Escolha do Ladrilhamento para um Simulador de Ondas Acústicas em GPUs por meio de Aprendizado de Máquina

Tiago da Silva[1], **Edson Gomi** [2], **Hermes Senger** [1]

[1]Departamento de Computação
Universidade Federal de São Carlos (UFSCar) – São Carlos, SP – Brazil
tiagos@estudante.ufscar.br, hermes@ufscar.br

[2]Escola Politécnica
Universidade de São Paulo (USP) – São Paulo, SP – Brazil
gomi@usp.br

Resumo. *The simulation of acousitc wave propagation is the kernel for important industrial applications like the Full-Waveform Inversion (FWI) and Reverse-Time Migration (RTM). The kernel solves partial differential equations (PDEs) based on the finite differences method, which can be significantly accelerated with the support of GPUs. One of the main challenges for accelerating this stencil computations on GPUs is to reduce the overhead of memory accesses, and tiling is an important optimization which can accelerate wave propagation kernels. However, deciding the tile sizes for these computations is not a straightforward question, which usually depend upon many architectural and application parameters. In the present work, we employ six machine learning methods for providing recommendations for the sizes of tiles to use. Our best strategy has achieved a improvement coefficient of 1.17 and 1.11 on two GPUs with Turing and Volta architectures.*

1. Introdução

Inversão da forma de onda completa (Full-Waveform Inversion, FWI) e a Migração Reversa no Tempo (Reverse-Time Migration, RTM) são exemplos de aplicações industriais que frequentemente ocupam *clusters* de HPC por períodos extensos, desde semanas até meses, para processar dados provenientes de um único local. O *kernel* dessas aplicações é o simulador de propagação de ondas acústicas, que resolve equações diferenciais parciais (PDEs) com base no método de diferenças finitas. Embora a utilização de GPUs ofereçam desempenho notável, um dos principais desafios desse tipo de aplicação reside na redução da sobrecarga de acessos à memória, que é predominante em cálculos do tipo *stencil*. *Stencil* é um padrão de operações em que o cálculo de cada ponto depende do acesso a uma vizinhança de pontos no seu entorno. A otimização de códigos *stencil* costuma ser bastante desafiadora com processadores de alto desempenho [Tadonki 2017, Haggui et al. 2018]. O poder de processamento dos processadores modernos está aumentando, mas seus sistemas de memória estão cada vez mais complexos. Mapear códigos *stencil* de forma eficiente em tais processadores é difícil e ainda está sob estudos intensivos. A otimização com ladrilhamento pode melhorar significativamente o reuso de dados e alivia essa sobrecarga de memória. No entanto, determinar o tamanho ideal

dos ladrilhos é uma questão complexa, especialmente em GPUs. Diversos fatores influenciam a escolha do melhor tamanho dos ladrilhos, como por exemplo o raio do *stencil*, a intensidade operacional, a precisão numérica (por exemplo, se 32-bits ou 64-bits), além de aspectos da arquitetura como o tamanho dos caches envolvidos, a vazão da memória, entre outros.

Neste trabalho, exploramos uma abordagem baseada no aprendizado de máquina para fornecer recomendações sobre os tamanhos de ladrilhos a serem utilizados. Exploramos seis métodos de aprendizado de máquina para recomendar o tamanho dos ladrilhos para um kernel de propagação de onda em duas arquiteturas de GPUs. Focamos na otimização do Simwave [Souza et al. 2022b], um pacote Python/C que permite aos pesquisadores modelarem a propagação de ondas acústicas. Avaliamos o impacto das otimizações de ladrilhamento utilizando duas GPUs distintas: RTX2080 e V100. Os modelos analisados incluem cinco algoritmos de regressão (*K-Nearest Neighbors*, *Árvore de Regressão*, *Random Forest*, *XGBoost* e *LightGBM*) e um modelo de classificação (*J48*).

A análise visa identificar quais modelos são mais eficazes na melhoria do desempenho do *kernel* em termos de tempo de execução e eficiência da recomendação. A avaliação comparativa dos modelos permitirá identificar o mais adequado para simulações acústicas e fornecer percepções para futuras pesquisas e aplicações práticas.

