Predição de Desempenho com *Graph Neural Networks*

Vanderson Martins do Rosario[1],
André Felipe Zanella[2], **Anderson Faustino da Silva**[2], **Edson Borin**[1]

[1]IC – Universidade de Campinas (UNICAMP) – Campinas – SP – Brasil

[2]DIN – Universidade Estadual de Maringá (UEM) – Maringá – PR – Brazil

{vanderson.rosario,edson}@ic.unicamp.br

aft.zanella@gmail.com, anderson@din.uem.br

Abstract. *Tuning application performance on modern computing infrastructures involves choices in a very large design space, because modern computing architectures have different structures that have impact in overall performance. Moreover, different applications use these resources in different ways, leading to a very complex performance function. Thus, making it hard for compilers or experts to find an optimal compilation plan for an application that maximizes such performance function. One possibility is to evaluate many possible plans and stick with the better, a technique known as autotuning. However, executing an application to measure its performance for every plan is very expensive. One approach is to not execute the application but to predict its performance. In this work, we evaluate the use of Graph Neural Networks (GCN), more specifically graph convolutional network, to make fast predictions without executing the application. We train a GCN using 30 thousand different compilation plans applied to 300 different applications and we showed that it can learn features from the control flow graph which outperforming non-graph-aware features in performance prediction.*

Resumo. *Melhorar o desempenho de uma aplicação em computadores modernos envolve escolhas em um largo espaço de busca, pois tais arquiteturas possuem variadas características que impactam significativamente o desempenho das aplicações. Além disso, diferentes aplicações tendem a utilizar essas características de forma diferente, tornando a função que mapeia o desempenho de uma aplicação em um hardware bastante complexa. O que torna a tarefa de gerar códigos que maximizem tal função, por parte de compiladores ou especialistas, uma tarefa difícil. Uma possibilidade é avaliar automaticamente várias possibilidades de compilação, técnica conhecida como autotuning. Porém, o custo de executar a aplicação para medir seu desempenho para cada possibilidade tem um elevado custo. Por isso, é comum a utilização de preditores de desempenho para acelerar essa exploração. Nesse trabalho, implementamos e avaliamos o uso de redes neurais de grafos, mais especificamente redes convolucionais de grafos, para a tarefa de predizer o desempenho de uma aplicação. Nós treinamos uma rede com 30 mil diferentes planos de compilação aplicados em 300 diferentes aplicações e mostramos que redes baseadas em grafo podem aprender sobre as características do grafo de fluxo de controle, obtendo desempenho melhor do que técnicas que não consideram tal grafo durante a predição de desempenho.*

1. Introdução

Embora sistemas modernos de geração de código, como os sistemas consolidados GCC e LLVM [Lattner and Adve 2004], forneçam planos de compilação[1] pré-definidos, nem sempre tais planos derivam o melhor código final acarretando na necessidade de gerar planos especializados. Tal necessidade ocasionou o desenvolvimento de *autotuners*, os quais são empregados como um mecanismo para encontrar bons planos de compilação [Ashouri et al. 2018].

O *autotuning* é uma abordagem efetiva na descoberta de bons planos de compilação [Filho et al. 2018]. Tal descoberta é consequência da comparação de diversos planos de compilação que são gerados utilizado-se de diferentes mecanismos como geradores aleatórios. No entanto, executar um número elevado de comparações possui um alto custo computacional. Cada comparação implica na execução dos dois códigos compilados com diferentes planos de compilação para medir seu desempenho. Em muitos casos as aplicações podem levar minutos ou até horas para serem executadas. Resultando em um alto custo computacional para as comparações. Uma abordagem diferente seria estimar o desempenho de uma aplicação dado suas características e um determinado plano de compilação. O que abre caminho para a utilização de modelos inteligentes que avaliem a qualidade de planos de compilação, sem a necessidade de executar a aplicação em questão.

Redes Neurais Artificiais tem se mostrado uma solução versátil e eficiente para a solução de problemas em diferentes áreas [Bastings et al. 2017, Satorras and Estrach 2018, Ying et al. 2018]. No contexto de alto desempenho Redes Neurais tem se mostrado usuais na descoberta de qual plataforma de hardware uma aplicação terá um melhor desempenho [Wen et al. 2014, Cummins et al. 2017]. Contudo, ainda existe uma lacuna no tocante ao uso de tais redes na descoberta de desempenho de uma aplicação mediante o uso de um determinado plano de compilação.

