CSE 4373/5373 - General Purpose GPU Programming

Parallel Graph Traversal

Alex Dillhoff

University of Texas at Arlington

Parallelization over Vertices

Parallelization over vertices

Parallel graph algorithms can operate on either the edges or the vertices.

In either case, such algorithms will require communication between threads due to dependencies in the graph or traversal algorithm.

Parallelization over vertices

In a breadth-first search, all vertices in one level must be explored before moving into the next.

This would require multiple kernel calls or some form of coarsening.

The first approach we look at will require multiple calls to the kernel, one for each level.

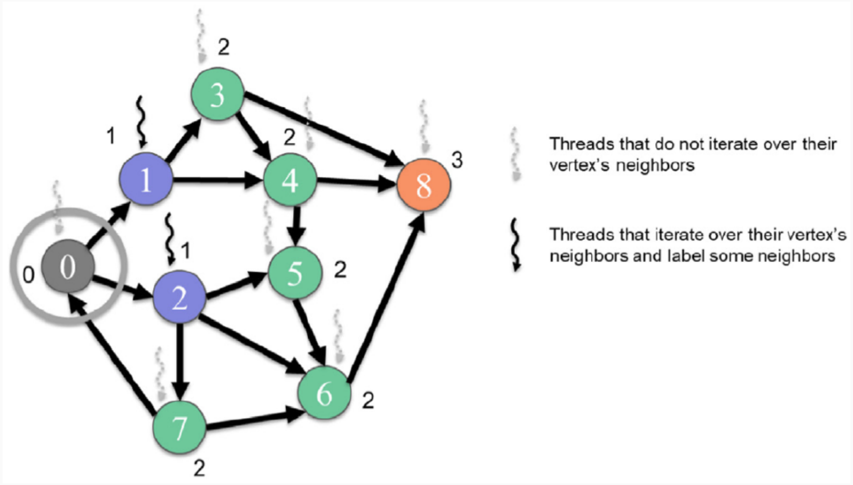
Top-Down Approach

```
__global__ void bfs_kernel(CSRGraph csrGraph, uint *level,
                           uint *newVertexVisited, uint *currLevel) {
    int vertex = blockIdx.x * blockDim.x + threadIdx.x;
    if (vertex < csrGraph.numVertices) {
        if (level[vertex] == currLevel - 1) {
            for (uint edge = csrGraph.srcPtrs[vertex];
                 edge < csrGraph.srcPtrs[vertex + 1]; edge++) {
                int neighbor = csrGraph.edges[edge];
                if (level[neighbor] == UINT_MAX) {  // Neighbor not visited
                    level[neighbor] = currLevel;
                    *newVertexVisited = 1;
                }
            }
        }
    }
}
```

Top-Down Approach

- Labels the vertices that belong on the given level.
- For each vertex, it iterates over the outgoing edges.
- These can be accessed as the nonzero elements in the adjacency matrix for each row.
- The CSR format is ideal for this case.

Top-Down Approach


0
1
2
3
4
5
6
7
8
0
1
1
2
2
2
2
2
2
3
Threads that do not iterate over their vertex's neighbors
Threads that iterate over their vertex's neighbors and label some neighbors

Breadth-first search top-down approach, iteration 1.

Top-Down Approach

- The boundary check ensures that the kernel only processes vertices that belong to the current level.
- Each thread has access to a global `newVertexVisited` and will set this to 1 if it finds a new vertex to visit.
- This is used to determine if the traversal should continue to the next level.

Bottom-Up Approach

The second kernel is also a vertex-centric approach, except it considers incoming edges rather than outgoing ones.

This is called the *pull* version.