2. Simulação da propagação da onda acústica

A propagação de ondas acústicas é fundamental para a indústria de exploração de petróleo e gás natural. Ela é a base de métodos que permitem o mapeamento e caracterização de materiais existentes no subsolo, tais como os reservatórios de petróleo e gás natural. Esta equação diferencial parcial descreve o comportamento da pressão acústica em função do tempo e das coordenadas espaciais, permitindo a análise da propagação de ondas acústicas em diversos meios, como ar, água, sólidos e estruturas complexas [Virieux and Operto 2009].

2.1. Kernel da propagação da onda

Existem várias equações de onda, baseadas em diferentes modelos físicos. Para este trabalho, usamos a equação simplificada da onda acústica, assumindo um meio isotrópico, densidade constante e desprezando tensões de cisalhamento, conforme definido em [Souza et al. 2022b]. Esta é uma equação diferencial que descreve o deslocamento de partículas, conforme segue:

$$\frac{1}{V^2}\frac{\partial^2 p}{\partial t^2} - \nabla^2 p = f \tag{1}$$

onde v_p é a velocidade do campo da onda de pressão, e $p(\mathbf{x}, t)$ pode ser expandido para 3 dimensões como $p(x, y, z, t)$, bem como o fonte $f(\mathbf{x}, t) = f(x, y, z, t)$. Na equação 2 temos sua forma discretizada para a segunda ordem espacial:

$$P_{i,j,k}^{(n+1)} = 2P_{i,j,k}^{(n)} - P_{i,j,k}^{(n-1)} + \nabla t^2 . v^2 \left(\frac{P_{i+1,j,k}^{(n)} - 2P_{i,j,k}^{(n)} + P_{i-1,j,k}^{(n)}}{\nabla x^2} + \frac{P_{i,j+1,k}^{(n)} - 2P_{i,j,k}^{(n)} + P_{i,j-1,k}^{(n)}}{\nabla y^2} + \frac{P_{i,j,k+1}^{(n)} - 2P_{i,j,k}^{(n)} + P_{i,j,k-1}^{(n)}}{\nabla z^2}\right) \tag{2}$$

A forma discretizada da equação da onda pode ser resolvida computacionalmente conforme mostrado no Algoritmo 1.

Algoritmo 1 Simulação da propagação da onda

Entrada: f: origem da pertubação
Saída: u^n: campo de ondas no intervalo de tempo n, para $n \leftarrow 1$ para T

1: $u^0 \leftarrow 0$
2: **Para** $n \leftarrow 1$ para T **Faça**
3: **Para** cada ponto no campo de ondas u^n **Faça**
4: Resolva Eq. 2 (lado esquerdo) ← Resolva Eq. 2 (lado direito) para campo de ondas u^n
5: **FimPara**
6: **Retorne** u^n
7: **FimPara**

Os valores acessados ao calcular cada célula usando um raio de *stencil* de 8 são exemplificados na Figura 1.

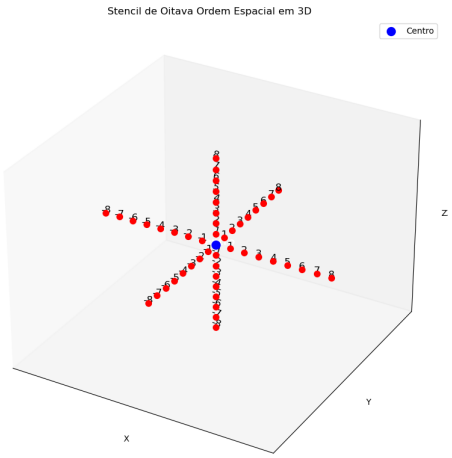

Figura 1. Um estêncil de 49 pontos (8ª ordem espacial)

Utilizamos o simulador de equações de ondas implementado pelo Simwave [Souza et al. 2022b]. O Simwave é distribuído como software aberto [1]. O pacote é composto por *kernels* modulares de *back-end* escritos em C, que fornecem tanto um solucionador numericamente preciso da equação da onda acústica para Geofísicos quanto um *kernel* de aplicação realista para pesquisadores de HPC.

2.2. Ladrilhamento

A otimização por ladrilhamento (do inglês, *tiling*) [Xue 2000, Allen and Kennedy 2001] é uma técnica para a divisão de matrizes em blocos menores chamados ladrilhos, que são processados de forma independente. Essa técnica é particularmente eficaz em situações em que há acessos repetido a dados situados na vizinhança (como é o caso de *stencil*). A figura 3 ilustra essa divisão. O objetivo é que, ao delimitar a computação a uma região menor (ladrilho), os dados trazidos para a memória cache sejam reutilizados na computação de pontos vizinhos, aumentando assim o reuso de dados que já estão nos caches e aumentando o desempenho.

