Deep Learning-based Approximate Graph-Coloring Algorithm for Register Allocation
- Dibyendu Das, Shahid Asghar Ahmad, Venkataramanan Kumar

LLVM-HPC2020 Workshop

Nov 12, 2020 • Atlanta, Georgia
Agenda

- Basic Concepts and Motivation
- Our DL-method using LSTM
- LLVM’s GRA vs our method
- Results
- Conclusion and Future Work
Graph-coloring is an important problem in CS with numerous applications.

A graph coloring is an assignment of labels, called colors, to the vertices of a graph such that no two adjacent vertices share the same color.

The chromatic number $\chi(G)$ of a graph $G$ is the minimal number of colors for which such an assignment is possible.

Solved using heuristics.

A graph coloring for a graph with 6 vertices. It is impossible to color the graph with 2 colors, so the graph has chromatic number 3.
Register Allocation as a Graph-Coloring Problem

- Register Allocation is an important problem in the area of compiler code generation
- Usually the number of registers available < number of variables used
- Create what is known as the *interference graph* which models registers which need to be *live* at the same time

```plaintext
live-in:  k  j
  g := mem[j+12]
  h := k - 1
  f := g * h
  e := mem[j+8]
  m := mem[j+16]
  b := mem[f]
  c := e + 8
  d := c
  k := m + 4
  j := b

live-out:  d  k  j
```

Example:
- Register allocation (RA) is an important application of graph coloring in compilers
  - Heuristics + lots of tuning
- Idea is to see whether we can build a simple deep-learning model that can color graphs and can be used for RA
- We build an LSTM-based model and experiment with some popular graphs as well as interference graphs generated by LLVM’s RA
Modeling Graph coloring using LSTMs

- Viewed as a sequence-2-sequence translation via LSTMs
- An input sequence where each item of the sequence corresponds to a node of the graph
  - Each item of the input sequence is an encoding of the adjacency vector of that vertex
- The output sequence is of the same length as the input sequence (number of nodes of the graph)
  - Each item of the output sequence is a *color* assignment for that node
- Adjacent nodes cannot carry the same color
  - But it is difficult to encode constraints in LSTM
- It is difficult to get perfect coloring for large graphs
  - Even perfect coloring for graphs with several hundred nodes is very time-consuming
- Three stacked layers of LSTM followed by a FC(dense) layer and ReLU
- The figure shows an unrolled LSTM

Adj Matrix

\[
\begin{bmatrix}
1 & 1 & 0 & 0 & \ldots \\
0 & 1 & 0 & 1 & \ldots \\
1 & 0 & 0 & 1 & \ldots
\end{bmatrix}
\]

1024 hidden units per LSTM

0 ≤ Color(v) ≤ n-1
Training the model (1)

- The aim is to apply such a model for moderate-sized graphs (few hundred nodes)
- Many interference graphs created by LLVM’s RA for SPEC CPU 2017 are of similar size
- Use supervised training by generating random graphs and coloring them optimally
  - Used a package called *nauty* (http://keithbriggs.info/very_nauty.html)
  - Generated random graphs up to 100 nodes of varying sparsity (very sparse to very dense)
  - Colored them optimally and generated the color assignments for the nodes
  - Used ~10000 samples for training
- Used 2 LONG INTS (128 bits) to encode the adjacency vertex of a node
Training the model (2)

- One sample of the input+output sequence for training looks like this:
  - \(<v_0_{\text{first 64bits}}, v_0_{\text{second 64bits}}>, \ldots, <v_{99}_{\text{first 64bits}}, v_{99}_{\text{second 64bits}}>, <\text{color}(v_0), \text{color}(v_1), \ldots, \text{color}(v_{99})>\>

- Sequences may be padded with zeros for graphs having a smaller number of vertices
- The model is trained for 100 epochs
- Up to a training error value of 5%
For inference the input needed is just the adjacency vector encoding

\[ <\langle v_0_{\text{first\_64bits}}, v_0_{\text{second\_64bits}}\rangle, \ldots, \langle v_{99}_{\text{first\_64bits}}, v_{99}_{\text{second\_64bits}}\rangle, 0, 0, \ldots, 0 > \]

