Making Caches Work for Graph Analytics

Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, Matei Zaharia

Fudong Fan, Chris Hoang, Sach Vaidya

Problem Statement

- Graphs can be much larger than cache
  - Working set does not fit into last level cache (LLC)
  - Ex. Twitter Graph: 41 million vertices, 1.5 billion edges has `rank` and `degree` arrays of about **656 MB >> 30-55 MB** LLC of current CPUs
- Access patterns are irregular
  - **60-80%** of cycles stalled on memory accesses
  - Random access to DRAM **6-8x more expensive** than random access to LLC

Motivation

Current graph frameworks

1. Lack of optimizations for cache utilization
    a. Random accesses to a large working set makes entire cache subsystem ineffective
2. Poor multi-core scalability
    a. Existing graph optimizations don't scale beyond 4-6 cores
3. High runtime overhead

Intuition - Frequency Based Clustering

- Many real-world graphs follow power law distribution
    - I.e., a small number of vertices have a large number of edges attached to them
- Pack popular vertices together in memory
- Spatial locality leads to improved cache line utilization
- Caveat: original ordering of vertices often exhibit some locality
    - Use a stable sort when clustering the vertices with above average out-degree

Intuition - Compressed Sparse Row Segmentation

- *Compressed Sparse Row* (CSR) Segmentation
  - Preprocess graph to divide vertices into cache-sized 1D segments
  - Partitions edges into subgraphs based on segments
- Subgraphs are processed in parallel
  - Limit random accesses to the cache
- Intermediate updates are locally merged and stored using buffer to avoid random writes to DRAM
  - All DRAM accesses are sequential
- Combine updates from all buffers within L1 cache

Background - Existing Frameworks

GraphMat, Ligra

- In-memory
- Do not optimize for caches
- GraphMat - fastest published implementation of PageRank, Collaborative Filtering

GridGraph

- Disk-based, optimized for memory/disk boundary
- 3x slower than in-memory frameworks

Background - PageRank

Algorithm used by Google Search to rank web pages in their search engine results

PageRank iteratively updates the rank of each vertex based on the rank and degree of its neighbors. (Pull-based algorithm)

The performance characteristics of PageRank can generalize to a large number of graph applications.

Algorithm 1 PageRank

```
procedure PAGERANK(Graph G)
    parallel for v : G.vertexArray do
        for u : G.edgeArray[v] do
            G.newRank[v] +=
                G.rank[u] / G.degree[u]
        end for
    end parallel for
end procedure
```

Proposed Method - CSR Segmentation

3 Steps:

1. Preprocessing
2. Parallel Segment Processing
3. Cache-Aware Merge

CSR Segmentation - Preprocessing

1. Partition vertices of graph into LLC-sized segments
2. For each segment, construct new subgraph with edges whose source vertices are in the segment

CSR Segmentation - Preprocessing

Algorithm 2 Preprocessing

Size of segment, Original Graph

Input: Number of vertices per segment N, Graph G

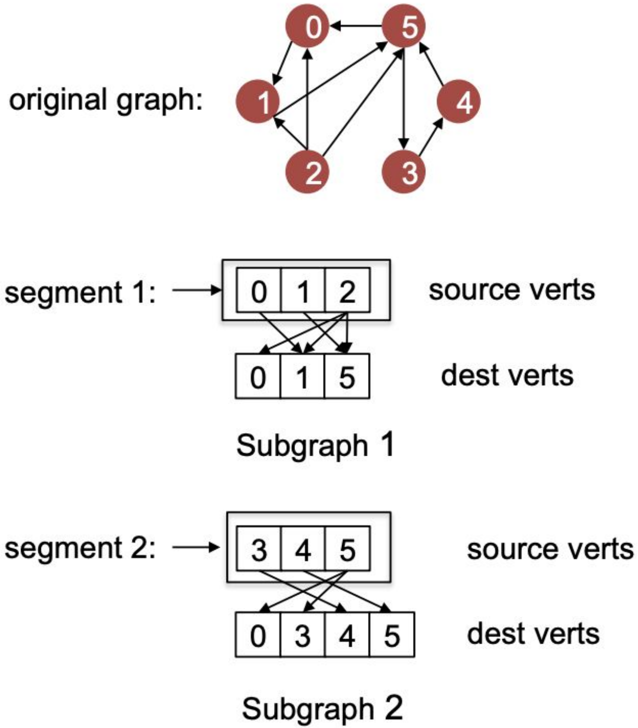
for $v : G.vertices$ **do**
    for $inEdge : G.inEdges(v)$ **do**
        $segmentID \leftarrow inEdge.src/N$
        $subgraphs[segmentID].addInEdge(v, inEdge.src)$
    end for
end for

