Universidade Federal de São Carlos– UFSCar
Centro de Ciências Exatas e de Tecnologia– CCET
Departamento de Computação– DC
Programa de Pós-Graduação em Ciência da Computação– PPGCC

Tiago da Silva

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

São Carlos
2024

Tiago da Silva

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

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Sistemas Distribuídos e Redes

Orientador: Hermes Senger

São Carlos
2024

Agradecimentos

Agradecemos pelo apoio recebido das seguintes agências de fomento:

- Fundação de Amparo à Pesquisa do Estado de São Paulo (Processos 2019/26702-8, 2021/00199-8 e 2023/00566-6).
- Conselho Nacional de Pesquisa e Desenvolvimento-CNPq (Processo 302296/2023-9);
- Financiadora de Estudos e Projetos - FINEP / Ministério da Ciência, Tecnologia e Inovações - MCTI / Fundo Nacional de Desenvolvimento Científico e Tecnológico – FNDC (Convênio 01.23.0575.00 - 0381/23);
- Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Código de Financiamento 001.

Resumo

A simulação da propagação da onda acústica é crucial em áreas como geofísica e imagem sísmica, sendo modelada por métodos numéricos, como o de diferenças finitas (FDM). Essas simulações são intensivas em recursos, especialmente em problemas de grande escala com *grids* 3D e múltiplos passos de tempo. O uso de GPUs tem se mostrado promissor devido ao seu poder de processamento paralelo, mas um desafio é a sobrecarga de acessos à memória. O *tiling*, que divide o *grid* em blocos menores, melhora a localidade dos dados, otimizando o acesso à memória e aumentando o desempenho. Entretanto, determinar o tamanho do *tile* para uma dada aplicação não é uma tarefa simples. Esse valor depende de diversos fatores, como a arquitetura da GPU, o tamanho do problema e as características específicas dos dados processados. A escolha do tamanho do *tile* é afetada diretamente pela utilização da memória cache, a largura de banda da memória e o paralelismo do cálculo, tornando a questão bastante complexa e sujeita a várias trocas de eficiência.

Neste estudo, utilizamos aprendizado de máquina para prever tamanhos otimizados de *tiles* na simulação de ondas acústicas. Avaliamos seis algoritmos (KNN, Árvore de Decisão, Random Forest, XGBoost, LightGBM e J48), e os resultados mostraram uma melhoria significativa, com o melhor modelo alcançando coeficientes de melhoria de 1,17 para a GPU Turing (RTX2080) e 1,11 para a Volta (V100), além de uma taxa de sucesso superior a 75% para ambas as GPUs.

Palavras-chave: acoustic wave simulation, stencil, GPU, OpenMP, performance, efficiency, parallel programming, loop tilling, loop blocking, partitioned matrix, blocking.

Abstract

The simulation of acoustic wave propagation is crucial in fields such as geophysics and seismic imaging, being modeled by numerical methods such as finite difference methods (FDM). These simulations are resource-intensive, especially in large-scale problems with 3D *grids* and multiple time steps. The use of GPUs has shown promise due to their parallel processing power, but one challenge is the memory access overhead. *Tiling*, which divides the *grid* into smaller blocks, improves data locality, optimizing memory access and increasing performance. However, selecting the optimal tile size for a given computation is not a trivial task. The optimal tile size depends on a variety of factors, including the specific architecture of the GPU, the size of the problem being solved, and the characteristics of the data being processed. In practice, the optimal tile size can vary significantly depending on the GPU's memory hierarchy, the bandwidth between the processor and memory, and the computational intensity of the kernel. Moreover, the choice of tile size can also affect the parallelism and load balancing of the computation, making it a complex trade-off that requires careful tuning.

In this study, we used machine learning to predict optimized *tile* sizes for acoustic wave simulations. We evaluated six algorithms (KNN, Decision Tree, Random Forest, XGBoost, LightGBM, and J48), and the results showed significant improvement, with the best model achieving improvement coefficients of 1.17 for the Turing GPU (RTX2080) and 1.11 for the Volta GPU (V100), as well as a success rate of over 75% for both GPUs.

Keywords: acoustic wave simulation, stencil, GPU, OpenMP, performance, efficiency, parallel programming, loop tilling, loop blocking, partitioned matrix, blocking.

Lista de ilustrações

Lista de tabelas

Lista de algoritmos

Sumário

Capítulo 1

Introdução

As ondas acústicas são uma forma de propagação de energia via um meio no espaço. Essas ondas viajam com uma velocidade característica e apresentam fenômenos como difração, reflexão e interferência ao interagir com o meio. A sua simulação é muito utilizada na resolução de problemas complexos, como na área sísmica, para localizar depósitos de matérias-primas, como petróleo e gás, em domínios que são difíceis de observar fisicamente. (Souza et al., 2022b)

A sua utilização implica no cálculo de domínios bidimensionais ou tridimensionais que representam um recorte do espaço físico, calculando ponto a ponto dessa representação em um algoritmo com alta aderência ao paralelismo. Com o aumento da complexidade dos desafios relacionados à compreensão e análise de ondas acústicas, como a modelagem de ambientes complexos e a detecção de padrões sonoros em dados massivos, torna-se imperativo o desenvolvimento de abordagens mais sofisticadas. (Virieux and Operto, 2009)

A utilização das arquiteturas GPU em algoritmos complexos, como a simulação da onda acústica, representa uma estratégia altamente vantajosa devido à sua capacidade em lidar com tarefas computacionais intensivas paralelamente. Em comparação com arquiteturas CPU, as GPUs proporcionam um aumento significativo no desempenho por contarem com uma quantidade muito maior de núcleos de processamento e terem sido desenvolvidas para permitir a execução simultânea de múltiplos cálculos. Nesse contexto, a paralelização proporcionada pelas GPUs acelera consideravelmente o tempo de processamento, viabilizando a modelagem precisa desses fenômenos acústicos complexos. (Weiss and Shragge, 2013)

Diversas interfaces de programação paralela foram desenvolvidas para aproveitar os recursos das arquiteturas GPU. Entre elas, destacam-se OpenMP, OpenACC, MPI e

CUDA. Essas interfaces oferecem recursos para distribuir, de maneira mais eficiente, a carga de trabalho e utilizar melhor os recursos dessa arquitetura.

Não somente a utilização do potencial paralelo dessas arquiteturas, mas também o uso de técnicas avançadas de programação paralela pode melhorar significativamente o desempenho no tempo de processamento. A abordagem de ladrilhamento, do inglês *tiling* (Xue, 2000), uma dessas técnicas avançadas, proporciona o reuso dos dados nas memórias caches, separando os cálculos de forma mais eficiente. Ela propõe um alinhamento dos cálculos conforme a sua proximidade na representação do espaço, considerando um algoritmo que calcula matrizes, e com isso, reutiliza os valores de posições vizinhas dessa matriz que já estão disponíveis na memória cache, não sendo necessário o tempo de uma nova cópia de dados da memória global.

O grande desafio na aplicação da abordagem de ladrilhamento é a escolha do tamanho do ladrilho, que influencia diretamente no desempenho de sua aplicação. Alguns autores como Korch and Werner (2019), Grosser et al. (2014) e Xu et al. (2009) comentam em seus estudos que os possíveis tamanhos ideais são diferentes dependendo do hardware e algoritmo utilizado, podendo ser encontrados por testes exaustivos com diferentes tamanhos.

Diversos trabalhos abordam o ladrilhamento com estratégia para a melhoria de desempenho de códigos estêncil como Li and Song (2004), Malik (2012), Liu et al. (2018), Rahman et al. (2010), Souza et al. (2022a), Korch and Werner (2019), Grosser et al. (2014) e Xu et al. (2009).

1.1 Objetivos da Pesquisa

Este trabalho visa investigar e propor uma solução automatizada, empregando técnicas de aprendizado de máquina, para a definição do tamanho do ladrilho no algoritmo de simulação de ondas Simwave (Souza et al., 2022b), uma das ferramentas utilizadas para realizar simulações nessa área. Para isso, avaliamos seis modelos de aprendizado, incluindo cinco de regressão (*K-Nearest Neighbors*, Árvore de Regressão, *Random Forest*, *XGBoost* e *LightGBM*) e um de classificação (*J48*). A proposta tem em vista encontrar tamalhos de *tile* que otimizem o desempenho do algoritmo, reduzindo o tempo de execução e os acessos à memória, com base em uma análise que considera tanto parâmetros arquiteturais das GPUs, quanto configurações específicas da aplicação.

1.2 Metodologia

Este trabalho se fundamenta em dados experimentais previamente coletados pelo nosso grupo, cujos resultados foram apresentados em Souza et al. (2022a). A partir desse conjunto de dados, propomos aplicar diferentes técnicas de aprendizado de máquina para

avaliar e identificar a abordagem mais precisa na seleção do tamanho do *tile*. O objetivo encontrar automaticamente tamanhos de *tile* que reduzam o tempo de execução, considerando as características específicas da arquitetura GPU onde o Simwave será executado.

1.3 Organização do Trabalho

Este trabalho está organizado em cinco capítulos. O Capítulo 1 apresenta a introdução e objetivos da pesquisa. O Capítulo 2 aborda os fundamentos teóricos e tecnológicos que embasam o trabalho. A seguir, o Capítulo 3 apresenta os principais trabalhos relacionados à pesquisa sobre ladrilhamento em GPUs. Os experimentos realizados e os resultados obtidos serão apresentados no Capítulo 4, destacando a aplicação das estratégias propostas e a análise detalhada conduzida ao longo desta pesquisa. Por fim, o Capítulo 5 apresenta as conclusões finais desta pesquisa, bem como uma discussão sobre as contribuições alcançadas e as possíveis direções para trabalhos futuros.

Capítulo 2

Fundamentos teóricos do trabalho

Este capítulo está dividido em: fundamentos da simulação da propagação da onda acústica, seção 2.1; fundamentos dos códigos estênceis, seção 2.2; fundamentos da abordagem explorada no estudo, seção 2.3; fundamentos das técnicas de aprendizagem de máquina, seção 2.4; e fundamentos do hardware utilizado para os estudos, seção 2.5.

2.1 Simulação da propagação da onda acústica

2.1.1 Equação da onda acústica

A equação da onda acústica é um conceito central na área da acústica e desempenha um papel fundamental na compreensão e modelagem da propagação das ondas sonoras em meios materiais. Essa equação diferencial parcial descreve o efeito da pressão acústica em função do tempo e das coordenadas espaciais, permitindo investigar a propagação de ondas acústicas em diferentes meios, como ar, água, sólidos e estruturas complexas (Virieux and Operto, 2009). A propagação de ondas acústicas em um meio de densidade constante pode ser descrita pela equação da propagação das ondas acústicas. Essa equação governa como as variações de pressão no meio evoluem ao longo do tempo e do espaço. Ela é dada por:

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

Onde p = p(x,y,z,t) é a pressão acústica como uma função das três coordenadas espaciais (x, y, z) e do tempo t.

$\frac{\partial^2 p}{\partial t^2}$ representa a segunda derivada de p em relação ao tempo t, descrevendo como a pressão varia ao longo do tempo, V é a velocidade do som no meio, que permanece constante nesse cenário e f é a fonte de perturbação (Igel, 2016).

A equação da onda acústica possui características fundamentais que influenciam a propagação das ondas sonoras. A velocidade de propagação V é uma das principais propriedades do meio e varia dependendo das características elásticas e compressíveis do material onde a onda se propaga. Além disso, a equação da onda acústica é linear, permitindo o uso de princípios de superposição, facilitando a análise de múltiplas fontes acústicas e suas interações (Igel, 2016).

2.1.2 Algoritmo de simulação

Para o estudo em questão será utilizado o Método das Diferenças Finitas e a equação da propagação das ondas acústicas será discretizada como abaixo:

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

Onde para cada $P_{i,j,k}^{(n+1)}$ na posição (i,j,k) do campo de ondas no intervalo de tempo n +1, representado em verde na figura 1, é calculado como uma função de seus vizinhos (em amarelo) e seu próprio valor no instante atual e no passo anterior do intervalo de tempo.

$$P_{1,2,3}^{(4)} = 2P_{1,2,3}^{(3)} - P_{1,2,3}^{(2)} + \nabla t^2 . v^2 \left(\frac{P_{2,2,3}^{(3)} - 2P_{1,2,3}^{(3)} + P_{0,2,3}^{(3)}}{\nabla x^2} + \frac{P_{1,3,3}^{(3)} - 2P_{1,2,3}^{(3)} + P_{1,1,3}^{(3)}}{\nabla y^2} + \frac{P_{1,2,4}^{(3)} - 2P_{1,2,3}^{(3)} + P_{1,2,2}^{(3)}}{\nabla z^2} \right) \quad (3)$$

Observe que a equação 3, forma discretizada da equação 1, acessa apenas 6 elementos vizinhos, Figura 1, e dois valores nos intervalos de tempo n e n -1 para obter o valor de $P_{i,j,k}^{(n+1)}$, isso porque a equação acústica é discretizada com segunda ordem no espaço e segunda ordem no tempo. No entanto, o número de operações de ponto flutuante, bem como os acessos à memória, dependem da física representada pela equação de onda e da ordem espacial com a qual foi discretizada, como demonstrado visualmente no estêncil de oitava ordem espacial na Figura 2.

O Algoritmo 1 generaliza a simulação para um domínio com qualquer número de dimensões.

2.2 Códigos estênceis

O termo estêncil se refere à configuração de pontos no *grid* acessados pelo algoritmo, que pode ser unidimensional ou multidimensional, onde cada ponto é calculado a partir de um subconjunto de seus vizinhos. Essas varreduras, que representam as atualizações de todos os pontos na *grid* conforme a regra computacional, são frequentemente empregadas na construção de solucionadores para equações diferenciais (Datta et al., 2009). Esses

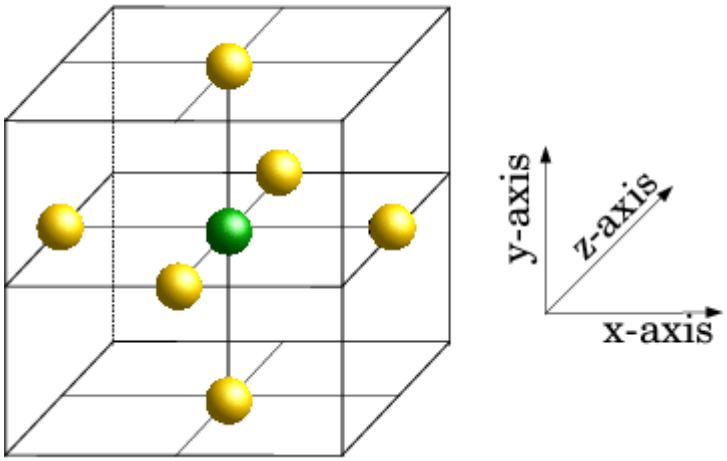


Figura 1 – Um estêncil de 7 pontos (2ª ordem espacial)

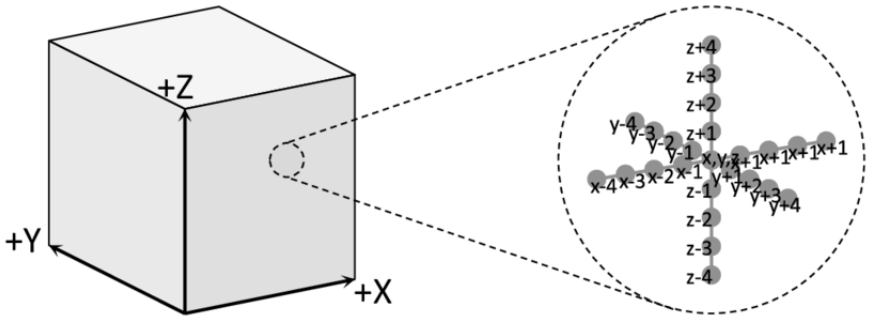


Figura 2 – Um estêncil de 25 pontos (8ª ordem espacial) (Krueger et al., 2012)

algoritmos são comuns em simulações físicas, análise de fluidos, processamento de imagens e outras aplicações que envolvem a discretização de problemas contínuos.

