Parallel Breadth First Search with On-GPU Asynchronous Scheduling

Anthony Ngo

December, 2024

Abstract

This paper explores the application of an on-GPU thread-data remapping method in evaluating a parallel breadth-first search algorithm, along with a CPU-synchronized queue-based implementation, and their behavior under different workloads. The performance between the on-GPU application and queue-based algorithms were compared against each other. Since graphs can be directed or undirected, performance characteristics were derived from each case, with traversal across directed graphs generally faster. In addition, two approaches to global synchronization were evaluated against each other.

1 Introduction

For context, breadth-first search (BFS) is common sub-problem in many graph-based applications. BFS is an algorithm that, starting at a source node, explores all nodes at the current depth before moving on to the nodes at the next depth level, etc.[2].

However, for large graphs, the amount of computation required to evaluate BFS is great, and thus requiring higher degrees of parallelism. A naive approach on the CPU for parallelism is to maintain a pool of threads, and assign nodes to any available threads. However, GPUs offer higher raw computational throughput compared to CPUs through higher parallelism, but they are more difficult to program because of global synchronization between threads essentially requires breaking computations across multiple kernel invocations.

This paper explores the applications and performance characteristics of on-GPU asynchronous processing, compared to a CPU-bound queue-based implementation. The performance characteristics of this problem will be investigated across various workloads, different processing methods, and different runtime configurations.

Central questions to be answered are (1) if we could avoid global synchronization, would that give us better performance? (2) what graphs have better or worse performance if we avoid global synchronization?

2 Related Work

The Harmonize framework, developed by CEMeNT, allows programs to abstract execution as a system of asynchronous functions, and offers two methods for evaluating algorithms: (1) an event-based method with global synchronization, that synchronizes GPU work via the CPU, and (2) an asynchronous method which performs lock-free work-sharing, requiring no global synchronization. This research uses these two evaluation methods for the purpose of comparison.

Harmonize examines the operations that are due to be executed, separating them into similar-operations so that similar-work may be executed together in batches [4]. Harmonize was chosen for this research specifically because complete exploration of a graph could be done in batches of nodes.

3 Description of Research

An on-GPU application was built on top of Harmonize, based on example code provided in the code repository[1]. While developing this application, two sources of datasets were evaluated on. Lastly, performance characteristics were gathered through a script that made the application process each dataset with varying source nodes, launching with event or asynchronous evaluation methods, whether the graph provided was directed or undirected, and a set group count and cycle count.

3.1 Code/Algorithm

A node in a graph is represented by the following:

```
struct Node {
  int id;
  size_t edge_count;
  size_t edge_offset;
  unsigned int depth;
  unsigned int visited;

  Node() {}

  Node(int _id) : id(_id), edge_count(0), depth(0xFFFFFFFF), visited(0) {}
};
```

To store the edges for each respective node, an adjacency list data structure was used. A graph is represented by 2 arrays. (1) edge arr: a concatenation of all adjacency lists of all nodes, of size —E— where E is the total number of edges. (2) node arr: an array of size —V— where V is the number of nodes, where each node stores offsets into the edge arr. The ith offset marks the position where the adjacency list for the ith node begins. Each node stores the length of its adjacency list, and the use of a separate edge arr allows for efficient access of a node's associated edges.

The following is the program *operation*/BFS implementation on-GPU:

```
struct BFSProgramOp {
  using Type = void (*)(Node* node, int print_debug);

  // store edges/nodes in single arrays at offsets
  template <typename PROGRAM> __device__ static Node& get_edge_node(PROGRAM prog, Node* node, int i) {
    int edge_node_id = prog.device.edge_arr[node->edge_offset + i];
    Node& edge_node = prog.device.node_arr[edge_node_id];
    return edge_node;
  }

  template <typename PROGRAM> __device__ static void eval(PROGRAM prog, Node* node, int print_debug) {
    if (print_debug) {
      int tid = blockIdx.x * blockDim.x + threadIdx.x;
      printf("thread %i: node %i\n", tid, node->id);
    }

    // explore neighbors
    for (int i = 0; i < node->edge_count; i++) {
      Node& edge_node = get_edge_node(prog, node, i);

      // if neighbor not visited, mark for next visit
      if (edge_node.visited != 1) {
        atomicCAS(&edge_node.visited, 0, 1);
        atomicMin(&edge_node.depth, node->depth+1);
```

```
        prog.template async<BFSProgramOp>(&edge_node, prog.device.print_debug);
      }
    }
  }
};
```

In the helper function get edge node, it takes a Node pointer, and an index i. For example, if a node has two edges, the edge offset is the size of the edge arr before adding these two edges, thus the ith edge associated with this node is offset+i.