However there is a big catch with the inference output

- The coloring assignment may be **INVALID**
- Since constraints are not encoded in LSTM adjacent nodes having the same color can happen

Our solution:

- Use a post-pass called *color correction* to restore the validity of color assignment
  - May need additional colors
- Forest-Fire graph
  - has a chromatic number of 5
- LSTM-based model colors with 4 colors resulting in 2 invalid edges
- These edges are $v_2, v_3$ and $v_1, v_5$
- $v_5$ can reuse the color $c_3$ as none of its neighboring nodes use $c_3$. $v_2$ requires a new color $c_5$ as both $v_2$ and $v_3$'s neighbors use all the colors
- So finally we also get 5 colors 😊
Inference performance on some popular graphs

- Karate Graph – X(G) – 5
  - LSTM uses 4 colors but 23 invalid edges out of 79 edges which is ~30% of the edges. On applying color correction, we use only one extra color resulting in coloring the graph optimally using 5 colors.

- Baidu Graph – X(G) – 3
  - LSTM uses 3 colors with 35 out of 90 edges being invalid which is ~38%. We are unable to reach the optimal number of 3 colors. But we can match the coloring number of the best heuristics available today by being able to color with 4 colors (after correction).
GRA does not maintain an interference graph explicitly.
We create the interference graph at the end of the Live Interval Analysis phase.
The interference graph is then written out in the input format required for inferencing.
We collected the interference graphs for the functions of certain SPEC CPU® 2017 benchmarks. We ignored those functions which have more than 100 nodes in the interference graph.
We collect the register count of each function after codegen.
## DL-model vs LLVM’s GRA

### Some results of mcf and leela

<table>
<thead>
<tr>
<th>Functions</th>
<th>LLVM reg-alloc</th>
<th>DL before correction</th>
<th>DL after correction</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch_arcs</td>
<td>17</td>
<td>14</td>
<td>22</td>
</tr>
<tr>
<td>replace_weakerArc</td>
<td>16</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>insert_newArc</td>
<td>14</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>resize_prob</td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>marc_arcs</td>
<td>12</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>refreshPositions</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>refreshArcPositions</td>
<td>8</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>master</td>
<td>23</td>
<td>13</td>
<td>24</td>
</tr>
<tr>
<td>worker</td>
<td>25</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>markBaskets</td>
<td>11</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>primal_bea_mpp</td>
<td>21</td>
<td>14</td>
<td>26</td>
</tr>
<tr>
<td>primal_feasible</td>
<td>9</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>flow_org_cost</td>
<td>14</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>flow_cost</td>
<td>13</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>refresh_neighbour_lists</td>
<td>10</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>update_tree</td>
<td>19</td>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>primal_start_artificial</td>
<td>11</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>primal_infeas</td>
<td>7</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>write_objective_value</td>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>main</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>262</strong></td>
<td><strong>174</strong></td>
<td><strong>257</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>LLVM reg-alloc</th>
<th>DL before correction</th>
<th>DL after correction</th>
</tr>
</thead>
<tbody>
<tr>
<td>lzma_index_buffer_decode</td>
<td>11</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>index_decode</td>
<td>12</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>lzma_index_hash_append</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>lzma_index_hash_decode</td>
<td>14</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>lzma_stream_buffer_decode</td>
<td>14</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>stream_decode</td>
<td>14</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>lzma_stream_footer_decode</td>
<td>8</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>lzma_vli_decode</td>
<td>13</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>lz_encoder_prepare</td>
<td>13</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>lzma_lz_encoder_init</td>
<td>11</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>lz_encode</td>
<td>14</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>lzma_mf_hc3_find</td>
<td>21</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>732</strong></td>
<td><strong>500</strong></td>
<td><strong>715</strong></td>
</tr>
</tbody>
</table>
An architecture of a DL-based RA

- A plausible setup for LLVM
Conclusions + Future Work

- Preliminary and exploratory work to check whether an LSTM-based solution is feasible
  - Looks promising
- Baidu (2018) solution using Reinforcement Learning but costly and complicated
- One work using Graph NN (but not for color assignment)
- Can experiment with sequence input orders (ex: Topo-sort, DFS, BFS)
- Can experiment with newer LSTMs like Attention/Transformer
- Can use LLVM’s graphs as training inputs rather than random graph