Programming Massively Parallel Processors

Chapter15 Graph traversal

242AIG18 JiYeong Yi

Chapter15

Graph traversal

Table of contents

14^{+} Results

15.1 Background

15.2 Breadth-first search

15.3 Vertex-centric parallelization of breadth-first search

15.4 Edge-centric parallelization of breadth-first search

15.5 Improving efficiency with frontiers

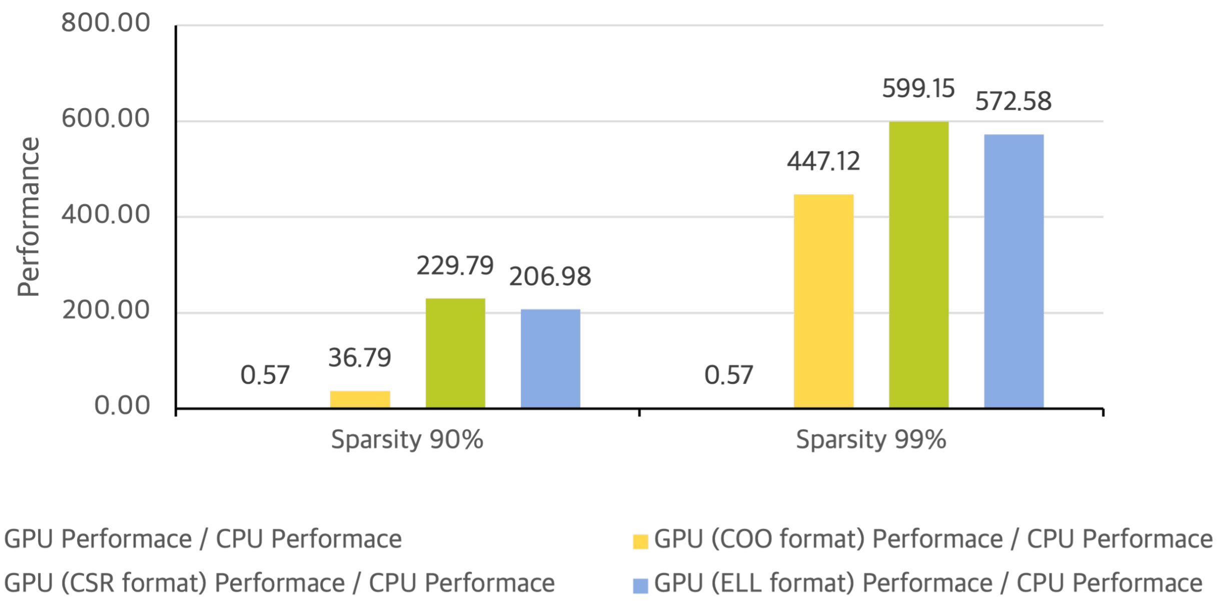
15.6 Reducing contention with privatization