Um exemplo é a aplicação da equação da onda acústica. Na modelagem da propagação de ondas sonoras, cada ponto em um *grid* representa a pressão em um determinado local. A equação discretizada dessa propagação envolve a atualização da pressão em um ponto no tempo t com base na pressão nos pontos vizinhos em $t - \Delta t$, onde Δt é o intervalo de tempo. A fórmula para essa atualização é determinada pelo estêncil utilizado na discretização, refletindo a dependência local dos pontos adjacentes (Igel, 2016).

Estes algoritmos apresentam um alto grau de paralelismo, o que os torna adequados para implementações eficientes em sistemas paralelos e distribuídos. A simplicidade dos estênceis, juntamente com sua localidade espacial, facilita a otimização de código e o aproveitamento de estruturas de dados para melhorias de desempenho. (Datta et al., 2009)

Algoritmo 1 Simulação da propagação da onda

Require: f: origem da perturbação, dx, dt: discretização espacial e temporal
Ensure: u^n: campo de ondas para $n = 1, \ldots, T$

Inicialize $u^0 \leftarrow 0$ e $u^1 \leftarrow 0$
for $n \leftarrow 1$ até T **do**
  for cada ponto (i, j) no *grid* espacial **do**
    Atualize $u_{i,j}^{n+1}$ usando a equação da onda discretizada:
      $u_{i,j}^{n+1} \leftarrow 2u_{i,j}^{n} - u_{i,j}^{n-1} + \frac{dt^2}{dx^2}\nabla^2 u_{i,j}^{n} + dt^2 f_{i,j}$
  end for
end for
return u^T

2.3 Otimização de código por ladrilhamento (*t*iling)

A abordagem de ladrilhamento, do inglês *tiling* (Xue, 2000), é uma técnica fundamental para otimizar o desempenho de operações de computação em grande escala, ao melhorar o uso da memória e possibilitar a paralelização eficiente em arquiteturas de processamento. Ela se concentra na divisão do *grid* em blocos menores chamados *tiles*, os quais são processados de forma independente. Essa técnica é particularmente eficaz em situações em que ocorre acesso repetido aos mesmos dados, o que é comum em algoritmos que trabalham com matrizes multidimensionais, como processamento de imagens.

Existe também a abordagem de ladrilhamento temporal (*space-time loop tiling*) (Bielecki and Palkowski, 2021), que envolve a subdivisão do espaço e do tempo de computação em blocos menores. Porém, neste trabalho aplicaremos apenas a abordagem de ladrilhamento no espaço.

Os princípios subjacentes à abordagem de ladrilhamento incluem a divisão de tarefas em blocos menores, como representado na Figura 3, cada um atribuído a uma equipe de unidades de processamento. Cada ponto do *grid* é processado de forma independente, permitindo a exploração eficaz da localidade dos dados e a minimização da latência de acesso à memória. Isso é alcançado mantendo os dados necessários para um bloco no cache, reduzindo a necessidade de buscar informações da memória global repetidamente. Além disso, a abordagem de ladrilhamento também minimiza o tráfego de memória, uma vez que cada bloco acessa apenas uma parte dos dados (Xue, 2000).

2.3.1 Implementação da abordagem de ladrilhamento

A implementação da abordagem de ladrilhamento segue uma estrutura:

- ❑ Identificação do *loop*: identifique o *loop* que processa o *grid*.
- ❑ Definição do tamanho do ladrilho: para a definição são normalmente realizados testes com diversos tamanhos, como citado por Korch and Werner (2019) o tamanho

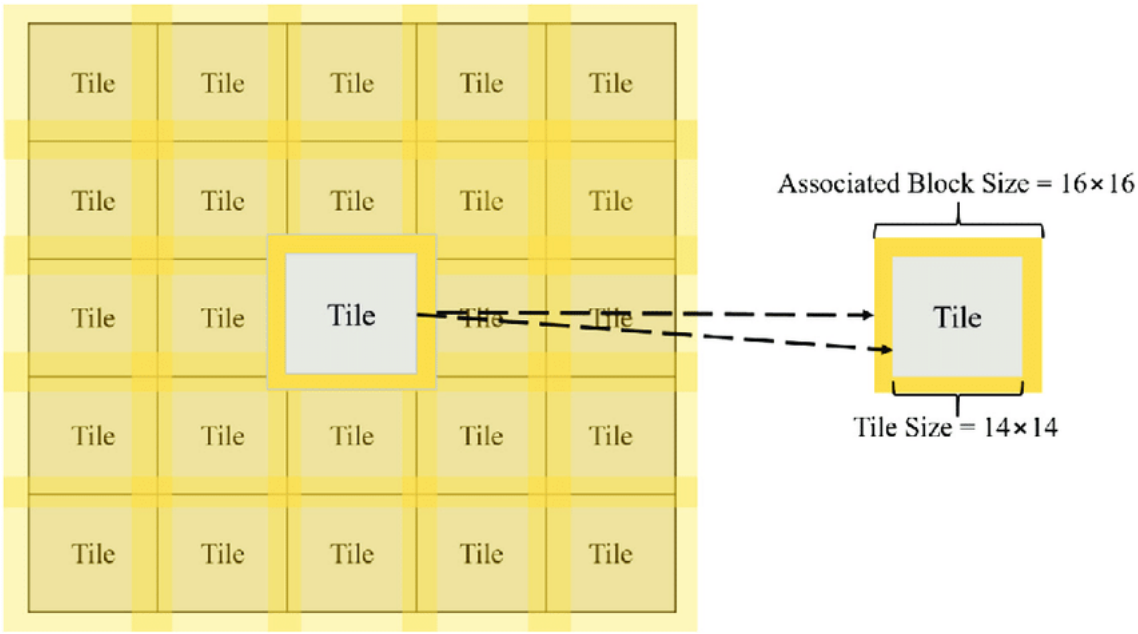


Figura 3 – Representação da abordagem de ladrilhamento (tiling)

Fonte: O Autor

do ladrilho pode ser encontrado por uma busca exaustiva com testes de diversos tamanhos.

- ❏ Implementação da abordagem de ladrilhamento: divida o *grid* em blocos menores utilizando a diretiva apropriada da linguagem de programação paralela que você está usando. Por exemplo, no OpenMP, você pode utilizar *#pragma omp tile sizes(X, Y, Z)*.
- ❏ Cálculos: certifique-se de que cada bloco calcula resultados independentes dos outros blocos.
- ❏ Coordenação e sincronização (se necessário): Em algoritmos paralelizados, coordene e sincronize os blocos, conforme necessário, para garantir a consistência dos resultados. Isso pode envolver o uso de diretivas de sincronização ou comunicação entre os blocos.

Exemplo de um loop sem ladrilhamento:

```
for (int i = 1; i <= 4; ++i)
    for (int j = 1; j <= 4; ++j)
        grid(i,j);
```

Exemplo de um loop com ladrilhamento, onde nos 2 primeiros loops temos a adição das variáveis i1 e j1 para receberem o tamanho do recorte de cada ladrilho, no eixo x e

y, e nos 2 loops subsequentes temos as iterações dentro de cada ladrilho, onde o primeiro loop representa o recorte de cada ladrilho e o segundo loop representa cada índice do eixo x e y do *grid*, representado na Figura 4:

```
for (int i1 = 0; i1 < 4; i1 += 2)
    for (int j1 = 0; j1 < 4; j1 += 2)
        # Loops para o processamento dentro dos ladrilhos
        for (int i2 = i1; i2 < i1 + 2; i2 += 1)
            for (int j2 = j1; j2 < j2 + 2; j2 += 1)
                grid(1+i2,1+j2);
```

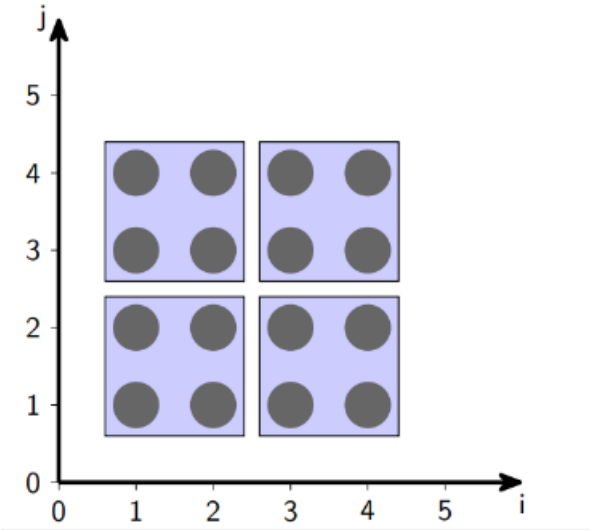


Figura 4 – Abordagem de ladrilhamento

Fonte: (Kruse, 2021)

Exemplo de um loop com ladrilhamento utilizando OpenMP:

```
#pragma omp tile sizes(2,2)
for (int i = 1; i <= 4; ++i)
    for (int j = 1; j <= 4; ++j)
        grid(i,j);
```

2.3.2 Desafios na aplicação em algoritmos estênceis

Algoritmos estênceis apresentam desafios únicos ao se aplicar a abordagem de ladrilhamento. A abordagem de ladrilhamento foi originalmente desenvolvida para otimizar algoritmos com acesso regular a dados, como *grids* multidimensionais, mas os algoritmos estênceis frequentemente envolvem acesso irregular a elementos do *grid*, como ilustrado

na Figura 5. O que significa que os elementos em uma vizinhança de um ponto podem ser acessados de maneira não linear. Isso torna a aplicação direta da abordagem de ladrilhamento mais complexa, uma vez que a divisão em blocos regulares pode não ser eficaz (Singh et al., 2020).

A sobrecarga computacional da divisão em blocos pode superar os benefícios obtidos pela redução do tráfego de memória. A aplicação de blocos menores pode levar a um aumento no número de cálculos de índices para acessar os vizinhos de cada ponto, o que pode resultar em um impacto negativo no desempenho geral do algoritmo. O software Devito (Luporini et al., 2020), por exemplo, propõe um ajuste empírico de tamanhos para encontrar tamanho de ladrilho que tenha o melhor desempenho.

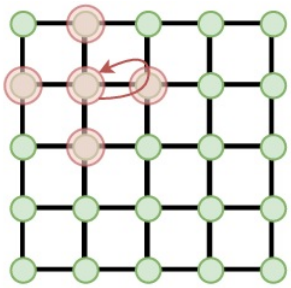


Figura 5 – Acesso de memória em algoritmo estêncil.

Fonte: (Denzler et al., 2021)

2.4 Técnicas de aprendizagem de máquina

Aprendizado de máquina é um campo da inteligência artificial que desenvolve algoritmos capazes de identificar padrões em dados e fazer previsões sem a necessidade de programação explícita para cada tarefa. Esses modelos ajustam seus parâmetros automaticamente com base nos exemplos apresentados, permitindo que melhorem seu desempenho ao longo do tempo. As principais abordagens incluem classificação, que atribui rótulos a novas amostras; regressão, que estima valores contínuos; e detecção de anomalias, que identifica padrões fora do comportamento esperado.

2.4.1 Classificação

O aprendizado por classificação é um tipo de aprendizado supervisionado que visa construir modelos preditivos capazes de categorizar instâncias em classes definidas, com base em exemplos previamente rotulados. Neste processo, o modelo recebe um conjunto de dados de treino, em que cada instância possui uma classe conhecida, e utiliza essas

informações para aprender a associar corretamente os atributos de entrada às classes de saída (Witten and Frank, 2005).

A supervisão ocorre porque o modelo recebe o resultado ou classe corretos para cada exemplo no conjunto de treino, orientando o processo de aprendizado e ajuda a ajustar o modelo para minimizar erros de classificação. Após o treinamento, o desempenho do modelo pode ser avaliado utilizando um conjunto de dados de teste independente, no qual as classes verdadeiras são conhecidas, mas não são fornecidas ao modelo durante o teste. A taxa de acerto ou precisão nos dados de teste fornece uma medida objetiva da qualidade do aprendizado, permitindo avaliar a generalização do modelo (Witten and Frank, 2005).

Além disso, em aplicações práticas, o sucesso do aprendizado por classificação pode ser medido subjetivamente, considerando a aceitabilidade ou interpretabilidade do modelo para os usuários humanos, por exemplo, a clareza das regras ou a simplicidade da árvore de decisão gerada. Isso é particularmente relevante quando a transparência no processo de classificação é desejada, como em domínios regulados ou em sistemas onde a confiança do usuário é fundamental (Witten and Frank, 2005).

2.4.2 Regressão

A previsão numérica, ou regressão, é uma forma de aprendizado supervisionado voltada para a estimativa de valores contínuos. Diferente do aprendizado de classificação, que associa exemplos a categorias discretas, a previsão numérica visa modelar a relação entre atributos e uma variável de resposta quantitativa. Um método comumente empregado para esse tipo de tarefa é a regressão linear, onde a variável alvo é expressa como uma combinação linear dos atributos do conjunto de dados, ponderada por coeficientes ajustados com base nos dados de treinamento (Witten and Frank, 2005).

Nesse contexto, o objetivo da previsão numérica é minimizar o erro entre os valores previstos e os valores reais da variável alvo, geralmente através da minimização da soma dos quadrados das diferenças entre essas quantidades para todas as instâncias de treinamento. O modelo resultante consiste em uma função linear cujos coeficientes refletem a influência de cada atributo na estimativa do valor contínuo. Esses coeficientes permitem que o modelo faça previsões para novas instâncias, com o ajuste baseado na estrutura linear subjacente dos dados fornecidos (Witten and Frank, 2005).