Redes Neurais de Grafo (do inglês *Graph Neural Networks* ou GNN) [Kipf and Welling 2016, Li et al. 2017] estão se popularizando pela sua capacidade de modelar as dependências existentes entre os nós do grafo e consequentemente inferir propriedades existentes entre diferentes entidades do mesmo. Grafos são estruturas comuns em diversos domínios, tais como moléculas, redes sociais e sistemas de geração de código, entre outros. Para um sistema de geração de código, como um compilador, uma aplicação normalmente é representada como um grafo sob o qual planos de compilação são aplicados. Como por exemplo, o Grafo de Fluxo de Controle e Grafo de Fluxo de Dados. Portanto, Redes Neurais baseadas em grafo se adequam perfeitamente ao problema em questão.

Nesse trabalho desenvolvemos um sistema baseado em GNN, o qual é capaz de classificar o ganho de desempenho de uma aplicação dado um determinado plano de compilação. O objetivo é mostrar que GNNs são algoritmos que podem ser utilizados

Agradecimentos: a Google Cloud pelo *grant* fornecido, ao CNPq (313012/2017-2) e CAPES (PROCAD 2966/2014), e também a Fapesp (2013/08293-7) e Petrobras pelo apoio financeiro.

[1]Um plano de compilação compreende os valores dos parâmetros do sistema, durante o processo de geração de código final.

para aprender a medição de qualidade de código. Estando assim um passo além dos sistemas de *autotuning*, removendo a necessidade de executar a aplicação diversas vezes. Nossos resultados demostram que o uso de Redes Neurais baseadas em grafo melhora a previsão em até 30%, frente a outras abordagens que não consideram a estrutura do grafo. De fato, nosso sistema possui uma acurácia de 91% e um erro médio menor que 1%. Por fim, para realizar uma predição em uma `NVIDIA P100`, precisou-se de apenas 16ms.

As principais contribuições desse trabalho são descritas a seguir.

1. Um sistema baseado em `GNN` que classifica o ganho de desempenho de um código.
2. Um sistema capaz de obter 91% de acurácia na classificação do nível de ganho de desempenho.
3. Disponibilização do sistema como uma plataforma aberta[2].
4. Demonstração do potencial do uso de `GNNs` para sistemas de geração de código de alto desempenho, dado que grande parte das estruturas de representação de códigos nos mesmos são feitos por meio de grafos.
5. Avaliação do desempenho do nosso sistema em um conjunto de dados de mais de 30 mil diferentes códigos, gerados com diferentes planos de compilação utilizando 300 aplicações diferentes provenientes da suite de teste `LLVM`.

O restante desse trabalho está organizado da seguinte forma: a Seção 2 apresenta os trabalhos relacionados a predição de desempenho de aplicações para aceleração *autotuning*; a Seção 3 apresenta a definição do problema de predição de desempenho para aceleração de *autotuning* e como utilizamos `GNNs` para tal tarefa; a Seção 5 apresenta as máquinas, configurações e metodologias utilizadas para geração do nosso conjunto de treino, teste e dos nossos experimentos; a Seção 6 apresenta os resultados do desempenho da `GNN` e, por fim, a Seção 7 traz nossas conclusões e trabalhos futuros.

2. Trabalhos Relacionados

Autotuning para geração de código de alto desempenho. Sistemas de *autotuning* empregam mecanismos baseados em compilação iterativa para determinar os melhores parâmetros para a aplicação, para um *hardware* específico. Compilação iterativa avalia diversos planos do espaço de busca, quanto maior a quantidade de planos a serem avaliados, maior o custo do sistema. A avaliação de um plano corresponde a gerarmos o código final da aplicação e medirmos o seu desempenho para o *hardware* em questão. Com o objetivo de minimizar tal custo, sistemas modernos empregam estratégias de aprendizagem de máquina com o objetivo de reduzir a quantidade de planos avaliados. Filho *et al* [Filho et al. 2018] apresentam diferentes estratégias para reduzir o espaço de busca juntamente com um motor que emprega raciocínio baseado em casos. Os autores demonstraram que aprendizagem de máquina é uma boa estratégia para prover *autotunings* eficientes. Ashouri *et al* [Ashouri et al. 2014] empregam uma rede Bayesiana para criar um modelo capaz de prever um bom plano de compilação para uma determinada aplicação, com um tempo de busca melhor que aos tradicionais *autotunings* que utilizam métodos mais simples de busca. Ogilvie *et al* [Ogilvie et al. 2017] empregam aprendizagem ativa para criar um modelo de predição de planos de compilação, que minimize a quantidade de vezes que um plano precisa ser avaliado. Embora tais trabalhos apresentem resultados

[2]https://github.com/vandersonmr/Code-Performance-Predictor

significativos, ainda existe a necessidade de avaliar diversos planos de compilação para descobrir o melhor. Seguindo um outro caminho, reduzindo o custo das comparação ao invés do número de comparações, o objetivo do nosso trabalho é, dadas as características de uma aplicação (na forma de um grafo) e um determinado plano de compilação, inferir o desempenho de tal plano para esta aplicação, considerando as particularidades do *hardware* em questão.

