Making Caches Work for Graph Analytics

Author(s): Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe

15.1 General Notes

The authors of this paper present a scalable cache-optimized approach to graph analytics (Cagra). The paper recalls previous discussions about the difference between cache-aware and cache-oblivious algorithms, which had the takeaway that (a) I/O operations are extremely costly and optimizing for cache-only random access is incredibly beneficial (especially for graphs, where the last level cache miss rate is 45%, and 60-80% of the time spent by graph algorithms is caught up in memory access according to the author's profiling) and (b) complex memory architectures with multiple levels of cache can be reasonably abstracted into a two-layer disk and cache model (so a cache-level optimization will work even for more complex real-world architectures). The implementation extends the Ligra API for programming graph algorithms (**EdgeMap**, **VertexMap**) with automatic cache optimizations.

Before plunging into the work, I want to briefly note that presenting the results in Figure 1, early in the paper, provides a very compelling conceptual overview. It quickly makes the case that this is a very powerful approach with respect to the baseline, and makes a strong case that the authors successfully approach a practical "lower bound" that has no DRAM access (achieved by writing an incorrect version of the algorithm that only reads from the 0 vertex, which is always in cache). It is also relatively abstract (baseline implementation vs segmented and clustered, in terms of normalized shell cycles), ensuring that readers do not get caught up in technical scrutiny before reading on.

Cagra's primary contribution is a segmented CSR representation. Compressed Sparse Row format is a way of compactly storing graphs as an array of vertices and an array of edges, where the array of edges is composed of a series of subarrays sequentially listing the neighbors of each vertex. Each vertex in the vertex array is stored alongside the index of its first neighbor in the edge array. This format lends itself to cache-efficient neighbor retrieval. Consider a set of vertices. Their neighbors can be loaded from main memory sequentially, as a single batch. This represents a subgraph. The subgraph, stored in cache, can then be cheaply processed by random access with a graph algorithm. Once the batch is processed, the next is read sequentially into memory. Over the course of a single pass of the graph, $O(E)$ edges and $O(V)$ vertices are sequentially read into memory. As multiple subgraphs are formed, some of which will share destination vertices, the authors perform a cache-aware merge once the buffers for individual subgraphs are filled up. This process can also be parallelized and divided across multiple processors.

Cagra can be further optimized by implementing frequency-based clustering. According to the power law, in many real world graphs there will be vertices that have extremely high connectivity compared to other vertices (think of a celebrity in a social network). The clustering technique identifies these superconnectors in preprocessing based on the number of their out-neighbors, and ensures that they are permanently located in fast-access.

This approach has a few advantages. One is that each segment can process all of its vertices in parallel. Another is that the authors do not require write operations to be performed in cache, which can result in poor scalability (2D partitioning) and instead allow writes directly to DRAM as long as they are sequential. This, together with the fact that individual vertices are not processed in parallel, avoids costly atomic synchronization across the graph.

The authors do discuss real-world constraints around cache architectures, noting that smaller segments for lower-level caches will reduce latency but require more merge operations. Following experimentation, they report that the last-level cache (L3) provided the best tradeoff.

Cagra vastly outperforms hand-optimized C++ implementations (up to 3x improvement) as well as compared to other state-of-the art graph processing methods like Hilbert Curve Ordering and GridGraph (which writes in cache) on real world graphs (up to 5x improvement). They find that algorithms like PageRank and Label Propagation experinece meaningful acceleration with segmentation only, and that adding clustering can provide additional speedups by making efficient use of L2 cache.

It would have been nice to see additional analysis performed on different computer architectures (especially ones with different cache sizes, since the segmentation size seems to be very impactful on the performance of this particular algorithm). Further work might explore the application of these methods in distributed settings (since there is no synchronization required, will the speedups be even more meaningful? Or will the additional complexity of network transfers cause unexpected bottlenecks?)