Making Caches Work for Graph Analytics

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

Reviewed by Miranda Cai
Graph Frameworks Are Limited

- Current graph frameworks do not reach full hardware potential
- Some frameworks store on disk
  - High overhead
- Others store in memory
  - Every access is a random access to DRAM
  - Not cache optimized
- 60-80% of cycles are stalled on memory access
Cagra

Idea: A graph framework that fully utilizes the cache to eliminate all DRAM random accesses and make all DRAM accesses sequential.

Main Contributions:

- CSR Segmenting
  - Partitioning system
- Cagra Framework
  - Ex: PageRank application
- Performance Benefits
Algorithm 2 Preprocessing

Input: Number of vertices per segment N, Graph G
for $v : G$.vertices do
  for $\text{inEdge} : G$.inEdges($v$) do
    segmentID ← $\text{inEdge}.src/N$
    subgraphs[segmentID].addInEdge($v, \text{inEdge}.src$)
  end for
end for

for $\text{subgraph} : \text{subgraphs}$ do
  subgraph.sortByDestination()
  subgraph.constructIdxMap()
  subgraph.constructBlockIndices()
  subgraph.constructIntermBuf()
end for
Algorithm 2 Preprocessing

Input: Number of vertices per segment N, Graph G

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

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

N=3

Segment 0

0 1 2

Segment 1

3 4 5
Preprocessing using CSR Segmenting

**Algorithm 2** Preprocessing

**Input:** Number of vertices per segment N, Graph G

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

for \( subgraph : \text{subgraphs} \) do
  subgraph.sortByDestination()
  subgraph.constructIdxMap()
  subgraph.constructBlockIndices()
  subgraph.constructIntermBuf()
end for

N=3

<table>
<thead>
<tr>
<th>Segment 0</th>
<th>Segment 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2</td>
<td>3 4 5</td>
</tr>
</tbody>
</table>

Diagram:

```
0 -- 5
|       |
1 -- 4
|       |
2 -- 3
```
Preprocessing using CSR Segmenting

**Algorithm 2 Preprocessing**

- **Input:** Number of vertices per segment $N$, Graph $G$
- **for** $v : G\.vertices$ **do**
  - **for** $\text{inEdge} : G\.inEdges(v)$ **do**
    - $\text{segmentID} \leftarrow \text{inEdge.src} / N$
    - $\text{subgraphs}[\text{segmentID}].add\text{InEdge}(v, \text{inEdge.src})$
  - **end for**
- **end for**
- **for** $\text{subgraph} : \text{subgraphs}$ **do**
  - $\text{subgraph}.sortByDestination()$
  - $\text{subgraph.constructIdxMap}()$
  - $\text{subgraph.constructBlockIndices}()$
  - $\text{subgraph.constructIntermBuf}()$
- **end for**

**N=3**

<table>
<thead>
<tr>
<th>Segment 0</th>
<th>Segment 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2</td>
<td>3 4 5</td>
</tr>
</tbody>
</table>

![Graph Diagram](attachment:image.png)
Preprocessing using CSR Segmenting

Algorithm 2 Preprocessing

Input: Number of vertices per segment N, Graph G
for v : G.vertices do
    for inEdge : G.inEdges(v) do
        segmentID ← inEdge.src/N
        subgraphs[segmentID].addInEdge(v, inEdge.src)
    end for
end for
for subgraph : subgraphs do
    subgraph.sortByDestination()
    subgraph.constructIdxMap()
    subgraph.constructBlockIndices()
    subgraph.constructIntermBuf()
end for
Preprocessing using CSR Segmenting

**Algorithm 2 Preprocessing**

*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

for $subgraph : subgraphs$ do
  $subgraph.sortByDestination()$
  $subgraph.constructIdxMap()$
  $subgraph.constructBlockIndices()$
  $subgraph.constructIntermBuf()$
end for
Preprocessing using CSR Segmenting

Algorithm 2 Preprocessing

Input: Number of vertices per segment N, Graph G
for v : G.vertices do
   for inEdge : G.inEdges(v) do
      segmentID ← inEdge.src/N
      subgraphs[segmentID].addInEdge(v, inEdge.src)
   end for
end for
for subgraph : subgraphs do
   subgraph.sortByDestination()
   subgraph.constructIdxMap()
   subgraph.constructBlockIndices()
   subgraph.constructInternBuf()
end for

N=3

Segment 0

0 1 2
0 1 5

Segment 1

3 4 5
0 3 4 5
CSR Segmenting Cache Benefits

Without segmenting, need to load all source vertices

With segmenting, load segments that fit into a cache
Processing Segments in Parallel

Algorithm 3 Parallel Segment Processing

```
for subgraph : subgraphs do
  parallel for v : subgraph.Vertices do
    for inEdge : subgraph.inEdges(v) do
      Process inEdge
    end for
  end parallel for
end for
```

Process in segments since each segment fits in cache. Then every vertex within
the same segment share the same working set.

