Aplicações do Aprendizado de Máquina na geração de código *spill*

Matheus Pires Vila Real[1], **Wesley Attrot**[1]

[1]Departamento de Computação – Universidade Estadual de Londrina (UEL)
Caixa Postal 10.011 – CEP 86057-970 – Londrina – PR – Brasil

{matheus.pires.vila, wesley}@uel.br

Abstract. *Register allocation is highlighted as one of the key code optimizations throughout a program's compilation process. It is, however, a NP-complete problem, due to the nature of the traditional implementations, which are unable to obtain optimal solutions in reasonable time. In this context, several heuristics and adjustments have been proposed over the course of the years in order to improve the precision and the speed of register allocators. Machine learning, in the other hand, is a field of artificial intelligence encompassing the development of complex classification and prediction models which are trained in an algorithmic way, that grew in relevance and became widely researched recently. Therefore, efforts to apply machine learning techniques to register allocation, aiming to improve spill code generation, arose. This work is intended to examine the current state-of-the-art of the integration of both fields and investigate the implementation possiblities of register allocators that widely employ machine learning models in their inner workings.*

Resumo. *A alocação de registradores é uma das otimizações de código mais significativas do processo de compilação de um programa. Mas, devido à natureza dos algoritmos empregados nas implementações tradicionais, ela caracteriza-se como um prolema NP-completo, o qual é difícil de se resolver de maneira ótima. Nesse contexto, ao longo dos anos foram sendo propostas heurísticas e ajustes visando tornar as técnicas de geração de código spill mais precisas e menos custosas. Com o crescimento da relevância do aprendizado de máquina (machine learning) — área da inteligência artificial, que engloba o desenvolvimento de modelos complexos de categorização e predição treinados de maneira algorítmica — surge a perspectiva de integração de ambas as áreas. Isto posto, este trabalho se propõe a vislumbrar o estado da arte da aplicação do aprendizado de máquina na alocação de registradores e investigar as possibilidades práticas de implementação de alocadores utilizando modelos treinados por machine learning.*

1. Introdução

Compiladores são um tipo de programa que tem como sua principal função a tradução de código-fonte escrito em uma linguagem de programação para um código equivalente em outra linguagem. Usualmente, a compilação é feita de uma linguagem de alto nível para linguagem de montagem, ou *assembly*, uma representação de baixo nível das instruções de máquina que exprime diversas especificidades da arquitetura-alvo [3].

O compilador efetua análise léxica, sintática e semântica no código e produz uma representação intermediária que será submetida a diversas otimizações, a fim de aprimorar e adequar o programa gerado às características da máquina de destino. Dentre essas otimizações, destaca-se a alocação de registradores, que consiste na tarefa de mapear as variáveis utilizadas pelo programador no código-fonte para os registradores físicos presentes na arquitetura-alvo [25].

Os registradores são os componentes de memória mais rápidos presentes nos sistemas computacionais, a frente de outras formas mais lentas, como a memória cache e a memória principal. No entanto, em muitos casos o número de registradores físicos é pequeno demais para comportar todos os valores em uso em um ponto do programa, criando a necessidade de se mapear alguns deles para a memória principal [8, 25].

Surge então a problemática da geração de um mapeamento com o menor número possível de variáveis armazenadas na memória principal, de modo a minimizar os acessos à memória que tornam o código mais lento [25]. Uma alocação sofisticada pode tornar um programa até 250% mais rápido do que o gerado por um alocador simples [28] e impactar positivamente o consumo energético da CPU, tendo em vista que operações em memória representam de 50% a 75% dos gastos de energia de um sistema computacional [33].

Todavia, os algoritmos para a alocação de registradores não são triviais. A obtenção de uma estratégia ótima de mapeamento é dificultada pela especificidade de cada arquitetura-alvo, e a abordagem clássica se vale da coloração de grafos — um problema NP-completo — como abstração para representar as interferências entre as variáveis [12, 11, 7]. Verifica-se então a necessidade pela criação de heurísticas para otimizar as técnicas existentes.

Nas últimas décadas, o avanço nas áreas da inteligência artificial e do aprendizado de máquina aliado ao aumento do poder computacional das máquinas no geral abriu portas para aplicações em diversas áreas [4], incluindo a alocação de registradores. Recentemente, trabalhos vêm sendo publicados explorando a interdisciplinaridade de ambas as áreas e trazendo resultados animadores ou, ao menos, de interesse, como é o caso das pesquisas de Amarasinghe *et al.*, Das *et al.* e VenkataKeerthy *et al.* [31, 14, 32].