The program operation defined above is run in batches, and the algorithm is as follows: each thread handles a single node, each edge that isn't visited yet has a thread spawned for that edge, and each edge is marked as visited so that if exploration isn't duplicated.

prog.device refers to the device state, as follows:

```
struct MyDeviceState {
  Node* node_arr;
  int* edge_arr;
  int root_node;
  int print_debug;
  iter::AtomicIter<unsigned int>* iterator;
};
```

The device state, itself, is an immutable struct, but can contain references and pointers to non-const data. This device state is used by each thread to access the node graph in device memory, and so atomic operations were used to prevent race conditions.

To begin work, a root node is needed. This node can be anywhere on the graph, provided there are edge nodes associated.

3.2 Datasets Used

For this research, two sources of data were used: Network Repository [5], and Stanford Large Network Dataset Collection[3]. For Network Repository, I chose graphs from the DIMACS collection because the graphs had an average density of 5-6 levels. I chose to use networks from the Stanford Large Network Dataset Collection because the data is based on real-world interactions, such as social networks.

3.2.1 The Network Repository

Datasets provided by The Network Repository[5] are usually in the Matrix Market format. A file in this format is comprised of 4 parts: a header line, comment lines, size lines, and the remaining are data lines.

A typical .mtx file is as follows:

```
%%MatrixMarket matrix coordinate pattern symmetric
45 45 918
2 1
3 1
3 2
4 1
4 2
4 3
5 1
5 2
5 3
...
```

The header line contains an identifier and 4 fields. For simplicity, the datasets with the following header line were chosen:

```
%%MatrixMarket matrix coordinate pattern symmetric
```

Comment lines each begin with % and can be ignored.

After the header and comment lines is the size line. This specifies the number of rows, columns, and nonzero elements[5].

The rest of the file are the data lines. They are in the format of:

```
i j
```

where i is the node, and j is the edge.

3.2.2 Stanford Large Network Dataset Collection

Datasets provided by Stanford Large Network Dataset Collection[3] are usually CSV files, with data lines in the form of

```
id,target
```

3.3 CPU-bound Queue-based Implementation

For completeness, the performance characteristics of the on-GPU application were compared against a queue-based implementation of BFS.

The following is the code for the queue-based implementation:

```
std::queue<Node*> queue;
root_node->depth = 0;
root_node->visited = 1;
queue.push(root_node);

 while (!queue.empty()) {
    Node* node = queue.front();
    queue.pop();

    for (int i = 0; i < node->edge_count; i++) {
      int edge_node_id = node_graph.edges[node->edge_offset + i];
      Node& edge_node = node_graph.nodes[edge_node_id];
      if (edge_node.visited != 1) {
        edge_node.visited = 1;
        edge_node.depth = std::min(edge_node.depth, node->depth+1);
        queue.push(&edge_node);
      }
    }
}
```

Starting at the root node, if adjacent nodes aren't marked as visited yet, they are marked as so and enqueued. In comparison with the on-GPU application, nodes must wait in queue before being accessed.

4 Results

Evaluation of performance for both implementations is measured by the number of milliseconds taken to explore all nodes in the undirected graph, or explore all directed nodes in the directed graph, for each given dataset.

The results were generated by a Python script that looped through a list of datasets, and captured the output of the runtime generated by either implementation through the `subprocess` package.

For each dataset, a single, randomly chosen node was selected as the root node. This was done so the root node would be consistent across the results for the implementations, and consistent in either the directed or undirected case.