Predição de desempenho. Uma questão importante em sistemas de alto desempenho é a predição de desempenho, cujo foco é *entender* o comportamento de uma aplicação em uma plataforma de *hardware*. Neste contexto, Sarkar e Mitra [Sarkar and Mitra 2014] propõem um método para analisar aplicações e inferir o tempo gasto por diversas porções do código. Desta forma, tal método é capaz de identificar porções do código que limitam o desempenho da aplicação para plataformas com `GPUs`. Ardalani *et al* [Ardalani et al. 2015] desenvolvem uma plataforma baseada em um modelo de regressão cujo objetivo é prever o desempenho de aplicações `GPUs` a partir do código para `CPU`. Ambos trabalhos extraem características do programa para desenvolverem suas propostas, explorando tanto as relações existentes no código fonte quanto o seu impacto no *hardware* em questão. Fan *et al* [Fan et al. 2019] propõem um modelo cujo objetivo é encontrar a melhor frequência entre núcleos e memória para códigos `OpenCL` em `GPUs`. Para alcançar tal objetivo, Fan *et al* [Fan et al. 2019] analisam o comportamento do consumo energético e desempenho de aplicações executadas em `GPU`. A partir de tal observação os autores criaram um modelo capaz de prever a melhor configuração de frequência para uma determinada aplicação. O trabalho de Fan *et al* se baseia na extração de características estáticas da aplicação, precisamente na quantidade de instruções inteiras, de ponto flutuante e de acesso à memória. Portanto, sem considerar a representação tradicional utilizada para gerar código final, grafos, como ocorre em nosso trabalho.

Redes Neurais Artificiais. As redes `GNNs` tem sido aplicadas em áreas como Processamento de Linguagem Natural [Bastings et al. 2017], Visão Computacional [Satorras and Estrach 2018] e Sistemas de Recomendação [Ying et al. 2018], entre outros. No entanto, `GNNs` ainda não foram exploradas em ambientes de alto desempenho. Em contraste, redes neurais *feed-forward* são comumente utilizadas para criação de modelos de previsão. Curtis-Maury *et al* [Curtis-Maury et al. 2007] fazem uso destas redes para prever o consumo de energia de aplicações `OpenMP` [OpenMP 2020] em sistemas *multi-core*. Por sua vez, Wen *et al* [Wen et al. 2014] empregam tais redes para estimar o potencial ganho em tempo de execução a partir do agendamento de tarefas de múltiplos programas `OpenCL` [OpenCL 2020] competindo recursos na `GPU`. Cummins *et al* [Cummins et al. 2017] utilizam redes recorrentes do tipo `LSTM` (*Long Short-term Memory*) para criar um modelo de predição de granularidade de *threads* para programas `OpenCL`, como também o mapeamento em `CPU` ou `GPU`. Um diferencial deste trabalho é o fato dos autores não empregarem *features* pré-definidas por humanos, mas permitirem que a rede aprenda e extraia *features* do código fonte na fase de treinamento, contudo sem utilizar representações baseadas em grafo. Seguindo a mesma abordagem de Cummins *et al*, Brauckmann *et al* [Brauckmann et al. 2020] mostram como utilizar as representações de grafo decorrrentes do sistema de compilação para os mesmos objetivos estabelecidos no trabalho de Cummins. Brauckmann demonstra que o ganho de desempenho em utilizar grafos é superior a outras abordagens. Diferente destes trabalhos, nossa estratégia explora o uso de `GNNs` como mecanismo para prever o desempenho de aplicações. Por

outro lado, nosso trabalho se assemelha ao de Brauckmann *et al* por utilizar grafos como mecanismos de aprendizado.

