited = atomicExch(&(visited[destinations[i]]), 1);
      if (!alreadyVisited) {
        distances[destinations[i]] = distances[i] + 1;
        const int sharedQueueIndex = atomicAdd(&sharedCurrentQueueSize, 1);
        if (sharedQueueIndex < BLOCK_QUEUE_SIZE) {
          // there is space in the local queue
          sharedCurrentQueue[sharedQueueIndex] = destinations[i];
        } else {
          // go directly to the global queue
          sharedCurrentQueueSize = BLOCK_QUEUE_SIZE;
```

Challenges

Remaining challenges:

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

Highly Irregular Memory Access



Texture memory - quick and dirty

What is texture memory?

- 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

Features:

- ✅ Hardware-level support for 1-, 2-, or 3-D layouts and interpolated reads
- ✅ The texture cache is also spatial layout-aware
- ✅ 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

Problem: 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

Solutions:

- 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

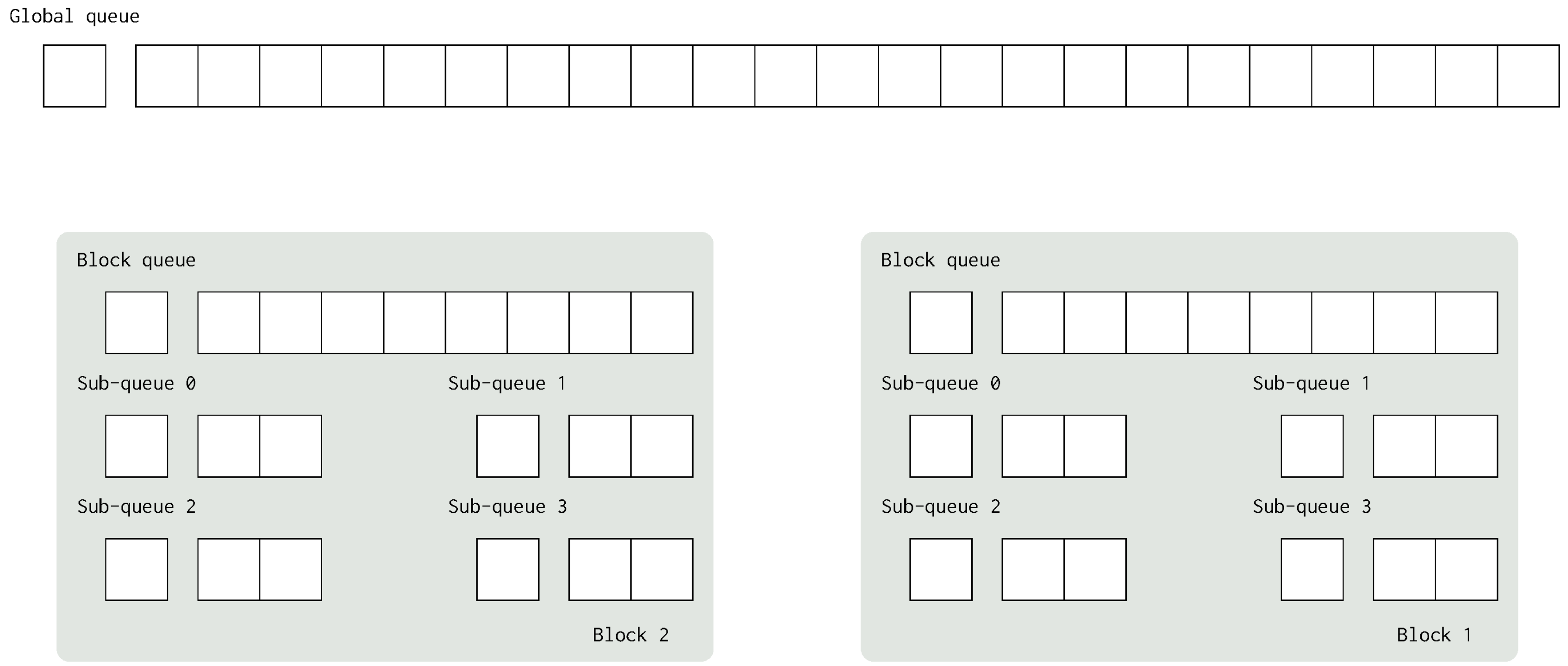
Remaining issue:

- While the block-level queues reduced contention for global memory
