Green-Marl
A DSL for Easy and Efficient Graph Analysis
S. Hong, H. Chafi, E. Sedlar, K. Olukotun
Background
Background

• Graph Analysis - Extract information from a graph dataset
Background – Graph Analysis

Strongly Connected Components

![Graph Diagram]

- A
- B
- C
- D
- E
- F
- G
- H
- I
- J
- K
- L
Background – Graph Analysis

Conductance

4 edges cut
20 edge end points
edge conductance = 4 / 20
Background – Graph Analysis

PageRank
Motivation
Motivation

• Large graph datasets
Motivation

- Intuitive Implementation vs Capturing Parallelism

```c
BFS(G, s) {
  initialize vertices;
  Q = {s};
  while (Q not empty) {
    u = RemoveTop(Q);
    for each v ∈ u->adj {
      if (v->color == WHITE)
        v->color = GREY;
        v->d = u->d + 1;
        v->p = u;
        Enqueue(Q, v);
    }
    u->color = BLACK;
  }
}
```

**Algorithm 1. CUDA_BFS (Graph G(V,E), Source Vertex S)**

1: Create vertex array V_a from all vertices and edge array E_a from all edges in G(V,E),
2: Create frontier array F_a, visited array X_a and cost array C_a of size V_a
3: Initialize F_a, X_a to false and C_a to ∞
4: F_a[S] ← true, C_a[S] ← 0
5: while F_a not Empty do
6:     for each vertex V in parallel do
7:         Invoke CUDA_BFS_KERNEL(V_a, E_a, F_a, X_a, C_a) on the grid.
8:     end for
9: end while
Motivation

• Challenges - Capacity, Performance, Implementation
Green-Marl
Green-Marl

• DSL (Domain-specific language)
  • Separation of programming and optimization
  • Intuitive implementation of graph algorithms
  • Exposes data-level parallelism

```java
// Computing Conductance

Procedure conductance(G: Graph, member: N_P<Int>(G), num: Int) : Double {
    Int Din, Dout, Cross;
    Din  = Sum(u:G.Nodes) (u.member==num) {u.Degree()};           // Compute degree sum of inside nodes.
    Dout = Sum(u:G.Nodes) (u.member!=num) {u.Degree()};            // Compute degree sum of outside nodes.
    Cross = Sum(u:G.Nodes) (u.member==num) {
        Count(j:u.Nbrs) (j.member!=num)};
    Double m = (Din < Dout) ? Din : Dout;
    If (m ==0) Return (Cross == 0) ? 0.0 : +INF;
    Else Return (Cross / m);
}
```
Green-Marl

• Simple language constructs
  • Primitive types: Bool, Int, Long, Float, Double
  • Graph types: Undirected, Directed
  • Types bounded to graphs: Node, Edge
  • Collection types: Set, Order, Sequence
  • Traversal Schemes: BFS, DFS

Procedure foo(G1, G2:Graph, n:Node(G1)){
  Node(G2) n2; // a node of graph G2
  n2 = n;  // type-error (bound to different graphs)
  Node.Prop<Int>(G1) A; //integer node property for G1
  n.A = 0;
  Node.Set(G1) S; // a node set of G1
  S.Add(n);
}
Green-Marl

• Deferred assignment and Reductions

```plaintext
Foreach(s:G.Nodes) {
    // no conflict.  t.X gives 'old' value
    s.X <= Sum(t:s.Nbrs) {t.X} @ s
}
// All the writes to X becomes visible simultaneously
// at the end of the s iteration.
```
Optimizations & Compilation
Optimizations

• Architecture independent optimizations
  • Loop Fusion, Hoisting Definitions, Reduction Bound Relaxation, Flipping Edges

Example of Flipping Edges Optimization

133 \texttt{Foreach(t:G.Nodes) (f(t))}
134 \texttt{Foreach(s:t.InNbrs) (g(s))}
135 \hspace{1em} t.A += s.B;

becomes

136 \texttt{Foreach(s:G.Nodes) (g(s))}
137 \texttt{Foreach(t:s.OutNbrs) (f(t))}
138 \hspace{1em} t.A += s.B;

Example of Flipping Edges Optimization
Optimizations

- Architecture dependent optimizations
  - Selection of Parallel Regions, Deferred Assignment and Saving BFS Children, Set-Graph Loop Fusion

```c
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

```c
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   )
```

Example of Set-Graph Loop Fusion Optimization
Compilation

• Into general-purpose languages, e.g. C++ (using graph library)

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

becomes

```
225    OMP(parallel for)
226    for(index_t s = 0; s < G.numNodes(); s++) {
227        // iterate over node’s edges
228        for(index_t t_=G.edge_idx[s]:t_<G.edge_idx[s+1];t_++){
229            // get node from the edge
230            index_t t = G.node_idx[t];
232        } }
```
Experiments & Comparisons
Concise Representation

• Fewer lines-of-code (LOC) for many problems

<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.
Experiments

- Betweenness Centrality
- Compared to SNAP library

```plaintext
Procedure Compute_BC(
    G: Graph, BC: NodeProp<Float>(G)) {
    G.BC = 0; // initialize BC
    ForEach(s: G.Nodes) {
        // define temporary properties
        NodeProp<Float>(G) Sigma;
        NodeProp<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 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.
Experiments

• Conductance

Figure 5. Speed-up of Conductance. Speed-up is over the SNAP library [9] version running on a single-thread. NoLM and NoSRDC means disabling the Loop Fusion (Section 3.3) and Reduction on Scalars (Section 3.5) optimizations, respectively.

• Vertex Cover

Figure 6. Speed-up of Vertex Cover implemented in Green-Marl and two versions of the corrected SNAP implementation SNAP which had a data-race. The first version, SNAP(correct) utilizes a simple locking approach. The second version, SNAP(optimized), uses a more advanced test and test-and-set scheme. A small instance (100k nodes, 800k edges) was used in this experiment.
Critique
Major Contributions

1. Intuitive, concise implementation of algorithms
2. Transparent, automatic optimizations through compilation
3. Wider range of optimizations using domain-specific knowledge
4. Architecture-dependent optimizations
5. High architecture portability
6. Easy integration into current workflow
Criticism

1. Limited to graphs which fits into RAM
2. Backend optimized for CPU execution only
3. Limited comparison with related works
Conclusion

• A domain-specific language that is
  • Portable
  • Concise and intuitive
  • Efficient
  • Easy to integrate into workflow

• Require more work on
  • Scalability
  • Performance evaluation