3. O Problema de Predição de Desempenho de Aplicações

Mecanismos de *autotuning* pretendem encontrar a melhor configuração e parâmetros para uma aplicação em um determinado *hardware*, com o objetivo de obter o melhor desempenho possível. Um exemplo é encontrar um bom plano de compilação, isto é, sequência de otimizações que serão utilizadas pelo compilador, que leva ao melhor desempenho do código gerado para uma aplicação em um *hardware*. É conhecido que, para cada par (aplicação, *hardware*), existe uma solução ótima que pode ser diferente para outros pares. Em geral, os mecanismos de *autotuning* normalmente avaliam diversas possibilidades durante a busca. Para cada tentativa é necessário conhecer o desempenho das configurações avaliadas para então poder ordená-las, e, assim, encontrar uma boa ou a melhor configuração. No entanto, executar aplicações para medir o seu desempenho possui um custo elevado já que uma aplicação pode levar minutos, horas ou dias para ser executada.

Outra possibilidade é prever o desempenho. Ou seja, para um par de configurações c_1 e c_2 e uma aplicação p, queremos uma função de predição de desempenho P tal que, se o tempo de execução T da aplicação com as diferentes configurações respeita a seguinte ordem: $T(p, c_1) < T(p, c_2)$, então deve existir uma alta probabilidade de que $P(p, c_1) \leq P(p, c_2)$. Em outras palavras, a função de predição P deve conseguir com alta probabilidade ordenar o desempenho das diferentes configurações c_i de forma correta, podendo, o mecanismo que use P concluir, com também alta probabilidade, quais os melhores candidatos de configurações. No entanto, essa não é a única restrição. É preciso que o tempo para se calcular P seja significativamente menor do que o tempo de T. Desta forma, acelerando o processo de *autotuning*, enquanto mantendo com alta probabilidade a ordenação das possíveis configurações.

Esse trabalho propõe o uso de `GNNs` para o problema da busca por bons planos de compilação para uma dada aplicação. Nesse cenário, para cada plano, o sistema de compilação gera um código diferente para uma mesma aplicação. Cada um desses diferentes códigos gerados possuem desempenhos diferentes. Encontrar o melhor plano implica em avaliar o desempenho dos respectivos códigos gerados. Para tal cenário, nosso objetivo é desenvolver um preditor que seja capaz de dizer entre dois códigos gerados com planos diferentes, para uma mesma aplicação, qual tem o melhor desempenho. Essa predição deve ser feita mais rapidamente do que executar as aplicações.

Partindo da premissa que o desempenho dos códigos gerados não está relacionado apenas com a quantidade de instruções e quais instruções da arquitetura alvo foram utilizadas, mas também com o fluxo de controle, desenvolvemos um modelo baseado em `GNNs` para aprender a realizar tal predição tendo como entrada Grafos de Fluxo de Controle (`GFC`). Por exemplo, dois códigos podem ter a mesma quantidade de instruções, mas um pode conter um laço (apenas uma aresta no `GFC`) a mais do que outra, afetando, assim, o desempenho.

Nossa hipótese é que `GNNs` podem aprender a identificar essas características do `GFC` e consequentemente as redes serem mais precisas em predições do que técnicas que não se utilizem do grafo e sua estrutura.

4. *Graph Neural Networks* para Predição de Desempenho

Redes Neurais Profundas (RNPs) [LeCun et al. 2015] utilizam vetores numéricos como entradas. Portanto, para ensinar a uma RNP a predizer a diferença de desempenho entre duas aplicações, precisamos representar tais códigos no formato de vetores numéricos, como por exemplo a proposta de Namolaru *et al* [Namolaru et al. 2010]. Tal vetor precisa conter informações suficientes para que a RNP aprenda a fazer a predição de forma eficaz. Portanto, quanto maior a quantidade de informações relevantes ao desempenho representado pelo vetor, melhor será a predição.

Uma abordagem simples seria criar um vetor onde cada elemento representa um tipo de instrução da arquitetura e seu valor representa a quantidade dessa instrução no código. No entanto, essa abordagem não leva em consideração características do fluxo de controle da aplicação, por exemplo. Nossa abordagem utiliza um vetor de contagem de instruções para caracterizar cada bloco básico do GFC da aplicação, e não para representar o código inteiro da aplicação. Portanto, além de utilizar uma representação que descreve as características das instruções que compõem o código da aplicação, utilizamos representações que indicam as relações entre tais instruções.

Nossa proposta é utilizar a técnica conhecida como *Graph Convolutional Layer* [Wu et al. 2019] para aumentar a quantidade de informação contida nesses vetores de cada bloco básico, unindo informações dos vizinhos. Realizamos essa operação olhando para vizinhos com profundidade de até 2 saltos no GFC. Ou seja, depois de aplicado essa transformação nos vetores dos blocos básicos, eles não mais representam apenas as informações de um bloco básico mas também informações da relação do bloco básico com seus vizinhos. A forma com que essas informações são unidas é aprendida durante o processo de treinamento.