15.7 Other optimizations

15.8 Summary

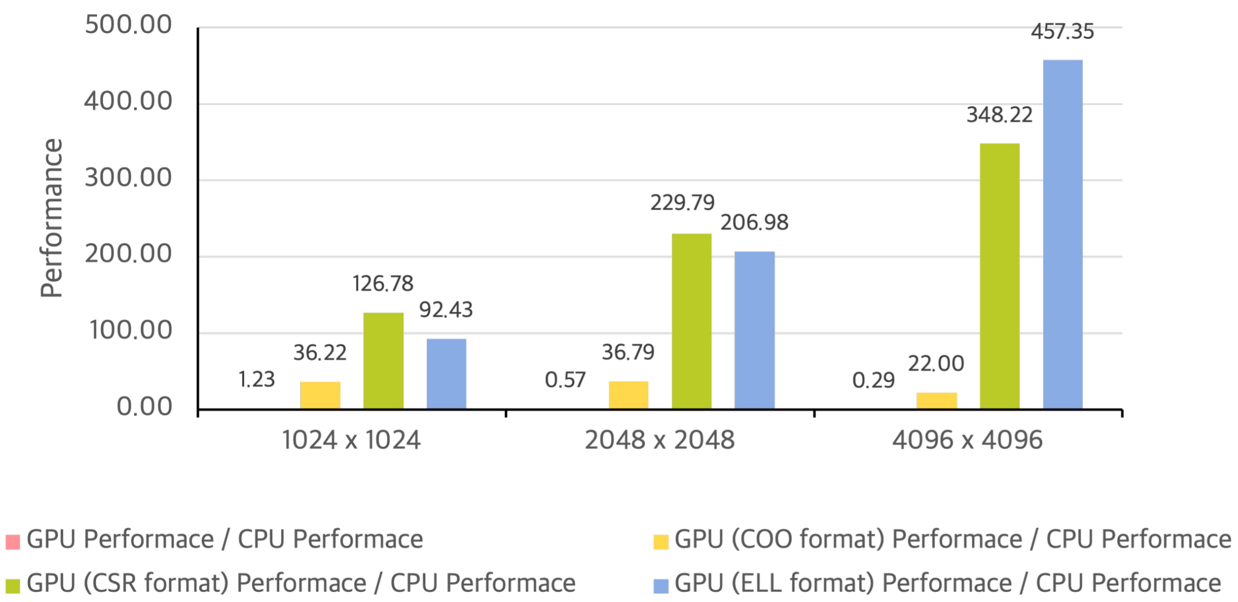
14+ results

		CPU Execution Time	GPU Execution Time	GPU (COO format) Execution Time	GPU (CSR format) Execution Time	GPU (ELL format) Execution Time
Matrix Size 2048 x 2048	Sparsity 90%	5.020 (ms)	8.872 (ms)	136.445 (us)	21.846 (us)	24.254 (us)
	Sparsity 99%	5.061 (ms)	8.872 (ms)	11.319 (us)	8.447 (us)	8.839 (us)



14+ results

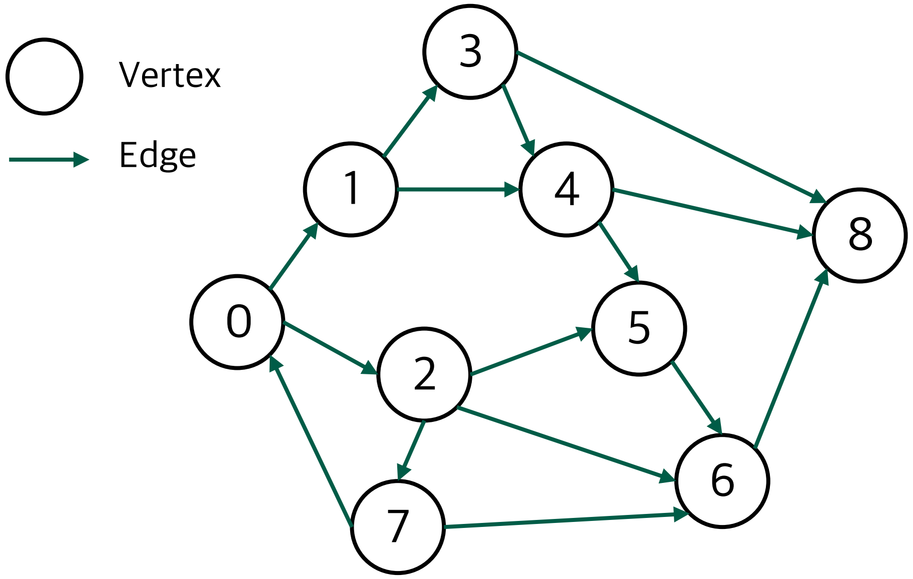
		CPU Execution Time	GPU Execution Time	GPU (COO format) Execution Time	GPU (CSR format) Execution Time	GPU (ELL format) Execution Time
Sparsity 90%	1024 x 1024	1.390 (ms)	1.133 (ms)	38.375 (us)	10.964 (us)	15.038 (us)
	2048 x 2048	5.020 (ms)	8.872 (ms)	136.445 (us)	21.846 (us)	24.254 (us)
	4096 x 4096	20.371 (ms)	70.466 (ms)	925.923 (us)	58.500 (us)	44.541 (us)



15.1 Background

- A graph is a data structure that represents the relationships between entities. The entities involved are represented as **vertices**, and the relations are represented as **edges**.

- An intuitive representation of a graph is an **adjacency matrix**. If there is an edge going from a source vertex i to a destination vertex j, the value of element A[i][j] of the adjacency matrix is 1.



