Register Allocation via Graph Neural Networks

Project Report for EECS 583: Advanced Compilers (Fall 2023)

Anuj Tambwekar, Austin Nguyen, Creighton Glasscock, Jacob Sansom
{anujt, ngaustin, creiglas, jhsansom}@umich.edu
Computer Science and Engineering
University of Michigan, Ann Arbor

***Abstract*—Register allocation is a critical component of any compiler and remains primarily heuristic-driven. Viewing register allocation as a graph coloring problem opens up multiple possibilities for improving allocation strategies. Still, the NP-Hard nature of the graph coloring problem makes coming up with efficient solutions challenging. In this project, we use graph neural networks to color interference graphs and obtain physical register assignments for virtual registers. More formally, given an interference graph G, and a fixed number of colors K, our GNN algorithm colors each node of G with one of the K colors, such that no two connected nodes are colored the same color, while attempting to minimize the number of spills needed to ensure the coloring is valid. We find that while our approach outperforms random coloring, it is still frequently outperformed by Chaitin's algorithm. However, we do find that we can outperform Chaitin's occasionally on smaller graphs with low values of K.**

I. Introduction

Register allocation is the process of mapping virtual registers in the intermediate representation of code to a physical hardware register. LLVM has two main register allocators built-in - linear scan and greedy. Both these algorithms compute live ranges of variables but do not explicitly treat the problem as a graph coloring problem. Rather, they use queues of live ranges and assign registers to them based on an available pool, checking to see if an assignment can be done using the available registers or if the virtual register must be spilled to memory.

Chaitin's algorithm approaches register allocation as a constrained graph coloring problem. Virtual registers are treated as graph nodes, with edges between nodes indicating overlapping live ranges. This graph, known as the interference graph, is then colored using K colors, where K is the number of physical registers available for the algorithm to use. Chaitin's algorithm uses a greedy approach to first color nodes of higher priority. This priority is computed using profile data, but measuring the number of lines over which the virtual register is accessed and dividing it by the number of virtual registers it overlaps with. While Chaitin's algorithm is efficient, its priority mechanism is built on having profile data. The greedy nature of the algorithm can result in suboptimal coloring, particularly when profile data is unavailable for prioritization.

In this project, we propose that Graph Neural Networks (GNNs) can decrease our reliance on profile data and perform efficient register allocations without the need for a fully heuristic-driven algorithm. By using a combination of spectral embeddings and soft-kmeans clustering, we train a graph neural network to cluster nodes of a graph into groups based on their distance across the graph, with nodes that are farther apart in the graph being closer together in embedding space. These clusters of nodes are then mapped to physical register labels where we choose to assign nodes to a particular register or spill it instead.

II. Background and Related Work

A. Machine learning for register allocation

Machine learning for register allocation is a new emerging topic of interest. The most recent approaches can be split into reinforcement learning-based algorithms and deep-learning approaches. In the reinforcement learning approaches [1], [2], an agent learns to estimate an optimal sequence of actions by multiple rounds of self-play. [1] explicitly targets an RL technique for register allocation in LLVM, whereas [2] uses RL to tackle the graph coloring problem at large. These approaches are usually computationally expensive and rather demanding, resulting in long compile times. However, [1] is a relatively new work that does show a lot of promise. The deep-learning approaches operate on the entire interference graph and aim to create a machine-learning model that can map an input graph to a set of colors. [3] is our main reference for this approach. They use an LSTM [4] model that captures the history and sequential relationships between nodes. By modeling the graph as a sequence of adjacency vectors, their approach converts a sequence of adjacency vectors into a sequence of colors. While their approach did achieve lower total register usage compared to LLVM's register allocation, the training regime of this sequence modeling approach is flawed as it fails to preserve the equivariance of a coloring. In a register allocation scenario, the final color given to a node is not important - the set of nodes with the same color determines the efficiency of the approach. The LSTM-based sequence modeling approach is unable to capture this invariance. As such, in this project, we focus on a method aimed at creating groups of nodes, where the learning objective is to learn how to group nodes together into the same color, as opposed to worrying about what color the group must be assigned.

B. Graph Neural Networks

Graph Neural Networks are a subset of neural networks designed specifically to operate on graphs by being permuta-

tionally equivariant. The input to the GNN is a set of vertices and edges, and the GNN's primary objective is to obtain embeddings for these nodes for further use in downstream tasks, such as classification or clustering. GNNs have two key types of layers - message passing layers and pooling layers. Message-passing layers connect nodes together by performing a permutation invariant operation on the features of all adjacent nodes. Simply put, the message passing layer aggegrates all the information around a node. This message-passing step can be repeated across multiple layers, creating a mechanism similar to the forward pass of a regular deep learning layer, with the additional benefit of being invariant to the order of the features. The pooling layers are similar to those found in a convolutional network and allow downsampling of the features in the message-passing layer.