Em nosso modelo, como mostra a Figura 1, utilizamos duas camadas de *Graph Convolution*. A primeira aplicada no grafo original e a segunda aplicada no resultado da primeira depois de passar por um *pooling* de redução do grafo para sua forma espectral mínima [Bianchi et al. 2019].

Uma vez que cada vetor dos blocos básicos contem informações sobre a sua relação estrutural no grafo com os vizinhos e o grafo foi reduzido a forma espectral mínima, aplicamos uma técnica de *pooling* para unir todos os vetores dos nós do grafo em um só e assim represente, agora, não cada bloco básico, mas o código inteiro como um vetor. A técnica de *pooling* soma os valores de todos os vetores dos blocos básicos com um peso de atenção [Li et al. 2017], o qual é aprendido durante o treinamento. O resultado de tais operações é um vetor apenas que representa o inteiro e as informações de todos os blocos básicos. Esse vetor final possui a mesma dimensão que o vetor dos blocos básicos, mas suas dimensões representam diferentes características do código que foram aprendidas a serem extraídas durante o treinamento pela camada de *Graph Convolution* e as camadas de *pooling*. Isto é, no final da aplicação das técnicas temos um vetor que contem informações sobre o código, além de seu GFC, que é utilizado como entrada de uma RNP para previsão de desempenho.

O modelo de RNP utilizado para aprender a extrair o vetor de características do código e prever o desempenho dos códigos está representado no digrama da Figura 1.

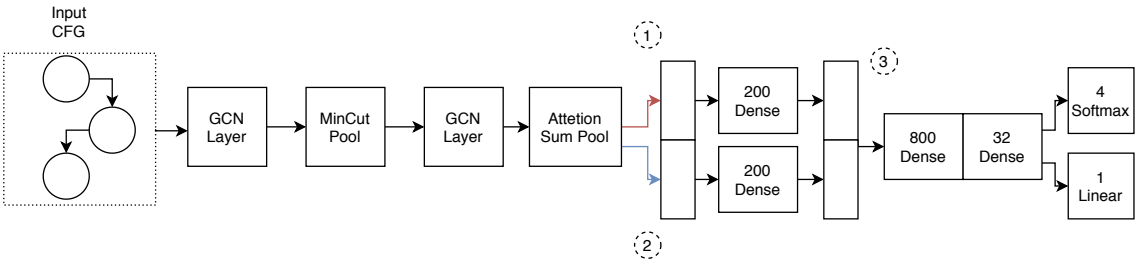

Figura 1. Modelo `GNN` para predição de desempenho

O modelo tem como entrada dois `GFCs` de dois códigos de uma mesma aplicação, otimizados com planos de compilação diferentes. Os blocos básicos desse grafo são anotados com a contagem de cada tipo de instrução. O primeiro `GFC` passa pela camada do `GCN` e de *pooling* gerando o primeiro vetor representando o primeiro código (marcado como (1) no diagrama). Em seguida, o segundo `GFC` passa pelo mesmo processo gerando um segundo vetor que representa o segundo código (marcado com (2) o diagrama). Cada um dos dois vetores passa por uma rede densa de neurônios gerando uma segunda versão de cada vetor. Os dois vetores processados pelas redes densas são então concatenados e o único vetor gerado (marcado com um (3) no diagrama) que serve como entrada para uma rede neural densa com 3 camadas que tem como saída o ganho de desempenho estimado entre as duas aplicações.

Foram gerados planos de compilação para o LLVM para 300 aplicações. Para cada, foram selecionados dois planos para gerar uma tupla (GFC_1, GFC_2, *speedup*), onde GFC_i é o `GFC` para um código ao qual foi aplicado um plano de compilação selecionado e *speedup* é o ganho de desempenho de GFC_1 em relação ao GFC_2.

Para a maior parte dos planos de compilação há pouca ou nenhuma diferença entre o desempenho da aplicação. Portanto, a seleção dos planos foi feito de forma a ter uma proporção igual de tuplas com *speedups* entre 4 classes apresentados na Tabela 1. No fim, cada uma das 4 classes contem 7500 tuplas. Nossa rede foi treinada tanto para prever o valor do *speedup*, valor de regressão, quanto para dizer a classe que o *speedup* entre ambos $CFGs$ se enquadra. Para o problema de classificação um chute teria 25% de chance de dizer a classe correta, dado a distribuição uniforme entre as categorias.

