CSE 599 I
Accelerated Computing - Programming GPUs

Parallel Patterns: Graph Search
Objective

- 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
Data Parallelism / Data-Dependent Execution

<table>
<thead>
<tr>
<th>Data Parallel</th>
<th>Data-Independent</th>
<th>Data-Dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stencil</td>
<td></td>
<td>SpMV</td>
</tr>
<tr>
<td>Histogram</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefix Scan</td>
<td></td>
<td>Merge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Graph Search</td>
</tr>
</tbody>
</table>
Massive Graph Applications

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

Massive graphs tend to be sparse!
Graph Review

Graph

Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Graph Review

Graph

Adjacency Matrix (CSR)

Values

Destinations

Row Pointers

1
1
1
1
1
1
1
1
1
1
1

4
1
2
3
4
5
6
7
8
9

15
15
12
13
11
9
7
4
8
5

15
15
13
12
11
9
7
4
8
5

0
1
2
3
4
5
6
7
8
9

0
1
2
3
4
5
6
7
8
9

6
Breadth-First Search

**Problem:**

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 $T$. 

Breadth First Search

Source = 0
Round 0
Breadth First Search

Source = 0
Round 1
Breadth First Search

Source = 0
Round 2
Breadth First Search

Source = 0
Round 3
Breadth First Search

Source = 0
Breadth First Search

Source = 2
Round 0
Breadth First Search

Source = 2
Round 1
Breadth First Search

Source = 2
Round 2
Breadth First Search

Source = 2
Round 3
Breadth First Search

Source = 2
Round 4
Breadth First Search

Source = 2
Sequential BFS

Most computer scientists are familiar with C++ / Java / Python / etc - style BFS implementations using language - provided data structures (e.g. queue)

We’ll look at a C-style implementation to ease the translation into CUDA
Sequential BFS

void BFS_sequential(int source, const int * rowPointers, const int * destinations, int * distances) {

    int frontier[2][MAX_FRONTIER_SIZE];
    int * currentFrontier = &frontier[0];
    int currentFrontierSize = 0;
    int * previousFrontier = &frontier[1];
    int previousFrontierSize = 0;

    insertIntoFrontier(source, previousFrontier, &previousFrontierSize);
    distances[source] = 0;

    while (previousFrontierSize > 0) {
        // visit all vertices on the previous frontier
        for (int f = 0; f < previousFrontierSize; f++) {
            const int currentVertex = previousFrontier[f];
            // check all outgoing edges
            for (int i = rowPointers[currentVertex]; i < rowPointers[currentVertex + 1]; ++i) {
                if (distances[destinations[i]] == -1) {
                    // this vertex has not been visited yet
                    insertIntoFrontier(destinations[i], currentFrontier, &currentFrontierSize);
                    distances[destinations[i]] = distances[currentVertex] + 1;
                }
            }
            swap(currentFrontier, previousFrontier);
            previousFrontierSize = currentFrontierSize;
            currentFrontierSize = 0;
        }
    }
}

In practice, we’d want to check for and handle overflow here
Sequential BFS

```c
void insertIntoFrontier(int vertex, int * frontier, int * frontierSize) {
    frontier[*frontierSize] = vertex;
    ++(*frontierSize);
}
```
One Parallelization 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
A More Work-Efficient Approach

- Parallelize each individual iteration of the while loop in the sequential BFS code
- Assign a section of the vertices in the previous frontier to each thread
- Introduce a synchronization point at the end of each iteration
void BFS_host(int source, const int * rowPointers, const int * destinations, int * distances) {

    int dFrontier[2][MAX_FRONTIER_SIZE];
    int * dCurrentFrontierSize;
    int * dPreviousFrontierSize; int hPreviousFrontierSize;
    int * dVisited;

    int * dCurrentFrontier = &frontier[0];
    int * dPreviousFrontier = &frontier[1];

    // allocate device memory, copy memory from device to host, initialize frontier sizes, etc.
    ...