[1] `https://github.com/HPCSys-Lab/simwave.git`

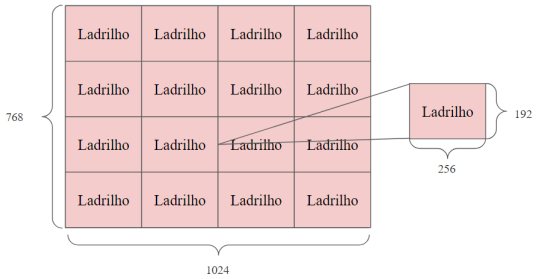

Figura 2. Abordagem de ladrilhamento

Em nossos experimentos utilizamos a abordagem de ladrilhamento implementada no OpenMP em sua versão 5.1 [OpenMP 2020] e implementada no compilador Clang na versão 13.0 [Kruse 2021].

3. Escolha do Tamanho do Ladrilho

Diversos trabalhos abordam o ladrilhamento com estratégia para a melhoria de desempenho de códigos estêncil [Xu et al. 2009], [Korch and Werner 2020] e [Luporini et al. 2020]. Por exemplo, [Luporini et al. 2020] utilizam testes exaustivos em um subespaço de busca com muitos tamanhos de ladrilhos até encontrarem aquele que tem o tempo de execução mais otimizado.

O desempenho da execução do *kernel* é bastante sensível ao tamanho do ladrilho utilizado [Souza et al. 2022a]. Este estudo investigou o desempenho do Simwave com 54 configurações combinando *grids* tridimensionais de tamanhos 256^3, 512^3 e 1024^3, precisões de float32 e float64, ordens espaciais de 2^a, 8^a e 16^a e hardwares GPU dos modelos RTX 2080 Super (Turing), V100 (Volta) e A100 (Ampere), porém a GPU RTX 2080 Super (Turing) não suportou *grids* de 1024, portanto 6 configurações foram retiradas, restando 48 configurações. Foram feitos testes com diversos tamanhos de ladrilhos tridimensionais, resultando em 13.816 experimentos, para encontrar o melhor tamanho de ladrilho. Cada experimento foi repetido 3 vezes. Na tabela 1 temos um exemplo dos resultados obtidos com os experimentos realizados por [Souza et al. 2022a]. Utilizamos esse conjunto de dados para extrair conhecimento sobre o melhor tamanho do ladrilho.

GPU	Grid	dtype	ordem espacial	tile 1	tile 2	time 3	Tempo de execução
a100	256	64	2	8	1	4	0.1489
a100	256	64	8	1	8	2	0.1535
a100	256	64	16	1	4	2	0.1674
a100	512	32	2	32	1	4	0.3016
a100	512	32	8	64	2	1	0.3258
a100	512	32	16	8	4	1	0.3597
rtx2080	256	32	2	2	16	4	0.2050
rtx2080	256	32	8	8	4	1	0.2188
rtx2080	256	32	16	64	2	1	0.2204
rtx2080	512	64	2	2	4	4	1.7565
rtx2080	512	64	8	16	2	1	1.8283
rtx2080	512	64	16	64	1	1	1.9292
v100	256	64	2	2	4	1	0.2310
v100	256	64	8	4	1	1	0.2329
v100	256	64	16	32	1	1	0.2462
v100	512	32	2	64	4	1	0.6302
v100	512	32	8	32	2	1	0.6946
v100	512	32	16	16	1	1	0.7136

Tabela 1. Desempenho em segundos para diferentes GPUs e configurações

4. Trabalhos Relacionados

O trabalho de [Souza et al. 2022a] avaliou o desempenho das técnicas de ladrilhamento e desenrolamento de laços em GPUs utilizando OpenMP. Os testes foram realizados em três GPUs diferentes: RTX 2080 Super, V100 e A100. A combinação dessas técnicas resultou em um aumento de desempenho de até 2.93 vezes em comparação ao código original, destacando a sensibilidade do desempenho ao tamanho do ladrilho escolhido. O presente trabalho tem o objetivo de investigar técnicas para a seleção automática do tamanho do ladrilho em GPUs, utilizando aprendizagem de máquina, visto que o tamanho do ladrilho impacta diretamente no desempenho da execução [Souza et al. 2022a, Korch and Werner 2020, Luporini et al. 2020]. No contexto de GPUs, os primeiros trabalhos surgiram nos anos 2000, a partir do lançamento do CUDA [Kirk et al. 2023].