Tabela 1. Categorias dos Speedups Utilizado

	Slowdow Alto	**Slowdown Baixo**	**Insignificativo**	**Speedup Alto**
Valor do Speedup (sp):	$sp < 0.45$	$0.45 \leq sp \leq 0.8$	$0.8 < sp \leq 1.3$	$sp > 1.3$

5. Materiais e Métodos

Para gerar dados para avaliar nosso modelo, utilizamos as aplicações contidas na suite de teste da infraestrutura `LLVM`[3]. Precisamente, extraímos 300 aplicações teste desta infraestrutura, as quais compreendem aplicações de diversas áreas. Após geramos 100 planos de compilação para cada aplicação, e por fim utilizando a infraestrutura `LLVM` geramos o código final e medimos o seu desempenho. Tal desempenho é o valor médio do tempo de

[3]www.llvm.org

execução, para 10 execuções distintas. Como pode ser observado geramos 30000 grafos distintos. O objetivo é ter diversidade e desta forma treinar nosso modelo para distintos casos.

Em uma segunda etapa, extraímos o GFC de cada código gerado. Este GFC representa as relações estáticas contidas nas instruções do código binário final. Cada nó do grafo contém a quantidade de instruções para cada classe de instruções do *hardware* em questão. Mais detalhadamente, cada nó armazena um vetor unidimensional contendo 94 elementos, onde cada elemento representa uma classe de instruções do processador Intel(R) Core(TM) i7-3820 CPU @3.60GHz, arquitetura na qual os experimentos foram realizados.

A rede descrita anteriormente foi treinada com *batch size* de 128 grafos, um *learning rate* de 1×10^{-2}, com ativação *relu* em todas as camadas e com uma taxa de redução de 50% na camada de *pooling* de espectro mínimo. O treinamento foi executado por 2000 épocas em uma instância da Google Cloud com uma NVIDIA P100 e utilizando-se das implementações da biblioteca Spektral[4] com o Tensorflow 2.2 [5].

Nós comparamos nossa abordagem de utilizar uma GCN e o *pooling* com atenção (GCN + *Attention Sum Pool*) com 3 outras abordagens de gerar o vetor que representa os códigos, a saber:

1. somar todos os vetores dos blocos básicos usando um mecanismo de atenção (*Attention Sum Pool*);
2. simplesmente somar os vetores de todos os blocos básicos (*Sum Pool*); e
3. utilizar o vetor do maior bloco básico (*Max Pool*).

A avaliação do nosso modelo de predição é norteada pelas questões descritas a seguir:

- Qual a acurácia alcançada pelo nosso modelo?
- O uso de GCN aumenta a acurácia da predição?
- Qual o valor do erro médio do desempenho?
- Para qual cenário a predição é mais precisa?

A resposta a tais questões indicará se nossa premissa, apresentada na Seção 3, está correta ou não.

6. Resultados e Discussão

Como apresentado na Tabela 2, nosso preditor baseado em GCN foi capaz de classificar corretamente em média 91% dos ganhos nas classes propostas na Seção 4. Além disso, nosso preditor foi capaz de prever o ganho de desempenho com um erro médio relativamente baixo, 0.0766. Outra questão importante a destacar é o fato do modelo de predição ser rápido, em uma NVIDIA P100 o modelo levou em média $16ms$ para realizar cada predição.

Em nossos experimentos, utilizar-se da GCN para aprender a extrair informações da estrutura do fluxo de controle dos códigos se demonstrou significativamente superior a todas as outras formas testadas que não utilizam de tal mecanismo.

[4]https://github.com/danielegrattarola/spektral

[5]https://www.tensorflow.org/

Tabela 2. Desempenho dos modelos de predição

Modelo	Acurácia	Erro Médio
GCN	**91% ± 1.21**	**0.0766 ± 0.0854**
Attention Sum Pool	79% ± 0.92	0.1430 ± 0.1237
Sum Pool	63% ± 1.53	0.3651 ± 0.327
Max Pool	58% ± 1.02	0.4472 ± 0.286

Quando olhamos para a matriz de confusão, apresentada na Figura 2, notamos que a predição é mais precisa para as classes com significativa diferença de desempenho dos códigos. Isto é, quanto maior a diferença de desempenho entre dois códigos, mais fácil de se prever que há uma diferença. A mesma conclusão se dá quando observamos o erro médio do ganho de desempenho predito para as aplicações de cada classe, como apresentado na Tabela 3. Ainda, é possível notar que os erros são sempre maiores nas classes mais próximas. Ou seja, a rede, quando erra a classe do ganho de desempenho tende a cometer esse erro em classes de desempenho similares.

