Green-Marl: A Domain-Specific Language for Easy and Efficient Graph Analysis

Oct 25th 2023
Wenxuan Li
Context & Motivation

- Context: large amount of graph data to be analysed and mined
- Challenges for efficient graph analysis on large-scale graph data:
  - Capacity: limited memory space
  - **Performance**: when run on large scale data
  - **Implementation**: difficult to implement correctly and efficiently
- Main focus: tight coupling between high-level graph analysis algorithm design and underlying hardware architecture
Overview

Green-Marl:

- A high-level domain specific **language**
- An associated **compiler** for producing optimized and parallelized low-level implementation

=> for both performance (optimization) and implementation (decoupling)
Language Design: Domain-specific Syntaxes

- **Data-Types:**
  - Graph
  - Nodes
  - Node_Prop

- **Traversal:**
  - InBFS (BFS)
  - InRBFS (reverse-order BFS)
  - UpNbrs and DownNbrs

---

**Figure 1.** Betweenness Centrality algorithm described in Green-Marl
Language Design: Parallelism

- Implicit Parallelism
  - Group Assignment: `G.BC=0`

- (Explicit) Parallel Execution Region
  - `Foreach`
  - following fork-join style

---

```
Procedure Compute_BC(
  G: Graph, BC: Node Prop<Float>(G)) {
  G.BC = 0;       // initialize BC
  Foreach (s: G.Nodes) {
    // define temporary properties
    Node Prop<Float>(G) Sigma;
    Node Prop<Float>(G) Delta;
    s.Sigma = 1;    // Initialize Sigma for root
    // Traverse graph in BFS-order from s
    InBFS(v: G.Nodes From s)(v!=s) {
      // sum over BFS-parents
      v.Sigma = Sum(w: v.UpNbrs) { w.Sigma};
    }
    // Traverse graph in reverse BFS-order
    InRBFS(v!=s) {
      // sum over BFS-children
      v.Delta = Sum (w:v.DownNbrs) {
        v.Sigma / w.Sigma * (1+ w.Delta)
      };
      v.BC += v.Delta @s; //accumulate BC
    }
  }
}
```

**Figure 1.** Betweenness Centrality algorithm described in Green-Marl
Figure 3. Overview of Green-Marl DSL-compiler Usage
Compiler: Loop Fusion

103    Foreach (s: G.Nodes) (f(s))
104       s.A = X(s.B);
105    Foreach (t: G.Nodes) (g(t))
106       t.B = Y(t.A)

becomes

107    Foreach (s: G.Nodes) (  
108       if (f(s)) s.A = X(s.B);  
109       if (g(s)) s.B = Y(s.A);  
110   )
Compiler: Set-Graph Loop Fusion

139  Node_Set S(G);  // ...
140  Foreach(s: S.Items)
141     s.A = x(s.B);
142  Foreach(t: G.Nodes)(g(t))
143     t.B = y(t.A)

becomes

144  Foreach(s: G.Nodes)(
145     if (S.Has(s)) s.A = x(s.B);
146     if (g(s)) s.B = y(s.A);
147  )
Compiler: Code Generation and Architecture Portability

Compiler emits out target code using code-generation templates

- Example: `Foreach` implementation with backend OpenMP

```c
Foreach(s: G.Nodes)
  For(t: s.Nbrs)
    s.A = s.A + t.B;
```

becomes

```c
OMP(parallel for)
for(index_t s = 0; s < G.numNodes(); s++) {
  // iterate over node’s edges
  for(index_t t=G.edge_idx[s]: t_<G.edge_idx[s+1]; t_++){
      // get node from the edge
      index_t t = G.node_idx[t];
  }
}
```

Allows replacement of backends for code-generations, e.g. CUDA
Evaluation: Productivity

- Measured by Line of Codes (LoC)
- Compared with implementations in existing graph analysis libraries

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC Original</th>
<th>LOC Green-Marl</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>350</td>
<td>24</td>
<td>[9] (C OpenMp)</td>
</tr>
<tr>
<td>Conductance</td>
<td>42</td>
<td>10</td>
<td>[9] (C OpenMp)</td>
</tr>
<tr>
<td>Vertex Cover</td>
<td>71</td>
<td>25</td>
<td>[9] (C OpenMp)</td>
</tr>
<tr>
<td>PageRank</td>
<td>58</td>
<td>15</td>
<td>[2] (C++, sequential)</td>
</tr>
<tr>
<td>SCC(Kosaraju)</td>
<td>80</td>
<td>15</td>
<td>[3] (Java, sequential)</td>
</tr>
</tbody>
</table>

Table 3. Graph algorithms used in the experiments and their Lines-of-Code (LOC) when implemented in Green-Marl and in a general purpose language.
Evaluation: Performance Gain

- Measured by Speed-up with number of threads growing
- Compared with implementations in SNAP library (Bader et al. 2008)
- Ablation study by disabling some optimizations of the compiler (e.g. FlippingEdge)

Figure 4. Speed-up of Betweenness Centrality. Speed-up is over the SNAP library [9] version running on a single-thread. NoFlipBE and NoSaveCh means disabling the Flipping Edges (Section 3.3) and Saving BFS Children (Section 3.5) optimizations respectively.

(Evaluated on randomly synthesized graphs with 32 million nodes and 256 million edges)
Limitations

- No backends supported for distributed environments when the paper was released
  - Later works introduced Pregel (Hong et al. 2014), CUDA (Shashidhar and Nasre. 2017) and MPI (Rajendran and Nandivada 2020)
- No baseline provided for evaluating speed-ups on PageRank and Kosaraju’s algorithm
- Still extra cost for mastering this language
Reference

- Hong et al. 2014. Simplifying Scalable Graph Processing with a Domain-Specific Language
- Shashidhar and Nasre 2017. LightHouse: An Automatic Code Generator for Graph Algorithms on GPUs
- Rajendran and Nandivada 2020. DisGCo: A Compiler for Distributed Graph Analytics
Thank you