Breadth First Search:
A Performance Study of Parallel and Sequential Algorithms

Daniel Simon
Grant Baker
Lawrence Gunnell
Yubin Fan

Portland State University

Abstract:

This paper considers the performance speedups experienced when searching for the shortest path between two nodes in an unweighted bi-directed graph using the Breadth First Search (BFS) algorithm. Our sequential BFS program served as the control group and was used to establish the speedup achieved through the application of various parallel programming technologies. The different parallel technologies utilized and addressed within this paper were CUDA, Unified Memory, Warp Queues, and OpenACC. Timing utilized CUDA events, and test runs were averaged across 25 runs for each implementation. Graph sizes ranged from 16 to 256 million nodes and were stored in Compressed Sparse Row format. Graphs were unweighted, bi-directional, and were relatively sparse, with an average degree of 3 across all nodes in a graph. Although the sequential version performed more than 1000x slower at large graph sizes, it did out-perform CUDA, Unified Memory, and Warp Queues for small graphs of approximately 4096 nodes or less. OpenACC performed similarly well on small graphs, but far better on large graphs. While OpenACC still trailed behind the Simple CUDA and Warp Queue implementations, it still outperformed our Unified Memory implementation. The simplistic CUDA and Warp Queues produced the best performance results and generated very similar performance profiles. Out of the parallel implementations, Unified Memory performed the worse, however the overall time calculation includes copying the graph into memory. If utilized on live systems, because the data may already be resident in the memory space, its performance relative to other parallel implementations may be better than what is documented in our results.

Methods:

Sequential:

Traditional breadth first search (BFS) is a fundamental graph-searching algorithm used for finding the shortest path between 2 nodes. The compressed sparse row (CSR) format was used across all of our algorithms to ensure that the sparseness of the graph did not waste computational resources. The sequential version also served as the control group that the other tests were benchmarked against. All groups were tested against the same graphs, which were sparse, bi-directional, unweighted graphs, with an average diameter of approximately 3 edges per node. The BFS algorithm employed was the conventional top-down algorithm, which consisted of the following steps:

1. A source node is designated.
2. Using an array of labels, mark the source node as distance 0 from itself, and all other nodes as distance -1 from the source. The value of -1 indicates that the node has not been processed in a previous frontier.

3. Start at the source node and Identify all neighboring nodes, which are connected by edges originating from the source node.
4. From each of the neighboring nodes, mark each unvisited connected node with the current distance from the source node. The nodes identified at a specific distance level comprise a frontier.
5. Repeat this process for each new frontier, incrementing the distance by 1, and stop when no nodes have been added to the frontier.

There are benefits and drawbacks to this style of BFS. One benefit is the use of ping-pong buffering. Traditionally, 2 frontiers are tracked, the current frontier and the previous frontier. These pointers are used to switch the roles of the 2 frontiers to avoid new allocations as well as having to copy the resultant values from the current frontier into the previous frontier. Additionally, there are 2 tail pointers which are used to indicate the number of elements in the previous frontier and the insertion point in the current frontier. The stopping condition is whether any new nodes were added to the current frontier after processing and before the buffer swap. This indicates that the graph has reached maximum spread. Any unvisited nodes remaining in the graph are disconnected from the source node.

Performance for the sequential version is expected to be considerably good at smaller graph sizes where the graph and the entire program fit conveniently into the computer's working-memory. However, since the CPU under test also performs various OS-related tasks such as context switching, processing interrupts, etc. and these distractions are likely to impact the performance results. This gets costlier as the program size increases, largely due to the size of the graph exceeding cache and RAM sizes. Instances of that occurring may require fetching from disk-based swap memory, etc. which would come at a significant performance cost. However, the machine used for testing did not have swapping enabled.

OpenACC:

Our implementation of breadth first search with OpenACC was very similar to our sequential implementation. The largest change was specifying the data requirements for the parallel region beforehand. This was critical because full synchronization is required after an iteration of the frontier before the next search iteration is launched. As the graph data is constant throughout the execution of the kernel, it was only copied into the region. Because the output label array and initial parent array had specific initialization states, they needed to be copied both in and out. Lastly the current frontier array only needed to be allocated as it acts only as a copying destination.