A eficácia da previsão numérica é medida pela precisão das previsões e pela adequação da estrutura linear à relação entre os atributos e a variável alvo (Witten and Frank, 2005).

2.4.3 Detecção de anomalias

A detecção de anomalias é a identificação de instâncias nos dados que se desviam de um padrão esperado ou não se ajustam ao modelo subjacente aplicado. Esse processo é utilizado para reconhecer possíveis erros ou dados atípicos que podem interferir na

análise, embora também envolva riscos, como a eliminação de dados válidos que apenas não seguem o modelo esperado. Na prática, é complexo discernir automaticamente entre um erro e uma instância que não se conforma ao modelo.

Em abordagens estatísticas como a regressão, visualizações podem auxiliar na detecção de anomalias, facilitando a identificação de dados que fogem da curva esperada. No entanto, para modelos mais complexos, onde o tipo de anomalia é sutil, estratégias de filtragem baseadas em múltiplos algoritmos de aprendizado, como árvores de decisão, vizinhos mais próximos e funções discriminantes lineares, são aplicadas. Esse processo busca consenso entre diferentes algoritmos para marcar uma instância como anômala apenas se todos falharem em classificá-la corretamente, minimizando o risco de descartar dados importantes (Witten and Frank, 2005).

Apesar de úteis, essas técnicas de filtragem podem, inadvertidamente, favorecer certas classes em detrimento de outras, dependendo das características dos algoritmos empregados. O ideal seria minimizar a necessidade de filtragem automática, garantindo a qualidade dos dados desde o início (Witten and Frank, 2005).

2.4.4 Formas de representação do conhecimento

A representação do conhecimento define a maneira como os padrões estruturais nos dados são descritos e compreendidos. Diferentes métodos de aprendizado produzem estruturas que capturam relações e características inerentes aos dados, possibilitando a análise e a inferência. Essas representações não apenas moldam como os algoritmos operam, mas também influenciam a compreensão humana dos resultados (Witten et al., 2016).

Modelos como árvores de decisão e regras de classificação exemplificam formas básicas de representação, amplamente utilizadas para expressar padrões discretos e categorizações. Estruturas mais avançadas podem incorporar exceções ou relações entre atributos, enquanto técnicas baseadas em instâncias priorizam os próprios dados ao invés de regras derivadas. Além disso, abordagens que geram agrupamentos de instâncias e árvores especializadas para previsão numérica demonstram a diversidade das metodologias empregadas (Witten et al., 2016).

2.4.4.1 Árvores de decisão

As árvores de decisão representam uma abordagem estruturada e eficiente para a classificação de instâncias em aprendizado de máquina, baseando-se no princípio de "dividir e conquistar". Cada nó da árvore realiza um teste em um atributo, como representado na Figura 6, que geralmente compara seu valor com uma constante ou, em alguns casos, entre atributos ou funções derivadas deles. As ramificações resultantes direcionam as instâncias para novos nós, até que atinjam uma folha, onde uma classificação ou uma distribuição de probabilidades é atribuída (Witten et al., 2016).

Como os testes são definidos varia conforme o tipo de atributo. Para atributos nominais, o número de ramificações reflete os valores possíveis do atributo, enquanto para atributos numéricos é comum utilizar divisões binárias (acima ou abaixo de uma constante) ou trinarias (exemplo: intervalos). Valores ausentes podem ser tratados como uma categoria específica ou por métodos mais sofisticados, como dividir a instância proporcionalmente entre os ramos, utilizando pesos baseados no conjunto de treinamento (Witten et al., 2016).

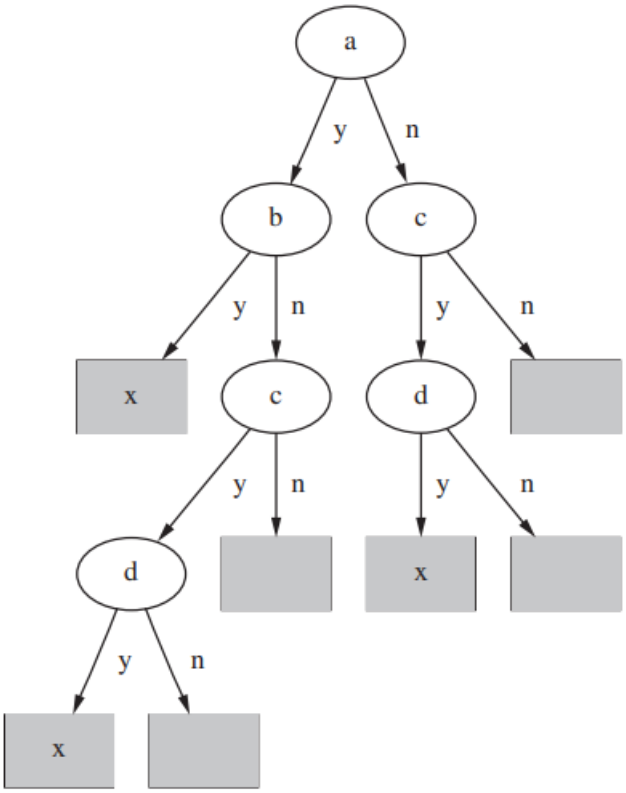


Figura 6 – Árvore de decisão para uma disjunção simples.

Fonte: (Witten et al., 2016)

Apesar da possibilidade de construir árvores manualmente, o processo é complexo e exige boa visualização dos dados e critérios bem definidos para escolher atributos relevantes. Ferramentas como o Weka oferecem suporte interativo para essa construção e permitem que métodos automatizados assumam o controle em etapas mais avançadas, otimizando a divisão dos atributos. Árvores de decisão exemplificam uma forma clara e interpretável de representação do conhecimento, sendo amplamente aplicadas em tarefas de classificação e predição (Witten et al., 2016).

2.4.4.2 Regras de Classificação

As regras de classificação constituem uma abordagem alternativa às árvores de decisão para representar o conhecimento, como demonstrado na Figura 7. Cada regra é composta por um antecedente—uma série de condições que devem ser atendidas—e um consequente, que define a classe associada ou a distribuição de probabilidade correspondente às instâncias que satisfazem as condições. Normalmente, as condições são combinadas usando a lógica E (AND), e as regras, como um conjunto, são tratadas como disjunções lógicas (OR) (Witten et al., 2016).

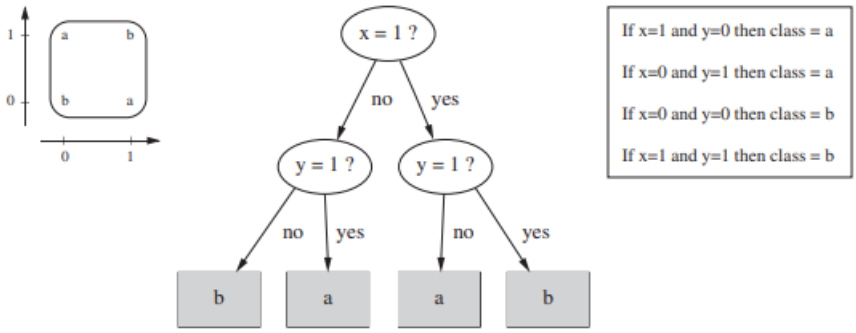


Figura 7 – O problema do ou-exclusivo (XOR).

Fonte: (Witten and Frank, 2005)

Embora regras possam ser derivadas diretamente de árvores de decisão, o processo frequentemente resulta em regras excessivamente complexas, devido à inclusão de condições redundantes, as quais podem ser removidas para simplificar o modelo. Por outro lado, a transformação de um conjunto de regras em uma árvore apresenta desafios, como o problema de subárvores replicadas, em que estruturas similares precisam ser duplicadas devido à falta de um único ponto de decisão (Witten et al., 2016).

As regras de classificação são atrativas por sua suposta modularidade, permitindo a adição de novas regras sem alterar as existentes. Contudo, conflitos podem surgir quando diferentes regras atribuem classificações distintas à mesma instância. Estratégias para lidar com esses conflitos incluem priorizar regras mais frequentemente ativadas ou adotar uma classe padrão para casos não cobertos (Witten et al., 2016).

Em cenários binários com a suposição de um "mundo fechado"—onde a ausência de uma classe implica automaticamente pertencimento à outra—, as regras são simples de interpretar e podem ser expressas em forma normal disjuntiva (OR de condições conjuntivas). No entanto, em contextos pluriclasses ou mais complexos, a ordem de execução das regras torna-se crucial, e a modularidade aparente pode ser comprometida (Witten et al., 2016).

Apesar de sua simplicidade inicial, conjuntos de regras podem se tornar sofisticados em aplicações práticas, sendo necessário abordar cuidadosamente questões de ambiguidade e eficiência para maximizar sua utilidade como forma de representação do conhecimento (Witten et al., 2016).

2.4.4.3 Regras de Associação

As regras de associação são uma extensão das regras de classificação, diferenciando-se por sua capacidade de prever qualquer atributo, incluindo combinações de atributos, e não apenas uma classe específica. Cada regra identifica uma regularidade distinta nos dados, sendo analisada isoladamente, sem a necessidade de compor um conjunto, como ocorre com as regras de classificação (Witten et al., 2016).

A avaliação das regras de associação baseia-se em dois critérios principais:

- **Cobertura**: refere-se ao número de instâncias corretamente previstas pela regra, também chamado de suporte.
- **Precisão**: indica a proporção de instâncias corretamente previstas em relação ao total de instâncias às quais a regra se aplica, frequentemente denominada confiança.

Exemplo: Considere um conjunto de dados sobre o desempenho de GPUs, com atributos como *largura de banda*, *cache L2*, e *arquitetura da GPU*, e um resultado *tamanho do ladrilho recomendado*. Uma regra de associação extraída pode ser:

Se largura de banda > 600GB/s, então tamanho do ladrilho = '4x4x2'

(Cobertura = 12, Precisão = 92%).

Essa regra indica que, em 12 instâncias no conjunto de dados, uma largura de banda superior a 600GB/s está associada à recomendação do ladrilho 4x4x2, e em 92% desses casos a regra foi válida (Witten et al., 2016).

Regras com múltiplas consequências requerem atenção especial ao implicarem relações complexas entre os atributos e as recomendações. Por exemplo, uma regra mais complexa pode ser:

Se cache L2 > 5MB e largura de banda > 500GB/s, então tamanho do ladrilho = '16x2x32'

e arquitetura da GPU = 'Ampere'.

Essa regra deve ser interpretada com cuidado, pois não apenas implica associações entre *cache L2* e *largura de banda*, as também entre *tamanho do ladrilho* e *arquitetura da GPU*.

Além disso, algumas regras podem implicar outras, sendo mais eficiente apresentar somente a regra mais forte para evitar redundâncias (Witten et al., 2016).

2.4.4.4 Regras com exceção

As regras de classificação podem ser estendidas para incluir exceções, permitindo ajustes incrementais em conjuntos de regras sem a necessidade de reformulação completa. Essa abordagem é útil, por exemplo, quando uma instância nova é mal classificada por uma regra existente. Em vez de modificar os critérios das regras originais, pode-se adicionar exceções específicas com base em outras características da instância (Witten et al., 2016).

Exemplo: Considere a escolha do tamanho do ladrilho com base em características de hardware. Suponha a seguinte regra:

Se largura de banda $\geq$ 600GB/s e cache L2 $\geq$ 6MB, então tamanho do ladrilho = '4x4x2'.

Agora, imagine um hardware com largura de banda de 650GB/s e cache L2 de 6.5MB, onde o tamanho ideal do ladrilho é '16x2x32'. A regra original recomendaria incorretamente o tamanho '4x4x2'. Em vez de modificar os limites da regra, uma exceção pode ser adicionada:

Se largura de banda $\geq$ 600GB/s e cache L2 $\geq$ 6MB, então ladrilho = '4x4x2',

exceto se cache L2 $>$ 6.4MB, então ladrilho = '16x2x32'.

Essa regra revisada acomoda o novo caso sem comprometer a precisão das demais recomendações.

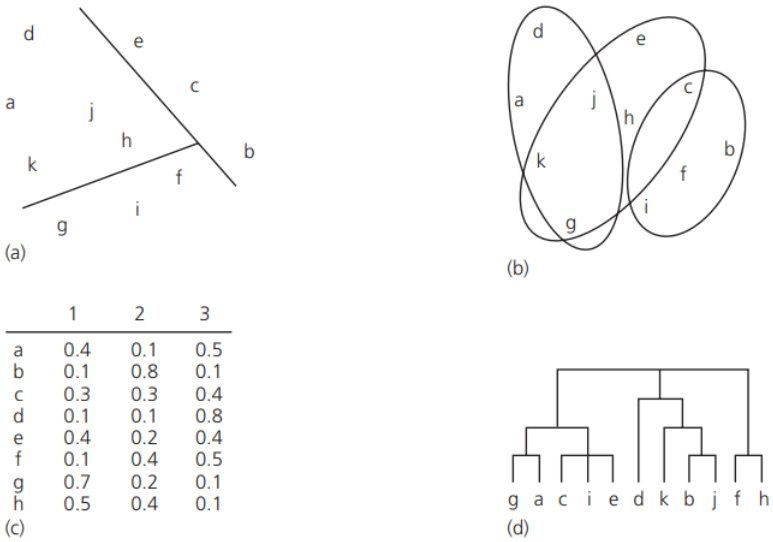
Esse tipo de estrutura permite a construção de conjuntos de regras mais flexíveis, que podem ser representados como árvores hierárquicas. Por exemplo, ao escolher tamanhos de ladrilho, uma regra pode ser modificada para incorporar uma exceção, especificando que certas características de hardware devem levar a uma recomendação diferente (Witten et al., 2016).

Embora a leitura dessas regras seja inicialmente complexa, elas se tornam mais compreensíveis com o tempo, principalmente pela sua capacidade de representar casos mais comuns e raros de maneira intuitiva. A principal vantagem desse formato reside na escalabilidade e na possibilidade de análise local das regras, tornando o entendimento e a modificação de grandes conjuntos de regras mais eficientes. Além disso, essa abordagem facilita a adaptação de regras em domínios dinâmicos, onde as exceções podem ser mais frequentes e específicas (Witten et al., 2016).

Embora a estrutura *se, então, senão* tradicional também possa ser aplicada, a formulação baseada em exceções apresenta vantagens ao facilitar a interpretação das regras gerais e suas exceções como um reflexo natural do pensamento humano (Witten et al., 2016).