[Xu et al. 2009] investigou a otimização de desempenho em GPUs utilizando o ladrilhamento. Os testes comparativos entre diferentes modelos de GPUs revelaram que as dimensões ótimas de ladrilho variam conforme o modelo. O objetivo deste trabalho foi mostrar a possibilidade de aumento do desempenho ao aplicar o ladrilhamento em GPU. O autor destaca que é recomendando ajustes específicos no tamanho do ladrilho para cada GPU, um mesmo tamanho pode ser otimizado pra um modelo e para o outro não.

[Rahman et al. 2010] desenvolveu um preditor de ladrilho baseado em uma rede neural treinada com tempos de execução empíricos em CPU. A técnica superou a busca aleatória, identificando tamanhos de ladrilhos com bom desempenho de forma eficiente. O autor apresenta em sua conclusão que os ladrilhos recomendados por essa rede neural ficam dentro de 10% do tempo de execução globalmente ideal.

[Malik 2012] utilizou redes neurais artificiais para criar um modelo de seleção de tamanhos de ladrilho para CPU, baseado em dados de desempenho e características dinâmicas do programa. A técnica demonstrou eficácia, alcançando desempenhos próximos ao ótimo em diversas arquiteturas e compiladores. O autor destaca em sua conclusão que que um modelo de seleção de tamanho de ladrilho razoavelmente preciso pode ser aprendido automaticamente por máquina nesse contexto, e que seus resultados ficaram em torno de 4% entre o recomendado e o mais otimizado.

[Grosser et al. 2014] apresentou uma abordagem híbrida de ladrilhamento hexagonal e clássico para cálculos iterativos de *stencil* em GPUs, integrando-a no compilador poliédrico PPCG. A técnica mostrou melhorias significativas, com um aumento de velocidade de até 920% para certos kernels 2D, embora a complexidade da modelagem e a dependência de funções manualmente derivadas representem desafios. O foco desse trabalho foi mostrar como o ladrilhamento hexagonal combinado com clássico pode entregar mais desempenho em relação a compiladores de *stencil* de última geração. Nota-se que o autor usou *loops* temporais e espaciais nos experimentos.

[Liu et al. 2018] apresentou uma abordagem de aprendizado de máquina para prever tamanhos ótimos de ladrilhos para CPU, utilizando características dos *loops* e uma rede neural de regressão. Os testes em diferentes CPUs mostraram que o

modelo alcança desempenho próximo ao ideal. O foco deste trabalho foi em extrair características dos *loops* que empregam ladrilhamento para capturar a localidade dos acessos aos dados em todas as dimensões do *loop* e o efeito da vetorização e com isso treinarem uma rede neural para prever um tamanho de ladrilho otimizado para aquele contexto.

5. Experimentos

O trabalho de [Souza et al. 2022a] realizou um conjunto de 13.816 experimentos que avaliam diferentes tamanhos de ladrilhos em três arquiteturas de GPUs em diversos cenários que executam uma aplicação semelhante. Nosso trabalho partiu desse conjunto de dados para buscar métodos de extração de conhecimento capazes de fornecer recomendações sobre o melhor tamanho de ladrilho para um novo problema. O primeiro passo foi remover os outliers utilizando o método de Tukey [Tukey et al. 1977], que remove os outliers do conjunto de dados filtrando valores que estão fora de 1.5 vezes o intervalo interquartil. O resultado é ilustrado na Fig. 5. Foram retirados 1.701 experimentos, que representam 12% da base, restando 12.117 experimentos.

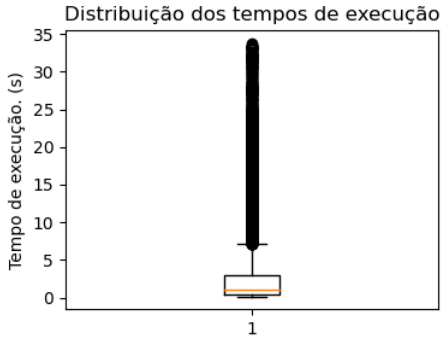

Figura 3. Dispersão dos tempos de execução do conjunto de experimentos de [Souza et al. 2022a].