Este trabalho se propõe a investigar as perspectivas de integração entre a alocação de registradores e o aprendizado de máquina. O restante deste trabalho está organizado da seguinte forma: a Seção 2 apresenta uma revisão da evolução dos conceitos levando a trabalhos atuais combinando aprendizado de máquina e alocação de registradores. A Seção 3 formaliza os objetivos a serem alcançados com o desenvolvimento deste trabalho e a Seção 4 especifica as metodologias, métricas e critérios para a avaliação dos resultados obtidos com respeito ao que foi definido como objetivo. A Seção 1 detalha o cronograma a ser seguido e a Seção 6 apresenta uma breve nota sobre as expectativas em relação ao trabalho.

2. Fundamentação Teórico-Metodológica e Estado da Arte

2.1. Alocação de Registradores

O algoritmo tradicional empregado na alocação de registradores abstrai a tarefa como um problema de coloração de grafo, onde tenta-se colorir um grafo com um número arbitrário de cores sem usar a mesma cor em nós adjacentes. Na geração da representação

intermediária do código-fonte, as variáveis são convertidas em registradores virtuais que são representados no grafo como nós, enquanto as interferências são expressas na forma de arestas [12].

Verifica-se que um grafo é k-colorável se, ao colorir iterativamente, é possível atribuir uma das k cores para cada um dos nós sem violar as restrições de interferência. Quando isso não for possível, a variável é mapeada para a memória principal e ocorre a geração de código *spill*, isto é, a inserção de instruções *store* e *load* no código de modo a fragmentar a presença daquele valor em registradores [11].

Contudo, a coloração de grafos é um clássico problema NP-completo [19] — um problema de difícil resolução no qual a obtenção de uma solução ótima é consideravelmente dispendiosa [16]. Por consequência, as implementações convencionais valem-se de aproximações ou resoluções quase-ótimas, não sendo capazes de garantir a utilização do menor número possível de cores e comumente introduzem código *spill* desnecessário [31]. Além disso, decidir qual variável mapear para a memória é uma decisão complexa, e uma escolha errada pode produzir um resultado indesejado [8]. Nesse cenário foram concebidas heurísticas visando a otimização da seleção de cores e da geração de *spill* [22].

Logo em 1982, Chaitin [11] introduziu uma heurística que consistia em computar o custo de *spill* de cada variável, para assim evitar o envio de variáveis de acesso frequente para a memória e que isso prejudicasse a performance do programa. Bernstein *et al.* [6] introduziram três novas heurísticas para o cálculo de custo, tal que o compilador escolhe a que produza menor custo total na alocação, além de uma estratégia gulosa para atribuição de cores.

Briggs *et al.* [9] ampliaram as técnicas para evitar a introdução de código *spill* para valores facilmente recomputáveis com poucas instruções, propondo a utilização uma forma de representação intermediária chamada *static single-assignment*, que seria revisitada em trabalhos posteriores [18, 27]. Bergner *et al.* [5] propuseram um método mais fino para inserir instruções *load/store* somente na região de interferência de duas variáveis, ao invés do código todo. Por fim, Cooper *et al.* [13] aprimoraram uma técnica para fragmentar a vivacidade de variáveis ao longo do código de modo a reduzir a pressão sobre os registradores e produzir menos *spill*.

2.2. Aprendizado de Máquina

O aprendizado de máquina é a área da inteligência artificial destinada ao desenvolvimento de algoritmos, modelos e agentes inteligentes capazes de realizar certas tarefas sem serem explicitamente programados e que se auto-aperfeiçoam com base em amostras de dados para treinamento. Após processar os dados, o modelo é capaz de descrever padrões de associação contidos nos dados e classificar uma entrada com base no conhecimento extraído [4].

No paradigma supervisionado, os dados do conjunto de treinamento são tratados como entradas de uma função de aproximação, que os mapeiam para o conjunto de saídas esperadas. A cada predição realizada, o algoritmo computa o erro associado e é capaz de ajustar seus próprios parâmetros a fim de reduzi-lo. Na abordagem não-supervisionada, não existe mapeamento prévio dos dados. Essa técnica busca extrair informações sobre o

comportamento do problema através da modelagem estatística do conjunto de treinamento [23].

Ainda há a aprendizagem por reforço, onde não é especificado ao algoritmo como realizar a aprendizado. Nesse caso o modelo é aperfeiçoado recompensando comportamentos que se aproximam do resultado desejado, enquanto variações indesejadas são descartadas [24].

Com o barateamento de recursos computacionais, tornaram-se disseminadas implementações de *deep learning*, isto é, abordagens que apresentam redes neurais artificiais com camadas intermediárias de neurônios entre as camadas de entrada e de saída. O processamento da entrada por várias camadas permite a extração e o reconhecimento de padrões contidos profundamente nos dados [23, 30].