III. Using GNNs for Register Allocation

A. GNN Architecture

The GNN architecture begins by converting a graph into a set of spectral embeddings, one per node. Spectral embeddings are equivariant over node permutations, meaning that the order in which nodes are labeled imparts no difference on the end result of the coloring. These spectral embeddings are then fed into a transformer model—a popular neural network architecture for arbitrary-length sequences—to yield one embedding per node. Finally, a soft K-means clustering algorithm is used to group these embeddings into K distinct groups, which subsequently become our colors. Unlike a conventional K-means clustering algorithm, soft K-means provides a probability value for each embedding-cluster pair representing the probability that the embedding belongs to the cluster. The coloring itself can be obtained by simply choosing the most probable color for each node.

To train this network, we utilize a loss function that penalizes the network for assigning bad colors and rewards it for assigning good colors. In particular, we define a node's color as "good" if it has no neighbors assigned the same color. Accordingly, a "bad" color occurs when a node has at least one neighbor assigned the same color. The intention behind this approach is to teach the network how to create embeddings that, when clustered, induce a good coloring. The loss function is shown in Equation 1 and the overall training algorithm is shown in Algorithm 1.

$$L(\theta) = -\sum_{i \in \mathcal{G}} \log P(c|n_i) + \sum_{i \in \mathcal{B}} \log P(c|n_i) \tag{1}$$

In the above equation, $\mathcal{G}$ and $\mathcal{B}$ refer to the sets of good and bad nodes, respectively. The probability values $P(c|n_i)$ are the probability assigned to color c for node n_i. Simply described, this loss function minimizes the probability assigned to all bad colors and maximizes the probability assigned to all good colors.

At testing time, the algorithm feeds the trained model G_θ an adjacency matrix A and executes Soft K-Means on the outputted embeddings V_θ to produce an initial coloring C, assigning the highest probability color for each node. Note that the coloring C may not be a valid coloring; there is no mechanism to ensure that neighboring nodes are not assigned the same color. As a result, we must apply corrections, which we opt to do by spilling any nodes with similarly colored neighbors in order of spill cost (lower spill costs first). The process is repeated for each color until the mapping C is a valid coloring. This is detailed in Algorithm 2.

Algorithm 1 GNN Training Algorithm

Inputs: Adjacency Matrix Dataset $\mathcal{M}$, Number of Physical Registers K, Epochs E
Outputs: Trained Model $\mathcal{G}_\theta$

for Epochs E **do**
  Get randomized data loader $\mathcal{D}(\mathcal{M})$
  for adjacency matrix batch $\{A_i\}_{i=1}^{b} \sim \mathcal{D}$ **do**
    Get embeddings $V_\theta = \mathcal{G}_\theta(\{A_i\}_{i=1}^{b})$
    Calculate $L(\theta)$ using SoftKMeans(V_θ) and Equation 1
    $\theta = \theta - \nabla L(\theta)$
  end for
end for

Algorithm 2 GNN Test-time Logic

Inputs: Trained Model $\mathcal{G}_\theta$, Adjacency Matrix A , Number of Physical Registers K, Spill Costs $\mathcal{C}$
Outputs: Coloring Map C, Spill Set S

Get embeddings $V_\theta = \mathcal{G}_\theta(A)$
Get initial coloring C =SoftKMeans(V_θ)
Initialize spill set $S = \emptyset$
for each color c of total K **do**
  Define V_c where $v \in V_c \Leftrightarrow C[v] = c$
  while any virtual registers with color c are neighbors **do**
    $v_{min} = \text{argmin}_{v \in V_c} \mathcal{C}[v]$
    C[v] = *spill*
    $S \leftarrow S \cup v$
  end while
end for
  return Coloring C

B. Training Pipeline

To train this GNN, we use randomly-generated synthetic graphs. For each synthetic graph, we let our GNN obtain a soft coloring, and then we use the loss function outlined in Equation 1 to adjust the GNN weights to improve future performance.

When creating graph colorings, it is reasonable to assume we will only be dealing with graphs where each node is only connected to at least one other node. In the event that a real-world graph did not possess this property (i.e., at least one node in the interference graph is an "island", overlapping with no other live range), it could be assigned an arbitrary color

without spilling. Thus, we only generate graphs where each node is connected to at least one other node.