dataset	root	program	directed	runtime
network-list.csv	16	async	True	3.72787
johnson8-2-4.mtx	22	async	True	4.60323
johnson8-2-4-nocomment.mtx	21	async	True	4.03213
MANN-a9.mtx	43	async	True	6.04336
san400-0-5-1.mtx	308	async	True	23.2174
brock200-1.mtx	195	async	True	15.6662
C500-9.mtx	423	async	True	35.8544
C1000-9.mtx	728	async	True	47.0593
C4000-5.mtx	3903	async	True	112.832
lastfm-asia-edges.csv	13	async	True	44.0903
lastfm-asia-target.csv	717	async	True	3.01939
artist-edges.csv	25658	async	True	70.069
athletes-edges.csv	9053	async	True	5.43072
company-edges.csv	653	async	True	32.7974
government-edges.csv	3536	async	True	30.7836
new-sites-edges.csv	22467	async	True	6.76643
politician-edges.csv	2077	async	True	3.15293
public-figure-edges.csv	5549	async	True	36.2846
tvshow-edges.csv	1297	async	True	3.29882

Table 1: Async, Directed Graph

When running the Harmonize-based application, the following parameters were configured for each dataset:

- Group count = 240
- Cycle count = 10000
- Arena size = 100000

Group count is how many multiples of a group of 32 threads (warps). For example, if group count is 2, then 64 threads are run, if group count is 3, then 96 threads are run, and so on[1].

Cycle count is how many times each warp executes a batch of function calls during a kernel invocation[1].

Arena size is the size of the arena/IO buffer used[1].

5 Analysis

Comparing the performance characteristics between asynchronous and event-based methods, they perform similarly, however asynchronous performs better when graphs are more interconnected. Event-based method performs better when graphs are more sparse or smaller in size.

The event-based method is generally faster than the asynchronous method, however in some cases graphs with more nodes, longer paths, and greater variance in path lengths, then asynchronous method is faster. The largest dataset did better when evaluated with the asynchronous method, suggesting that larger datasets may derive further benefits.

Harmonize runtimes run slower than custom implementation, however these runtimes require tuned resource allocation and scheduling parameters to behave efficiently, but the logistics of per-graph tuning were beyond this research.

dataset	root	program	directed	runtime
network-list.csv	16	async	False	6.67891
johnson8-2-4.mtx	22	async	False	5.21296
johnson8-2-4-nocomment.mtx	21	async	False	5.02787
MANN-a9.mtx	43	async	False	6.22688
san400-0-5-1.mtx	308	async	False	40.3582
brock200-1.mtx	195	async	False	18.9298
C500-9.mtx	423	async	False	54.2806
C1000-9.mtx	728	async	False	99.1065
C4000-5.mtx	3903	async	False	230.522
lastfm-asia-edges.csv	13	async	False	31.2898
lastfm-asia-target.csv	717	async	False	209.548
artist-edges.csv	25658	async	False	73.2489
athletes-edges.csv	9053	async	False	33.5344
company-edges.csv	653	async	False	27.5521
government-edges.csv	3536	async	False	35.1548
new-sites-edges.csv	22467	async	False	40.0481
politician-edges.csv	2077	async	False	33.7288
public-figure-edges.csv	5549	async	False	34.6964
tvshow-edges.csv	1297	async	False	27.1835

Table 2: Async, Undirected Graph

dataset	root	program	directed	runtime
network-list.csv	16	event	True	3.50541
johnson8-2-4.mtx	22	event	True	4.20634
johnson8-2-4-nocomment.mtx	21	event	True	3.61485
MANN-a9.mtx	43	event	True	5.328
san400-0-5-1.mtx	308	event	True	20.0471
brock200-1.mtx	195	event	True	12.7987
C500-9.mtx	423	event	True	34.8497
C1000-9.mtx	728	event	True	87.7272
C4000-5.mtx	3903	event	True	715.154
lastfm-asia-edges.csv	13	event	True	39.0512
lastfm-asia-target.csv	717	event	True	3.09002
artist-edges.csv	25658	event	True	260.956
athletes-edges.csv	9053	event	True	5.20464
company-edges.csv	653	event	True	26.6518
government-edges.csv	3536	event	True	36.4667
new-sites-edges.csv	22467	event	True	6.56042
politician-edges.csv	2077	event	True	2.72019
public-figure-edges.csv	5549	event	True	28.6116
tvshow-edges.csv	1297	event	True	3.0719