2.3. Aplicações do Aprendizado de Máquina na Alocação de Registradores

Trabalhos como o de Lemos *et al.* [21], Goudet *et al.* [17], Schuetz *et al.* [29] e Dodaro *et al.* [15] demonstram a aplicabilidade das abordagens de *deep learning* para resolução da coloração de grafos a um nível básico, e sugerem a possibilidade de se generalizar essas soluções para os problemas aplicados.

Amarasinghe *et al.* [31] criaram um sistema de aprendizagem utilizando algoritmo evolutivo para automaticamente encontrar heurísticas de alocação de registradores, obtendo uma melhora média de 23%. Cavazos *et al.* [10] desenvolveram, utilizando aprendizagem supervisionada, uma heurística para a decisão em tempo de compilação entre diferentes algoritmos de alocação, permitindo ao compilador alternar entre eles conforme necessário. De maneira análoga, Musliu *et al.* [26] aplicaram aprendizagem não-supervisionada para obter uma heurística de decisão entre algoritmos de coloração de grafos.

Das *et al.* [14] apresentaram um modelo de *deep learning* baseado em rede neural com várias camadas *long short-term memory* (LSTM), capaz de atribuir uma cor a cada nó do grafo. Como esse procedimento gerava em torno de 10%–20% de colorações inválidas, era aplicada uma etapa subsequente de correção das cores. O alocador obteve resultados em performance comparáveis ao alocador *greedy* (GRA) do *framework* LLVM.

Um sistema multi-agentes baseado em aprendizagem por reforço foi desenvolvido por VenkataKeerthy *et al.* [32] como solução independente de arquitetura, resolvendo diferentes etapas do processo de alocação, como a coloração do grafo e geração de código *spill*. Os autores reportaram desempenho semelhante aos dos alocadores do LLVM em termos de acesso a memória e velocidade do código produzido, sendo ligeiramente superiores em alguns *benchmarks*.

3. Objetivos

Este trabalho tem como objetivo integrar, de maneira eficiente, aplicações de aprendizado de máquina na alocação de registradores por coloração de grafos, visando otimizar o processo de alocação através de:

1. Técnicas para auxiliar na escolha das heurísticas a serem utilizadas nas etapas da alocação;
2. Novas maneiras de se computar custos de *spill* utilizando aprendizado de máquina;

3. Implementação de um alocador completo que empregue amplamente recursos de inteligência artificial.

A meta é implementar totalmente ao menos um dos tópicos supracitados, mas realizar uma investigação detalhada a respeito de todos eles.

4. Procedimentos metodológicos/Métodos e técnicas

Primeiramente, será realizada uma análise compreensiva da literatura recente sobre o uso de aprendizado de máquina para otimizar o processo de alocação de registradores por coloração de grafo, além do estudo das funcionalidades do LLVM, um *framework* modular que dispõe de diversos recursos destinados ao desenvolvimento de compiladores reutilizáveis [2, 20].

Será então implementado um modelo de aprendizado com o propósito atuar nas etapas da alocação, e que será integrado com as ferramentas disponíveis no LLVM. Em seguida, resultados experimentais serão coletados através da realização de *benchmarks*, como o *SPEC CPU* [1], e serão comparados aos produzidos pelos alocadores do LLVM [34].

As métricas utilizadas nas comparações consistirão basicamente na quantidade de código *spill* introduzido e na velocidade dos executáveis gerados. Elas permitirão investigar se os métodos propostos no trabalho apresentam um resultado satisfatório perante o estado da arte.

5. Cronograma de Execução

A execução deste trabalho foi dividida em um conjunto de atividades bem delimitadas que descrevem as etapas do andamento da pesquisa:

1. Levantamento bibliográfico;
2. Análise das técnicas e estudo das ferramentas;
3. Escrita da versão preliminar;
4. Planejamento da implementação;
5. Implementação;
6. Coleta e análise dos resultados;
7. Escrita da versão para banca/final.

A tabela 1 apresenta o cronograma de execução das atividades enumeradas acima ao longo do ano:

Tabela 1. Cronograma de Execução

$Ativ. \diagdown mês$	ago.	set.	out.	nov.	dez.	jan.	fev.	mar.	abr.	mai.
Ativ. 1	•									
Ativ. 2	•	•	•							
Ativ. 3	•	•	•	•	•					
Ativ. 4		•	•	•						
Ativ. 5				•	•	•	•	•		
Ativ. 6						•	•	•		
Ativ. 7						•	•	•	•	•

6. Contribuições e/ou Resultados esperados

É esperado que a pesquisa vislumbre novos métodos para se integrar as técnicas do aprendizado de máquina na alocação de registradores, de modo a demonstrar a aplicabilidade prática dessas técnicas e possibilitar o desenvolvimento de alocadores mais eficientes, rápidos e confiáveis. Também é previsto contemplar o estado da arte das pesquisas sobre alocação de registradores, apresentando uma síntese sobre a vanguarda da produção intelectual na área.