Inside of the scope of the data declaration, was the loop that called the breadth first search kernel. This kernel declaration was kept simple, only indicating that the enclosed region should be parallelized for the nvidia device. However, even the device specification turned out to be superfluous as libgomp7.5 only can offload to NVidia devices via PTX. We attempted a more specific set of parallelization declarations, requesting each gang to search one node, while having each child node investigated by a worker of the parents gang. However, this was invalid for the implementation of OpenACC we compiled with. This is because the number of child

nodes is not consistent from parent to parent, meaning the two parallelization declarations were requesting dynamic parallelism. In theory this is supported by the OpenACC 2.0 specification, however, implementation support seems limited.

Beyond these changes the only other consideration made was for the race condition in generating the new frontiers. This is a concern with any parallel implementation of breadth first search. We specified that changes to the tail of the current frontier are atomic updates to ensure no thread could store while another was reading and vise versa. However, the updates to the actual current frontier array were only specified as atomic writes as there is no need to ever read from the current frontier within the kernel. Because any thread would be writing the same data, and writes would need to be serialized, the labels array required no synchronization specification.

CUDA:

Our CUDA implementation of BFS was extended from our sequential implementation. Note in step 5 of our implementation, we check to see if no new nodes have been visited. If any new nodes have been visited, we put the newly visited nodes in our previous frontier and we go back to step 3, which identifies all new nodes connected to the nodes in the previous frontier that have not been visited. Since we still have to iterate over each node in the previous frontier, we can parallelize the computation done with each node in the previous frontier. Each node in the previous frontier has to visit all the nodes that they are connected to by an edge in the graph. So, for each node in the previous frontier we put the computation it has to do in a separate thread. In our CUDA implementation, we keep the check that new nodes have been visited serialized. This allows all launched threads to synchronize before the next expansion of the frontier.

Parallelizing the computation gives us a race condition. There is the possibility of two different nodes in the previous frontier connecting the same node that was not previously visited. Now, we have to make sure that they aren't both doing computation on the node. All we need is for one of them to do the computation on the node. So, we pick one node to do the computation and the other nodes we tell them that there is no need to do the computation, and to carry on with the rest of their computation. The way we implement the lock is with an atomic exchange. For this, we use an additional array which we call visited. The visited array will have one entry for every node. If the node has been visited before, the visited array at the nodes location is set to one. Otherwise, if it hasn't been visited the visited array at the nodes location is set to zero. Now, when we attempt to visit a node with one of our threads we do an atomic exchange with the visited array at the nodes location. We set the visited array at the nodes location to one, and we get back the previous value of the visited array at the nodes location. Since this is an atomic operation, the value cannot be written to in the middle of our operation. If we get that the previous value of the visited array is one, then the node has been visited so there is no further computation needed to be done for this node. If the previous value of the visited array is zero, then the node has not been visited before, and the atomic exchange will have updated the visited array at this node's location to one. This way if any other threads try to

access the node this thread is working on, it will know that the node has already been visited so there is nothing to do.

Now, we are free to fill in the label array at the nodes location, since we are the only thread to access the node. We also do an atomicAdd on the current frontier tail. This is an atomic operation that increments the tail by one, and returns back the previous current frontier tail. This lets us write to the previous current frontier tail, and prevents other threads from writing to it. Now, an optimization that we make is we set up an array in shared memory so that we can write out to the current frontier in a coalesced manner. If the array in the shared memory is filled up, then we will have to write it out to global memory.

So, in our host code we launch a kernel that processes the computation that is needed to be done for the previous array. After this is done, the current array is updated with the nodes visited by the previous array. If no new nodes were visited, then the BFS is done. Otherwise, if at least one new node was visited, then the previous current frontier becomes the new previous frontier. The previous current frontier tail also becomes the new previous frontier tail. This lets us know how many nodes we have to iterate over for the next iteration. We just swap pointers to do this. Then, we launch another kernel with the updated previous frontier. We keep doing this until no new nodes are visited in a kernel.