- The block-level counter is now the bottleneck

Solution:

- Extend the hierarchy by further splitting the block-level queue
- Create sub-queues within each block

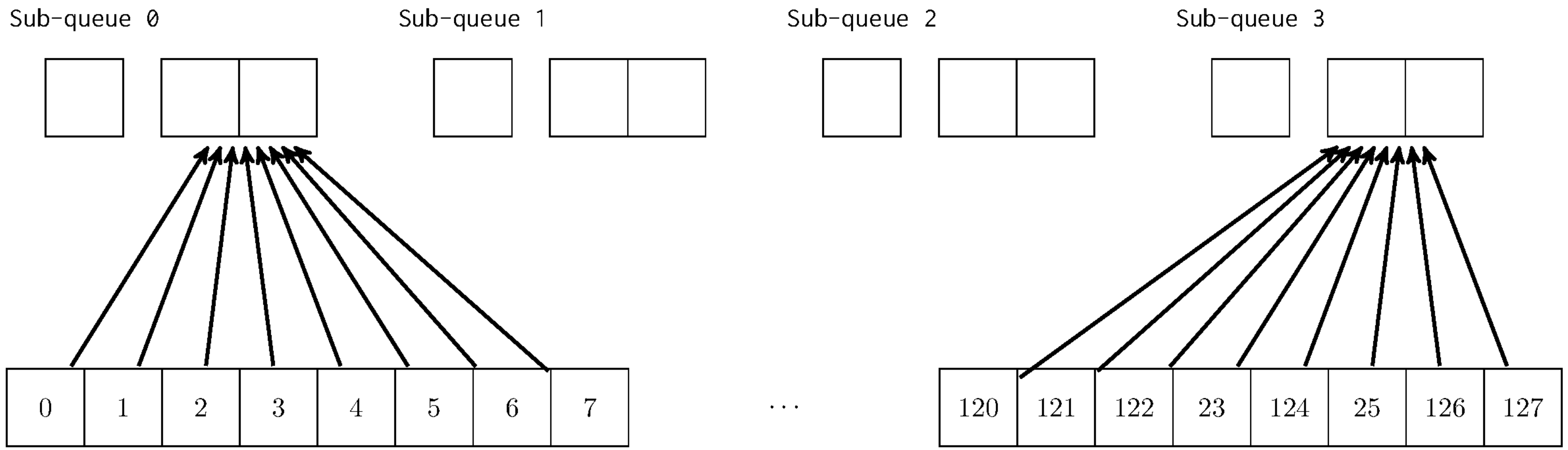
Three-Level Queue Hierarchy



Assignment to subqueues

Simple approach (BAD):

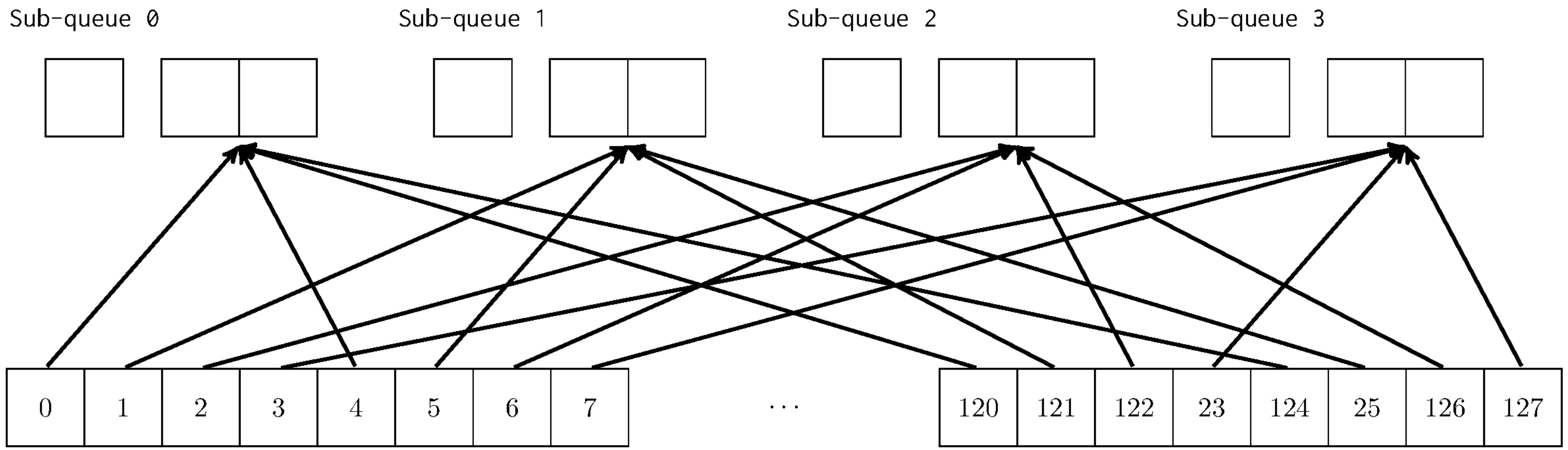
```
subQueueIndex = threadIdx.x / (blockDim.x / NUM_SUB_QUEUES);
```



Avoid queue conflicts

Better approach:

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



Dealing with load imbalance

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

Two potential strategies:

1. **Delay work assignment:** Delay the assignment of work to threads until after the total amount of work to be done is known

2. **Dynamic thread spawning:** Spawn new threads when needed to account for additional work

Conclusion

Key takeaways:

✅ Graphs can be processed in parallel!

✅ Texture memory can help with large, read-only memory with 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

Optimization strategies:

- Multi-level queue hierarchies
- Texture memory for irregular access
- Hybrid CPU/GPU execution
- Smart subqueue assignment