Return: Fills up subgraph.interimBuf with processed edges.
Algorithm 4 Cache-Aware Merge

parallel for block : blocks do
  for subgraph : G.subgraphs do
    blockStart ← subgraph.blockStarts[block]
    blockEnd ← subgraph.blockEnds[block]
    intermBuf ← subgraph.intermBuf
    for localIdx from blockStart to blockEnd do
      globalIdx ← subgraph.idxMap[localIdx]
      localUpdate ← intermBuf[localIdx]
      merge(output[globalIdx], localUpdate)
    end for
  end for
end parallel for
return output

Interim Buffers

<table>
<thead>
<tr>
<th>Segment 0</th>
<th>Segment 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Output

Blocks are L1-cache sized
Segment Size Selection

- Trade-off when choosing segment size
  - Smaller segments → Lower random access latency, More interim buffer merges
  - Larger segments → Higher random access latency, Less interim buffer merges
- Experiments show L3 cache (LLC) is the best
- Expansion factor metric

\[ q = \frac{s_{\text{adj}}}{s} \]

where \( s = \) no. of vertices per segment, \( s_{\text{adj}} = \) avg no. of edges to segment

\( q \) describes avg no. of segments that contribute data to each vertex, which is same as the no. of merges per vertex
Memory Access Costs Analysis

\( k \) segments

\( q \) expansion factor

\( V/k \) no. source vertices per segment

\( qV/k \) no. interim buffer updates per segment
Memory Access Costs Analysis

$k$  segments

$q$  expansion factor

$V/k$  no. source vertices per segment

$qV/k$  no. interim buffer updates per segment

Phase 1 Traffic: $E + V + qV$

Phase 2 Traffic: $V + qV$
Memory Access Costs Analysis

\(k\) segments

\(q\) expansion factor

\(V/k\) no. source vertices per segment

\(qV/k\) no. interim buffer updates per segment

Phase 1 Traffic: \(E + V + qV\)

Phase 2 Traffic: \(V + qV\)

Total Traffic: \(E + 2qV + 2V\)
Frequency Based Clustering

- Before CSR segmenting, reorder the vertices such that high degree vertices are clustered together
- Only vertices with degree > avg degree get clustered
- Most of the original locality is preserved
- The advantages:
  - Most graphs follow power-law degree distribution
  - Better cache-line utilization
  - Keep frequently accessed vertices in fast cache
PageRank Algorithm

**Algorithm 5 PageRank in Cagra**

```c
typedef double vertexDataType
contrib ← \{1/outDegree[v], ...\}
newRank ← \{0.0, ...\}

procedure EDGEUPDATE(bufVal, srcVal, dstVal)
  bufVal+ = srcVal
  return true
end procedure

procedure MERGE(newDstVal, bufVal)
  newDstVal+ = bufVal
end procedure
```

```c
procedure VERTEXUPDATE(v)
  newRank[v] ← 0.15 + 0.85 \times newRank[v]
  newRank[v] ← newRank[v]/outDegree[v]
  contrib[v] ← 0.0
  return true
end procedure

procedure PAGE RANK(G, maxIter)
  iter ← 0
  A ← V
  while iter ≠ maxIter do
    A ← EdgeMap(G, A, EdgeUpdate, EdgeMerge)
    A ← VertexMap(G, A, VertexUpdate)
    Swap(contrib, newRank)
    iter ← iter + 1
  end while
end procedure
```

Example of easy to implement algorithm using Cagra Interface, some ideas borrowed from Ligra.
Cagra shows up to 5x speedup against the most competitive existing frameworks, and performs better on larger graphs.
Segmenting on its own already provides 2x speedup. Cycles stalled on memory per edge increases for graph size on Hand Optimized C++, but stays consistent for Cagra with segmenting.
Cagra is much more scalable than other cache optimized frameworks like GridGraph and Hilbert ordering ones.
Preprocessing time is insignificant. GridGraph’s preprocessing time was up to 9-11x slower than Cagra’s.
Comparison to Existing Models

- **GridGraph, X-Stream**
  - Use 2D partitioning into subgraphs
  - Some subgraphs can be small → bad scalability
  - High overhead run-times

- **Disk-based systems (GraphChi)**
  - Slow compared to cache optimizations

- **Distributed Systems**

- **Hilbert Ordering**
  - Edge traversal method
  - Cache contention → bad scalability
Conclusion

Strengths

● Novel Approach and optimizations that meshed well together
● Very in depth evaluation

Weaknesses

● Only algorithms with certain features were used for comparison

Future Directions

● Introducing more parallelism
● Minimizing preprocessing time