[Directed graph]

destination vertex

source vertex

	0	1	2	3	4	5	6	7	8
0	0	1	1	0	0	0	0	0	0
1	0	0	0	1	1	0	0	0	0
2	0	0	0	0	0	1	1	1	0
3	0	0	0	0	1	0	0	0	1
4	0	0	0	0	0	1	0	0	1
5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	0	0	0	0	1
7	1	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0

[Adjacency matrix]

15.1 Background

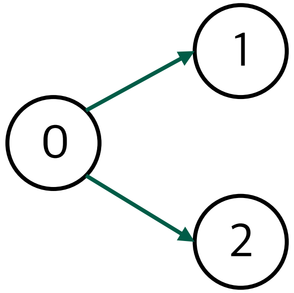
- Sparsely connected graphs can probably benefit from a sparse matrix representation. Using a compressed representation of the matrix can drastically reduce the amount of storage required and the number of wasted operations on the zero elements.

- Three different storage format:
  1. Compressed sparse row (CSR)
  2. Compressed sparse column (CSC)
  3. Coordinate (COO)

15.1 Background

- Compressed sparse row (CSR):

  The CSR representations give easy access to the **outgoing edges of a given vertex**.



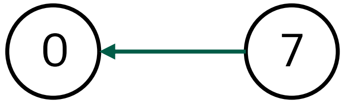
row pointers srcPtrs[10]	0	2	4	7	9	11	12	13	15	15					
column indices dst[15]	1	2	3	4	5	6	7	4	8	5	8	6	8	0	6
nonzero elements data[15]	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

[CSR representation]

15.1 Background

- Compressed sparse column (CSC):

  The CSC representation gives easy access to the **incoming edges of a given vertex**.



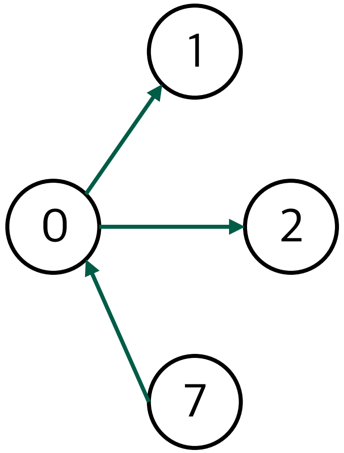
column pointers dstPtrs[10]	0	1	2	3	4	6	8	11	12	15					
row indices src[15]	7	0	0	1	1	3	2	4	2	5	7	2	3	4	6
nonzero elements data[15]	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

[CSC representation]

15.1 Background

- Coordinate (COO):

  The COO representation gives easy access to the **source and destination vertices of a given edge**.



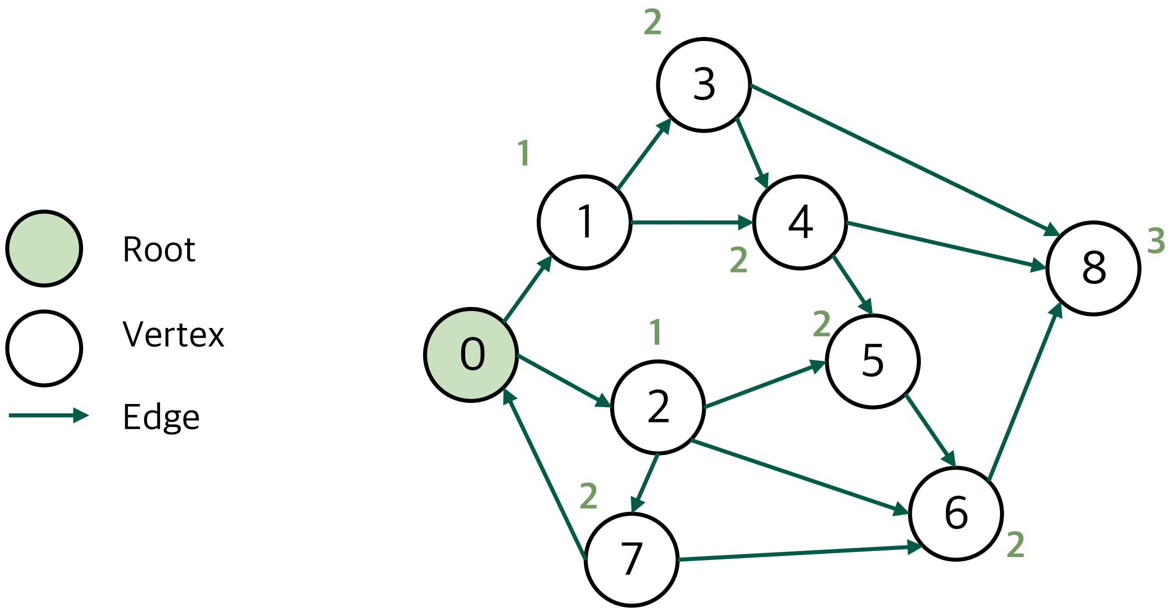
row indices src[15]	0	0	1	1	2	2	2	3	3	4	4	5	6	7	7
column indices dst[15]	1	2	3	4	5	6	7	4	8	5	8	6	8	0	6
nonzero elements data[15]	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

