Elixir
A System for Synthesizing Concurrent Graph Programs

Prountzos D., Manevich R. & Pingali K.

Christopher Little
Motivation

Best solution to problems depends on:

- Data
- Machine Architecture
- Intra-algorithm tuning
- ...

**Dream:** let the compiler worry about it all
Running Example: SSSP
(Single-Source Shortest Path)

Dijkstra

Bellman-Ford
Graph Algorithm

Operators

Schedule

Order Activity Processing

Identify New Activities

Static Schedule

Dynamic Schedule
SSSP Elixir Specification

Graph [
    nodes(node: Node, dist: int)
    edges(src: Node, dest: Node, wt: int)
]

relax = [ nodes(node a, dist ad)
    nodes(node b, dist bd)
    edges(src a, dest b, wt w)
    bd > ad + w ] ->
    [bd = ad + w]

sssp = iterate relax >> schedule
SSSP Elixir Specification

Graph [ nodes(node: Node, dist: int) edges(src: Node, dest: Node, wt: int) ]

relax = [ nodes(node a, dist ad) nodes(node b, dist bd) edges(src a, dest b, wt w) bd > ad + w ] -> [bd = ad + w]

sssp = iterate relax >> schedule
SSSP Elixir Specification

Graph [  
    nodes(node: Node, dist: int)  
    edges(src: Node, dest: Node, wt: int)  
  ]

relax = [  
    nodes(node a, dist da)  
    nodes(node b, dist db)  
    edges(src a, dest b, wt w)  
    db > da + w ] ->  
    [db = da + w]

sssp = iterate relax >> schedule
Scheduling

- Metric
- Group
- Fuse
- Unroll
- Ordered/unordered
Graph Algorithm

Operators

Schedule

Order Activity Processing

Static Schedule

Dynamic Schedule

Identify New Activities
assume \((da + w < db)\)
assume \(!((dc + w' < db))\)
new\(_{db}\) = \(da + w\)
assert \(!((dc + w' < new\_db))\)
assume (da + w < db)
assume !(db + w' < dc)
new_db = da + w
assert !(new_db + w' < dc)
Evaluation
Experiments

Explored Dimensions

group
unroll k
dynamic scheduler

Statically group multiple instances
Statically unroll operator applications
different worklist policy/implementation
...

...
Definition 3.1 (Graph). A graph $G = (V^G, E^G, \text{Att}^G)$ where $V^G \subseteq \text{Nodes}$ are the graph nodes, $E^G \subseteq V^G \times V^G$ are the graph edges, and $\text{Att}^G : \left((\text{Attrs} \times V^G) \rightarrow \text{Vals}\right) \cup \left((\text{Attrs} \times V^G \times V^G) \rightarrow \text{Vals}\right)$ associates values with nodes and edges. We denote the set of all graphs by Graph.

Definition 3.2 (Pattern). A pattern $P = (V^P, E^P, \text{Att}^P)$ is a connected graph over variables. Specifically, $V^P \subseteq \text{Vars}$ are the pattern nodes, $E^P \subseteq V^P \times V^P$ are the pattern edges, and $\text{Att}^P : \left((\text{Attrs} \times V^P) \rightarrow \text{Vars}\right) \cup \left((\text{Attrs} \times V^P \times V^P) \rightarrow \text{Vars}\right)$ associates a distinct variable (not in $V^P$) with each node and edge. We call the latter set of variables attribute variables. We refer to $(V^P, E^P)$ as the shape of the pattern.

Let $\mu_R$ and $\mu_{R'}$ be two matchings corresponding to the operators above. We say that $\mu_R$ and $\mu_{R'}$ overlap, written $\mu_R \land \mu_{R'}$, if the matched subgraphs overlap: $\mu_R(V^R) \cap \mu_{R'}(V^R') \neq \emptyset$. Then, the following equality holds:

\begin{align*}
\text{Def}[\text{op}, \text{op}'] (G, \mu_R) = \\
\text{let } G' = \left[\text{op}\right] (G, \mu_R) \\
\text{in } \{ \mu_{R'} \mid \mu_{R'} \land \mu_R, \quad \\
(G, \mu_{R'}) \neq \text{Re}^{op}, \text{Gd}^{op}, \\
(G', \mu_{R'}) \models \text{Re}^{op}, \text{Gd}^{op} \} .
\end{align*}
Conclusion

- Elixir can beat hand-written implementations
- “High-level” specification could be simpler
- Not very accessible paper (unhelpful formalisms)
- Dynamic graphs unsupported
- Is auto-tuning integrated yet?