Bottom-Up Approach

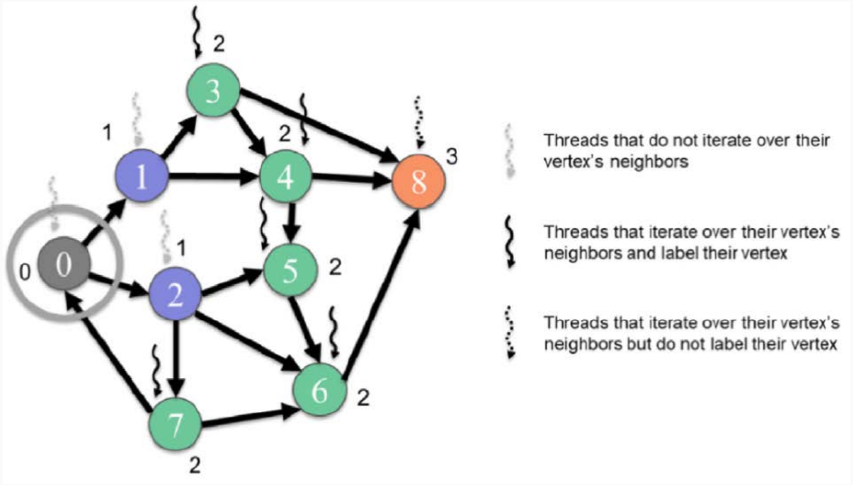
```
__global__ void bfs_kernel(CSCGraph cscGraph, uint *level,
                           uint *newVertexVisited, uint *currLevel) {
    int vertex = blockIdx.x * blockDim.x + threadIdx.x;
    if (vertex < cscGraph.numVertices) {
        if (level[vertex] == UINT_MAX) {
            for (uint edge = cscGraph.dstPtrs[vertex];
                 edge < cscGraph.dstPtrs[vertex + 1]; edge++) {
                int neighbor = cscGraph.edges[edge];
                if (level[neighbor] == currLevel - 1) {  // Neighbor visited
                    level[vertex] = currLevel;
                    *newVertexVisited = 1;
                    break;
                }
            }
        }
```

Bottom-Up Approach

Note the use of the Compressed Sparse Column (CSC) format for the graph.

The kernel requires that each thread be able to access the incoming edges, which would be determined by the nonzero elements of a give column of the adjacency matrix.

Bottom-Up Approach


0
1
2
3
4
5
6
7
8
Threads that do not iterate over their vertex's neighbors
Threads that iterate over their vertex's neighbors and label their vertex
Threads that iterate over their vertex's neighbors but do not label their vertex

Bottom-up traversal pulls from the visited vertices.

Bottom-Up Approach

- If there is an incoming edge at the previous level then it will be visited at the current level.
- In that case its job is done and it can break out of the loop.
- This approach is more efficient for graphs with a high average degree and variance.

Bottom-Up Approach

- *Push* is more efficient in early levels: smaller number of vertices per level.
- *Pull* is more efficient in later levels: higher chance of exiting early.
- Each level is a separate kernel call $\implies$ simple to combine.
- Combined solution requires both CSR and CSC representations of the graph.

Parallelization over Edges

Parallelization over Edges

As the name suggests, the edge-centric approach processes the edges in parallel.

If the source vertex belongs to a previous level and the destination vertex is unvisited, then the destination vertex is labeled with the current level.

Parallelization over Edges

```
__global__ void bfs_kernel(COOGraph cooGraph, uint *level,
                           uint *newVertexVisited, uint *currLevel) {
    int edge = blockIdx.x * blockDim.x + threadIdx.x;

    if (edge < csrGraph.numEdges) {
        uint vertex = cooGraph.src[edge];
        if (level[vertex] == currLevel - 1) {
            uint neighbor = cooGraph.dst[edge];
            if (level[neighbor] == UINT_MAX) {
                level[neighbor] = currLevel;
                *newVertexVisited = 1;
            }
        }
    }
}
```

Parallelization over Edges

- The edge-centric approach has more parallelism than the vertex-centric approaches.
- Graphs typically have more edges than vertices, so the largest benefit is on smaller graphs.
- Another advantage related to load imbalance: some vertices have more edges than others.

Parallelization over Edges

Tradeoffs

- There may be many edges that are not relevant for a particular level.
- The vertex-centric approach could skip these entirely.
- Since every edge needs to be indexed, the COO format is used.
- Requires more space than the CSR or CSC formats.

Parallelization over Edges

Since these sparse representations are already used, we could perform these operations using sparse matrix multiplications.

Libraries such as cugraph provide implementations of these algorithms.

Improving Work Efficiency

Improving Work Efficiency

The previous two approaches have a common problem: it is likely that many threads will perform no useful work.

Vertex-centric

- The threads that are launched only to find out that their vertex is not in the current level will not do anything.

Improving Work Efficiency

The previous two approaches have a common problem: it is likely that many threads will perform no useful work.

Vertex-centric

- The threads that are launched only to find out that their vertex is not in the current level will not do anything.
- Those threads should not be launched in the first place.

Improving Work Efficiency

The previous two approaches have a common problem: it is likely that many threads will perform no useful work.