[COO representation]

15.2 Breadth-first search

- BFS is often used to discover the shortest number of edges that one needs to traverse to go from one vertex to another vertex of the graph.

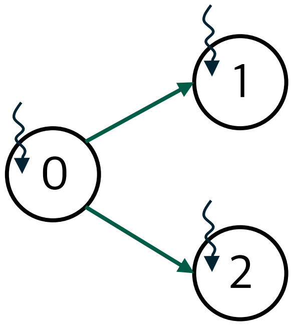
- There are different ways of summarizing the outcome of a BFS traversal. One way is, given a vertex that is referred to as the root, to label each vertex with the smallest number of edges that one needs to traverse to go from the root to that vertex.



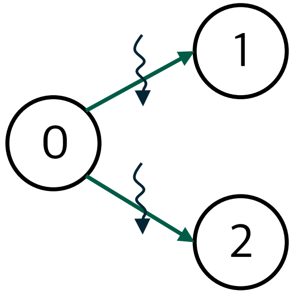
[BFS results]

15.3 Vertex-centric parallelization of breadth-first search

- In fact, many parallel implementations of graph algorithms can be classified as vertex-centric or edge-centric.
- A **vertex-centric** parallel implementation assigns threads to vertices and has each thread perform an operation on its vertex, which usually involves iterating over the neighbors of that vertex.
- An **edge-centric** parallel implementation assigns threads to edges and has each thread perform an operation on its edge, which usually involves looking up the source and destination vertices of that edge.



[Vertex-centric parallelization]



[Edge-centric parallelization]

15.3 Vertex-centric parallelization of breadth-first search

- **vertex-centric push (top-down)** parallel implementation of BFS assigns each thread to a vertex to iterate over the vertex's outgoing edges.

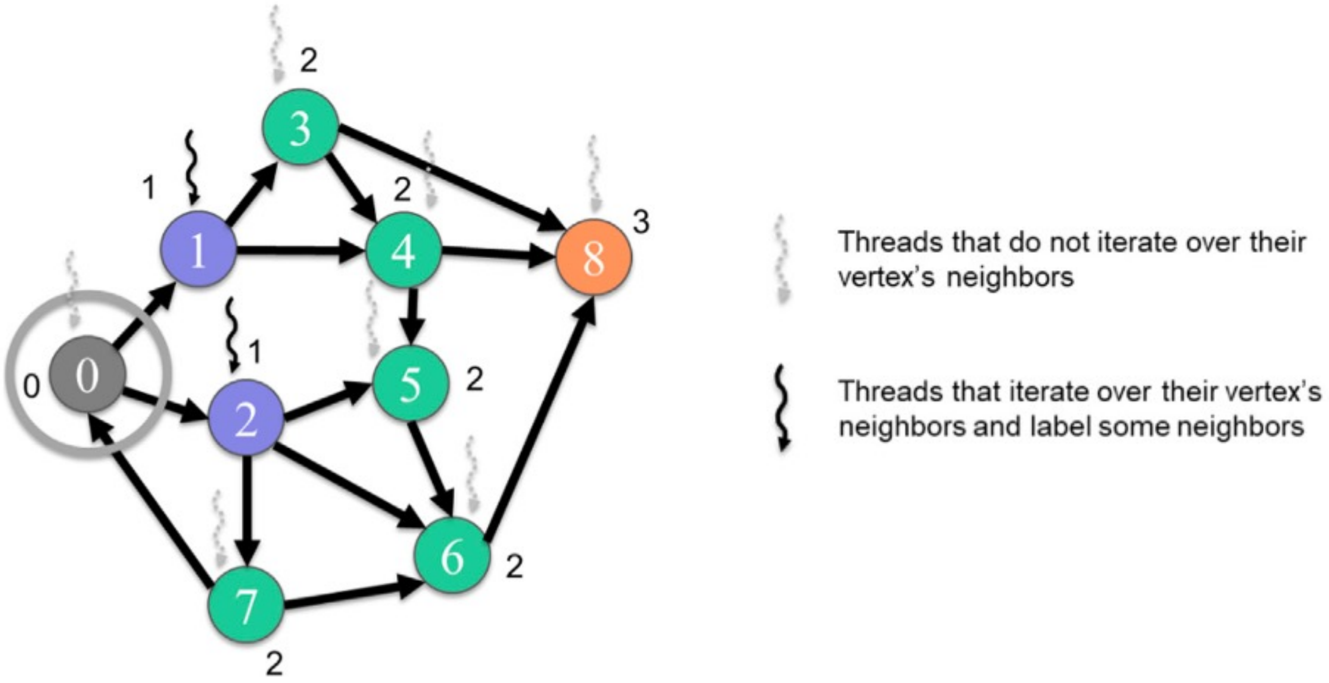
  1. Each thread first checks whether its vertex belongs to the previous level.