Table 3: Event, Directed Graph

dataset	root	program	directed	runtime
network-list.csv	16	event	False	6.11955
johnson8-2-4.mtx	22	event	False	4.60278
johnson8-2-4-nocomment.mtx	21	event	False	4.46058
MANN-a9.mtx	43	event	False	5.70598
san400-0-5-1.mtx	308	event	False	36.5738
brock200-1.mtx	195	event	False	16.156
C500-9.mtx	423	event	False	66.4251
C1000-9.mtx	728	event	False	214.624
C4000-5.mtx	3903	event	False	1340.56
lastfm-asia-edges.csv	13	event	False	151.608
lastfm-asia-target.csv	717	event	False	127.84
artist-edges.csv	25658	event	False	1949.15
athletes-edges.csv	9053	event	False	299.439
company-edges.csv	653	event	False	213.885
government-edges.csv	3536	event	False	221.745
new-sites-edges.csv	22467	event	False	577.307
politician-edges.csv	2077	event	False	136.157
public-figure-edges.csv	5549	event	False	246.813
tvshow-edges.csv	1297	event	False	77.1085

Table 4: Event, Undirected Graph

dataset	root	program	directed	runtime
network-list.csv	16	None	True	0.002321
johnson8-2-4.mtx	22	None	True	0.012906
johnson8-2-4-nocomment.mtx	21	None	True	0.007081
MANN-a9.mtx	43	None	True	0.036786
san400-0-5-1.mtx	308	None	True	0.173501
brock200-1.mtx	195	None	True	0.10559
C500-9.mtx	423	None	True	0.536991
C1000-9.mtx	728	None	True	1.53032
C4000-5.mtx	3903	None	True	24.0705
lastfm-asia-edges.csv	13	None	True	0.29794
lastfm-asia-target.csv	717	None	True	0.000911
artist-edges.csv	25658	None	True	2.24383
athletes-edges.csv	9053	None	True	0.025357
company-edges.csv	653	None	True	0.270285
government-edges.csv	3536	None	True	0.297635
new-sites-edges.csv	22467	None	True	0.041422
politician-edges.csv	2077	None	True	0.001974
public-figure-edges.csv	5549	None	True	0.289017
tvshow-edges.csv	1297	None	True	0.003149

Table 5: Queue, Undirected Graph

dataset	root	program	directed	runtime
network-list.csv	16	None	False	0.029704
johnson8-2-4.mtx	22	None	False	0.012998
johnson8-2-4-nocomment.mtx	21	None	False	0.023179
MANN-a9.mtx	43	None	False	0.02473
san400-0-5-1.mtx	308	None	False	0.528066
brock200-1.mtx	195	None	False	0.215601
C500-9.mtx	423	None	False	1.42298
C1000-9.mtx	728	None	False	5.50055
C4000-5.mtx	3903	None	False	50.1399
lastfm-asia-edges.csv	13	None	False	1.29383
lastfm-asia-target.csv	717	None	False	0.842476
artist-edges.csv	25658	None	False	22.843
athletes-edges.csv	9053	None	False	2.9468
company-edges.csv	653	None	False	2.50272
government-edges.csv	3536	None	False	2.00509
new-sites-edges.csv	22467	None	False	7.02565
politician-edges.csv	2077	None	False	1.22846
public-figure-edges.csv	5549	None	False	2.39057
tvshow-edges.csv	1297	None	False	0.675057

Table 6: Queue, Undirected Graph

6 Conclusions

In this research project, lessons learned were to communicate clearly with the advisor when help is needed or if something is beyond the scope of the project, experiment with how to use Harmonize properly before diving straight ahead, and lastly utilize features of git to integrate with advisor's suggested changes.

In answering the central questions: (1) avoiding global synchronization through the asynchronous method does yield better performance but only on specific graphs. (2) large graphs have better performance if we avoid global synchronization, and suffers on sparser graphs.

Goals that were accomplished include learning more about CUDA, sticking to a set schedule every week, and building a decently performant implementation of breadth-first search running on a GPU

7 Future Work

Future work could be done with different datasets (larger datasets, datasets with greater path variants), different graph algorithms, reducing runtime overhead, and optimizations that may improve one method or the other.

8 References

References

[1] Center for Exascale Monte Carlo Neutron Transport (CEMeNT), Braxton Cuneo, and Joanna Piper Morgan. harmonize, April 2024.

[2] Susanna S. Epp. *Discrete Mathematics with Applications*. Brooks/Cole Publishing Co., USA, 4th edition, 2010.

[3] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[4] Joanna Piper Morgan, Ilham Variansyah, Braxton Cuneo, Todd S. Palmer, and Kyle E. Niemeyer. Performance portable monte carlo neutron transport in mcdc via numba, 2024.

[5] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.