Unified Memory:

For unified memory, there is very little implementation difference between the Unified Memory version and the normal CUDA version. In fact, the kernel is exactly the same. The difference is in the normal CUDA version we have the graph in the CPU and transfer it to the GPU. In the unified memory version, the memory doesn't get transferred over until it is needed that triggers a page fault. Then some time will be taken in the kernel to transfer over the memory that is needed. The motivation for doing this is in the interest of doing a graph search on a sparse non-connected graph. For instance, when we do BFS we do not know what parts of the graph will be needed so we have to transfer the entire graph to the GPU apriori. With Unified Memory, we do not have to transfer the entire graph over the GPU. We only need to transfer the nodes that are needed to do the BFS. This is the connected component of the graph that contains the source node. Since, less nodes have to be transferred over to the GPU, we hope that less time is spent transferring data to the GPU. This would improve the performance of our parallel BFS.

Warp Queues:

In the CUDA implementation, we set up a queue in shared memory so that we can write out to the current frontier array in global memory in a coalesced fashion. These block level queues can still have a lot of threads accessing the queue. All threads in a warp will access the same block queue. To remedy this bottleneck, we create warp level queues. During kernel execution, we classify threads into warp-level queues using the least significant bits of their

threadIdx.x values. This evenly distributes the atomic operations executed by threads in a warp to the warp-level queues.

Testing and Results:

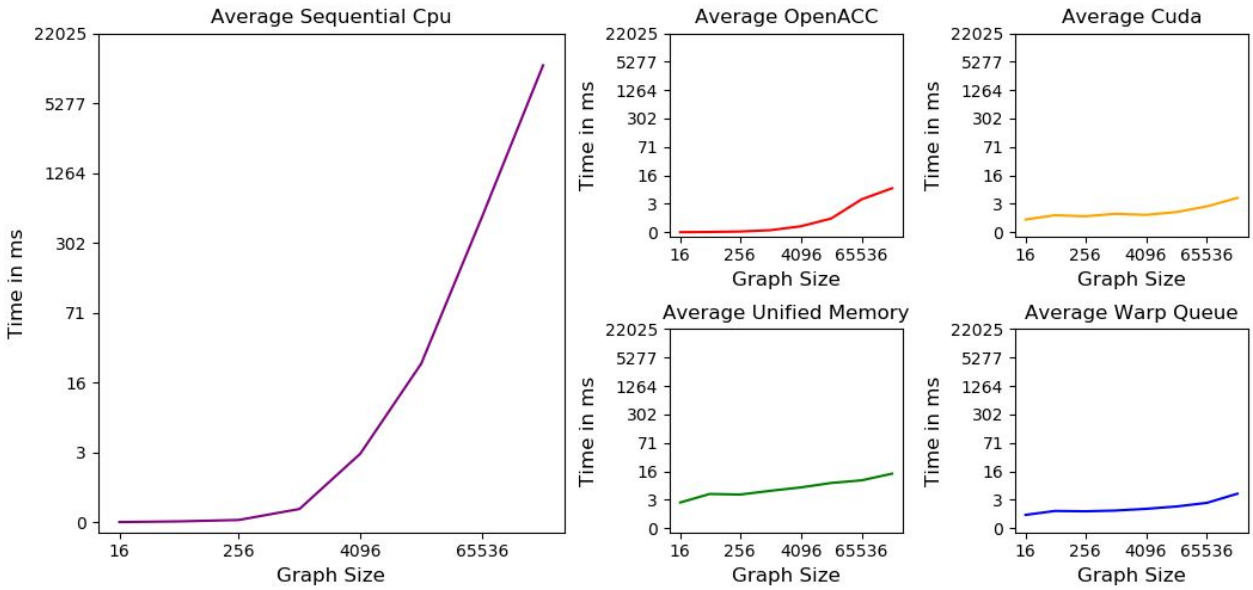
Experimental Design:

A test harness was created to control the sequential, OpenACC, and various CUDA implementations of the breadth first search algorithm. Central to this design was a random graph generation method that would create a graph of a given number of nodes, and then randomly add edges. Each node generated a random number of edges determined by a normal distribution originating from it to any other uniformly randomly sampled node. Edges were considered bidirectional, and were therefore added once in each direction to the CSR representation. Edges that already existed in the graph as well as self connections were ignored. We selected graph sizes ranging exponentially from small 16 node graphs to half a billion node graphs for our testing. In addition to this, graphs of 4 different diameters were tested. Unlike our various graph search implementations, the graph generation portion was performed sequentially. However, that decision did impact the size of the graphs we were capable of generating. When comparing the search times to the time spent generating graphs, parallelizing that portion of the code base would not only allow us greater freedom to test significantly larger graphs, but having that flexibility would increase our ability to test against different graph densities at that size.