Vertex-centric

- The threads that are launched only to find out that their vertex is not in the current level will not do anything.
- Those threads should not be launched in the first place.
- **Solution:** have each thread build a **frontier** of vertices that they visit, so that only the vertices in the frontier are processed.

Improving Work Efficiency

```
__global__ void bfs_kernel(CSRGraph csrGraph, uint *level,
                           uint *prevFrontier, uint *currFrontier,
                           uint numPrevFrontier, uint *numCurrFrontier,
                           uint currLevel) {
    uint i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < numPrevFrontier) {
        uint vertex = prevFrontier[i];
        for (uint edge = csrGraph.srcPtrs[vertex];
             edge < csrGraph.srcPtrs[vertex + 1]; edge++) {
            uint neighbor = csrGraph.dst[edge];
            if (atomicCAS(&level[neighbor], UINT_MAX, currLevel) == UINT_MAX) {
                uint currFrontierIdx = atomicAdd(numCurrFrontier, 1);
                currFrontier[currFrontierIdx] = neighbor;
            }
        }
```

Improving Work Efficiency

- When launched, only the threads corresponding to a frontier will be active.

Improving Work Efficiency

- When launched, only the threads corresponding to a frontier will be active.
- Start by loading the elements from the previous frontier.

Improving Work Efficiency

- When launched, only the threads corresponding to a frontier will be active.
- Start by loading the elements from the previous frontier.
- Only iterate over the outgoing edges.

Improving Work Efficiency

- When launched, only the threads corresponding to a frontier will be active.
- Start by loading the elements from the previous frontier.
- Only iterate over the outgoing edges.
- If the neighbor has not been visited, then it is labeled with the current level and added to the current frontier.

Improving Work Efficiency

- When launched, only the threads corresponding to a frontier will be active.
- Start by loading the elements from the previous frontier.
- Only iterate over the outgoing edges.
- If the neighbor has not been visited, then it is labeled with the current level and added to the current frontier.
- The atomic operation ensures that the size of the frontier is updated correctly.

Improving Work Efficiency

The call to `atomicCAS` prevents multiple threads from adding the same vertex to the frontier.

It checks whether the current vertex is unvisited.

Not every thread is going to visit the same neighbor, so the contention should be low for this call.

Reducing Contention

Reducing Contention

The use of atomic operations in the previous example introduces contention between threads.

Reducing Contention

The use of atomic operations in the previous example introduces contention between threads.

Privatization can be applied in these cases to reduce that contention.

- Each block will have its own private frontier.
- Contention is reduced to atomic operations within a block.
- The local frontier is in shared memory, resulting in lower latency atomic operations.

Reducing Contention

```
__global__ void bfs_kernel(CSRGraph csrGraph, uint *level,
                           uint *prevFrontier, uint *currFrontier,
                           uint numPrevFrontier, uint *numCurrFrontier,
                           uint currLevel) {
    // Initialize privatized frontier
    __shared__ uint currFrontier_s[LOCAL_FRONTIER_CAPACITY];
    __shared__ uint numCurrFrontier_s;

    if (threadIdx.x == 0) {
        numCurrFrontier_s = 0;
    }
    __syncthreads();

    uint i = blockIdx.x * blockDim.x + threadIdx.x;
```

Reducing Contention

```
if (i < numPrevFrontier) {
    uint vertex = prevFrontier[i];
    for (uint edge = csrGraph.srcPtrs[vertex]; edge < csrGraph.srcPtrs[vertex
        uint neighbor = csrGraph.dst[edge];
        if (atomicCAS(&level[neighbor], UINT_MAX, currLevel) == UINT_MAX) {
            uint currFrontierIdx_s = atomicAdd(&numCurrFrontier_s, 1);
            if (currFrontierIdx_s < LOCAL_FRONTIER_CAPACITY) {
                currFrontier_s[currFrontierIdx_s] = neighbor;
            } else {
                numCurrFrontier_s = LOCAL_FRONTIER_CAPACITY;
                uint currFrontierIdx = atomicAdd(numCurrFrontier, 1);
                currFrontier[currFrontierIdx] = neighbor;
            }
        }
    }
```

Reducing Contention

```
    __syncthreads();
    // Allocate in global frontier
    __shared__ uint currFrontierStartIdx;
    if (threadIdx.x == 0) {
        currFrontierStartIdx = atomicAdd(numCurrFrontier, numCurrFrontier_s);
    }
    __syncthreads();
    // Commit to global frontier
    for (uint currFrontierIdx_s = threadIdx.x;
         currFrontierIdx_s < numCurrFrontier_s; currFrontierIdx_s += blockDim.x)
        currFrontier[currFrontierStartIdx + currFrontierIdx_s] =
            currFrontier_s[currFrontierIdx_s];
    }
}
```

Reducing Contention

The main BFS block of the kernel above will write to the local frontier as long as there is space.