  2. If so, the thread will iterate over the outgoing edges to label all the unvisited neighbors as belonging to the current level.

- Since this implementation requires accessibility to the **outgoing edges** of a given source vertex (i.e., nonzero elements of a given row of the adjacency matrix), a **CSR** representation is needed.

15.3 Vertex-centric parallelization of breadth-first search

- A vertex-centric push BFS traversal from level 1 to level 2.



Threads that do not iterate over their vertex's neighbors

Threads that iterate over their vertex's neighbors and label some neighbors

15.3 Vertex-centric parallelization of breadth-first search

- **vertex-centric pull (bottom-up)** parallel implementation of BFS assigns each thread to a vertex to iterate over the vertex's incoming edges.

1. Each thread first checks whether its vertex has been visited yet.

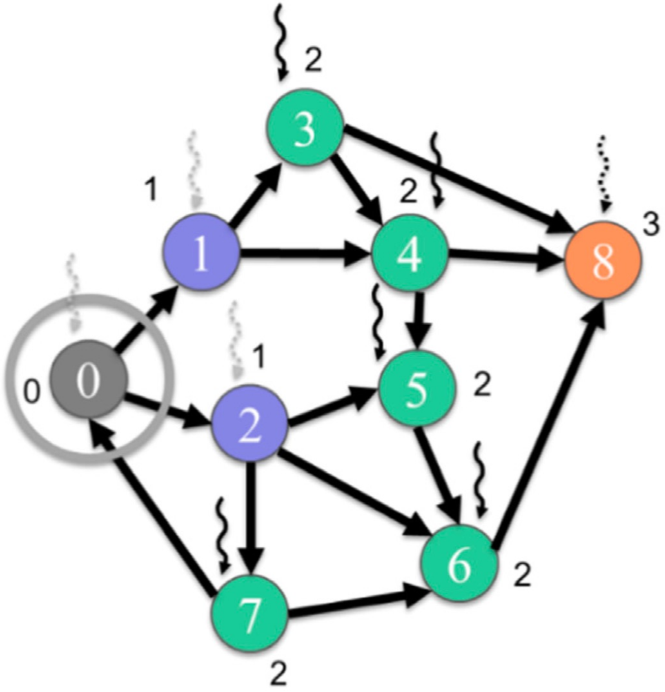
2. If not, the thread will iterate over the incoming edges to find whether any of the neighbors belong to the previous level.

3. If the thread finds a neighbor that belongs to the previous level, the thread will label its vertex as belonging to the current level.

- Since this implementation requires accessibility to the **incoming edges** of a given destination vertex (i.e., nonzero elements of a given column of the adjacency matrix), a **CSR** representation is needed.

15.3 Vertex-centric parallelization of breadth-first search

- A vertex-centric pull BFS traversal from level 1 to level 2.



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

15.4 Edge-centric parallelization of breadth-first search

- Edge-centric parallel implementation of BFS assigns each thread to an edge.