2.4.4.5 Agrupamento

O agrupamento, do inglês *clustering*, é uma técnica de aprendizado que organiza instâncias em grupos chamados *clusters*, sendo representado graficamente por diagramas que demonstram como as instâncias estão distribuídas, como na Figura 8. No caso mais simples, cada instância é associada a um único cluster, com a visualização geralmente realizada em duas dimensões, particionando o espaço para indicar os grupos. Em métodos mais complexos, uma instância pode pertencer a múltiplos clusters, com representações como diagramas de Venn ou modelos probabilísticos que indicam graus de associação a cada cluster (Witten et al., 2016).



	1	2	3
a	0.4	0.1	0.5
b	0.1	0.8	0.1
c	0.3	0.3	0.4
d	0.1	0.1	0.8
e	0.4	0.2	0.4
f	0.1	0.4	0.5
g	0.7	0.2	0.1
h	0.5	0.4	0.1

Figura 8 – Diferentes representações de *clusters*

Fonte: (Witten et al., 2016)

Alguns algoritmos de agrupamento produzem estruturas hierárquicas, onde *clusters* maiores se subdividem em *subclusters* menores em níveis sucessivos. Essa organização é representada por dendrogramas, que mostram relações de proximidade entre as instâncias em diferentes níveis de granularidade. Essa abordagem é amplamente aplicada em áreas como biologia, devido à sua capacidade de representar relações hierárquicas naturalmente (Witten et al., 2016).

Além disso, o agrupamento pode ser utilizado como uma etapa intermediária para a inferência de descrições estruturais, como árvores de decisão ou conjuntos de regras. Nesse contexto, o agrupamento fornece uma base inicial para a alocação de instâncias a categorias específicas, contribuindo para a construção de modelos explicativos mais robustos e interpretáveis (Witten et al., 2016).

2.4.5 O classificador C4.5

O C4.5 é uma das ferramentas mais influentes no aprendizado de máquina para construção de árvores de decisão. Desenvolvido por J. Ross Quinlan, o C4.5 destaca-se por sua ampla aplicação em mineração de dados. Seu sucessor, o C5.0, embora comercial e mais rápido na geração de regras, mantém os fundamentos da indução de árvores do C4.5, com melhorias marginais em precisão (Witten and Frank, 2005).

O C4.5 utiliza parâmetros ajustáveis para aprimorar a poda e eliminar testes irrelevantes. Por padrão, a confiança é fixada em 25%, mas ajustes podem ser realizados para otimizar a poda e reduzir a complexidade das árvores, especialmente em cenários com ruídos nos dados. Além disso, o algoritmo evita testes em que quase todas as instâncias do conjunto de treinamento apresentam o mesmo resultado, contribuindo para a eficiência do modelo (Witten and Frank, 2005).

A abordagem *top-down* para indução de árvores de decisão, como a implementada no algoritmo C4.5, é amplamente estudada na literatura devido à sua combinação de simplicidade e robustez. Essa abordagem inicia a construção da árvore a partir da raiz, escolhendo recursivamente o melhor atributo para dividir os dados em cada nó, até que um critério de parada seja atendido, como um número mínimo de instâncias por folha ou a pureza dos subconjuntos resultantes. Embora variações no critério de divisão e poda reduzam o tamanho das árvores ou ajustem a precisão, o impacto dessas mudanças no desempenho global é geralmente limitado (Witten and Frank, 2005).

Adicionalmente, o C4.5 utiliza árvores univariadas, onde cada nó baseia-se em um único atributo. Contudo, versões multivariadas, como as introduzidas pelo sistema CART, podem ser mais compactas e precisas ao considerar combinações de atributos em cada divisão. Apesar disso, essas versões são mais complexas e exigem maior tempo de processamento, limitando sua aplicabilidade em cenários que demandam alta interpretabilidade (Witten and Frank, 2005).

2.4.6 O algoritmo *k*-vizinhos mais próximos

O algoritmo k-vizinhos mais próximos (k-NN) é um método de aprendizado supervisionado não paramétrico, desenvolvido inicialmente por Evelyn Fix e Joseph Hodges em 1951, e posteriormente aprimorado por Thomas Cover. Ele é amplamente utilizado para tarefas de classificação e regressão, sendo caracterizado por sua simplicidade e eficiência local (Cover and Hart, 1967).

2.4.6.1 Funcionamento Geral

O k-NN baseia-se em identificar os k exemplos mais próximos de um ponto de consulta em um espaço de atributos multidimensional:

- **Classificação**: A classe do ponto é determinada pela votação majoritária entre os k vizinhos mais próximos. No caso de $k = 1$, o rótulo do vizinho mais próximo é diretamente atribuído.

- **Regressão**: O valor predito é calculado como a média dos valores dos k vizinhos mais próximos, ou apenas o valor do vizinho mais próximo se $k = 1$.

O cálculo de proximidade geralmente utiliza métricas de distância, como a distância euclidiana para variáveis contínuas ou a distância de Hamming para variáveis discretas (Cover and Hart, 1967).

O algoritmo k-NN apresenta uma abordagem simples e intuitiva, caracterizada pela ausência de uma fase explícita de treinamento. Em vez disso, os vetores de características e seus rótulos são armazenados como base de consulta, e todo o processamento ocorre em tempo real durante a classificação ou regressão. Essa característica permite que o k-NN seja utilizado de forma flexível em uma ampla gama de aplicações, especialmente quando há necessidade de rapidez na implementação inicial (Cover and Hart, 1967).

Uma das vantagens do k-NN é sua capacidade de incorporar pesos baseados na distância entre os vizinhos, atribuindo maior relevância aos mais próximos. Métodos como o uso de pesos proporcionais ao inverso da distância podem aumentar significativamente a precisão do modelo, conferindo ao algoritmo uma adaptabilidade que o torna útil mesmo em cenários complexos (Cover and Hart, 1967).

No entanto, o k-NN também enfrenta desafios inerentes à sua simplicidade. O desequilíbrio de classes, por exemplo, pode levar a previsões enviesadas em favor das classes mais frequentes. Para contornar esse problema, estratégias como a ponderação por distância ou o uso de representações abstratas, como mapas auto-organizáveis (*Self-Organizing Maps*, SOM), ajudam a distribuir os dados de maneira mais uniforme e a reduzir o impacto do desequilíbrio. Além disso, o algoritmo é sensível à estrutura local dos dados, sendo influenciado pela escala dos atributos. Para melhorar o desempenho nesses casos, é recomendável aplicar normalizações adequadas ou adotar métricas aprendidas, como o *Large Margin Nearest Neighbor*, que aprimoram a análise da proximidade entre os dados (Cover and Hart, 1967).

2.4.7 O algoritmo Árvore de Regressão

A árvore de regressão é um modelo estatístico utilizado para prever valores contínuos de uma variável dependente Y, dividindo o espaço de dados em subgrupos por meio de divisões baseadas em variáveis independentes X. Esses modelos compartilham a estrutura hierárquica das árvores de classificação, mas diferem no objetivo: em vez de classificar observações em categorias, a árvore de regressão ajusta um modelo em cada nó terminal para gerar previsões numéricas (Breiman et al., 1986).

Historicamente, a árvore de regressão surgiu com o algoritmo AID, que, junto ao método CART, utiliza a soma dos desvios quadráticos em relação à média como métrica de impureza e a média amostral de Y como predição. Esse processo resulta em modelos constantes por partes, que são intuitivos, mas podem apresentar baixa precisão preditiva quando comparados a métodos mais complexos (Breiman et al., 1986).

Modelos modernos, como o M5', introduzem maior sofisticação ao combinar árvores constantes com ajustes de modelos lineares nas folhas, permitindo previsões lineares por partes eficientemente. Por outro lado, o GUIDE combina características de árvores de classificação, ajustando modelos residuais em cada nó para gerar divisões mais flexíveis e imparciais (Breiman et al., 1986).

A árvore de regressão é amplamente reconhecida por sua capacidade de capturar interações entre variáveis e de identificar relações não lineares nos dados, características que as tornam ferramentas valiosas em análises preditivas. Modelos baseados em divisões constantes por partes são especialmente úteis para interpretações simples e estruturais, enquanto modelos lineares ou polinomiais aumentam a precisão preditiva ao reduzir a complexidade estrutural. Essa versatilidade permite que a árvore de regressão seja ajustada para atender às necessidades específicas de cada análise (Breiman et al., 1986).

A eficácia da árvore de regressão é evidente em aplicações práticas, como estudos de função pulmonar, onde variáveis como idade, altura e sexo interagem de maneira complexa. Além disso, sua flexibilidade se estende a adaptações para diferentes critérios de perda, incluindo mínimos quadrados, medianas e modelos voltados para dados censurados. Essa adaptabilidade amplia ainda mais seu alcance e relevância em diferentes contextos analíticos (Breiman et al., 1986).

Com sua abordagem interpretável e ajustável, a árvore de regressão equilibra simplicidade estrutural e precisão preditiva, destacando-se como uma escolha robusta para uma ampla gama de problemas. Modelos como GUIDE, CART e M5' ilustram bem essa versatilidade, sendo amplamente utilizados em diferentes áreas. Além disso, a disponibilidade de ferramentas de software, como RPART, WEKA e pacotes específicos para CRUISE, GUIDE e QUEST, facilitam sua implementação (Breiman et al., 1986).

2.4.8 O algoritmo *Random Forest*

Random Forest é um método de aprendizado de máquina baseado em conjuntos que utiliza múltiplas árvores de decisão para realizar tarefas de classificação e regressão. Fundamentada na introdução de aleatoriedade controlada durante a construção das árvores, a técnica busca reduzir sobreajustes, explorando a Lei dos Grandes Números para melhorar a robustez e precisão preditiva (Breiman, 2001).

O modelo é caracterizado pela construção de diversas árvores independentes, onde cada uma é treinada com amostras geradas por *bagging*, e uma seleção aleatória de características. Essa abordagem reduz a correlação entre as árvores, promovendo um ganho

de acurácia. Durante a inferência, as previsões das árvores são combinadas por métodos de agregação, como a média para regressão ou votação majoritária para classificação (Breiman, 2001).

O *Random Forest* é eficaz na modelagem de interações entre variáveis e fronteiras de decisão não lineares devido à sua construção baseada em múltiplas árvores de decisão. Essas árvores podem capturar relações complexas e interações entre variáveis sem a necessidade de explicitamente especificar essas interações.

O método *out-of-bag* (OOB) é uma técnica embutida no *Random Forest* para avaliar o desempenho do modelo sem a necessidade de um conjunto de validação separado. Durante o treinamento, cada árvore é construída com um subconjunto aleatório dos dados (amostragem com reposição), e os dados que não foram usados para treinar a árvore são utilizados para calcular o erro OOB. Isso fornece uma estimativa confiável do erro de generalização do modelo.

Sua competitividade em relação a técnicas como *boosting* e *adaptive bagging* reside na capacidade de reduzir viés sem necessidade de modificar iterativamente os dados de treinamento (Breiman, 2001).

Embora o método seja mais eficiente em tarefas de classificação, apresenta menor desempenho em algumas situações de regressão. Estudos sugerem que a introdução de formas alternativas de aleatoriedade, como combinações booleanas de características, pode aprimorar seus resultados. Adicionalmente, interpretações teóricas indicam que o *Random Forest* atua como um *kernel* dinâmico no espaço de margem, equilibrando correlação e força preditiva para otimizar decisões em problemas de classificação binária (Breiman, 2001).

2.4.9 O algoritmo XGBoost

O XGBoost, abreviação de *Extreme Gradient Boosting*, é um sistema de aprendizado de máquina escalável projetado para aprimorar o desempenho do método de *gradient tree boosting*. Este algoritmo é amplamente reconhecido por sua eficácia em capturar dependências complexas nos dados e por sua capacidade de lidar com grandes volumes de informação, tornando-se uma escolha dominante em diversas aplicações práticas e competições de ciência de dados (Chen and Guestrin, 2016).

Diferentemente de outros métodos de aprendizado, o XGBoost combina inovações algorítmicas e de sistemas para alcançar alta escalabilidade e eficiência. Suas contribuições incluem um algoritmo otimizado para aprendizado de árvores com dados esparsos, um procedimento de esboço de quantis ponderados para cálculo eficiente de propostas de divisão, e técnicas avançadas de computação paralela e distribuída. Além disso, ele suporta cálculos fora do núcleo, permitindo que cientistas de dados processem centenas de milhões de exemplos em máquinas convencionais, sem necessidade de grandes recursos computacionais (Chen and Guestrin, 2016).

O XGBoost também utiliza uma função de objetivo regularizada, que inclui termos para penalizar a complexidade do modelo e, assim, ajuda a controlar o sobreajuste, além de estratégias específicas para tratar dados esparsos e valores ausentes e aumentar a eficiência computacional. Essas inovações tornaram o XGBoost um sistema de aprendizado altamente escalável e adaptável, capaz de lidar com problemas complexos como classificação de textos, previsão de comportamento de clientes, detecção de anomalias e categorização de produtos (Chen and Guestrin, 2016).

O impacto do XGBoost é amplamente evidenciado por sua adoção em competições e aplicações reais. Por exemplo, o XGBoost foi utilizado em 17 das 29 soluções vencedoras de desafios publicados pelo Kaggle em 2015, muitas vezes como componente principal dos modelos de predição (Chen and Guestrin, 2016).

2.4.10 O algoritmo LightGBM

O LightGBM é uma implementação otimizada do algoritmo de Árvores de Decisão com *Gradient Boosting Decision Tree* (GBDT), projetado para abordar os desafios associados ao aprendizado de máquina em cenários de grandes volumes de dados. O GBDT é amplamente reconhecido por sua precisão, eficiência e interpretabilidade, sendo aplicado em tarefas como classificação multiclasse, predição de cliques e aprendizado de ranqueamento. No entanto, as implementações convencionais de GBDT apresentam alta complexidade computacional, devido à necessidade de analisar todas as instâncias de dados para cada característica durante o cálculo do ganho de informação (Meng et al., 2017).

Para superar essas limitações, o LightGBM adota duas técnicas fundamentais que melhoram significativamente a eficiência computacional sem comprometer a acurácia do modelo. A primeira técnica, denominada *Gradient-based One-Side Sampling* (GOSS), realiza a seleção das amostras com base nos gradientes, priorizando aquelas que apresentam gradientes mais elevados. Essas amostras são consideradas mais relevantes, por possuírem maior impacto no cálculo do ganho de informação. Com isso, a técnica preserva instâncias cruciais para o modelo, enquanto descarta aquelas com gradientes menos significativos, garantindo uma estimativa mais precisa do ganho de informação e otimizando o tempo de processamento (Meng et al., 2017).

A segunda técnica, chamada *Exclusive Feature Bundling* (EFB), aborda a alta dimensionalidade em espaços de características esparsos. Em cenários práticos, é comum que muitas características sejam quase mutuamente exclusivas, como ocorre em representações *one-hot*. O EFB explora essa propriedade agrupando características exclusivas em uma única dimensão, reduzindo efetivamente o número de características a serem processadas. Esse agrupamento é realizado por meio de algoritmos de coloração de grafos, garantindo uma redução quase sem perdas na representatividade das características e otimizando o uso de recursos computacionais (Meng et al., 2017).