C. Extracting Interference Graphs from Code

While synthetic graphs are sufficient for training our model, synthetic graphs are not necessarily sufficient for the validation step. The synthetic graphs cannot be assumed to be perfectly representative of real interference graphs in characteristics such as rank, sparsity, and so on. As such, in order to demonstrate how our GNN specifically performs on register allocation tasks, rather than just generic graph-coloring tasks, it is necessary that we validate our model on interference graphs generated from real code. While there do exist several datasets for general graph-coloring approaches, to our knowledge, there is no publicly available dataset consisting of representative interference graphs for the study of benchmarking register allocation approaches.

Given this lack of availability, we found it necessary to develop a novel pipeline involving a custom LLVM pass in order to generate these real interference graphs from any repository of source (.c) files. Our LLVM pass generates an adjacency matrix representation of the interference graph of any given source file. This pass is divided into three parts: live range analysis, instruction frequency analysis, and matrix creation.

A virtual register's live range is defined as the instructions in the intersection of its liveness and reaching definition sets. Both of these sets were manually constructed using classical methods (GEN, KILL, IN, OUT sets) and combined to create each live range. Static-single assignment (SSA) allowed some simplications in our code. For example, KILL sets for reaching definitions were set to the empty set, as SSA enforces that a virtual register can only be assigned to once, implying its definition can never be killed.

With live ranges constructed, the pass then determines edges, edge weights, and node spill costs. An edge between two nodes is present when there is a non-empty intersection between the two corresponding live ranges. Edge weights are calculated by multiplying each instruction in the intersecting live ranges each by their execution frequencies (provided by LLVM's Block Frequency Information) and summing the result. Node spill costs are similarly calculated: multiplying each instruction in a live range by their execution frequencies and summing the result.

The last step of the LLVM pass is to populate the adjacency matrix. Our adjacency matrix is slightly different from common convention. Diagonal elements denote the spill costs of nodes while off-diagonals denote the edge weights. For example, entry (i, j) corresponds to the edge weight between node i and j. If there is no edge between two nodes, the entry has value -1. We allow edge weights to have weight 0, which corresponds to a non-empty intersection of live ranges that was never executed during profiling. This case must be accounted for as a lack of execution in profiling does not imply its lack of execution in subsequent runs. This adjacency is subsequently written to a file for later use by the GNN training pipeline.

Algorithm 3 Interference Graph Generation

Inputs: InstructionFrequencyInformation IFI, Number Virtual Registers n
Outputs: Adjacency Matrix A

Initialize live $\mathcal{L}$, reaching definition $\mathcal{D}$, and live range $\mathcal{R}$ sets to $\emptyset$.
Initialize spill cost $\mathcal{C}$ and edge weight $\mathcal{E}$ sets to $\emptyset$.
Initialize empty square matrix A of size n
for v virtual register **do**
  $\mathcal{L}[v] \leftarrow LiveAnalysis(v)$
  $\mathcal{D}[v] \leftarrow ReachingDefsAnalysis(v)$
  $\mathcal{R}[v] \leftarrow \mathcal{L}[v] \cap \mathcal{D}[v]$
end for
for (v, u) pair of virtual registers **do**
  $\mathcal{R}_{vu} \leftarrow \mathcal{R}[v] \cap \mathcal{R}[u]$
  Execution frequency $E \leftarrow \sum_{i \in \mathcal{R}_{vu}} IFI(i)$
  if v = u **then**
    $\mathcal{L}[v] = E$
  else
    $\mathcal{E}[v][u] = E$
    $\mathcal{E}[u][v] = E$
  end if
end for
for entry A_{ij} in A **do**
  if i = j **then**
    $A_{ii} \leftarrow \mathcal{L}[i]$
  else
    if $\mathcal{R}_{ij} = \emptyset$ **then**
      $A_{ij} = -1$
    else
      $A_{ij} = \mathcal{E}[i][j]$
    end if
  end if
end for
return Adjacency Matrix A

Method	Performance
Random	0.681
Chaitin	0.417
GNN (Ours)	0.517

TABLE I
AVERAGE NUMBER OF SPILLED NODES FOR A DATASET OF SYNTHETIC GRAPHS WITH LESS THAN 10 NODES.

This graph generation is executed on a repository of source (.c) files provided by LLVM's test suite [5]. This test suite is comprised of programs designed to benchmark LLVM performance (efficiency, compilation speed, etc.). The combined pipeline is shown in Figure 1 below.