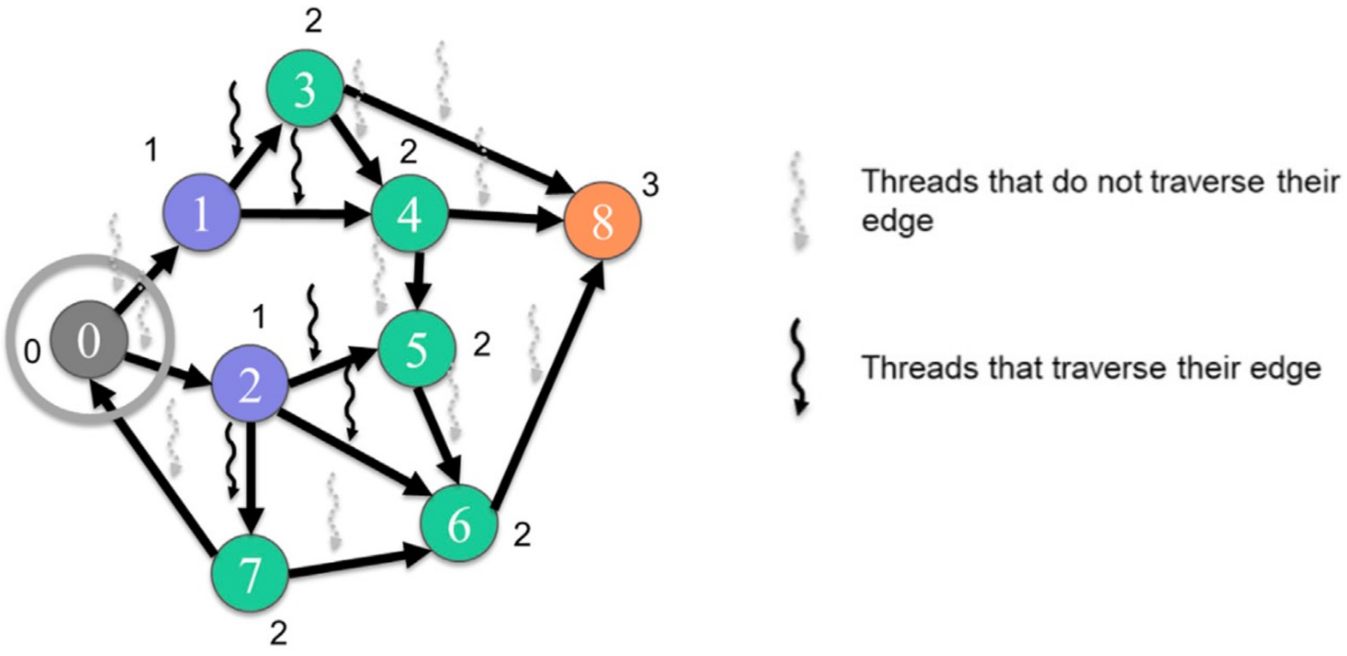
    1. Each thread first checks whether the source vertex of the edge belongs to the previous level and whether the destination vertex of the edge is unvisited.

    2. If so, it labels the unvisited destination vertex as belonging to the current level.

- Since this implementation requires accessibility to the source and destination vertices of a given edge (i.e., row and column indices of a given nonzero), a COO data structure is needed.

15.4 Edge-centric parallelization of breadth-first search

- An edge-centric BFS traversal from level 1 to level 2.


2
3
1
2
3
1
4
8
0
0
1
2
5
2
6
2
7
2
Threads that do not traverse their edge
Threads that traverse their edge

15.4 Edge-centric parallelization of breadth-first search

- Advantage of edge-centric implementation
  1. The first advantage is that the edge-centric implementation exposes more parallelism.
  2. The second advantage of the edge-centric implementation over the vertex-centric implementations is that it exhibits less load imbalance and control divergence.

- Disadvantage of edge-centric implementation
  1. The first disadvantage is that edge-centric implementation checks every edge in the graph.
  2. The second disadvantage of the edge-centric implementation is that it uses COO, which requires more storage space to store the edges compared to CSR and CSC, which are used by the vertex-centric implementations.