    hPreviousFrontierSize = 1;

    while (hPreviousFrontierSize > 0) {
        int numBlocks = (hPreviousFrontierSize-1) / BLOCK_SIZE + 1;

        BFS_Bqueue_kernel<<<numBlocks, BLOCK_SIZE>>>(dPreviousFrontier, dPreviousFrontierSize, 
                                             dCurrentFrontier, dCurrentFrontierSize, 
                                             dRowPointers, dDestinations, dDistances, dVisited);

        swap(dCurrentFrontier,dPreviousFrontier);
        cudaMemcpy(dPreviousFrontierSize, dCurrentFrontierSize, sizeof(int), cudaMemcpyDeviceToDevice);
        cudaMemcpy(&hPreviousFrontierSize, dPreviousFrontierSize, sizeof(int), cudaMemcpyDeviceToHost);

    }
}

Output Interference in BFS

- We’ll use flags to mark whether or not a vertex has been visited.
- From a correctness perspective, output interference on flags can be ignored.
- However, this will lead to additional work.

**Without synchronization:**

- Previous queue: A, B
- Current queue: C, C
- `alreadyVisited? 0`

**With atomicExch:**

- Previous queue: A, B
- Current queue: C
- `alreadyVisited? 1`
Basic Parallel BFS Kernel

```c
__global__ void BFS_Bqueue_kernel(const int *previousFrontier, const int *previousFrontierSize,
                                  int *currentFrontier, int *currentFrontierSize,
                                  const int *rowPointers,
                                  const int *destinations, int *distances, int *visited) {

    const int t = threadIdx.x + blockDim.x * blockIdx.x;
    if (t < *previousFrontierSize) {

        const int vertex = previousFrontier[t];
        for (int i = rowPointers[vertex]; i < rowPointers[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(currentFrontierSize, 1);

                // place the vertex in line
                currentFrontier[queueIndex] = destinations[i];

            }
        }
    }

    __syncthreads();
}
```
Output Interference in Frontier Queue

- There is also output interference in placing vertices in the queue
- Synchronization is strictly required here for correct output
- This is a bottleneck of the basic kernel
Privatization of the Frontier Queue

- We can make a private, block-level copy of the frontier queue
- Once complete, the private queues are combined to form the global queue
Parallel BFS Kernel with Privatization

```c
__global__ void BFS_Bqueue_kernel(const int * previousFrontier, const int * previousFrontierSize,
                                   int * currentFrontier, int * currentFrontierSize, const int * rowPointers,
                                   const int * destinations, int * distances, int * visited) {

  __shared__ int sharedCurrentFrontier[BLOCK_QUEUE_SIZE];
  __shared__ int sharedCurrentFrontierSize, blockGlobalQueueIndex;

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

  const int t = threadIdx.x + blockDim.x * blockIdx.x;
  if (t < *previousFrontierSize) {
    const int vertex = previousFrontier[t];
    for (int i = rowPointers[vertex]; i < rowPointers[vertex+1]; ++i) {
      const int alreadyVisited = atomicExch(&(visited[destinations[i]]),1);
      if (!alreadyVisited) {
        distances[destinations[i]] = distances[i] + 1;
        const int sharedQueueIndex = atomicAdd(&sharedCurrentFrontierSize, 1);
        if (sharedQueueIndex < BLOCK_QUEUE_SIZE) {
          // there is space in the local queue
          sharedCurrentFrontier[sharedQueueIndex] = destinations[i];
        } else {
          // go directly to the global queue
          sharedCurrentFrontierSize = BLOCK_QUEUE_SIZE;
          const int globalQueueIndex = atomicAdd(currentFrontierSize, 1);
          currentFrontier[globalQueueIndex] = destinations[i];
        }
      }
    }
  }
  __syncthreads();

  if (threadIdx.x == 0) blockGlobalQueueIndex = atomicAdd(currentFrontierSize, sharedCurrentFrontierSize);
  __syncthreads();

  for (int i = threadIdx.x; i < sharedCurrentFrontierSize; i += blockDim.x) {
    currentFrontier[blockGlobalQueueIndex + i] = sharedCurrentFrontier[i];
  }
}
```
Remaining Issues