Essas inovações tornam o LightGBM altamente eficiente e escalável. Ao combinar o GOSS e o EFB, o algoritmo consegue acelerar o treinamento em até 20 vezes, mantendo níveis de precisão competitivos em comparação com as implementações tradicionais. Além disso, o LightGBM utiliza algoritmos baseados em histogramas para a seleção dos pontos de divisão, o que permite organizar os valores das características em intervalos discretos. Isso não apenas melhora o uso de memória, mas também acelera o treinamento (Meng et al., 2017).

2.5 Utilização de GPUs para alto desempenho

Durante décadas, microprocessadores baseados em uma única unidade central de processamento (CPU), como os das famílias Intel Pentium e AMD Opteron, impulsionaram avanços significativos no desempenho computacional, principalmente por meio do aumento da frequência de *clock* e melhorias nas arquiteturas internas. Contudo, por volta de 2003, questões relacionadas ao consumo de energia e à dissipação de calor começaram a limitar o crescimento da frequência de *clock*, forçando os fabricantes a adotar novos paradigmas de *design*, como a inclusão de múltiplos núcleos de processamento. Essa transição marcou o início da era dos processadores *multicore*, alterando significativamente o panorama do desenvolvimento de software. Para explorar o potencial desses novos microprocessadores, tornou-se necessário adotar técnicas de programação paralela. Essa mudança de paradigma exigiu que desenvolvedores e engenheiros de software se adaptassem às complexidades da programação paralela para acompanhar os avanços tecnológicos (Kirk et al., 2023).

Desde então, a indústria de semicondutores tem seguido duas trajetórias principais para projetar microprocessadores, demonstradas visualmente na Figura 9. A trajetória de *multicore* visa manter a velocidade de execução de programas sequenciais, enquanto avança para múltiplos núcleos. Os multicores começaram com processadores de dois núcleos e o número de núcleos dobrou a cada geração de processadores. Em contraste, a trajetória de *many-core* foca mais no *throughput* de execução de aplicações paralelas. Os muitos núcleos começaram com inúmeros núcleos menores, e o número de núcleos dobrou a cada geração. Enquanto as CPUs têm foco em desempenho sequencial, as GPUs são projetadas como motores de cálculo numérico, otimizadas para o *throughput* de execução em massa de *threads* (Kirk et al., 2023).

Além do desempenho, outros fatores desempenham um papel crucial na escolha de processadores pelos desenvolvedores. A ampla adoção de uma arquitetura no mercado influencia diretamente o custo de desenvolvimento de software e o tamanho da base de clientes potenciais. Nesse contexto, a popularidade das GPUs, amplamente presentes em computadores pessoais e servidores, torna-as uma escolha economicamente atrativa. Características como custo acessível, disponibilidade e suporte a padrões amplamente

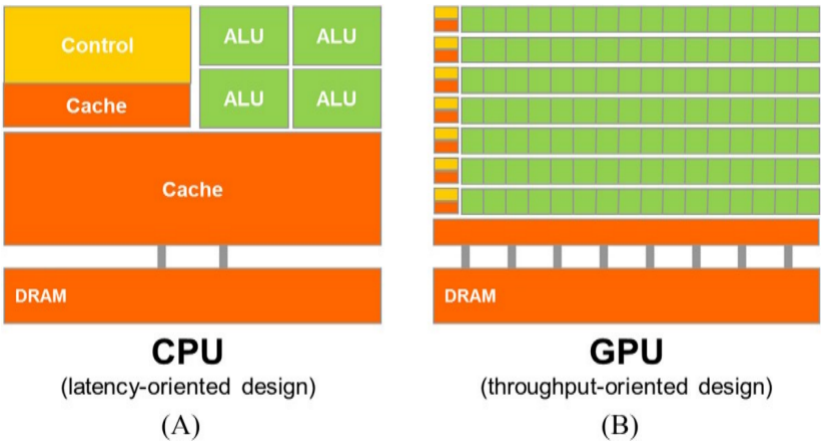


Figura 9 – Diferenças de design entre CPU e GPU

Fonte: (Kirk et al., 2023)

adotados, como o IEEE para aritmética de ponto flutuante, são essenciais para garantir resultados previsíveis em cálculos numéricos. GPUs modernas oferecem suporte a esses padrões de forma comparável às CPUs, ampliando sua viabilidade em aplicações que exigem precisão e desempenho (Kirk et al., 2023).

O principal objetivo da programação massivamente paralela é permitir que as aplicações desfrutem de aumentos contínuos de velocidade nas futuras gerações de hardware. Aplicações bem implementadas em uma GPU podem alcançar um aumento de velocidade de mais de 100 vezes em relação à execução sequencial em um único núcleo de CPU. Muitas das aplicações do futuro, como simulações biológicas ao nível molecular e processamento de vídeo e áudio em alta definição, demandarão cada vez mais poder de processamento. O aumento da velocidade computacional também possibilitará interfaces de usuário mais avançadas, como telas sensíveis ao toque de alta resolução e interfaces baseadas em voz e visão computacional. Além disso, a simulação e modelagem precisas de fenômenos físicos estão impulsionando novas aplicações, como a utilização de técnicas de aprendizado de máquina para melhorar o desempenho dessas simulações (Kirk et al., 2023).

2.5.1 Programação paralela heterogênea

Até 2006, o uso de GPUs para computação geral era limitado, pois os programadores precisavam recorrer a APIs gráficas, como OpenGL e Direct3D, o que impunha desafios significativos ao desenvolvimento de aplicativos fora do domínio gráfico. Em 2007, o lançamento da *Compute Unified Device Architecture* (CUDA) trouxe uma mudança paradigmática. O CUDA não apenas introduziu um novo modelo de programação de software, mas também incorporou modificações de hardware diretamente no chip, como memórias

compartilhadas e unidades de controle otimizadas para computação geral. Essa abordagem eliminou a necessidade de utilizar interfaces gráficas, permitindo aos desenvolvedores criar uma ampla gama de aplicações diretamente para GPUs. Além disso, a compatibilidade do CUDA com ferramentas de programação familiares, como C e C++, facilitou sua adoção e contribuiu para expandir significativamente o uso de GPUs na computação de alto desempenho (Kirk et al., 2023).

A estrutura de um programa CUDA reflete a coexistência de um hospedeiro (CPU) e um ou mais dispositivos (GPUs) no computador, com código de hospedeiro e código de dispositivo misturados em um mesmo arquivo-fonte. A execução de um programa CUDA começa com o código do hospedeiro (CPU) e, quando uma função de *kernel* é chamada, Figura 10, inúmeras *threads* são lançadas em um dispositivo para executar o *kernel*. Todas as *threads* lançadas por uma chamada de *kernel* são coletivamente chamadas de *grid*. A execução do *grid* termina quando todas as *threads* terminam suas execuções, e a execução continua no hospedeiro até que outro *grid* seja lançado. É importante notar que muitas aplicações de computação heterogênea gerenciam a sobreposição da execução da CPU e GPU para aproveitar ambos os recursos. Ao lançar um *grid*, muitas *threads* são geradas para explorar o paralelismo de dados. Na conversão de cor para escala de cinza, por exemplo, cada *thread* pode ser usado para calcular um píxel do *array* de saída (Kirk et al., 2023).

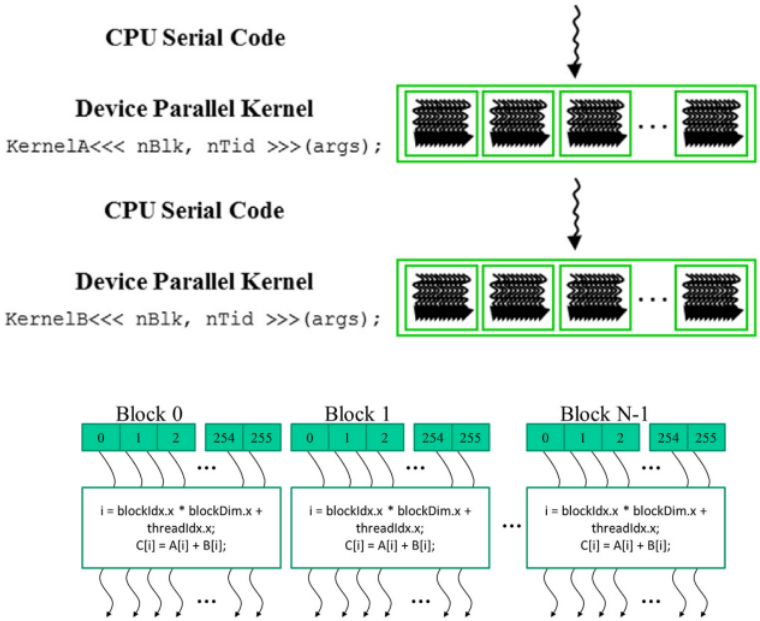


Figura 10 – Execução de um programa em CUDA.

Fonte: (Kirk et al., 2023)

Todas as *threads* em um *grid* executam a mesma função de *kernel* e usam índices de coordenadas para se distinguir e identificar a parte apropriada dos dados para processar.

As *threads* são organizados em uma hierarquia de dois níveis: um *grid* contém um ou mais blocos, e cada bloco contém uma ou mais *threads*, como mostra a Figura 11. Cada bloco possui um índice de bloco, acessível pela variável *blockIdx*, e cada *thread* possui um índice de *thread*, acessível pela variável *threadIdx*. A configuração de execução em uma chamada de *kernel* especifica as dimensões do *grid* e dos blocos, acessíveis pelas variáveis *gridDim* e *blockDim*. Geralmente, um *grid* é uma matriz tridimensional de blocos, e cada bloco é uma matriz tridimensional de *threads*. Os parâmetros de configuração de execução especificam as dimensões do *grid* e dos blocos em cada dimensão, utilizando o tipo *dim3*, por exemplo, cada elemento do vetor representa as dimensões x, y e z (Kirk et al., 2023).

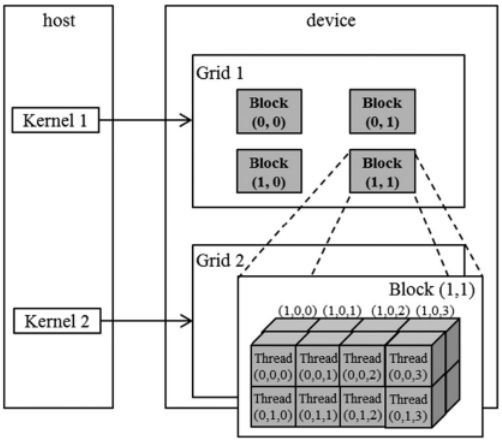


Figura 11 – Organização do *grid* em CUDA.

Fonte: (Kirk et al., 2023)

Existem quatro tipos principais de memória: memória global, memória constante, memória local e memória compartilhada. A memória global e constante podem ser escritas e lidas pelo hospedeiro. A memória constante oferece acesso de leitura de baixa latência. A memória local é usada por cada *thread* para armazenar dados privados, enquanto a memória compartilhada é compartilhada entre *threads* de um bloco. Registros e memória compartilhada são memórias *on-chip*, acessíveis com alta velocidade e eficiência. Na Figura 12 podemos observar como é feita a movimentação entre essas memórias.

A Figura 13 oferece uma visão geral da arquitetura de uma GPU compatível com CUDA. Essa GPU é organizada em uma matriz de multiprocessadores com vários *Streaming Multiprocessor* (SM). Cada SM contém vários núcleos de processamento chamados de processadores de *streaming* ou núcleos CUDA, que compartilham lógica de controle e recursos de memória. Por exemplo, a GPU Ampere A100 possui 108 SMs com 64 núcleos cada, totalizando 6912 núcleos na GPU inteira (Kirk et al., 2023).

Uma importante funcionalidade do CUDA são as barreiras. O seu uso permite a sincronização de *threads* em um mesmo bloco. Quando uma *thread* chama a função *syncthreads*,

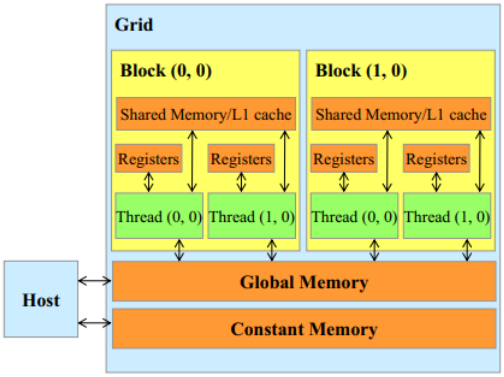


Figura 12 – Hierarquia da memória em CUDA.

Fonte: (Kirk et al., 2023)

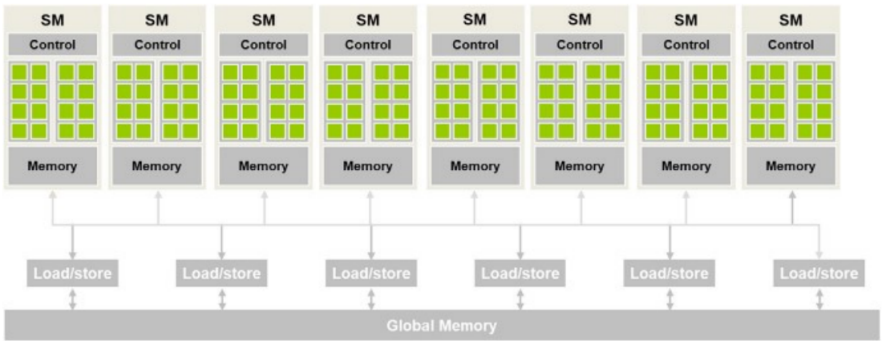


Figura 13 – Arquitetura GPU compatível com CUDA

Fonte: (Kirk et al., 2023)

ela é mantida na localização do programa da chamada até que todas as *threads* no mesmo bloco alcancem essa localização, garantindo que todas tenham concluído uma fase de sua execução antes de passar para a próxima, como vemos na Figura 14. Essa sincronização é essencial para evitar resultados inconsistentes e *deadlocks*. Por exemplo, se quatro amigos forem juntos ao shopping em um carro e cada um for às lojas separadamente, eles precisam se encontrar de volta ao carro antes de saírem, caso contrário, alguém pode ficar para trás. O uso inadequado de barreiras de sincronização pode levar a resultados incorretos ou à *threads* esperando indefinidamente umas pelas outras.