15.5 Improving efficiency with frontiers

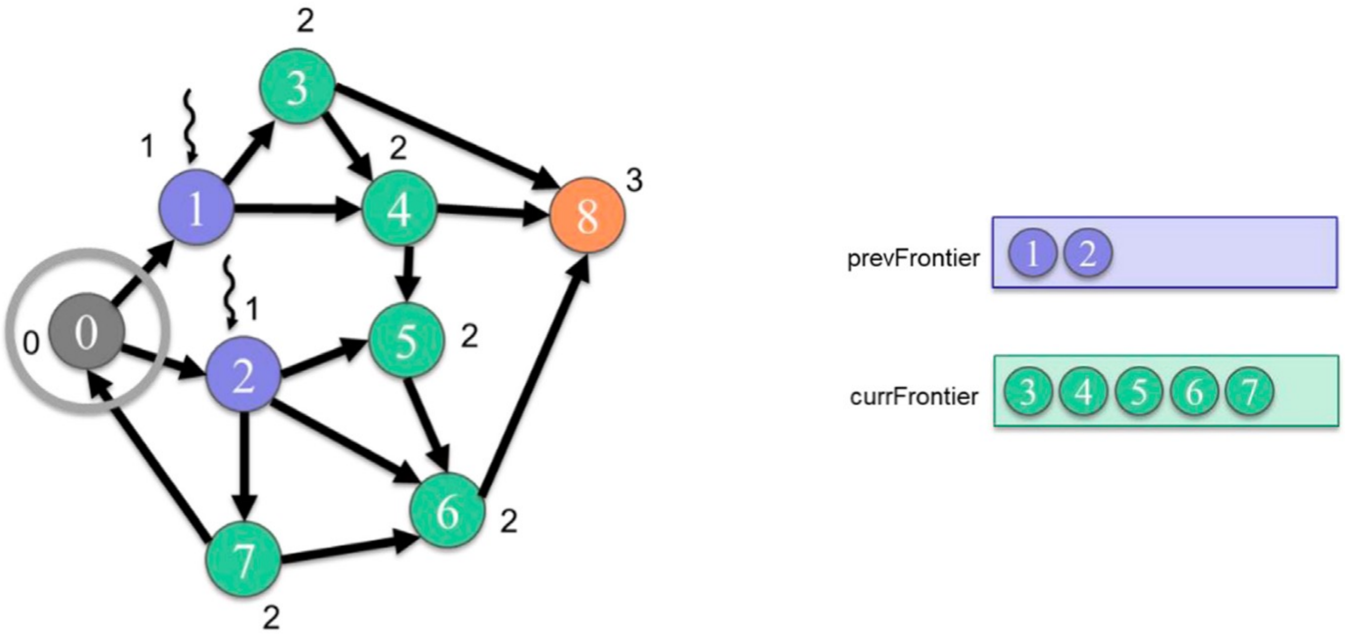
- The advantage of vertex-centric and edge-centric implementation is that the kernel are highly parallel and do not require any synchronization across threads.

- The disadvantage is that many unnecessary threads are launched and a lot of wasted work is performed.

- To avoid launching these unnecessary threads, we can have the threads processing the vertices in the previous level collaborate to construct a **frontier of the vertices that they visit**. Hence for the current level, threads need to be launched only for the vertices in that frontier

15.5 Improving efficiency with frontiers

- A vertex-centric push BFS traversal from level 1 to level 2 with frontiers.