Espera-se ainda incentivar trabalhos futuros apoiando-se nas descobertas desta pesquisa, através da construção coletiva do conhecimento.

7. Espaço para assinaturas

Londrina, *30 de agosto de 2023*.

Aluno	Orientador

Referências

[1] SPEC CPU®2017 Overview / What's New? `https://www.spec.org/cpu2017/Docs/overview.html`. Acesso em: 16 de junho de 2023.

[2] The LLVM Compiler Infrastructure. `https://llvm.org`. Acesso em: 16 de junho de 2023.

[3] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Alternative eText Formats Series. Addison-Wesley, 2007.

[4] E. Alpaydin. *Introduction to Machine Learning, fourth edition*. Adaptive Computation and Machine Learning series. MIT Press, 2020.

[5] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. *SIGPLAN Not.*, 32(5):287–295, may 1997.

[6] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. *SIGPLAN Not.*, 24(7):258–263, jun 1989.

[7] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the np-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how. volume 4382, pages 283–298, 11 2006.

[8] Preston Briggs. *Register allocation via graph coloring*. Rice University, 1992.

[9] Preston Briggs, Keith D Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 311–321, 1992.

[10] John Cavazos, J. Eliot B. Moss, and Michael F. P. O'Boyle. Hybrid optimizations: Which optimization algorithm to use? In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, pages 124–138, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[11] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, page 98–105, New York, NY, USA, 1982. Association for Computing Machinery.

[12] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[13] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In Kai Koskimies, editor, *Compiler Construction*, pages 174–187, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[14] Dibyendu Das, Shahid Asghar Ahmad, and Venkataramanan 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)*, pages 23–32, 2020.

[15] Carmine Dodaro, Davide Ilardi, Luca Oneto, and Francesco Ricca. Deep learning for the generation of heuristics in answer set programming: A case study of graph coloring. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors, *Logic Programming and Nonmonotonic Reasoning*, pages 145–158, Cham, 2022. Springer International Publishing.

[16] M. R. Garey and D. S. Johnson. The complexity of near-optimal graph coloring. *J. ACM*, 23(1):43–49, jan 1976.

[17] Olivier Goudet, Cyril Grelier, and Jin-Kao Hao. A deep learning guided memetic framework for graph coloring problems. *Knowledge-Based Systems*, 258:109986, 2022.

[18] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in ssa-form. In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, pages 247–262, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[19] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[20] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[21] Henrique Lemos, Marcelo Prates, Pedro Avelar, and Luís Lamb. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. pages 879–885, 11 2019.

[22] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of algorithms for local register allocation. In Stefan Jähnichen, editor, *Compiler Construction*, pages 137–152, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[23] Amitha Mathew, P. Amudha, and S. Sivakumari. Deep learning techniques: An overview. In Aboul Ella Hassanien, Roheet Bhatnagar, and Ashraf Darwish, editors, *Advanced Machine Learning Technologies and Applications*, pages 599–608, Singapore, 2021. Springer Singapore.

[24] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[25] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, pages 481–482. Morgan Kaufmann, 1st edition, 1997.

[26] Nysret Musliu and Martin Schwengerer. Algorithm selection for the graph coloring problem. In Giuseppe Nicosia and Panos Pardalos, editors, *Learning and Intelligent Optimization*, pages 389–403, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[27] Fernando Pereira and Jens Palsberg. Register allocation after classical ssa elimination is np-complete. pages 79–93, 03 2006.

[28] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. *SIGPLAN Not.*, 43(6):216–226, jun 2008.

[29] Martin J. A. Schuetz, J. Kyle Brubaker, Zhihuai Zhu, and Helmut G. Katzgraber. Graph coloring with physics-inspired graph neural networks. *Phys. Rev. Res.*, 4:043131, Nov 2022.

[30] Neha Sharma, Reecha Sharma, and Neeru Jindal. Machine learning and deep learning applications-a vision. *Global Transitions Proceedings*, 2(1):24–28, 2021. 1st International Conference on Advances in Information, Computing and Trends in Data Engineering (AICDE - 2020).

[31] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 77–90, New York, NY, USA, 2003. Association for Computing Machinery.

[32] S. VenkataKeerthy, Siddharth Jain, Anilava Kundu, Rohit Aggarwal, Albert Cohen, and Ramakrishna Upadrasta. Rl4real: Reinforcement learning for register allocation. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, CC 2023, page 133–144, New York, NY, USA, 2023. Association for Computing Machinery.

[33] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):802–815, 2006.

[34] Tiago Xavier, George Souza Oliveira, Ewerton Lima, and Anderson Faustino. A detailed analysis of the llvm's register allocators. pages 190–198, 11 2012.