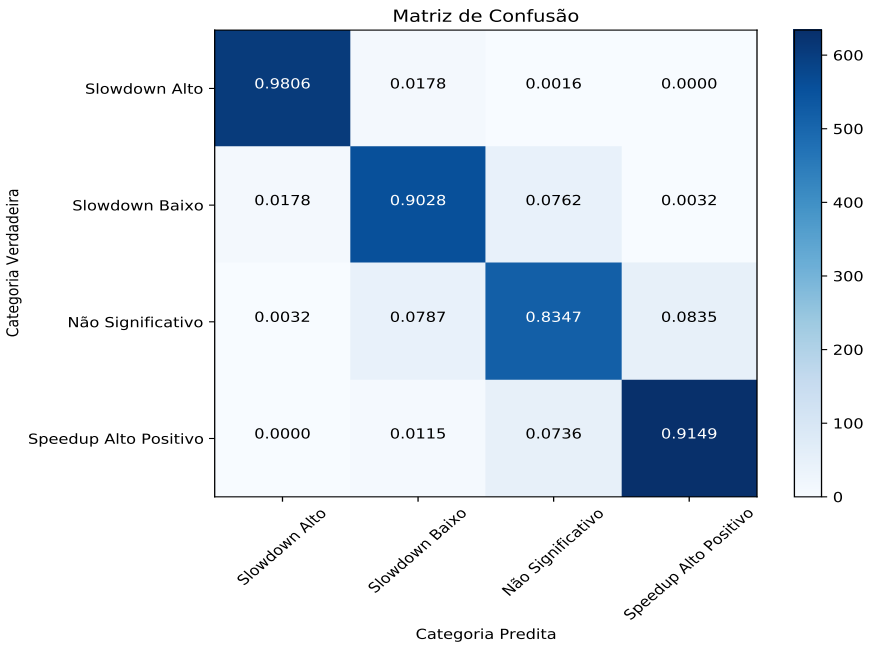

Figura 2. Matriz de confusão

Tabela 3. Erro Médio do Ganho de Desempenho

	Classe			
	Slowdown Alto	**Slowdown Baixo**	**Não Significativo**	**Speedup Alto Positivo**
Erro	0.0583	0.0759	0.1116	0.0602

Por fim, a Figura 3 mostra que no processo de aprendizado o modelo de RNP com a GCN foi capaz de convergir tanto para os dados de treino, quanto para os dados de validação. Além disso, o gráfico mostra que 200 épocas é o suficiente para a convergência da rede. Mesmo treinando por 2 mil épocas, após 200 não se obteve resultados melhores.

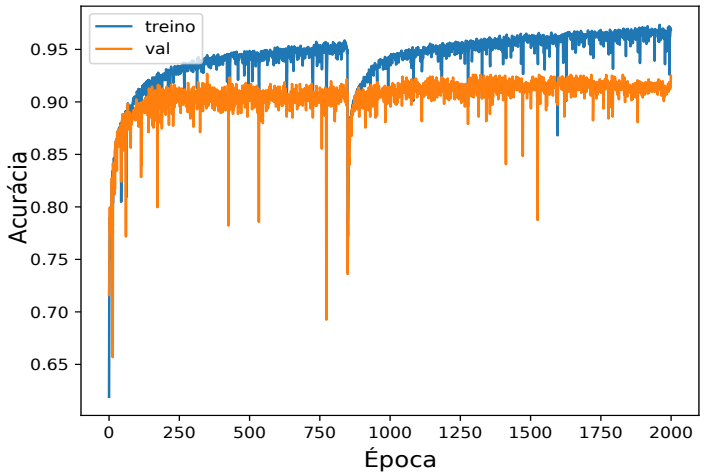

Figura 3. Acurácia no decorrer das épocas. "Val"representa a acurácia para o conjunto de validação e "treino"a acurácia para o conjunto de treino.

7. Conclusões e Trabalhos Futuros

Recentemente, redes neurais baseadas em grafo, ou *Graph Neural Networks* (GNNs), tem ganhado popularidade em diversas áreas, devido a sua capacidade de operar sobre estruturas de grafo de forma a inferir propriedades entre diferentes entidades. Uma área na qual tais redes ainda foram pouco exploradas é a predição de desempenho de aplicações. Programas normalmente são representados na forma de grafos, como de controle ou fluxo, e seus desempenhos estão diretamente ligados a estrutura desses grafos. Portanto, o uso de GNNs parece promissor.

