Literature Review

Cheng Liu

2016-10-27

1 Gunrock

Gunrock [7] is a data-centric graph processing framework which mainly handles graphs that can be expressed with iterative convergent processes. It is also based on the BSP model. Basically, the graph problems are divided into steps and each step must be synchronized. Advance-filter-compute are the typical primitive steps used in Gunrock. These operations are performed on top of graph frontiers which is used in many graph processing framework. Major optimization strategies used in Gunrock are listed below.

- Kernel fusion: Inspired by the primitive GPU optimization strategy which leverages the producer-consumer locality between operations, Gunrock tries to integrate advance, filter and user-specific functor into a thread to improve performance as well as memory access efficiency.

- Workload balance: Gunrock's advance step which is applied on graph frontier result in severe load balancing problem especially for graphs with power-law distribution just as any other similar framework. To solve this problem, Gunrock roughly divides the vertices in the frontier into three groups based on the size of the associated neighbor lists. Vertices with larger size of neighbor lists will be executed in parallel in a cooperative thread array (CTA). Vertices with medium size will be executed in parallel in a warp. Vertices with small size will be executed in separate threads. When multiple vertices are assigned to a CTA or warp, the vertex with largest size of associated neighbor list will be distributed to all the threads of the CTA or warp and be executed first. Then all the vertices will be executed sequentially with the same logic.

- Idempotent vs. non-idempotent operations: As vertices in current frontier may share the same neighbors, there are duplicate vertices when producing the next frontier. Gunrock removes some of the duplicate vertices with inexpensive heuristics for applications that allows duplicated vertices in frontier. This strategy helps to improve performance. For applications that can't tolerate duplicated vertices, it can also remove duplicate vertices completely.

- Pull and push traversal: This is the same with the top-down and bottom-up

traversal strategy used in Ligra.

- Priority queue: Usually all the vertices in the frontier are treated equally in BSP model while Gunrock divides them into two queues based on a criterion to save work.

2 Work-efficient Parallel GPU Methods for SSSP

The algorithm developed in this work [3] is based on delta-stepping algorithm. It consists of the following steps.

- The vertices are divided into one or multiple buckets. Vertices within a bucket is processed in parallel.
- Traverse the vertices in a bucket. Load balancing is the key challenge in this step.
- Decide vertices to be processed in next iteration.

This work comes from the same group of Gunrock and it focuses on SSSP optimization on GPUs. Although the optimization techniques used target SSSP, they are generalized and ported to Gunrock. Thus this work is reviewed for more details about the graph optimization techniques. Here are the highlights of this SSSP optimization techniques.

- load balanced graph traversal:

  Group blocking: The edges of the vertices within a block are stripped from each vertex's edge list and processed by a cooperative thread array (CTA) in parallel. Threads within a block is load-balanced, but load-imbalance may still exist between the blocks. Particularly, when the vertex degree is small, this method is not efficient.

  CTA+Warp+Scan [5]: The basic idea is to divide the vertices into three categories based on the size of the edge list. The method is applied in Gunrock as described in last section.

  Edge partition: Instead of grouping equal number of vertices in each block, this method organizes the groups of edges with equal length to ensure strict load balance within a block.

- Work organization which essentially decides the vertices to be processed in next iteration. Again three different methods are proposed.

  Workfront sweep: The basic idea of Workfront is to compute on vertex frontier instead of all the vertice in Bellman-Ford. In addition, with the frontier queue and a vertex id table, redundant vertice in the queue are completely removed to minmize the computing work.

  Near-Far Pile: This is used in Ligra and also known as priority queue strategy in Gunrock. The basic idea is to process some of the vertices in the work queue or frontier first which helps to save the work of the computing. Particularly, the Near-Far method divides the vertices in the work queue in two piles based on the distances to the source. The distance threshold is named as delta which can be customized. When the vertices in the near pile are processed, vertices in the far pile will be analyzed to update the new near pile with the threshold increased by anther delta. Meanwhile, duplicated vertices in the far piles will be removed.

  Bucketing with far pile: This method follows the same design philosophy with Near-Far pile. While delta in near-far method may impose diverse number of vertices in near pile in different iterations. The bucketing method selects a determined number of vertices for the near pile when the number of vertices that meets the distance criteria is than the predefined number (i.e. 110 of the active vertices in this work). The rest active vertices are considered as the far pile. In next iteration, the vertices in the far pile are further selected with a increased distance criteria similarly. This method will ensure the number of near pile varies in a relatively small range which fits the GPU hardware.

By adding priority information to the vertices in the active list, we can avoid updating or computing vertices associated with current vertices with long edges, as they will probably be updated by other paths with smaller edges and more hops. Essentially, this reduces the amount of computing and the overall run-time can be reduced as long as the GPU processing elements remain high utilization.

