Warp-Centric CUDA Programming for SSSP Algorithms

15-418 Project Proposal

Yiwen(Victor) Song (`yiwenson`), Xingran Du (`xingrand`)

April 11, 2021

1 Summary

We are going to explore the benefits of a warp-centric GPU programming paradigm in optimizing CUDA implementations of two SSSP (Singel Source Shortest Path) algorithms: Dijkstra's and Bellman-ford.

2 Background

Graph algorithms typically involves iterating through graph nodes or edges. Many nodes in the graph can often be processed in parallel, such as the WSP problem from assignment 3&4, so they can benefit from a parallel implementation. In this project, we want to explore solving a particular graph problem - the SSSP (Singel Source Shortest Path) problem, using a particular computer architecture - CUDA programming on NVIDIA GPUs.

Naive CUDA implementations to parallelize graph algorithms would be to assign each node to a task running on a CUDA thread, assuming that the graph algorithm performs the same operation on each node in one iteration. Since GPUs can schedule hundreds or thousands of threads to run at the same time, this allows us to potentially achieve a lot of parallelism. However, GPUs execute instructions in a special way: threads are grouped into warps, and a warp executes as a SIMD program. This means that GPUs' performance suffer greatly from branch divergence and imbalanced workloads, which, unfortunately, are very common in graph problems.

We are exploring the benefits of a warp-centric CUDA programming paradigm targeting workload imbalance caused by irregular accesses, proposed in the paper "Accelerating CUDA Graph Algorithms at Maximum Warp" by Sungpack Hong et al. [1]. This paper presents an idea for CUDA programming that reduces workload imbalance and utilizes locality in memory accesses, including code snippets for an optimized implementation for the BFS algorithm. We will adopt this programming paradigm and try to implement and optimize a CUDA implementation of the

Dijkstra's algorithm, which is an adaptation of the BFS algorithm. At a very high level, Dijkstra's algorithm maintains a mapping of nodes to their distances from the origin and a set of visited nodes; at each iteration, the algorithm visits one additional node that has the shorted distance with the current set of visited nodes.

We will also see if this approach can be generalized to other graph algorithms that are not BFS-based, such as the Bellman-Ford algorithm, which is another algorithm for the SSSP problem. Here we will not be exploring the difference in the scope of problems they can solve; we are only focused on optimizing their GPU implementations. At a very high level, the Bellman-Ford algorithm maintains a mapping of nodes to their distances from the origin; at each iteration, the algorithm updates the mapping by looking at every pair of vertices connected by an edge. The algorithm terminates after at most n iterations.

3 The Challenge

3.1 Workload

The workload is BFS and we will use a RMAT graph as input, which is a generated, irregular graph. In BFS, each level in the tree depends on the previous level, but nodes in the same level could be visited in parallel; divergence happens when some neighbors are already visited while others are not. The memory accesses are highly irregular with low locality, because any two nodes could have an edge between them. Communication to computation ratio could be high if all threads access global memory to access the graph with no explicit caching.

We will adopt this BFS implementation to write a CUDA implementation for the Dijkstra's algorithm. We will not be maintaining a priority queue, but rather follow the simple BFS algorithm and find the shortest path from one of the visited nodes to one of the unvisited nodes. This can be done by doing some kind of a reduce, or updating a global variable in a critical section.
For the Bellman-Ford algorithm, the workload is much more predictable. However, it still suffers from huge imbalance because of the variation in terms of the degrees of the nodes. Another difference is that the Bellman-Ford algorithm does not require a reduction or a critical section. We will implement and benchmark these two algorithms with and without the warp-centric paradigm.

3.2 Constraints

Each node can have varying degrees (number of neighbors) following the power law degree distribution, so if we, for example, naively assign one node to one thread, there could be large workload

imbalance within the warp; memory accesses for following pointers to neighbors can also create high memory bottleneck if the graph is irregular. If there are branch conditions for node operations or choosing which neighbors to explore, threads in a warp can also suffer from divergence.

Using a warp-centric algorithm to decrease divergence and coalesce memory accesses within the warp seems like a natural approach, but since we also used `scan` in assignment 2 for filtering large inputs in parallel based on some condition, we also want to see how this slightly different, data-centric approach works on graphs.

4 Resources

We will follow the Hong paper[1], and we will use the NVIDIA GPUs on GHC cluster machines. We will start coding from the BFS code provided in the paper, but will code the two SSSP algorithms on our own, only using the same optimization idea.

5 Goals and Deliverables

- Implement a naive Dijkstra's algorithm based on BFS implementation in CUDA.
- Implement and benchmark a warp-based version of Dijkstra's algorithm, discussing different approaches such as using the `scan` library in assignment 2 vs. having a critical section. Based on the results presented in the paper [1] at page 276, we should expect to see a 4-6x speedup after applying the warp-centric paradigm.
- Add timer code or do experiments to explore the fine-grained memory access time, barrier wait time, SIMD divergence, etc. to confirm the qualitative statements in the paper about improved work balance and memory locality.
- Implement a naive Bellman-Ford algorithm based on BFS implementation on CUDA.
- Implement a warp-based version of Bellman-Ford algorithm and measure its performance. We are not sure what speedup we should expect for this, but we will compare and discuss the workload difference of these two algorithms along with the benchmark results.
- Stretch goal: implement and benchmark additional optimizations to these algorithms, such as deferring outliers and dynamic workload distribution

6 Platform

Real-world graphs like website pages and the links between them have a large number of nodes. CUDA on GPU allows a high computation throughput, since most of the computation in graph algorithms is often traversing neighbors, which is not expensive but requires a very large throughput if the graph is large. What is more, the operations performed on each node or each neighbor is often identical, so SIMD execution is a natural choice.

7 Schedule

- Apr 11: Implement a naive Dijkstra's algorithm based on BFS implementation on CUDA. Find sample inputs and verify correctness.
- Apr 18: Implement a warp-based version of Dijkstra's algorithm and verify correctness. Consider different possible implementations and benchmark results.
- Apr 25: Work on project checkpoint report. Analyze the effects of work balance and memory locality and the results obtained. Start with the naive CUDA implementation of the Bellman-Ford algorithm.
- May 2: Implement a warp-based version of Bellman-Ford algorithm and measure its performance. Analyze and compare differences.
- May 9: Work on project report. If time permits, work on the stretch goals.

References

[1] Sungpack Hong et al. "Accelerating CUDA Graph Algorithms at Maximum Warp". In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPoPP '11. San Antonio, TX, USA: Association for Computing Machinery, 2011, 267–276. ISBN: 9781450301190. DOI: 10.1145/1941553.1941590. URL: https://doi.org/10.1145/1941553.1941590.