Find and add this edge to a particular subgraph

for $subgraph : subgraphs$ **do**
    $subgraph.sortByDestination()$
    $subgraph.constructIdxMap()$
    $subgraph.constructBlockIndices()$
    $subgraph.constructIntermBuf()$
end for

Algorithm Metadata:
IdxMap - local index to global index
BlockIndices - block starts/ends for merge step
IntermBuf - store intermediate result for destination vertex

Preprocessing Example



CSR Segmentation - Parallel Segment Processing

1. Process each subgraph
    a. Shared read-only working set

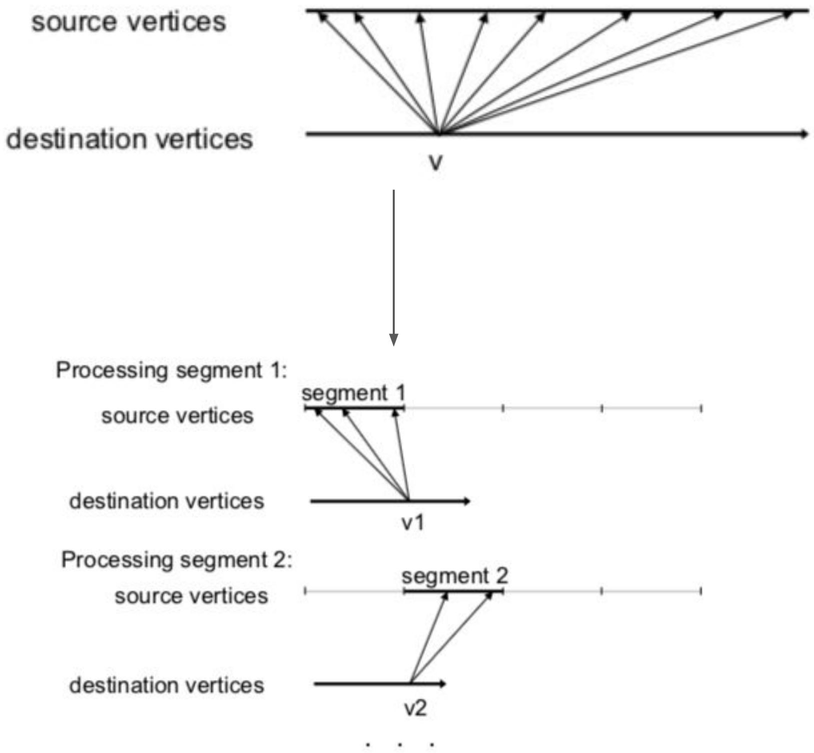
2. Parallelize across different vertices

CSR Segmentation - Parallel Segment Processing

Algorithm 3 Parallel Segment Processing

for $subgraph : subgraphs$ **do** Source Vertices
    parallel for $v : \boxed{subgraph.Vertices}$ **do** Edges to destination vertices
        for $inEdge : \boxed{subgraph.inEdges(v)}$ **do**
            Process $inEdge$
        end for
    end parallel for
end for

Parallel Segment Processing Example



CSR Segmentation - Cache-Aware Merge

- Access intermediate output buffers for each segment sequentially
- Divide range of vertex IDs into cache-sized blocks
- Worker thread reads a range of Vertex IDs from intermediate buffers and updates dense output vector using local to global index mapping (IdxMap)