IV. RESULTS

A. Results on Synthetic Interference Graphs

Table I reports the average number of spilled nodes for a dataset of synthetic graphs with less than 10 nodes and we fix the number of physical registers (colors) to 3. In this case, our

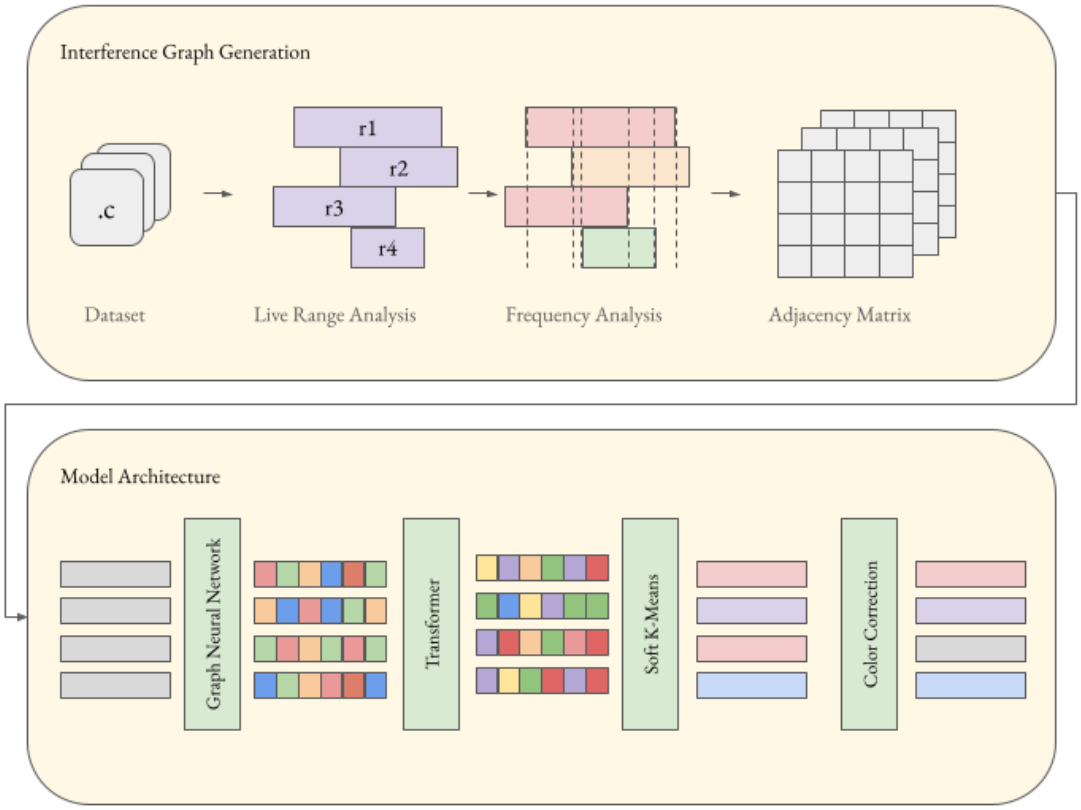


Fig. 1. Our approach's training pipeline can be divided into two parts: graph generation and model training. The former takes a dataset of source (.c) files and generates interference graphs using live range analysis and profiling data. The latter feeds generated adjacency matrices into a GNN and Transformer model to output graph colorings using Soft K-Means and a color correction scheme. Note that color correction is only executed at test time.

Method	Performance
Random	0.876
Chaitin	0.430
GNN (Ours)	0.778

TABLE II
AVERAGE NUMBER OF SPILLED NODES ON THE LLVM-GENERATED DATASET OF INTERFERENCE GRAPHS.

GNN approach tends to outperform random coloring but falls slightly short of Chaitin's performance.

While these results do not support that our approach is necessarily successful, increased performance over random coloring does suggest that the GNN architecture is extracting some useful information. This claim is limited, in the sense that performance is not striking nor better than the chosen benchmark. Regardless, this is a promising signal that approaches similar to ours can be utilized with possible tuning, architectural changes, or dataset choices.

V. RESULTS ON REAL INTERFERENCE GRAPHS

Table II reports the average number of spilled nodes for a dataset of interference graphs generated by our custom LLVM pass. Chaitin's algorithm outperforms our approach by a more significant margin when compared to that on synthetically generated graphs. It is of note that the GNN still outperforms random coloring in this case, against suggesting that the architecture is somewhat informed in its decisions.