As barreiras de sincronização podem ser classificadas como inerentes ou explícitas. As barreiras inerentes ocorrem naturalmente devido à estrutura do algoritmo, sem necessidade de comandos explícitos, como em dependências de dados e sincronizações implícitas de laços paralelos. Já as barreiras explícitas, como a função *___syncthreads()* no CUDA, são inseridas deliberadamente no código para coordenar a execução das *threads*, garan-

tindo que todas alcancem um determinado ponto antes de prosseguir. O CUDA assegura que todas as *threads* de um bloco tenham acesso aos recursos necessários para atingir a barreira de sincronização, reduzindo a possibilidade de *deadlocks* e garantindo a consistência dos dados compartilhados entre as *threads*.

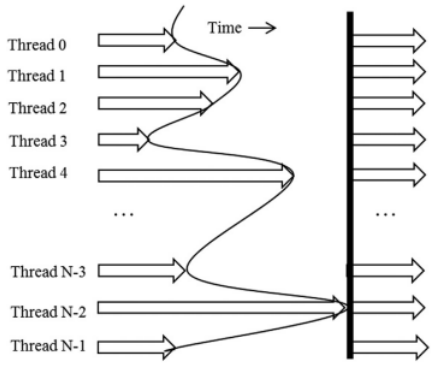


Figura 14 – Barreira de sincronização em CUDA

Fonte: (Kirk et al., 2023)

Os blocos podem executar em qualquer ordem relativa entre si, permitindo escalabilidade transparente em diferentes dispositivos. Conceitualmente, as *threads* em um bloco também podem executar em qualquer ordem entre si. O agendamento de *threads* em GPUs CUDA é um conceito de implementação de hardware e deve ser discutido no contexto de implementações específicas. Em muitas implementações, os blocos são divididos em unidades de 32 *threads* chamadas *warps*, representados na Figura 15, os quais são a unidade de escalonamento de *threads* em SMs. Os *warps* são particionados com base nos índices de *threads*, sendo utilizados para agendar instruções em todas as *threads* de um bloco. A vantagem do modelo SIMD, que significa *Single Instruction, Multiple Data*, é compartilhar o custo do hardware de controle, permitindo um maior *throughput* aritmético. Espera-se que a partição de *warps* continue sendo uma técnica popular de implementação, com cada *warp* consistindo em 32 *threads*, como todas as GPUs CUDA até o momento.

2.5.2 OpenMP

Sistemas heterogêneos geralmente consistem em CPUs, unidades vetoriais SIMD e GPUs. A ideia de *offloading* é transferir computação da CPU para a GPU, mas deixar a CPU ociosa durante esse processo é ineficiente. O objetivo ideal é a programação heterogênea, onde partes do problema são executadas em diferentes componentes de hardware, como CPU, GPU e unidades vetoriais SIMD. Isso maximiza o desempenho, pois cada parte do problema é executada na plataforma mais adequada. OpenMP é uma

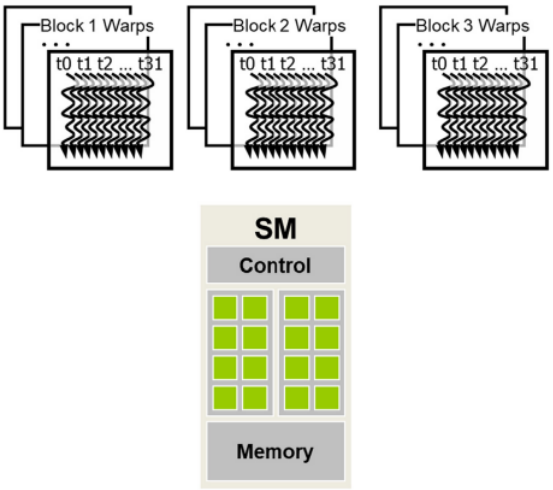


Figura 15 – Blocos particionados em *warps*

Fonte: (Kirk et al., 2023)

interface de programação que suporta esses tipos de processadores heterogêneos em um único programa, incluindo *multithreading* para CPUs, programação direta de unidades vetoriais SIMD e programação de GPUs. Com suporte amplo entre compiladores de alto desempenho, o OpenMP facilita a portabilidade do código entre plataformas e gerações de hardware (Deakin and Mattson, 2023).

O OpenMP permite que um programa sequencial seja convertido incrementalmente em um programa paralelo, como vemos na Figura 16. A paralelização no OpenMP é explícita, onde o programador especifica o que deve ser feito para definir uma execução paralela, enquanto o compilador é responsável por gerar o código paralelo com base nas diretivas inseridas pelo programador. As diretivas do compilador são expressas como `#pragma` em C ou C++. Geralmente, uma diretiva está associada a um bloco de código, onde esse bloco é convertido em uma função pelo compilador. Esse bloco deve ser estruturado, ou seja, ter um único ponto de entrada e um único ponto de saída. A combinação de uma diretiva e seu bloco estruturado é chamada de construção, e todo o código executado dentro dela, incluindo funções chamadas, é chamado de região (Deakin and Mattson, 2023).

Em 2013, o OpenMP adicionou suporte para sistemas heterogêneos em resposta à necessidade de lidar com sistemas baseados em GPUs. Isso foi motivado pela ascensão dos sistemas baseados em GPUs, especialmente após o anúncio de um sistema baseado em GPU no topo da lista TOP500 em 2012. Esse evento chamou a atenção da comunidade de HPC, que percebeu a inevitabilidade dos sistemas baseados em GPU e a necessidade de adaptação do software para ambientes heterogêneos. Essa tendência de incorporação de GPUs permanece forte nos anos subsequentes (Deakin and Mattson, 2023).

Para transferir a execução de um programa para um dispositivo, como uma GPU,

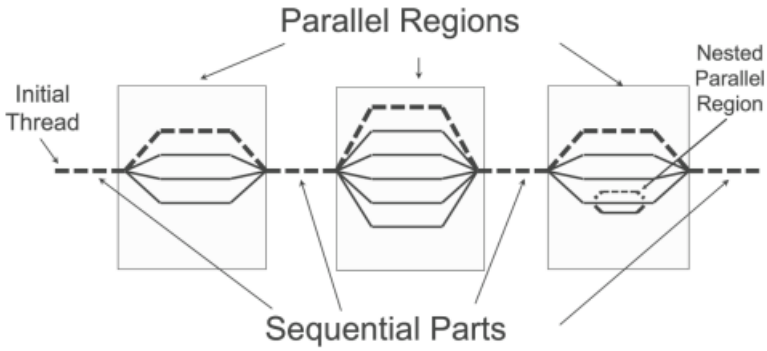


Figura 16 – Blocos particionados em *warps*

Fonte: (Kirk et al., 2023)

utiliza-se a diretiva *#pragma omp target* do OpenMP. Ao encontrar essa diretiva, os dados necessários para a execução da região alvo são transferidos para o dispositivo, e o código especificado é executado no dispositivo. A região alvo inclui o código na diretiva e quaisquer funções chamadas por ele. Por padrão, a *thread* que encontra a diretiva *target* aguarda a conclusão da região alvo antes de continuar a execução, a menos que seja especificado o modificador *nowait*. O movimento de dados entre o host e o dispositivo é gerenciado automaticamente, conforme regras explícitas ou implícitas de mapeamento de dados definidas pelo OpenMP, permitindo que desenvolvedores explorem a paralelização em GPUs de maneira mais eficiente. A Figura 17 mostra as duas ações principais que a diretiva *target* realiza, sendo: transferência da execução e transferência dados no início e no final da região alvo (Deakin and Mattson, 2023).

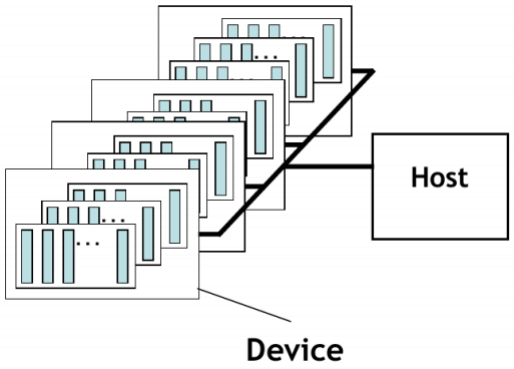


Figura 17 – Modelo Hospedeiro/Dispositivo

Deakin et al. (2022)

Embora o OpenMP simplifique o processo de programação para GPUs, a otimização de desempenho ainda é uma consideração importante. A escolha das diretivas corretas, o

mapeamento adequado de dados e a distribuição eficiente de trabalho entre os processadores são pontos importantes para garantir que o código aproveite todo o potencial das GPUs.

Para nosso estudo, utilizaremos a combinação das diretivas abaixo, que serão detalhadas em seguida:

#pragma omp target teams distribute parallel for collapse(3)

#pragma omp tile sizes(X, Y, Z)

A diretiva *target* é utilizada para especificar regiões do código que devem ser executadas na GPU. O código dentro desse bloco é convertido em um *kernel* e enviado para execução no dispositivo, conforme a especificação do OpenMP (Deakin and Mattson, 2023). A diretiva *teams* permite a criação de múltiplas equipes de *threads* na GPU (Deakin and Mattson, 2023). Isso é essencial para arquiteturas baseadas em SPMD (*Single Program Multiple Data*), onde diversas threads executam o mesmo programa, mas operam sobre dados diferentes. Dessa forma, cada equipe pode conter várias *threads* que processam os dados de maneira independente.

A diretiva *distribute* compartilha as iterações do *loop* entre as equipes de forma equitativa, seguindo um agendamento estático. Isso possibilita distribuir a carga de trabalho pelos recursos paralelos da GPU, garantindo um balanceamento adequado entre as equipes (Deakin and Mattson, 2023).

A diretiva *parallel* cria um conjunto de *threads* dentro de cada equipe, permitindo a execução paralela das tarefas. No contexto de GPUs, essa diretiva segue o modelo SPMD, mas internamente pode ser implementada como SIMD (*Single Instruction Multiple Data*) dentro dos grupos de execução chamados de *warps* (na arquitetura NVIDIA) (Deakin and Mattson, 2023). Isso significa que, embora diferentes *threads* possam executar fluxos de controle distintos, aquelas no mesmo warp seguem a mesma instrução simultaneamente.

A diretiva *for* é utilizada para distribuir as iterações do *loop* entre as *threads* de uma equipe (Deakin and Mattson, 2023). Assim, combinando as diretivas *teams*, *distribute*, *parallel* e *for*, é possível explorar ao máximo o paralelismo da GPU:

- *teams* cria equipes de execução;
- *distribute* compartilha iterações entre equipes;
- *parallel* cria *threads* dentro de cada equipe;
- *for* distribui as iterações entre *threads* dentro da equipe.

Dessa forma, essa combinação permite que milhares de *threads* processem grandes volumes de dados, aproveitando o paralelismo massivo característico das GPUs (Deakin and Mattson, 2023).

A diretiva *collapse* é usada para combinar múltiplos *loops* aninhados, permitindo uma exploração mais eficiente do paralelismo. A quantidade de *loops* aninhados a ser combinada é especificada ao passar um número para a diretiva, como em *collapse(3)*, que indica que três *loops* aninhados serão colapsados em um único *loop* para otimizar a execução paralela.

A diretiva *tile sizes* é usada para dividir o *grid* em ladrilhos menores, que podem ser processados de forma mais eficiente em GPUs, explorando a localidade dos dados. Em nosso caso, teremos 3 eixos da matriz, X, Y e Z (Kruse, 2021).

Em resumo, a primeira combinação de diretivas explora a otimização do código para aproveitar o paralelismo disponível na GPU, já a segunda combinação explora os benefícios da abordagem de ladrilhamento, recortando o *grid* em ladrilhos menores para reaproveitar os pontos vizinhos já existentes na memória.

Este capítulo apresentou os fundamentos teóricos essenciais para o desenvolvimento deste trabalho, abordando desde a modelagem da propagação da onda acústica até técnicas de otimização computacional e aprendizagem de máquina. A revisão das metodologias empregadas na simulação numérica, bem como das abordagens para aprimoramento de desempenho por meio de GPUs, estabelece a base para as estratégias exploradas nos capítulos seguintes. Esses conceitos são fundamentais para a implementação e análise dos algoritmos discutidos posteriormente, garantindo uma compreensão sólida dos desafios e soluções investigadas.

Capítulo 3

Trabalhos relacionados

Neste capítulo, apresentaremos os principais trabalhos relacionados a esta pesquisa. Nosso objetivo é situar o presente estudo dentro do contexto da pesquisa existente e para destacar lacunas e oportunidades na literatura. A seleção dos trabalhos relacionados foi realizada por meio de uma pesquisa bibliográfica utilizando bases de dados acadêmicas, como IEEE Xplore, ACM Digital Library e Google Scholar. Foram empregadas palavras-chave como *stencil*, *GPU* e *tiling*, visando identificar estudos relevantes sobre otimização de códigos estênceis em arquiteturas paralelas utilizando *tiling*.

A pesquisa conduzida por Souza et al. (2022a) explorou o impacto das técnicas de ladrilhamento e desenrolamento de laços (*loop unrolling*) no desempenho de GPUs, utilizando o *offloading* do OpenMP. Os experimentos foram realizados com a versão simplificada do software Simwave (Souza et al., 2022b), analisando três arquiteturas distintas: RTX 2080 Super, V100 e A100. A implementação foi desenvolvida em C, utilizando o compilador Clang 13 para aplicar as otimizações de *unroll* e ladrilhamento (Kruse, 2021). Os resultados indicaram que a combinação dessas técnicas foi particularmente eficaz, alcançando até 2,93 vezes o desempenho do código original. No entanto, observou-se uma alta sensibilidade do desempenho ao tamanho do ladrilho escolhido, evidenciando a necessidade de estratégias mais robustas para determinar tamanhos otimizados.

No contexto de GPUs, a investigação sobre técnicas de ladrilhamento começou a ganhar atenção nos anos 2000, impulsionada pelo advento do CUDA (Kirk et al., 2023). Desde então, várias abordagens têm sido exploradas para determinar o tamanho do ladrilho, dada sua influência direta no desempenho.

O presente trabalho propõe uma abordagem baseada em aprendizado de máquina para a seleção automática do tamanho do ladrilho em GPUs. Diferentemente da busca exaustiva, que pode ser computacionalmente custosa, o uso de aprendizado de máquina

permite modelar as relações entre os parâmetros do sistema e o desempenho esperado. Ao fundamentar-se nos dados obtidos por Souza et al. (2022a), este estudo avança na otimização de execução em GPUs, explorando o potencial do aprendizado de máquina para lidar com a complexidade dessa tarefa.

Essa seção foi dividida em trabalhos que utilizaram uma busca exaustiva e trabalhos que utilizaram técnicas de aprendizado de máquina.