CSR Segmentation - Cache-Aware Merge

Algorithm 4 Cache-Aware Merge

parallel for $block : blocks$ **do**

Iterate over subgraphs

    for $subgraph : G.subgraphs$ **do**

        $blockStart \leftarrow subgraph.blockStarts[block]$

        $blockEnd \leftarrow subgraph.blockEnds[block]$

        $intermBuf \leftarrow subgraph.intermBuf$

        for $localIdx$ **from** $blockStart$ **to** $blockEnd$ **do**

            $globalIdx \leftarrow subgraph.idxMap[localIdx]$

            $localUpdate = intermBuf[localIdx]$

            merge$(output[globalIdx], localUpdate)$

        end for

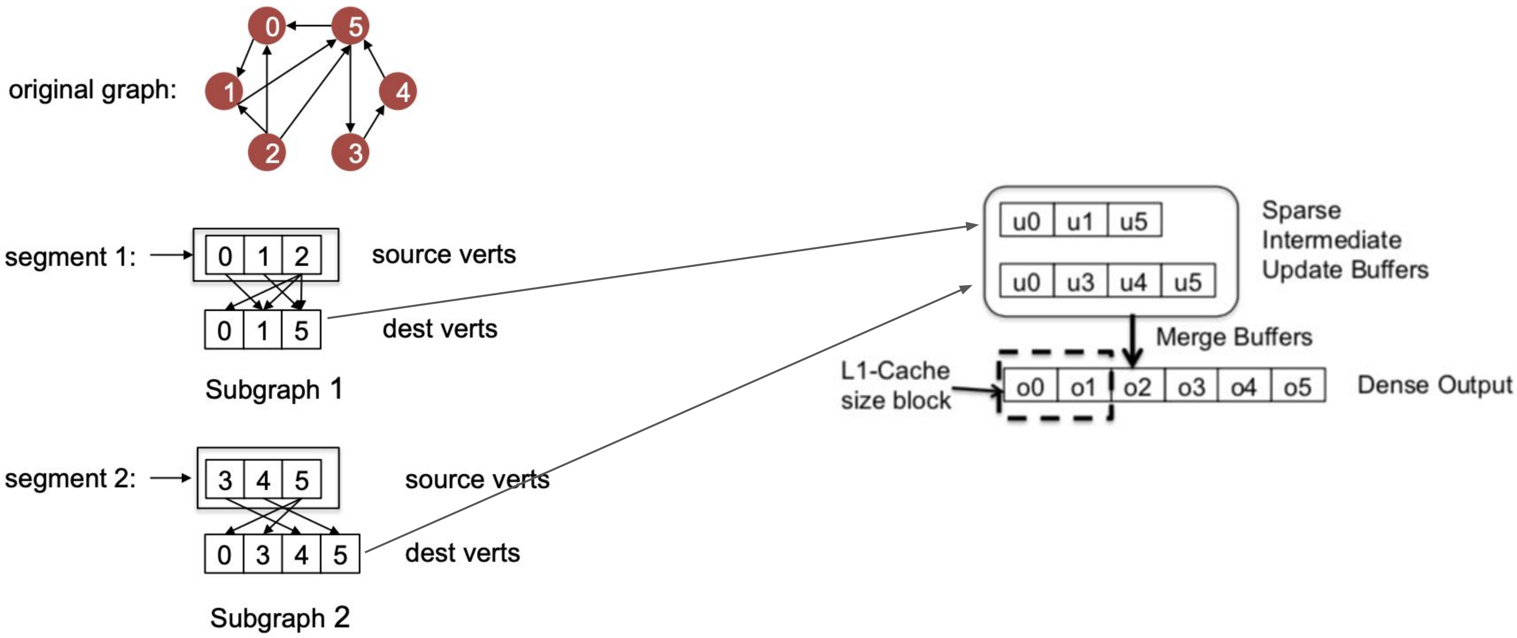
    end for

end parallel for

return $output$

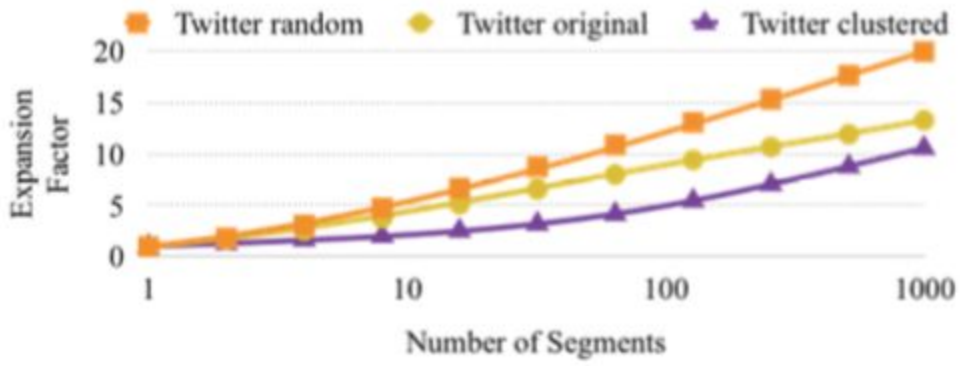
Use idxMap to update global output

Cache-Aware Merge Example



Tradeoff - Segment Size

- Smaller segments will reduce latency (will fit into lower level L1,L2 caches)
- Will require more merges
- Expansion factor - how many segments, on average, contribute data to a vertex (number of merge operations required for each vertex)



Proposed Method - Frequency-Based Clustering

1. Reordering of physical layout of vertex data
    a. Fast cache utilization
2. Out-degree clustering
    a. Vertices with above average out-degree
    b. Locality of original ordering