5.1. Abordagem por classificação

Predizer o melhor tamanho de ladrilho significa fornecer uma tripla que indica o tamanho do ladrilho nas suas três dimensões. São necessárias três dimensões, porque a compilação irá produzir blocos tridimensionais de *threads* na GPU. Na abordagem de classificação, a classe a ser predita pelo algoritmo é a tripla que contém a recomendação do tamanho do ladrilho nessas três dimensões.

Utilizamos o classificador J48 disponibilizado na ferramenta Weka [Witten and Frank 2005], que é uma implementação do algoritmo C4.5. Esse classificador foi selecionado por utilizar árvores de decisão como forma de representação do conhecimento, que é altamente interpretável. Para avaliar o modelo, empregamos a técnica de validação cruzada com 10 pastas, um procedimento estatisticamente rigoroso que divide o conjunto de dados em 10 subconjuntos, utilizando 9 para o treinamento e 1 para o teste em cada iteração. Ao todo, são realizadas 10 iterações alternando a pasta utilizada para o teste.

Removidos os outliers, a base de dados com 12.117 instâncias foi submetida ao algoritmo. O classificador produziu uma árvore de decisão composta por 44 folhas. Cada folha da árvore representa uma regra de decisão derivada dos dados, fornecendo um modelo de fácil compreensão das condições que levam a cada resultado.

5.2. Abordagem por regressão

Uma segunda abordagem baseada em regressão foi empregada para a recomendação do melhor tamanho do ladrilho. Dado uma configuração do problema, utilizamos a regressão para prever os tempos de execução da aplicação para um conjunto de possíveis tamanhos de ladrilho. A seguir, buscamos a configuração que produz o menor tempo de execução dentre o conjunto analisado. Foram avaliados os seguintes algoritmos de regressão:

- **K-Nearest Neighbors (KNN):** Um modelo baseado em distâncias que faz previsões com base nos k vizinhos mais próximos [Fix and Hodges 1951].
- **Árvore de Regressão:** Um modelo que utiliza uma estrutura hierárquica de decisões para prever valores contínuos [Breiman et al. 1986].
- **Random Forest:** Um conjunto de árvores de regressão que melhora a robustez e a precisão por meio da agregação de múltiplas árvores [Breiman 2001].
- **XGBoost:** Um modelo que combina árvores de regressão em um processo iterativo para melhorar a precisão da previsão [Chen and Guestrin 2016].
- **LightGBM:** Um algoritmo que utiliza uma abordagem baseada em histogramas para melhorar a eficiência e a velocidade [Meng et al. 2017].

Para os modelos KNN, Árvore de regressão e *Random Forest* foi utilizada a biblioteca scikit-learn em sua versão 1.3.0, já para os modelos XGBoost e LightGBM foram utilizadas suas próprias bibliotecas, com seus mesmos nomes, nas versões 2.0.3 e 4.0.3 respectivamente.

Para a otimização dos hiperparâmetros dos modelos, foi utilizada a biblioteca **Optuna** [Akiba et al. 2019], que disponibiliza um *framework* para otimização de hiperparâmetros. Utilizou-se um total de 100 testes para explorar o espaço de hiperparâmetros e identificar as melhores configurações para cada modelo. Cada modelo foi avaliado com base na precisão das previsões dos tempos de execução, sendo o desempenho analisado para determinar qual abordagem ofereceu a maior capacidade preditiva.

Os melhores hiperparâmetros encontrados para cada modelo foram os seguintes:

- **K-Nearest Neighbors (KNN):** $k = 2$ vizinhos.
- **Árvore de Regressão:** *max_depth* igual a 19 e *min_samples_split* igual a 2.
- **Random Forest:** *n_estimators* igual a 23, *max_depth* igual a 10 e *min_samples_split* igual a 14.
- **XGBoost:** *eta* igual a 0.11974855714013455, *max_depth* igual a 8, *subsample* igual a 0.5 e *colsample_bytree* igual a 1.
- **LightGBM:** *num_leaves* igual a 0.11974855714013455, *learning_rate* igual a 8, *feature_fraction* igual a 0.5, *bagging_fraction* igual a 1 e *bagging_freq* igual a 0.

Na tabela 2 temos as *features* utilizadas no treinamento e predição dos modelos de regressão.