3.1 Trabalhos que utilizaram busca exaustiva

Xu et al. (2009) examina o uso de GPUs para ajuste de desempenho e otimização, com foco na técnica de ladrilhamento. Os autores destacam que diferentes modelos de GPUs possuem características distintas que impactam a eficácia do ladrilhamento. Eles implementam a interpolação bilinear como um caso de teste e comparam o desempenho do ladrilhamento em diferentes modelos de GPU, como a GTX 260 e a GeForce 8800. Observam que os tamanhos otimizados do ladrilho variam entre os modelos de GPU. Recomendam, portanto, que os programadores ajustem cuidadosamente o ladrilhamento para cada modelo de GPU visando alcançar o melhor desempenho. Em seus experimentos, constataram que dimensões de ladrilhamento de 32x4 proporcionaram melhor desempenho para imagens maiores em ambos os modelos de GPU.

Grosser et al. (2014) apresenta uma abordagem de ladrilhamento híbrido hexagonal/-clássico para paralelizar eficientemente cálculos iterativos de estêncil em GPUs. Essa abordagem combina ladrilhamento hexagonal ao longo da dimensão do tempo e da primeira dimensão espacial com ladrilhamento clássico nas dimensões restantes. O ladrilhamento hexagonal promove a reutilização ao longo do tempo, evitando cálculos redundantes, favorecendo acessos coalescidos à memória e evitando divergência de *threads*. Os autores integram sua proposta de ladrilhamento no compilador poliédrico PPCG, com otimizações adaptadas para o ladrilhamento híbrido. Resultados experimentais demonstram melhorias significativas de desempenho em relação a compiladores estênceis de última geração, alcançando um aumento de velocidade de até 920% para alguns kernels 2D. Além disso, os autores descrevem um método para encontrar valores adequados para os parâmetros de tamanho de ladrilho, utilizando um modelo baseado na relação carga-processamento. No entanto, a complexidade da modelagem e a dependência de funções derivadas manualmente podem representar desafios na implementação dessa abordagem, cuja eficácia depende da aplicação específica e estêncil utilizado.

Korch and Werner (2019) otimizaram métodos explícitos de *one-step* para a solução de equações diferenciais ordinárias (ODEs) em GPUs NVIDIA, utilizando a GeForce GTX Titan Black com microarquitetura Kepler, com o objetivo de aumentar a eficiência computacional por meio de estratégias avançadas de *tiling*, como o ladrilhamento trapezoidal e hexagonal. Para determinar os tamanhos ótimos dos ladrilhos, empregaram técnicas

eficientes de busca, como busca por gradiente e *simulated annealing*, assim, alcançando melhorias significativas no desempenho. Seus experimentos demonstraram acelerações de até 3,0 vezes para pequenas distâncias de acesso, destacando a eficácia da abordagem na otimização do uso da arquitetura da GPU para melhorar o desempenho computacional.

3.2 Trabalhos que utilizaram técnicas de aprendizado de máquina

Li and Song (2004) propõe técnicas de otimização de cache por meio do ladrilhamento automático de estênceis iterativos, abordando tanto o espaço quanto o tempo. O autor explora desafios como a natureza não linear dos aninhamentos de *loops* e a otimização do fator de ajuste para reduzir os custos de acesso à memória e minimizar as perdas de cache. Para determinar os fatores ótimos de ajuste, respeitando as dependências de dados, são sugeridos algoritmos baseados em programação linear inteira e teoria dos grafos. Uma análise de custo de memória é apresentada visando selecionar os tamanhos ideais de bloco, mitigando problemas de capacidade e conflitos no cache. Os experimentos realizados com 16 algoritmos demonstraram melhorias de até cinco vezes no desempenho, quando comparado as implementações originais. Em suma, o artigo oferece uma abordagem sistemática para otimizar algoritmos estênceis iterativos por meio do ladrilhamento automático, priorizando a minimização dos fatores de ajuste e a eliminação de conflitos de cache. Os experimentos foram conduzidos em uma estação de trabalho uniprocessada SUN Ultra I e em um único processador MIPS R10K de uma máquina multiprocessadora SGI Origin 2000, com os resultados comparados as implementações originais otimizadas pelos compiladores nativos e a outra técnica chamada *shift-and-peel*.

Rahman et al. (2010) propõe o uso de uma abordagem de aprendizado de máquina para construir um preditor de desempenho com base em um pequeno número de execuções empíricas. Eles treinam uma rede neural artificial (ANN) nos tempos de execução para diferentes configurações de tamanho do ladrilho, permitindo à ANN prever a distribuição de desempenho e identificar tamanhos de ladrilho com bom desempenho. Os autores desenvolveram uma técnica de busca em duas etapas, onde na primeira etapa amostram aleatoriamente uma fração do espaço de busca e treinam a ANN nos tempos de execução medidos. Na segunda etapa, utilizam a ANN para prever tamanhos de ladrilho com bom desempenho próximos de mínimos locais na distribuição de desempenho, avaliando empiricamente apenas esses pontos previstos. Seus experimentos, realizadas em uma arquitetura Nehalem com um processador Intel Core i7 860 de 2.8 GHz, mostram que essa técnica assistida por ANN supera a busca aleatória, melhorando significativamente o tempo de execução, encontrando tamanhos de ladrilho dentro de 10% do tempo de execução globalmente ideal usando um tempo de busca reduzido.

Malik (2012) aborda a dificuldade de desenvolver um modelo preciso para a seleção do tamanho ideal do ladrilho e propõe o uso de redes neurais artificiais para aprender automaticamente um modelo eficaz. O autor destaca que a coleta de dados pode ser demorada, mas pode ser facilitada com o uso de ferramentas como o gerador de código parametrizado com blocos, TLoG. A coleta de dados envolveu uma busca exaustiva pelos tamanhos ideais de ladrilho, levando de 50 a 60 horas para cada combinação de arquitetura de compilador. O autor utilizou os dados coletados juntamente com características dinâmicas do programa, obtidas por meio de contadores de desempenho de hardware, como entrada para treinar redes neurais artificiais. O modelo desenvolvido foi validado em diversas arquiteturas e compiladores, demonstrando desempenho próximo ao ótimo, com diferenças de até 4% em relação aos tamanhos ideais reais. Em conclusão, o autor destaca que um modelo de seleção de tamanho do ladrilho razoavelmente preciso pode ser aprendido automaticamente por máquina, prevendo consistentemente tamanhos de ladrilho com desempenho próximo ao ótimo em diversas configurações.

Liu et al. (2018) propõem uma nova abordagem para prever tamanhos otimizados de ladrilho na técnica de ladrilhamento, utilizando redes neurais artificiais. Os autores extraem 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. Essas características são utilizadas como entradas em uma rede neural de regressão generalizada para aprender o modelo de previsão de tamanho do ladrilho. Programas sintéticos são usados para gerar dados de treinamento com características e tamanhos de ladrilhos otimizados correspondentes. Embora considerem apenas 4 *threads* durante o treinamento do modelo, eles podem adaptar os tamanhos dos ladrilhos previstos para diferentes números de *threads* por meio de um ajuste simples baseado no balanceamento de carga paralela. Em experimentos realizados em dois servidores, um Intel Xeon com 4 processadores octa-core e um IBM Power6 com 1 quad-core, o modelo de previsão de tamanho do ladrilho atinge desempenho próximo ao ideal. Em suma, os pesquisadores propõem uma abordagem eficaz de aprendizado de máquina para prever tamanhos ideais de ladrilho para o ladrilhamento, baseada em características do *loop*.

A Tabela 1 sintetiza os principais estudos que abordaram a técnica de ladrilhamento para algoritmos estênceis. Observa-se que, embora existam trabalhos que empregam aprendizagem de máquina para prever o tamanho otimizado do ladrilho, tais abordagens têm sido aplicadas exclusivamente a arquiteturas baseadas em CPUs. Por outro lado, os estudos que focaram em arquiteturas de GPU utilizaram predominantemente métodos de busca exaustiva. Essa lacuna evidencia uma oportunidade para investigar o potencial de aplicação de algoritmos de aprendizado de máquina na determinação do tamanho otimizado de ladrilho especificamente para arquiteturas de GPU, ampliando o alcance e a eficiência das técnicas de otimização no contexto dessas plataformas.

Este capítulo apresentou uma revisão abrangente dos trabalhos relacionados à otimi-

Autores	**Aprendizagem de Máquina**	**Testes Exaustivos**	**CPU**	**GPU**
Li and Song (2004)	x		x	
Xu et al. (2009)		x	x	
Rahman et al. (2010)	x		x	
Malik (2012)	x		x	
Grosser et al. (2014)		x		x
Liu et al. (2018)	x		x	
Korch & Werner (2019)		x		x
Sousa et al. (2022)		x		x
Nossa proposta	x			x

Tabela 1 – Resumo dos trabalhos relacionados.

zação de estênceis em arquiteturas paralelas, com foco no uso de técnicas de ladrilhamento em GPUs e a aplicação de aprendizado de máquina para determinar tamanhos de ladrilhos otimizados. Observamos que, enquanto muitas pesquisas anteriores se concentraram na busca exaustiva para a seleção do tamanho ideal do ladrilho, a utilização de aprendizado de máquina tem sido mais prevalente em contextos de CPUs. Em particular, a lacuna identificada nas aplicações em arquiteturas de GPU aponta para uma oportunidade significativa de pesquisa, que visa explorar o potencial do aprendizado de máquina para melhorar a eficiência e a precisão na seleção de tamanhos de ladrilhos para essas plataformas. O trabalho apresentado contribui para essa área, propondo uma abordagem que utiliza aprendizado de máquina para otimizar o desempenho em GPUs, representando um avanço sobre as técnicas convencionais de busca exaustiva. Dessa forma, espera-se que esta pesquisa abra novos caminhos para otimizações mais eficientes e adaptáveis.

Capítulo 4

Escolha do tamanho do ladrilho por aprendizado de máquina

4.1 Experimentos

O ladrilhamento é amplamente reconhecido como uma técnica fundamental para a otimização de desempenho em programas que realizam operações iterativas sobre estruturas de dados multidimensionais, como os usados em cálculos de estêncil. Essa técnica tem sido objeto de estudo em diversos trabalhos clássicos e contemporâneos, como demonstrado em Xu et al. (2009), Grosser et al. (2014), Korch and Werner (2019).

Trabalhos recentes, como o de Luporini et al. (2020), realizaram testes exaustivos em subespaços de busca, avaliando diversas combinações de tamanhos de ladrilhos. A estratégia utilizada envolveu a seleção iterativa do melhor tamanho de ladrilho com base em métricas empíricas de tempo de execução, permitindo identificar configurações que maximizassem o desempenho computacional.

O software Devito, como descrito em Luporini et al. (2020), realiza otimizações utilizando *tiling* para resolver equações diferenciais parciais (PDEs) de forma eficiente em arquiteturas modernas. Ele emprega uma abordagem iterativa para otimizar o tamanho dos ladrilhos, baseada em métricas empíricas de tempo de execução, permitindo selecionar configurações que maximizem o desempenho computacional. O Devito realiza testes exaustivos em subespaços de busca, avaliando diferentes combinações de tamanhos de ladrilhos.

A escolha do tamanho do ladrilho é um fator crítico, ao influenciar diretamente o desempenho de aplicações em arquiteturas heterogêneas. Como demonstrado por Souza et al. (2022a), o desempenho de *kernels* baseados em estêncil é altamente sensível a

essa escolha, dado que ela afeta a eficiência da utilização de recursos computacionais, como unidades de processamento, caches e barramentos de memória. No referido estudo, os autores investigaram o desempenho do Simwave, uma aplicação representativa, em um extenso conjunto de configurações experimentais. Foram analisadas 54 combinações, variando dimensões de *grids* tridimensionais (256^3, 512^3 e 1024^3), precisões numéricas (*float32* e *float64*), ordens espaciais (2^a, 8^a e 16^a) e diferentes arquiteturas de GPUs: RTX 2080 Super (Turing), V100 (Volta) e A100 (Ampere). Ressalte-se que a RTX 2080 Super não suportou *grids* de tamanho 1024^3, levando à exclusão de 6 cenários, o que resultou em 48 configurações viáveis. Em cada configuração, foram realizados experimentos com múltiplos tamanhos de ladrilhos tridimensionais, totalizando 13.816 execuções. Cada experimento foi repetido três vezes para reduzir a variabilidade e garantir a robustez das análises. A Tabela 2 ilustra alguns dos resultados reportados em Souza et al. (2022a), que foram utilizados como base para derivar recomendações sobre tamanhos ideais de ladrilhos.

GPU	Grid	dtype	ordem espacial	tile 1	tile 2	tile 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 2 – Desempenho em segundos para diferentes GPUs e configurações Souza et al. (2022a)

O conjunto de dados gerado por Souza et al. (2022a) constitui um recurso valioso para a compreensão do impacto do ladrilhamento no desempenho computacional. A partir dessa base de dados, nosso estudo desenvolveu métodos de extração de conhecimento capazes de fornecer recomendações preditivas para novos cenários. Inicialmente, foi realizada uma etapa de pré-processamento para a remoção de *outliers*, a fim de garantir maior confiabilidade e representatividade dos resultados. Para tanto, aplicou-se o método de Tukey (Tukey et al., 1977), que identifica valores discrepantes como aqueles fora de 1,5 vezes o intervalo interquartil. Esse processo resultou na exclusão de 1.701 experimentos, correspondendo a 12% do total original, restando 12.117 experimentos. A distribuição dos dados é apresentada na Figura 18.

O problema de otimização do ladrilhamento, portanto, envolve uma interação com-

Figura 18 – Dispersão dos tempos de execução do conjunto de experimentos de Souza et al. (2022a).

plexa entre as características do hardware, a geometria do *grid* e a ordem espacial do estêncil. Estudos como o de Korch and Werner (2019) e Souza et al. (2022a) evidenciaram que a exploração empírica do espaço de busca, embora computacionalmente intensiva, revela padrões fundamentais e oferece subsídios críticos para o aprimoramento e desenvolvimento de heurísticas otimizadas. Nosso trabalho propõe uma abordagem inovadora para análise desses dados, utilizando técnicas de aprendizado de máquina para modelar o desempenho em função das variáveis experimentais. Essa metodologia possibilita a construção de modelos preditivos capazes de recomendar tamanhos de ladrilhos otimizados para novos problemas, reduzindo a necessidade de experimentação exaustiva.

4.2 Abordagem por classificação

Predizer o tamanho otimizado de ladrilho para execução eficiente em GPUs consiste em determinar uma tripla que especifica as dimensões do ladrilho nos eixos tridimensionais. Essa representação é fundamental, pois a compilação gera blocos tridimensionais de *threads* que serão processados pela GPU, e o alinhamento dessas dimensões com a arquitetura do hardware e a natureza do *kernel* é determinante para o desempenho. Dessa forma, a tarefa de classificação envolve associar cada instância do conjunto de dados a uma classe, onde a classe é representada por uma tripla que corresponde à recomendação otimizada para as dimensões do ladrilho.