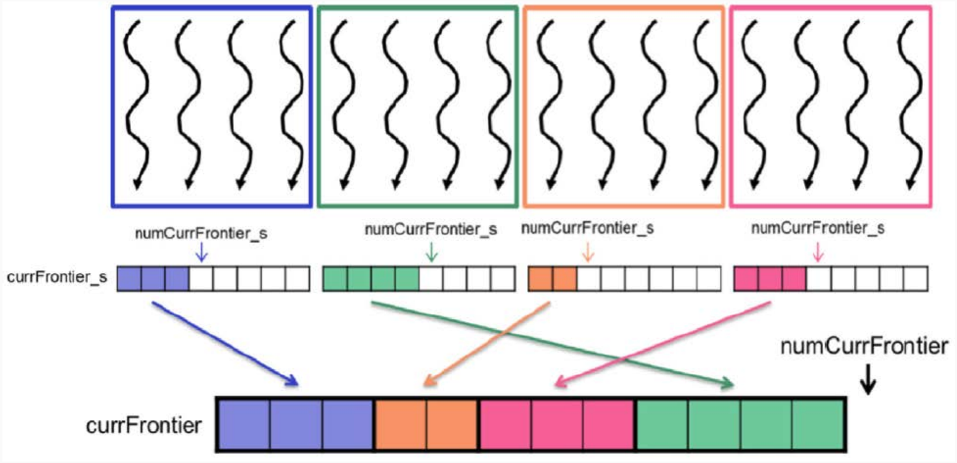
If the capacity is hit, all future writes will go to the global frontier.

Reducing Contention

After BFS completes, a representative thread (index 0) from each block will allocate space in the global frontier, giving it a unique starting index.

This allows each block to safely write to the global frontier without contention.

Reducing Contention



Privatization of the frontier.

Additional Optimizations

Reducing launch overhead

- If the frontiers of a BFS are small, the overhead of launching a kernel for each level can be significant.

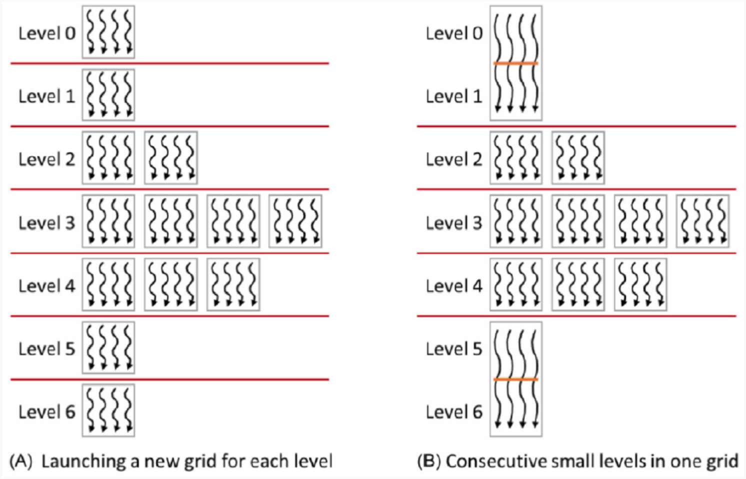
Reducing launch overhead

- If the frontiers of a BFS are small, the overhead of launching a kernel for each level can be significant.
- In such cases, a kernel with a grid size of 1 can be launched to handle multiple levels.

Reducing launch overhead

- If the frontiers of a BFS are small, the overhead of launching a kernel for each level can be significant.
- In such cases, a kernel with a grid size of 1 can be launched to handle multiple levels.
- This block would synchronize after each level to ensure that all threads have completed the current level before moving on to the next.

Reducing launch overhead



Kernel with grid size of 1 for multiple levels.

Improving load balance

In the first vertex-centric approach we look at, the threads were not evenly balanced due to the fact that some vertices have more edges than others.

For graphs that have high variability in the number of edges per vertex, the frontier can be sorted and placed into multiple buckets.

The buckets would be processed by a separate kernel.