Feature	Descrição
Quantidade de SMs	Quantidade de Multiprocessadores de Streaming (SM)
Total de Núcleos	Número total de núcleos CUDA
Largura de Banda da Memória	Taxa de transferência de dados da memória
Memória Compartilhada	Memória acessível por todos os threads de um bloco
Cache L2	Tamanho do cache de nível 2
Tamanho do *grid*	Quantidade de pontos em cada eixo do grid tridimensional
Tipo de Dados (dtype)	Tipo de dados usado nas operações
Ordem do Espaço	Precisão da discretização espacial
Tamanho do ladrilho eixo x	Quantidade de pontos no eixo x do ladrilho
Tamanho do ladrilho eixo y	Quantidade de pontos no eixo y do ladrilho
Tamanho do ladrilho eixo z	Quantidade de pontos no eixo z do ladrilho

Tabela 2. Tabela de *features* utilizadas nos modelos de regressão

5.3. Metodologia de testes

A escolha do melhor tamanho do ladrilho pode depender tanto de parâmetros da aplicação quanto da arquitetura do hardware, como tamanho dos caches, vazão da comunicação com a memória, hierarquia de memória, quantidade de *threads* por multiprocessador (SMs). Além disso, o desempenho do ladrilhamento também pode ser influenciado por diferentes parâmetros da aplicação, tais como a precisão (32 ou 64 bits), a ordem espacial, o raio do estêncil (que depende da ordem espacial utilizada na discretização), o tamanho do grid, etc. Assim, os experimentos foram realizados em duas arquiteturas de GPU, variando o tamanho do grid, a precisão e a ordem espacial. Assim, os experimentos comparam os tempos de execução obtidos com as recomendações do tamanho do ladrilho e os tempos de execução sem quaisquer otimizações (*baseline*). Os valores representam uma média de três execuções.

5.4. Configuração dos Experimentos

As recomendações dos tamanhos do ladrilho geradas pelos seis modelos foram testadas em diferentes configurações de GPU e da aplicação. Para cada configuração, comparamos os tempos de execução obtidos com a recomendação e sem o ladrilhamento (*baseline*). Nosso *baseline* consiste no mesmo kernel implementado em OpenMP, ela só não contempla a otimização por ladrilhamento. Abaixo temos o trecho de código utilizado como *baseline*:

```
for(int t = 0; t < time_steps; t++) {
  #pragma omp target teams distribute parallel for \ collapse(3)
  for(int i = radius; i < d1 - radius; i++){
    for(int j = radius; j < d2 - radius; j++){
      for(int k = radius; k < d3 - radius; k++){
        ...
        for(int ir = 1; ir <= radius; ir++){
          // stencil point calculation
```

Este é o trecho de código otimizado com ladrilhamento e implementado em OpenMP que foi utilizado nos experimentos:

```
for(int t = 0; t < time_steps; t++) {
  #pragma omp target teams distribute parallel for \ collapse(3)
  #pragma omp tile sizes(BLOCK1,BLOCK2,BLOCK3)
  for(int i = radius; i < d1 - radius; i++){
```

```
5     for(int j = radius; j < d2 - radius; j++){
6       for(int k = radius; k < d3 - radius; k++){
7         ...
8         for(int ir = 1; ir <= radius; ir++){
9           // stencil point calculation
```

Testamos representantes de 2 arquiteturas de GPU: RTX 2080 Super (Turing) e V100 (Volta), descritas na Tabela 3. Nosso código de referência é uma versão simplificada do Simwave [Souza et al. 2022b] implementado em C e otimizado usando ladrilhamento suportado pelo *Clang* 13.0 [Kruse 2021]. Usamos as flags -O3 -fPIC -ffast-math -fopenmp -fopenmp-version=51 -fopenmp-targets=nvptx64 -Xopenmp-target. O CUDA 11.0 e o ambiente de software foram configurados em um contêiner *Singularity* executando em um *cluster Slurm*.

A seguir, listamos as configurações testadas nos experimentos:

- **Técnicas de recomendação do ladrilho:** KNN, árvore de regressão, random forest, XGBoost, LightGBM e J48
- Os parâmetros utilizados nas configurações incluíram:
  - **Precisão:** Float32 e Float64
  - **Ordens Espaciais:** 2^a, 8^a e 16^a
  - **RTX2080:**
    - **Grids:** $200^3, 250^3, 300^3, 350^3, 400^3, 450^3, 500^3, 550^3, 600^3, 650^3$
  - **V100:**
    - **Grids:** $200^3, 250^3, 300^3, 350^3, 400^3, 450^3, 500^3, 550^3, 600^3, 650^3, 700^3, 750^3, 800^3, 850^3, 900^3, 950^3, 1000^3, 1050^3, 1100^3$