Técnicas de *autotuning* em compiladores, como por exemplo, para encontrar o melhor plano de compilação, são bastante caras porque avaliar seus resultados depende normalmente em executar a aplicação. Técnicas para prever o ganho de desempenho desses planos de compilação existem, mas elas normalmente não utilizam características das estruturas dos grafos das aplicações ou essas características tem que ser automaticamente inseridas e decidir quais inserir por um especialista. Nesse trabalho propomos o uso de GNNs para aprender a extrair automaticamente essas características a serem de um grafo de controle de fluxo e serem aplicadas em uma rede neural. Nosso modelo, baseado em GCN e *Attention Sum Pool*, indica qual a diferença de desempenho entre os dois códigos, como também a classe de desempenho para a qual o modelo acredita estar a entrada.

Os resultados apresentados indicam que nosso modelo possui uma acurácia de 91%, um baixo erro médio e é superior a outras três abordagens testadas que não utilizam uma representação baseada em grafos, sendo, portanto, uma evidência que nossa premissa inicial é verdadeira. O desempenho de um código não está relacionado apenas com a quantidade de instruções e quais instruções da arquitetura alvo foram utilizadas,

mas também com a estrutura do fluxo de controle, e que uma rede GNN é capaz de extrair e aproveitar dessas informações.

Os resultados são promissores e indicam que novas pesquisas podem ser desenvolvidas a partir desta. Um próximo desafio é utilizar esse sistema de previsão de desempenho em uma busca de *autotuning* para testarmos se conseguimos encontrar bons planos de compilação para aplicações.

Referências

Ardalani, N., Lestourgeon, C., Sankaralingam, K., and Zhu, X. (2015). Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 725–737, New York, NY, USA. Association for Computing Machinery.

Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., and Silvano, C. (2018). A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5).

Ashouri, A. H., Mariani, G., Palermo, G., and Silvano, C. (2014). A bayesian network approach for compiler auto-tuning for embedded processors. In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 90–97.

Bastings, J., Titov, I., Aziz, W., Marcheggiani, D., and Sima'an, K. (2017). Graph convolutional encoders for syntax-aware neural machine translation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1957–1967, Copenhagen, Denmark. Association for Computational Linguistics.

Bianchi, F. M., Grattarola, D., and Alippi, C. (2019). Mincut pooling in graph neural networks. *CoRR*, abs/1907.00481.

Brauckmann, A., Goens, A., Ertel, S., and Castrillon, J. (2020). Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, page 201–211, New York, NY, USA. Association for Computing Machinery.

Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. (2017). End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232.

Curtis-Maury, M., Singh, K., McKee, S. A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R., and Schulz, M. (2007). Identifying energy-efficient concurrency levels using machine learning. In *2007 IEEE International Conference on Cluster Computing*, pages 488–495.

Fan, K., Cosenza, B., and Juurlink, B. (2019). Predictable gpus frequency scaling for energy and performance. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA. Association for Computing Machinery.

Filho, J. F., Rodriguez, L. G. A., and da Silva, A. F. (2018). Yet another intelligent code-generating system: A flexible and low-cost solution. *Journal of Computer Science and Technology*, (5):940–965.

Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907.

Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. (2017). Gated graph sequence neural networks. *CoRR*, abs/1511.05493.

Namolaru, M., Cohen, A., Fursin, G., Zaks, A., and Freund, A. (2010). Practical aggregation of semantical program properties for machine learning based optimization. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '10, page 197–206, New York, NY, USA. Association for Computing Machinery.

Ogilvie, W. F., Petoumenos, P., Wang, Z., and Leather, H. (2017). Minimizing the cost of iterative compilation with active learning. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 245–256.

OpenCL (2020). OpenCL framework.

OpenMP (2020). OpenMP framework.

Sarkar, S. and Mitra, S. (2014). Execution profile driven speedup estimation for porting sequential code to gpu. In *Proceedings of the 7th ACM India Computing Conference*, COMPUTE '14, New York, NY, USA. Association for Computing Machinery.

Satorras, V. G. and Estrach, J. B. (2018). Few-shot learning with graph neural networks. In *International Conference on Learning Representations*.

Wen, Y., Wang, Z., and O'Boyle, M. F. P. (2014). Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Yu, P. S. (2019). A comprehensive survey on graph neural networks. *CoRR*, abs/1901.00596.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. (2018). Graph convolutional neural networks for web-scale recommender systems. *CoRR*, abs/1806.01973.