Evaluation - Setup

1. Datasets
   a. Social networks
   b. Web graphs
   c. Netflix
2. Applications
   a. PageRank, label propagation, collaborative filtering, betweenness centrality
   b. Unpredictable vertex data accesses
3. Comparison
   a. Hand optimized C++ implementations
   b. GraphMat, Ligra, GridGraph

Evaluation - PageRank

Dataset	Cagra	HandOpt C++	GraphMat	Ligra	GridGraph
Live Journal	0.017s (1.00×)	0.031s (1.79×)	0.028s (1.66×)	0.076s (4.45×)	0.195 (11.5×)
Twitter	0.29s (1.00×)	0.79s (2.72×)	1.20s (4.13×)	2.57s (8.86×)	2.58 (8.90×)
RMAT 25	0.15s (1.00×)	0.33s (2.20×)	0.5s (3.33×)	1.28s (8.53×)	1.65 (11.0×)
RMAT 27	0.58s (1.00×)	1.63s (2.80×)	2.50s (4.30×)	4.96s (8.53×)	6.5 (11.20×)
SD	0.43 (1.00×)	1.33 (2.62×)	2.23 (5.18×)	3.48 (8.10×)	3.9 (9.07×)

Evaluation - Label Propagation

Dataset	Cagra	HandOpt C++	Ligra
Live Journal	0.02s (1×)	0.01s (0.68×)	0.03s (1.51×)
Twitter	0.27s (1×)	0.51s (1.73×)	1.16s (3.57×)
RMAT 25	0.14s (1×)	0.33s (2.20×)	0.5s (3.33×)
RMAT 27	0.52s (1×)	1.17s (2.25×)	2.90s (5.58×)
SD	0.34 (1×)	1.05 (3.09×)	2.28 (6.71×)

Evaluation - Collaborative Filtering

Dataset	Cagra	HandOpt C++	GraphMat
Netflix	0.20s (1×)	0.32s (1.56×)	0.5s (2.50×)
Netflix2x	0.81s (1×)	1.63s (2.01×)	2.16s (2.67×)
Netflix4x	1.61s (1×)	3.78s (2.80×)	7s (4.35×)

Evaluation - Betweenness Centrality

Dataset	Cagra	Ligra
LiveJournal	1.2s (1×)	1.2s (1.00×)
Twitter	14.6s (1×)	17.5s (1.19×)
RMAT 25	7.08s (1×)	11.1s (1.56×)
RMAT 27	21.9s (1×)	42.8s (1.95×)
SD	15.0(1×)	19.7 (1.31×)