Tabela 3. Atributos dos Hardwares Utilizados

Atributo	RTX 2080 Super	V100
Arquitetura	Turing	Volta
SMs	48	80
CUDA Cores por SM	64	64
CUDA Cores Totais	3072	5120
Peak FP64 TFLOPS	0.35	7.8
Peak FP32 TFLOPS	11.2	15.7
Tamanho da Memória	8 GB	32 GB
Largura da Banda de Memória	496 GB/s	900 GB/s
Memória Compartilhada	64 KB	96 KB
Tamanho do Cache L2	4 MB	6 MB

Para a GPU RTX2080 foram um total de 60 configurações distintas, cada uma testada nos 6 modelos por 3 vezes, totalizando 360 experimentos e 1080 execuções. Para a GPU V100 foram um total de 114 configurações distintas, totalizando 684 experimentos e 2052 execuções. Cada configuração distinta é composta por precisão, ordem espacial e tamanho de *grid*.

5.5. Resultados dos experimentos

Neste experimento, avaliamos o desempenho de cinco modelos de regressão (KNN, Árvore de Regressão, *Random Forest*, *XGBoost* e *LightGBM*) e um modelo de classificação (*J48*). Os resultados são apresentados nas tabelas 4 e 5.

Tabela 4. Resultados dos experimentos de 60 configurações distintas na RTX2080.

	J48	KNN	Regr. tree	Random Forest	XGBoost	LightGBM
% de configurações c/ melhora	86,67%	87,78%	86,67%	80,00%	77,22%	56,67 %
Coeficiente de melhoria	1,12	1,14	1,17	1,10	1,12	1,07

Tabela 5. Resultados dos experimentos em 114 configurações distintas na V100.

	J48	KNN	Regr. tree	Random Forest	XGBoost	LightGBM
% de configurações c/ melhora	74,85%	76,90%	89,77%	73,68%	85,09%	48,83 %
Coeficiente de melhoria	1,08	0,78	1,11	1,05	1,10	1,09

O percentual de configurações com melhora é dado pelo número de experimentos que executaram mais rápido do que o *baseline*, ou seja, para o melhor resultado, modelo *regression tree*, a cada 100 experimentos na GPU V100, 89 tiveram um desempenho superior aos mesmos experimentos no *baseline*. O coeficiente de melhoria representa a média das razões entre os tempos de execução do experimento e *baseline*, exemplificando, o coeficiente de melhoria 1,11 para o modelo *regression tree* na GPU V100 significa que seus experimentos tiveram, em média, 11% de desempenho superior aos mesmos experimentos no *baseline*.

6. Conclusão

A escolha do tamanho do ladrilho tem grande impacto no desempenho de códigos *stencil* em GPUs. A escolha do melhor tamanho pode variar muito em função de vários parâmetros de cada arquitetura e de diversos parâmetros da aplicação. Neste estudo, avaliamos o desempenho de seis modelos de aprendizado de máquina (cinco de regressão e um de classificação) em termos de melhoria e piora no desempenho ao aplicar otimizações de ladrilhamento em duas GPUs distintas: RTX2080 e V100.

Dentre os métodos avaliados, concluímos que a Árvore de Regressão se mostrou consistentemente como o modelo mais eficaz em ambas as GPUs, apresentando melhorias significativas e mínimas perdas de desempenho. Esse modelo teve uma alta taxa de sucesso nas recomendações, mais de 75% em ambos os casos, e teve o maior coeficiente de melhoria em ambas GPUs. A estratégia proposta pode auxiliar na melhoria de desempenho das aplicações em cenários desconhecidos, para os quais o ladrilhamento ainda não foi testado, levando a um bom desempenho sem ter a necessidade de testar um grande número de configurações.

Agradecimentos

Os autores agradecem à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP - Projetos de número 2019/26702-8, 2021/00199-8 e 2023/00566-6). H.S. agradece ao Conselho Nacional de Pesquisa e Desenvolvimento - CNpq (Processo 302296/2023-9) e a FINEP (Convênio 01.23.0575.00 - 0381/23) pelo apoio recebido. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Referências

[Akiba et al. 2019] Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2623–2631, New York, NY, USA. ACM.

[Allen and Kennedy 2001] Allen, R. and Kennedy, K. (2001). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, San Francisco, CA.

[Breiman 2001] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

[Breiman et al. 1986] Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1986). *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA.

