Faculty of Engineering
Cairo University

Parallel Computing

#CMP4011

Graph Traversal Parallelization

Breadth First Search

Submitted to:

Eng. Mohamed Abdallah

Submitted By:

NAME	SEC	BN	ID
Yasmine Ashraf Ghanem	2	37	9203707
Yasmin Abdullah Nasser	2	38	9203717

Final System Description

The objective of our BFS implementation is to find the minimum distance to every vertex from a given source vertex. We implemented the BFS kernel using two approaches:

1. Vertex-Centric Approaches

In the vertex-centric approach parallelization is applied among vertices of the graph; each thread is assigned a graph vertex and is responsible for visiting all its neighbors and setting unvisited vertices to a level. The vertex centric approach can be implemented using three ways:

Basic Implementations:

- **Top-Down**
  Where we start from the source vertex and update the levels of the graph as we go; so the neighbors in the next level are updated

- **Bottom-Up**
  Where we reach the maximum depth of the graph and then update the neighbors of the previous vertex as we go.

- **Direction-Optimized**
  This approach takes the best of both, as the top down approach is better for shallower vertices since they don't have a lot of neighbors while it gets worse as we go deeper. On the other hand, the bottom-up approach is better at the end and worse in the beginning since the first few iterations no updates are made. The direction-optimized approach begins with the top down approach for a few levels then switches to the bottom-up approach.

Optimizations:

We noticed that we could apply a few optimizations to improve the kernel time, control divergence, privatization and memory coalescing, and latency.

- **Control Divergence & Redundancy**
  The simple top-down approach suffers from high control divergence and redundancy work due to the iterations that each thread performs on the neighbors of its vertex. Each thread iterates over all the neighbors of the current vertex and updates their level if they have not been visited.

  The problem is that some of the neighbors are common between vertices which results in unnecessary redundant checking thread. This is because the kernel is launched for every vertex in every level of the graph even if it is not

necessary and the checking is done inside the kernel even if it doesn't get updated.

To lessen control divergence and redundant work by threads we introduce queues into the kernel where we add a queue for the previous vertices and a queue for the current vertices. The threads are assigned vertices from the queue only so the kernel is launched much less.

The addition of the queues resulted in having atomic operations since multiple threads can access the same vertex when updating the queue which leads to serialization and would decrease the overall throughput.

- **Privatization**
  Due to the increase of atomic operations, all threads are atomically incrementing a global counter and inserting into the global memory which causes high latency.

  Privatization if the addition of a private local (shared) memory for each block that is shared by all threads in the block. We allocate a shared memory for the current queue and the index for accessing the queue since these are the variables that suffer from a race condition.

  The addition of shared memory required synchronizing threads after each part in the kernel to ensure that all threads reached the same point before proceeding.

  Finally, the copying of the data from the shared memory to the global memory achieves memory coalescence since each thread inserts a single element in the block and the stride is the block dimension.

- **Minimized Launch Overhead**
  Lastly, since our kernel is called for each level this causes kernel launch overhead that could reverse all the optimization done.

  The minimization of the launch overhead is as if you are merging two kernel launches into one. The condition on which this happens is that the combined current queue size of two consecutive kernel calls is less than the number of threads in a single block.

  In practice, this is achieved for the first and last levels of the graph, for example, if we have 6 levels the levels one and two could be merged and levels five and 6 could be merged.

  The "merging" of the kernels is done by writing a separate kernel to handle both levels at the same time.

2. Edge-Centric Approach

In this edge-centric approach we parallelize among edges; each thread is assigned an edge that explores the source vertices in the previous level and checks if the destination vertex is visited or not.

Optimizations:

- **Streams**
  The simple top-down approach suffers from high control divergence and redundancy work due to the iterations that each thread performs on the neighbors

Experiments

We conducted multiple experiments where we tried multiple ways to implement the BFS:

- **Graph Representation:**
  The graph is represented by an adjacency matrix but since it could be considered a sparse matrix we tried two different implementations:
  - Coordinate Representation
  - Compressed Sparse Row

  The choice of representation is dependent on the BFS approach implemented; for the vertex-centric approach we chose the CSR graph representation since for each vertex we loop on all outgoing edges from the current vertex. For the edge-centric approach we chose the coordinate representation since it represents the edges of the graph.

- **BFS Approaches:**
  We tried with both the vertex-centric and edge-centric approach as mentioned before.
  - Vertex-Centric Approach
  - Edge-Centric Approach

- **Optimizations:**
  We applied different optimizations and reported the time after each optimization and compared the results with each approach.

- **Graph Size:**
  Lastly, we tried different graph sizes with different numbers of vertices and edges. We kept the number of vertices constant and changed the edges to find the effect of dense graphs on the kernel. The results are reported in the performance analysis section.

Performance Analysis

- **Benchmarking**
  Taken benchmarks are in terms of algorithm complexity and execution time.

  CPU Benchmark:
  Sequential BFS traversal is O(V +E) where V is the number of vertices and E the number of edges.

  GPU Benchmark:
  The execution time for the parallel execution of BFS depends on the number of GPU cores N and FLOPs F
  The execution time is approximately O(V+E)/N*F

- **GPU Counter Parts and Speed-ups**

1. Vertex-Centric Approaches

Number of Vertices	CPU Time / ms	GPU Time / ms	Optimized GPU Time / ms	Speed-Up
100	0.08	0.51	0.99	0.081
1000	0.35	0.56	2.1	0.16
10000	2.89	1.93	1.07	2.70
50000	6.88	3.38	2.88	2.39
100000	30.24	12.49	10.53	2.87

2. Edge-Centric Approach

Number of Vertices	CPU Time / ms	GPU Time / ms	Optimized GPU Time / ms	Speed-Up
100	0.085	5.27	7.28	0.016
1000	5.035	39.13	12.54	0.128
10000	417.35	415.42	52.6	1.0
50000	12551.444	5524.69	775.56	2.27
100000	24682.96	6964.81	1551.61	3.5

Conclusion

Both vertex-centric and edge-centric approaches offer noticeable speed-ups over traditional CPU-based BFS implementations. Mainly we found that the vertex centric had more optimizations that can be applied than the edge centric one.

Both approaches showed higher performance at larger scaled data than CPU which is usually the goal solving bottlenecks when dealing with large-scale graphs in real-life problems, such as social networks, web graphs, or road networks.

Moreover, a huge factor that contributes to the performance of these parallel algorithms are the graph type and representation.
For example the Coordinate Representation (COO) was beneficial for the edge-centric approach, as it allowed for direct parallelization of edges. However, this representation sometimes leads to higher memory usage and redundant checks. On the other hand, the Compressed Sparse Row (CSR) representation, which organizes graph data for efficient neighbor traversal, was more suitable for the vertex-centric approach, enhancing data locality and reducing memory access latency.

For future work we can include:

- **Hybrid approaches to leverage the strengths of each methods based on the usage of the application,**
- **More optimization techniques as load balancing, algorithmic improvements or memory management**
- **Handle dynamic graphs which is a real-life implementation where vertices can**