Time points were set to include whatever data upload and download was needed for the implementation. For CUDA implementations, time points separately captured the upload, execution, and download which were then summed for a total implementation run time. Timing, for all implementations, was done using CUDA events captured in the default stream providing timings with the resolution of approximately half a microsecond. This was chosen over using the host's high resolution clock as the host scheduler seemed to add variability to the results. All CUDA tests were executed on the default stream to limit the effects of the CUDA library's internal threading. Because memcpys are only synchronous with the host thread when staging before the DMA transfer, the host thread was synchronized at the end of each upload to ensure the whole process was completed. This was to ensure that timings gathered for unified memory implementations, which are fully synchronous, were comparable to global memory copies.

Initial timings of the cuda implementations were on the order of milliseconds, this meant measurement noise was a significant issue. To reduce these effects, each test was run 25 times and the resulting time data is the average of those values. Additionally, after executing each run, the resulting label vectors were compared between implementations to ensure identical results.

Results:



	Sequential	OpenAcc	Simple CUDA	Unified Memory	Warp Queues
time	11419.9ms	8.12791ms	4.63423ms	14.5013ms	4.65992ms
speedup	N/A	1405.9x	2464.2x	787.5x	2450.6x

Benchmarked with a Graph diameter of approximately 3 edges per node on an Intel Core i9 9880h with an NVidia GeForce RTX 2070 Mobile card.

Analysis:

It is abundantly clear that a sequential implementation is a poor design choice for a breadth first search. But the comparisons between the parallel implementations are more interesting. It should be noted that a thorough test would include graph sizes that approach the size of the GPU memory. This was not done because the graph generation code was not parallelized and therefore was the longest part of each test by far.

Firstly, the performance of the OpenACC implementation at small graph sizes rivals that of the sequential implementation. However, at large graph sizes, the performance is slightly worse than a CUDA unified memory implementation. Unfortunately the kernels ran too quickly for a monitoring tool to determine what the OpenACC target device was. We hypothesize that a multi-threaded CPU implementation was generated which was used at smaller graph sizes. At

larger sizes OpenACC could have switched to using the CUDA device, however we were not able to prove this. Either way, the result was good performance across all graph sizes tested, which was unique among the implementations we tested.

When comparing the CUDA implementations, the results are not as clear. Firstly the timing for the unified memory includes data movement into the unified memory space. This is not representative of a true use case. As a result the timings for the unified memory implementation are higher than those of the other CUDA implementations; however, the timings for execution time alone are only slightly higher when compared to the others. This was expected as the kernel was exactly the same as the simple cuda implementation, but the underlying driver could page fault when needed to the CPU memory.

The timing differences between the simple CUDA implementation and the Warp Queue implementation are within sampling error of each other. This is completely useless for performance comparison. In order to see a difference in timings, there would need to be significant data racing for the current frontier array slowed by atomics. This would require both large graphs and high connectivity. As a result it seems that the extra operations required to manage the shared memory cache extend computation time a similar amount to atomic accesses to global memory.

Conclusion:

This set of implementations only represents a subset of the possible methods to do a breadth first search. And a breadth first search is generally not all that useful on its own. Given more time, it would be good to compare our implementations to public GPGPU graph processing libraries.

Until the release of CUDA 11 this May, NVidia offered nvGRAPH, a first party library for graph processing on a gpu utilizing sparse matrix representations. Rather usefully, the documentation references a Berkeley paper “Direction Optimizing BFS”. The essence of this paper is an algorithm that alternates between a breadth first search and a bottom up search. The bottom up search is run for all unvisited nodes, where each node checks if any of its neighbors is in the previous frontier. This has the benefit of not requiring any atomic operations or synchronization. However, the bottom up search is only efficient when the previous is sufficiently larger or connected to a significant portion of the unvisited nodes. This assumption seems fair for graphs representing social networks. A markov model is used to determine transitions between the breadth first search and the bottom up search. The algorithm loses significant efficiency when there are large non-connected components.