[Chen and Guestrin 2016] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM.

[Fix and Hodges 1951] Fix, E. and Hodges, J. (1951). Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review*, 20(1):1–30.

[Grosser et al. 2014] Grosser, T., Cohen, A., Sadayappan, P., Holewinski, J., and Verdoolaege, S. (2014). Hybrid hexagonal/classical tiling for gpus. In *Proc. of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Orlando, FL, USA. ACM.

[Haggui et al. 2018] Haggui, O., Tadonki, C., Lacassagne, L., Sayadi, F., and Ouni, B. (2018). Harris corner detection on a NUMA manycore. *Future Gener. Comput. Syst.*, 88:442–452.

[Kirk et al. 2023] Kirk, D. B., mei W. Hwu, W., and Hajj, I. E. (2023). *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Inc., 4nd edition.

[Korch and Werner 2020] Korch, M. and Werner, T. (2020). Improving locality of explicit one-step methods on gpus by tiling across stages and time steps. *Future Generation Computer Systems*, 102:889–901.

[Kruse 2021] Kruse, M. (2021). Loop transformations using clang's abstract syntax tree. In *50th Intl. Conference on Parallel Processing Workshop*, pages 1–7.

[Liu et al. 2018] Liu, S., Cui, Y., Jiang, Q., Wang, Q., and Wu, W. (2018). An efficient tile size selection model based on machine learning. *Journal of Computer Science and Technology*.

[Luporini et al. 2020] Luporini, F., Louboutin, M., Lange, M., Kukreja, N., Witte, P., Hückelheim, J., and Gorman, G. J. (2020). Architecture and performance of devito, a system for automated stencil computation. *ACM Transactions on Mathematical Software*, 46(1):1–28.

[Malik 2012] Malik, A. M. (2012). Optimal tile size selection problem using machine learning. In *2012 11th International Conference on Machine Learning and Applications*, USA. IEEE.

[Meng et al. 2017] Meng, Q., Ma, Z., Li, H., Leskovec, J., Zhang, X., and et al, K. H. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *31st Conference on Neural Information Processing Systems (NeurIPS)*.

[OpenMP 2020] OpenMP (2020). OpenMP Examples Updated with 5.1 Features. `https://www.openmp.org/openmp-updates/openmp-examples-updated-with-5-1-features`. Online; accessed 13 February 2024.

[Rahman et al. 2010] Rahman, M., Pouchet, L.-N., and Sadayappan, P. (2010). Neural network assisted tile size selection. *The Ohio State University*. {rahmanm,pouchet,saday}@cse.ohio-state.edu.

[Souza et al. 2022a] Souza, J. F. D., Machado, L. S., Gomi, E. S., Tadonki, C., McIntosh-Smith, S., and Senger, H. (2022a). Performance of openmp offloading for the acoustic wave stencil on gpus. In *Supercomputing*, Dallas, TX, USA.

[Souza et al. 2022b] Souza, J. F. D., Moreira, J. B. D., Roberts, K. J., di Ramos Alves Gaioso, R., Gomi, E. S., Silva, E. C. N., and Senger, H. (2022b). simwave - a finite difference simulator for acoustic waves propagation. *arXiv*.

[Tadonki 2017] Tadonki, C. (2017). Scalable numa-aware wilson-dirac on supercomputers. In *2017 International Conference on High Performance Computing & Simulation, HPCS 2017, Genoa, Italy, July 17-21, 2017*, pages 315–324. IEEE.

[Tukey et al. 1977] Tukey, J. W. et al. (1977). *Exploratory data analysis*, volume 2. Springer.

[Virieux and Operto 2009] Virieux, J. and Operto, S. (2009). An overview of full-waveform inversion in exploration geophysics. *Geophysics*, 74(6):WCC1–WCC26.

[Witten and Frank 2005] Witten, I. H. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann series in data management systems. Morgan Kaufmann, 2nd edition. Library of Congress Cataloging-in-Publication Data.

[Xu et al. 2009] Xu, C., Kirk, S. R., and Jenkins, S. (2009). Tiling for performance tuning on different models of gpus. In *Second International Symposium on Information Science and Engineering*, page 60. IEEE.

[Xue 2000] Xue, J. (2000). *Loop Tiling for Parallelism*, volume SECS 575 of *Kluwer International Series in Engineering and Computer Science*. Springer Science+Business Media New York, New York. Originally published by Kluwer Academic Publishers, New York in 2000.