Our limited results could be attributed to many factors. First, our training dataset is relatively small (around 250 interference graphs), meaning that the GNN has an extremely limited amount of resources to extract information from. Future work could be dedicated to increasing this dataset to measure how much that affected performance. Secondly, we opted for a fully unsupervised learning approach where, in actuality, a semi-supervised approach may be beneficial. For example, utilizing Chaitin's output to inform how our GNN could improve on such colorings would be an interesting approach to consider.

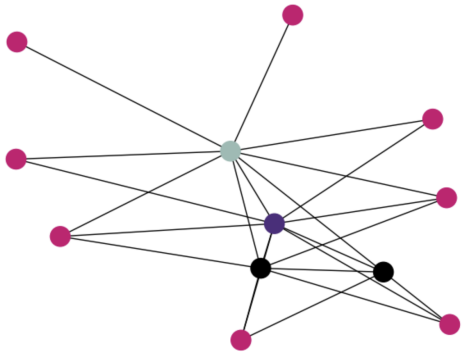
A. *Successful Examples*

There are cases in which our GNN achieves a preferable coloring, i.e. with fewer spills, compared to Chaitin's algorithm. We find this to occur on occasion for smaller graphs with fewer colors. Below, we compare the results on the interference graph generated from the source file sse.shift.c in the LLVM Test Suite, of which the source code is shown below:

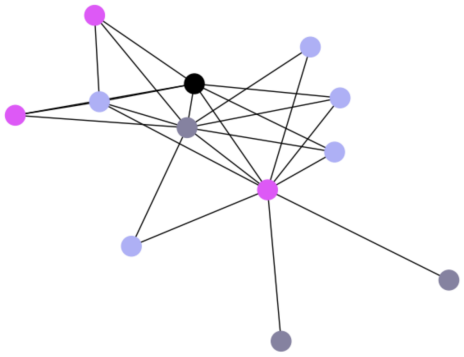
```
#include <emmintrin.h>
#include <stdio.h>

typedef union {
  __m128i V;
  int A[4];
```

```
} IV;

static void printIV(IV *F) {
  printf("%08x%08x%08x%08x\n",
         F->A[0],
         F->A[1],
         F->A[2],
         F->A[3]);
}

int main() {
  __m128i allones = _mm_set1_epi32(0);
  __m128i zeroones, onezeros;
  allones = _mm_cmpeq_epi32(allones, allones);
  zeroones = _mm_srli_epi16(allones, 8);
  printIV((IV*)&zeroones);
  onezeros = _mm_slli_epi16(allones, 8);
  printIV((IV*)&onezeros);
  return 0;
}
```

The colored interference graphs are shown below. In both the them, nodes colored black represent spilled nodes.



Chaitin's coloring for the sse.shift.c interference graph is shown above. The two black nodes were each assigned by Chaitin's algorithm as the same color of one of their neighbors, and thus were forced to spill.



In contrast, our GNN's coloring for the same graph is shown above. The coloring created by our GNN resulted in one fewer spilled node than the coloring created by Chaitin's algorithm with the same number of colors.

VI. Conclusion and Areas for Improvement

In this project, we show that a graph neural network approach for graph coloring has limited success in register allocation. We see that our approach outperforms random coloring but does not outperform Chaitin's algorithm, particularly on real interference graphs. Occasionally, however, we do find that our approach can outperform Chaitin's on small graphs with a low number of physical registers. These results do indicate some promise for GNN-based register allocation. Some future investigations could focus on the effect of dataset size on performance or interweaving deep learning approaches with procedural algorithms like Chaitin's. Other future extensions to this work could include finding a way to partition graphs into smaller subgraphs to find more effective ways of generalizing across graphs.

References

[1] S. VenkataKeerthy, S. Jain, A. Kundu, R. Aggarwal, A. Cohen, and R. Upadrasta, "Rl4real: Reinforcement learning for register allocation," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 133–144. [Online]. Available: https://doi.org/10.1145/3578360.3580273

[2] J. Huang, M. M. A. Patwary, and G. F. Diamos, "Coloring big graphs with alphagozero," *CoRR*, vol. abs/1902.10162, 2019. [Online]. Available: http://arxiv.org/abs/1902.10162

[3] D. Das, S. A. Ahmad, and V. Kumar, "Deep learning-based approximate graph-coloring algorithm for register allocation," in *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, 2020, pp. 23–32.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, nov 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[5] "GitHub - llvm/llvm-test-suite — github.com," https://github.com/llvm/llvm-test-suite, [Accessed 13-12-2023].