● 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 frontiers

● 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
BFS Results in Highly Irregular Memory Access

visited

destinations

rowPointers

previousFrontier

- visited: `1 1 0 0 0 0 1 1 0 0 0 ...`
- destinations: `2 5 5 6 0 4 5 2 5 7 1 2 4 4 8 ...`
- rowPointers: `0 2 4 7 9 10 13 16 17 20 21 ...`
- previousFrontier: `0 6`
Global Memory Bandwidth Limitations Got You Down?

- Is the usage pattern known, local, and not more than ~48KB? 
  - Yes: Shared memory!
  - No:
    - Is read-only memory OK? 
      - Yes: Is it less than ~64KB? 
        - Yes: Constant memory!
        - No: Texture memory!
      - No: Hope you’ve been nice to your L2 cache...
Texture Memory

- 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 Memory

Declaration:

```cpp
texture<int, 1, cudaReadModeElementType> rowPointersTexture;
```

Host side:

```cpp
int * hRowPointers;
int rowPointersLength;
cudaArray * texArray = 0;

cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<int>();

cudaMallocArray(&texArray, &channelDesc, rowPointersLength);

cudaMemcpyToArray(texArray, 0, 0, hRowPointers, rowPointersLength*sizeof(int), cudaMemcpyHostToDevice);

cudaBindTextureToArray(rowPointersTexture, texArray);
```

Device side:

```cpp
for (int i = tex1D(rowPointersTexture,vertex); i < tex1D(rowPointersTexture,vertex+1); ++i)  ...
```
Remaining Issues

- **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 frontiers
- **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
Kernel Launch Overhead

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

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

Some options:

1. Use the CPU if the frontier size dips below some threshold.
2. Create a single-block variant of the BFS kernel that iterates until its block-level queue is full before returning to the host.
Using the CPU for Small Frontiers

// is the most up-to-date frontier information on host or device?
bool currentDataOnDevice = false;

while (hPreviousFrontierSize > 0) {
    int numBlocks = (hPreviousFrontierSize-1) / BLOCK_SIZE + 1;

    if (numBlocks < NUM_BLOCKS_THRESHOLD) {
        if (currentDataOnDevice) {
            // copy data to host

            BFS_iterate_sequential(hPrevious Frontier, hPreviousFrontierSize,
                                   hCurrentFrontier, hCurrentFrontierSize,
                                   rowPointers, destinations, distances);
            currentDataOnDevice = false;
        }
    } else {
        if (!currentDataOnDevice) {
            // copy data to device

            BFS_Bqueue_kernel<<<numBlocks, BLOCK_SIZE>>>(dPreviousFrontier, dPreviousFrontierSize,
                                                        dCurrentFrontier, dCurrentFrontierSize,
                                                        dRowPointers, dDestinations, dDistances, dVisited);
            currentDataOnDevice = true;
        }
    }

    ...
Remaining Issues

✔ 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 frontiers

● 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
Block-Level Queue Contention

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:

Block queue:

Block 0

Sub-queue 0:

Sub-queue 1:

Sub-queue 2:

Sub-queue 3:

Block 1

Block queue:

Block 0

Sub-queue 0:

Sub-queue 1:

Sub-queue 2:

Sub-queue 3:
Sub-Queue Assignment

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

This mapping means threads in the same warp will be using the same queue!
Sub-Queue Assignment

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

Shortcut if the number of queues is a power of 2

Much better!
Remaining Issues

- 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 frontiers

- 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
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.

Tune in to the next lecture on CUDA dynamic parallelism!

This could result in additional code complexity, higher register usage, and/or more synchronization.
Conclusion / 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
Sources

https://www.wikipedia.org/