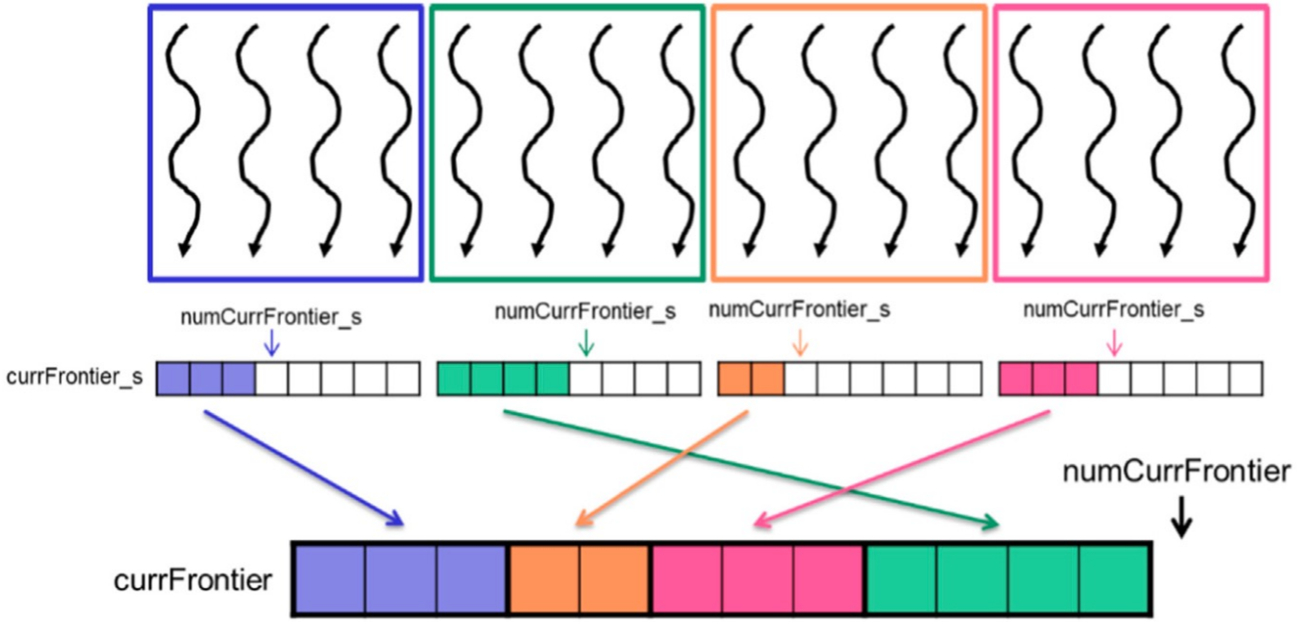


- The disadvantage of this frontier-based approach is the overhead of the long-latency atomic operations, especially when these operations contend on the same data.

15.6 Reducing contention with privatization

- Privatization reduces contention of atomic operation by applying partial updates to a private copy of the data, then updating the public copy when done.

- Threads will contend on the same data only with other threads in the same block.

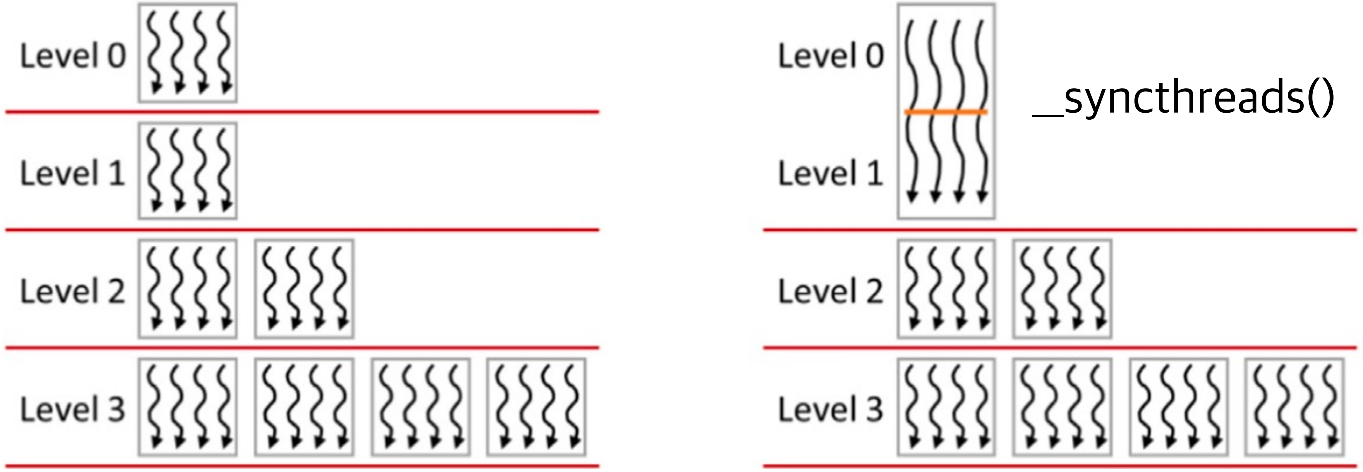


15.7 Other optimizations

Reducing launch overhead

- In most graphs, the frontiers of the initial iterations of a BFS can be quite small. For these iterations the overhead of terminating a grid and launching a new one may outweigh the benefit of parallelism.

- One way to deal with these iterations with small frontiers is to prepare another kernel that uses only one thread block but may perform multiple consecutive iterations.


Level 0
Level 1
Level 2
Level 3
Level 0
__syncthreads()
Level 1
Level 2
Level 3