NVidia’s graph processing library has since been spun out into the RAPIDS incubator and is receiving a massive overhaul. An alternate API, Gunrock, is an open source graph processing library developed by UC Davis that will eventually integrate with the RAPIDS ecosystem. Not only does it provide an implementation of “Direction Optimizing BFS”, but it has a large selection of generic graph processing algorithms that can be distributed across multiple GPUs on a single node.

We have no expectation that our implementations would stand up against these more developed libraries. Our CUDA implementations have several significant shortcomings. Firstly, the kernels fetch a significant amount of data for relatively few computations. More than that, those data accesses are not coherent, the reads from a set of threads will be striped across several rows. It would be more efficient to process all the children of a single node before moving on to a new node. This could be a good use case for dynamic parallelism, as a larger locad could be used and shared among threads. However, this could cause issues because of the inconsistent number of edges each frontier node has. Furthermore in all graphs we tested, the average number of edges was below the size of a warp which could cause divergence.

Beyond data access pattern issues, each expansion of the frontier is calculated with a new kernel launch. This is done more than just to fully synchronize all threads, but to also check the size of the frontier for the termination condition. This kernel launch adds overhead, especially for sparsely connected graphs. This could somewhat be reduced by launching the kernels in a stream and having the host wait to add the next kernel for an event recorded by the first thread adding a node to the frontier. However these modifications were beyond the scope of this project.

Future Research:

To expand upon this body of work, there are several subjects worth exploring. The GPU's tiling acceleration scheme implies that a hierarchical search method might show promise in some applications. This could be implemented as either coarse to fine optimization, or slicing a large graph into subgraphs. Both of these methods would improve coherence and reduce synchronization overhead. But moving away from breadth first searches, these algorithms only operate on unweighted graphs. Once the edges have weights, the search algorithm becomes more complicated, eg. Dijkstra and A*. This would require an expansion of our sparse array implementation. For a breadth first search there is no reason to store values of the sparse array as if the edge exists the value is 1. These algorithms would also have more significant synchronization issues due to the updating of the labels array and visiting nodes more than once.

One optimization we could do is using dynamic parallelism. In our implementation, we had the host code launch the kernels. Then we would transfer data from the device to the host to tell whether we accessed any new nodes. If we launched a kernel that did this work then we wouldn't have to transfer any data to the host until the end. We could have the launched kernel be in charge of launching kernels to update the current frontier. This would reduce the dependency on the CPU. At the end of the BFS, we would transfer the resulting distances back to the host. This eliminates the need to transfer data after each iteration of the BFS algorithm to be able to tell when to stop.

For a very narrow set of applications tensor cores could be leveraged. This is because the core concept of a GPU breadth graph search algorithm is a sparse matrix of edges times a sparse vector of nodes. Tensor cores are terrible for that. However, once a graph becomes densely connected and multiple searches are being executed against the same graph simultaneously, there is a realm of possibility. This could be used to generate a solution to the

traveling salesperson problem which is np-complete or worse. Effectively an all to all shortest path algorithm could be run by representing the frontiers of each search together as a matrix. However this is just theorizing, and far beyond the scope of this project.

References:

Team Github Repo: https://github.com/dbs4261/CS535_GraphSearch

Kirk, D., & Hwu, W. (2017). Chapter 12/Parallel patterns: Graph search. In Programming massively parallel processors: A hands-on approach. Cambridge, MA, United States: Morgan Kaufmann, an imprint of Elsevier.

Luo, L., Wong, M., & Hwu, W. (2010). An effective GPU implementation of breadth-first search. In: ACM/IEEE design automation conference (DAC).

Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12). IEEE Computer Society Press, Washington, DC, USA, Article 12, 1–10.

Burkhardt, Paul. (2019). "Optimal algebraic Breadth-First Search for sparse graphs." ArXiv abs/1906.03113