Evaluation - Analysis of Optimizations

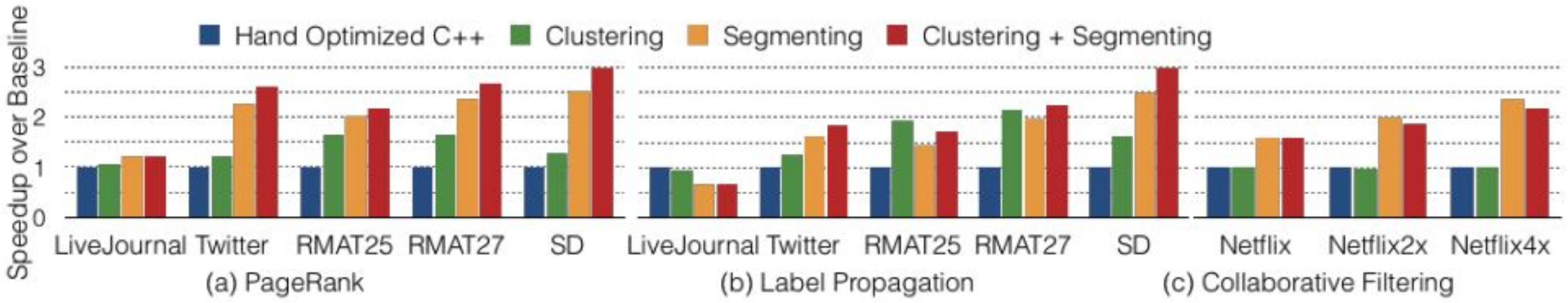
CSR Segmenting

- Eliminate random DRAM access

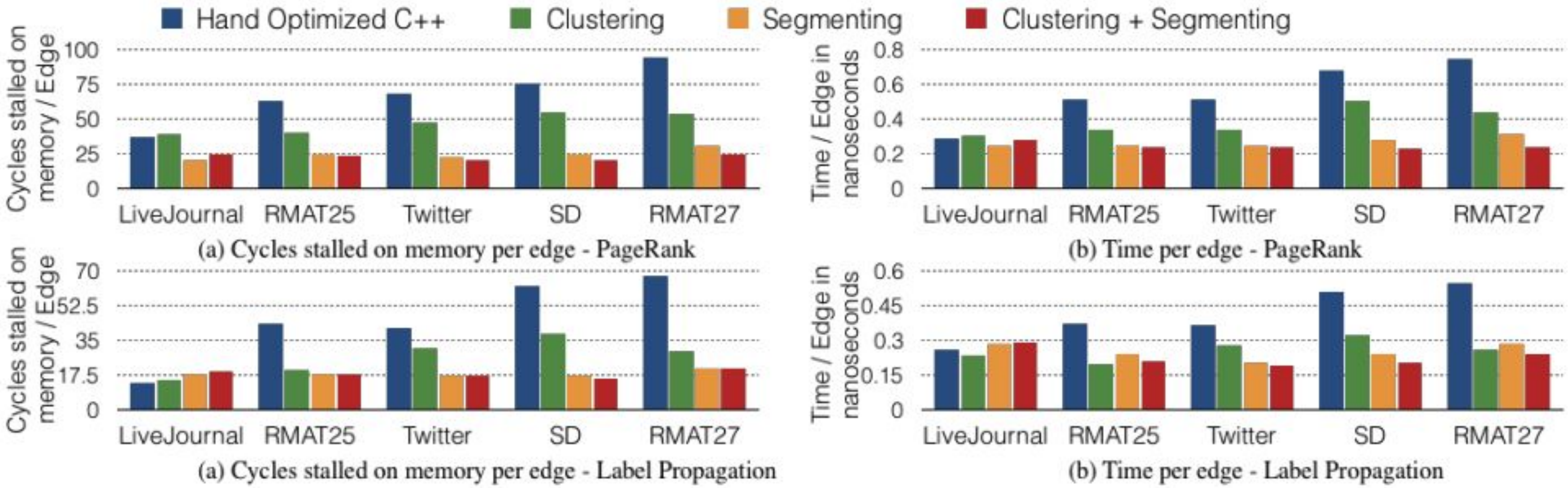
Frequency-Based Clustering

- Take advantage of higher level caches

Evaluation - Comparisons



Evaluation - Comparisons



Strengths

1. Speedups between 3x - 5x
2. Scalability with more cores via segmenting

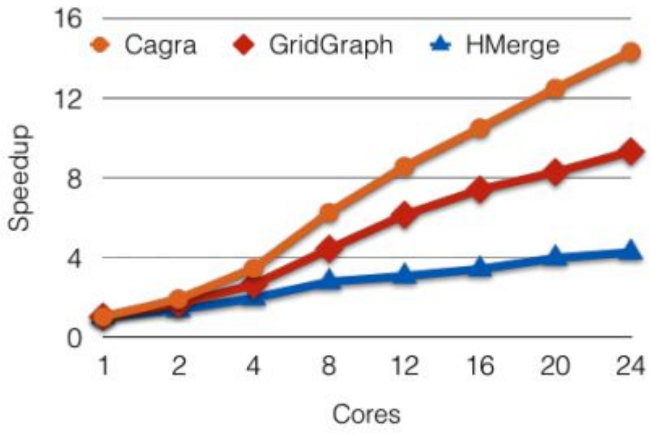


Fig. 10: Scalability for PageRank on Twitter

Weaknesses

1. Requires preprocessing
    a. CSR construction
    b. Vertex clustering
    c. Subgraph partitioning
2. Requires larger graphs to achieve greater speedups

Conclusion

Novel graph framework for improving cache utilization

Techniques

1. Frequency-based clustering
2. CSR segmenting + cache-aware merge

Speedups of up to 5x in comparison to status quo

Preprocessing

Dataset	Clustering	Segmenting	Build CSR
LiveJournal	0.1 s	0.2 s	0.48 s
Twitter	0.5 s	3.8 s	12.7 s
RMAT 27	1.4 s	6.3 s	39.3 s

TABLE VI: Preprocessing Runtime in Seconds.

Frameworks	Cagra	GridGraph	X-Stream
Partitioned Graph	1D-segmented CSR	2D Grid	Streaming Partitions
Sequential DRAM traffic	E + (2q+1)V	E + (P+2)V	3E + KV
Random DRAM traffic	0	0	shuffle(E)
Parallelism	within 1D-segmented subgraph	within 2D-partitioned subgraph	across many streaming partitions
Runtime Overhead	Cache-aware merge	E*atomics	shuffle and gather phase