Here is a summary of relevant SSSP algorithms in reference papers. Some of them are the basis of the algorithms proposed in this work.

Dijkstra algorithm implemented with a priority queue is efficient as a sequential algorithm but expose little parallelism for parallel computing architectures.

PHAST [4]: It has a Dijkstra-like preprocessing step to pre-compute distances to vertices of high-degree. Then the Dijkstra algorithm can start from these highly ranked vertices in parallel. This algorithm works well on low-degree and high-diameter graphs. (More details will be added later.)

Delta Step [6]: Instead of processing one vertex at a time in Dijkstra algorithm, it groups vertices in buckets and process vertices in a bucket in parallel. In delta-stepping, the vertices are grouped into buckets depending on distances of the vertices from the source.

Major challenges for Delta step algorithm on GPUs.

- Delta-stepping's bucket implementation requires dynamic array that can be quickly resized in parallel.
- Fine-grained renaming and moving vertices between buckets are difficult and inefficient.
- GPU memory hierarchy is not well explored.

Bellman-Ford: It is a standard parallel algorithm for SSSP problem. Each vertex maintains the distance to the source and has neighbor vertices information updated iteratively. The algorithm completes when the algorithm converges. This algorithm suffers load imbalance for graphs with power-law distribution. Race condition occurs when the update is parallelized and atomic update is needed.

Algorithms	**Wk. Complexity**	**Type**	**Parallelism**
Dijkstra	$O(v \log v + e)$	General	Serial
Bellman-Ford	$O(ve)$	High Degree	Parallel
Delta Step	$O(v \log v + e)$	General	Coarse Parallel
PHAST	$O(v \log v + e)$	Low Degree	Preprocessing Parallel

Table 1: SSSP Algorithms

2.1 NXgraph

NXgraph [1] is a graph processing framework on a single machine meaning that the original graph is stored in the disk. It follows a few general optimization rules that have been applied in a few different graph processing frameworks including GraphChi, TurboGraph, VENUS, GridGraph.

- Exploit the locality of the graph data
- Utilize the multi-thread of CPU
- Reduce the amount of data transfer
- Stream disk IO

Here are the highlights of NXgraph centering the above four design principles.

- In order to explore the locality of the graph processing, vertices of the graph are divided into intervals and the edges are split in to shards. In each interval, the shards are further divided into sub-shards. Graph computation is performed with granularity of a sub-shard.
- To produce the intervals and shards, the graph needs a two-step pre-processing. In the first step, vertex IDs are re-assigned to produce continuous vertex IDs. The continuous vertex IDs facilitate efficient interval generation and pre-shard generation. In the second step, the pre-shard result are further cut into pieces i.e. sub-shards. Note that the edges in each sub-shards are sorted with the destination vertex IDs.
- NXgraph update strategies: The basic idea is to have two intervals of vertices stored in main memory while the sub-shards are streamed from disk. As the vertices are updated in each iteration and new data are needed in next iteration, vertices must be synchronized. To approach the synchronization cost, each interval has two copies. One copy is used for read and used in current iteration and the other is updated. When the iteration moves forward, the two copies swapped. Thus the synchronization is reduced with the memory cost. In order to support the graphs of which the vertices are large than the main memory, only vertices that are computed will be

loaded into main memory. And the order of the sub-shard computation is carefully organized to make sure disk read/write are minimized. Finally, a mixed solution combined the previous strategy is proposed to balance the performance and the memory overhead.

- Finally, the graph processing performance is modeled and bottleneck of the graph processing system can be found from the analytical models.

2.2 FPGP

FPGP [2] is the simplified version of NXgraph implemented on FPGAs. The most interesting part of this work is to build specific analytical models targeting CPU-FPGA system. The CPU-FPGA system includes multiple FPGA chips with a big shared memory. Meanwhile, each FPGA board has a private DDR. This work basically has vertices stored on the shared memory while shards of edges stored on private memory. According to the experiment, the shared memory bandwidth is not a clear bottleneck, but the private memory bandwidth is currently the major performance bottleneck. Due to the memory bandwidth, the FPGP doesn't show significant performance speedup over the multi-core counterpart.

References

[1] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. Nxgraph: An efficient graph processing system on a single machine. *CoRR*, abs/1510.06916, 2015.

[2] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. Fpgp: Graph processing framework on fpga a case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 105–110, New York, NY, USA, 2016. ACM.

[3] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 349–359, Washington, DC, USA, 2014. IEEE Computer Society.

[4] Daniel Delling, Andrew V Goldberg, Andreas Nowatzyk, and Renato F Werneck. Phast: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

[5] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[6] Ulrich Meyer and Peter Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[7] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 11. ACM, 2016.