Para esse objetivo, utilizamos o algoritmo J48, uma implementação do classificador C4.5, disponível na ferramenta Weka (Witten and Frank, 2005). O J48 foi escolhido

devido à sua capacidade de construir árvores de decisão como modelo de aprendizado, proporcionando uma representação interpretável do conhecimento extraído.

A avaliação do modelo foi conduzida empregando a técnica de validação cruzada com 10 *folds*, um método estatisticamente robusto que garante uma estimativa confiável do desempenho do modelo. Nessa abordagem, o conjunto de dados é particionado em 10 subconjuntos, e em cada iteração, 9 subconjuntos são utilizados para treinar o modelo, enquanto o subconjunto restante é reservado para teste. Este processo é repetido 10 vezes, alternando os subconjuntos de teste em cada rodada, permitindo uma avaliação completa e balanceada do desempenho em todo o conjunto de dados.

Após a etapa de remoção de outliers, o conjunto de dados final, composto por 12.117 instâncias, foi submetido ao algoritmo de classificação. O classificador gerou uma árvore de decisão contendo 44 folhas, como demonstrada na Figura 19. Cada folha representa uma regra de decisão específica extraída do conjunto de dados, descrevendo as condições necessárias para determinar a classe de saída, ou seja, a tripla do tamanho do ladrilho ideal. Essas regras formam um modelo compacto e altamente interpretável, que permite identificar os fatores determinantes no desempenho e os padrões subjacentes na escolha do tamanho do ladrilho.

4.3 Abordagem por regressão

Uma segunda abordagem explorada neste estudo foi baseada em modelos de regressão, visando recomendar o tamanho ideal de ladrilho para uma configuração específica do problema. Essa abordagem consiste em prever os tempos de execução associados a diferentes tamanhos de ladrilhos e, em seguida, identificar a configuração que apresenta o menor tempo de execução entre as opções avaliadas. Tal metodologia proporciona uma análise mais granular e quantitativa, permitindo explorar a relação entre as características do problema e o desempenho observado.

Foram avaliados cinco algoritmos de regressão, cada um com características distintas para modelagem e previsão:

- **K-Nearest Neighbors (KNN):** Este modelo baseia-se na similaridade entre instâncias, utilizando os k-vizinhos mais próximos para realizar as previsões (Cover and Hart, 1967).
- **Árvore de Regressão:** Modelo hierárquico que utiliza divisões sucessivas nos dados para prever valores contínuos, permitindo uma interpretação direta das condições associadas a cada predição (Breiman et al., 1986).
- **Random Forest:** Um conjunto de árvores de regressão que combina previsões individuais por meio de agregação, aumentando a robustez e a precisão do modelo (Breiman, 2001).

```
J48 pruned tree
------------------

space_order <= 2
|   grid_size <= 256
|   |   dtype = float32
|   |   |   qtt_sm <= 48: 2x16x4 (343.0)
|   |   |   qtt_sm > 48
|   |   |   |   qtt_sm <= 80: 1x32x2 (343.0)
|   |   |   |   qtt_sm > 80: 16x1x4 (196.0)
|   |   dtype = float64
|   |   |   qtt_sm <= 48: 32x4x2 (343.0)
|   |   |   qtt_sm > 48
|   |   |   |   qtt_sm <= 80: 2x4x1 (343.0)
|   |   |   |   qtt_sm > 80: 8x1x4 (196.0)
|   grid_size > 256
|   |   dtype = float32
|   |   |   grid_size <= 512
|   |   |   |   qtt_sm <= 48: 16x2x4 (343.0)
|   |   |   |   qtt_sm > 48
|   |   |   |   |   qtt_sm <= 80: 64x4x1 (343.0)
|   |   |   |   |   qtt_sm > 80: 32x1x4 (196.0)
|   |   |   grid_size > 512
|   |   |   |   qtt_sm <= 80: 4x2x1 (343.0)
|   |   |   |   qtt_sm > 80: 2x4x4 (196.0)
|   |   dtype = float64
|   |   |   grid_size <= 512
|   |   |   |   qtt_sm <= 48: 2x4x4 (343.0)
|   |   |   |   qtt_sm > 48
|   |   |   |   |   qtt_sm <= 80: 2x16x1 (343.0)
|   |   |   |   |   qtt_sm > 80: 1x4x4 (196.0)
|   |   |   grid_size > 512
|   |   |   |   qtt_sm <= 80: 2x8x4 (343.0)
|   |   |   |   qtt_sm > 80: 16x1x4 (196.0)
space_order > 2
|   grid_size <= 256
|   |   dtype = float32
|   |   |   space_order <= 8
|   |   |   |   qtt_sm <= 48: 8x4x1 (343.0)
|   |   |   |   qtt_sm > 48
|   |   |   |   |   qtt_sm <= 80: 1x1x2 (343.0)
|   |   |   |   |   qtt_sm > 80: 4x1x2 (196.0)
|   |   |   space_order > 8
|   |   |   |   qtt_sm <= 48: 64x2x1 (343.0)
|   |   |   |   qtt_sm > 48
|   |   |   |   |   qtt_sm <= 80: 2x8x1 (343.0)
|   |   |   |   |   qtt_sm > 80: 1x2x1 (196.0)
|   |   dtype = float64
|   |   |   space_order <= 8
|   |   |   |   qtt_sm <= 48: 16x4x1 (343.0)
|   |   |   |   qtt_sm > 48
|   |   |   |   |   qtt_sm <= 80: 4x1x1 (343.0)
|   |   |   |   |   qtt_sm > 80: 1x8x2 (196.0)
|   |   |   space_order > 8
|   |   |   |   qtt_sm <= 48: 2x4x1 (343.0)
|   |   |   |   qtt_sm > 48
|   |   |   |   |   qtt_sm <= 80: 32x1x1 (343.0)
|   |   |   |   |   qtt_sm > 80: 1x4x2 (196.0)
|   grid_size > 256
|   |   space_order <= 8
|   |   |   dtype = float32
|   |   |   |   qtt_sm <= 80
|   |   |   |   |   grid_size <= 512
|   |   |   |   |   |   qtt_sm <= 48: 64x1x1 (343.0)
|   |   |   |   |   |   qtt_sm > 48: 32x2x1 (343.0)
|   |   |   |   |   grid_size > 512: 2x2x2 (343.0)
|   |   |   |   qtt_sm > 80: 64x2x1 (392.0)
|   |   |   dtype = float64
|   |   |   |   qtt_sm <= 48: 16x2x1 (343.0)
|   |   |   |   qtt_sm > 48
|   |   |   |   |   grid_size <= 512: 64x1x1 (539.0)
|   |   |   |   |   grid_size > 512
|   |   |   |   |   |   qtt_sm <= 80: 4x2x1 (343.0)
|   |   |   |   |   |   qtt_sm > 80: 64x1x1 (196.0)
|   |   space_order > 8
|   |   |   grid_size <= 512
|   |   |   |   qtt_sm <= 48
|   |   |   |   |   dtype = float32: 8x1x1 (343.0)
|   |   |   |   |   dtype = float64: 64x1x1 (343.0)
|   |   |   |   qtt_sm > 48
|   |   |   |   |   qtt_sm <= 80: 16x1x1 (686.0)
|   |   |   |   |   qtt_sm > 80
|   |   |   |   |   |   dtype = float32: 8x4x1 (196.0)
|   |   |   |   |   |   dtype = float64: 32x1x1 (196.0)
|   |   |   grid_size > 512
|   |   |   |   qtt_sm <= 80
|   |   |   |   |   dtype = float32: 2x2x1 (343.0)
|   |   |   |   |   dtype = float64: 4x4x1 (343.0)
|   |   |   |   qtt_sm > 80: 64x1x1 (392.0)

Number of Leaves  :     44

Size of the tree :      87
```

Figura 19 – Árvore de decisão gerada pelo algoritmo J48.

- **XGBoost:** Um modelo baseado em *boosting*, que constrói iterativamente árvores de regressão otimizadas para corrigir erros das anteriores, maximizando a precisão (Chen and Guestrin, 2016).

- **LightGBM:** Algoritmo que adota estratégias como histogramas de gradiente, para melhorar a eficiência do treinamento e o desempenho preditivo (Meng et al., 2017).

Para implementar os modelos, utilizamos a biblioteca *Scikit-learn* (versão 1.3.0) para KNN, Árvore de Regressão e *Random Forest*, enquanto os modelos XGBoost e LightGBM foram configurados por meio de suas bibliotecas dedicadas, nas versões 2.0.3 e 4.0.3, respectivamente.

A otimização dos hiperparâmetros foi realizada utilizando a biblioteca **Optuna** (Akiba et al., 2019), que oferece um *framework* eficiente para exploração automatizada de espaços de busca. Foram realizados 100 ensaios para cada modelo, ajustando parâmetros críticos com base em métricas de desempenho predefinidas, visando identificar as configurações mais adequadas para cada algoritmo.

4.3.1 Hiperparâmetros Otimizados

Os hiperparâmetros finais ajustados para cada modelo são detalhados abaixo:

- **KNN:** $k = 2$, selecionando os dois vizinhos mais próximos para balancear precisão e custo computacional.
- **Árvore de Regressão:** Profundidade máxima de 19 (*max_depth*) e divisão mínima de 2 amostras (*min_samples_split*), promovendo uma modelagem detalhada e generalização eficaz.
- **Random Forest:** 23 estimadores (*n_estimators*), profundidade máxima de 10 (*max_depth*) e divisão mínima de 14 amostras (*min_samples_split*), oferecendo uma abordagem robusta contra sobreajustes.
- **XGBoost:** Configuração com *eta* = 0.12, *max_depth* = 8, *subsample* = 0.5 e *colsample_bytree* = 1, equilibrando eficiência computacional e capacidade preditiva.
- **LightGBM:** Ajuste com *num_leaves* = 31, *learning_rate* = 0.08, *feature_fraction* = 0.5 e *bagging_fraction* = 1. Essa configuração otimiza o desempenho ao minimizar sobreajustes e maximizar a velocidade de aprendizado.

4.3.2 Conjunto de Dados e *Features*

A base de dados utilizada contém 12.117 instâncias após a remoção dos *outliers*, conforme descrito anteriormente. As *features* consideradas no treinamento estão detalhadas na Tabela 3, abrangendo tanto características intrínsecas do hardware, como largura de banda e cache L2, quanto propriedades específicas do problema, como o tamanho do ladrilho em cada eixo.

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 3 – Tabela de *features* utilizadas nos modelos de regressão

4.4 Metodologia de testes

A determinação do tamanho ideal do ladrilho constitui um problema multidimensional que envolve tanto aspectos intrínsecos da arquitetura do hardware quanto características específicas da aplicação. No contexto arquitetural, fatores como o tamanho e a organização dos caches, a largura de banda da comunicação entre memória e processador, a hierarquia de memória e a capacidade de paralelização intrínseca, representada pela quantidade de *threads* por SM, desempenham papéis fundamentais. Esses aspectos determinam, em última instância, os limites superiores de desempenho e a eficiência com que os recursos computacionais podem ser utilizados.

Do ponto de vista da aplicação, variáveis como a precisão numérica (32 ou 64 bits), a ordem espacial do método de discretização e o raio do estêncil, sendo diretamente dependente da ordem espacial, têm impacto direto sobre o padrão de acessos à memória e os custos computacionais associados. A precisão numérica, por exemplo, não apenas influencia a quantidade de memória necessária para armazenar os dados, mas também afeta a quantidade de trabalho computacional devido ao maior custo de operações em ponto flutuante de maior precisão. Similarmente, a ordem espacial e o raio do estêncil determinam o número de pontos adjacentes acessados para cada cálculo, implicando em maior pressão sobre a largura de banda de memória e a eficiência dos caches. O tamanho do *grid*, por sua vez, governa a granularidade do trabalho e a distribuição das tarefas entre os recursos de hardware disponíveis, sendo, portanto, um fator determinante na saturação da capacidade de execução paralela da GPU.

A fim de avaliar a influência dessas variáveis no desempenho computacional, foi conduzido um conjunto abrangente de experimentos em duas arquiteturas de GPU distintas. Essas arquiteturas apresentam diferentes configurações de hierarquia de memória, capacidade de paralelismo e largura de banda, permitindo a análise da generalidade dos resultados obtidos. Os experimentos consistiram na variação sistemática de parâmetros

fundamentais, como o tamanho do *grid*, a precisão numérica e a ordem espacial, avaliando o desempenho do sistema em cenários diversos. Em cada caso, comparou-se o tempo de execução obtido a partir de recomendações otimizadas para o tamanho do ladrilho com aquele obtido a partir de configurações padrão, referidas como *baseline*, que não empregam técnicas de otimização.

Os tempos de execução foram registrados como a média de três execuções independentes para cada configuração experimental, de modo a reduzir a influência de variações não determinísticas introduzidas pelo sistema operacional, pelo estado do hardware ou por outros fatores externos. Essa metodologia minimizou potenciais vieses associados a ruídos aleatórios.

Os resultados experimentais obtidos permitiram uma análise quantitativa da eficácia das recomendações otimizadas de tamanho de ladrilho em comparação às configurações *baseline*. Além disso, os testes possibilitaram a identificação de cenários nos quais as otimizações resultaram em ganhos significativos de desempenho, bem como situações em que a influência das recomendações foi limitada.

4.5 Configurações dos experimentos

A avaliação experimental das recomendações de tamanhos de ladrilho geradas pelos seis modelos preditivos foi realizada considerando configurações distintas tanto do hardware (GPUs) quanto da aplicação. Para cada configuração, os tempos de execução foram comparados entre a implementação otimizada com ladrilhamento e uma implementação de referência, utilizada como *baseline*. Esta implementação *baseline* corresponde ao mesmo kernel desenvolvido em OpenMP, sem a aplicação de técnicas de ladrilhamento, assegurando que a comparação seja direcionada exclusivamente ao impacto das otimizações implementadas. Abaixo, apresentamos o 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
```

De forma análoga, o código otimizado com suporte a ladrilhamento, também implementado em OpenMP, é descrito abaixo. Essa versão incorporou a abordagem de ladrilhamento com a diretiva *#pragma omp tile sizes*, onde cada variável *BLOCK* representa o tamanho do *tile* em cada eixo sugerido pelo modelo:

```
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++){
  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
```

Os experimentos foram conduzidos em duas arquiteturas distintas de GPUs: NVIDIA RTX 2080 Super (arquitetura Turing) e NVIDIA V100 (arquitetura Volta). Essas GPUs foram escolhidas devido às diferenças significativas em seus atributos arquiteturais, como capacidade computacional em ponto flutuante de dupla precisão, largura de banda de memória, tamanho de caches e memória compartilhada, conforme detalhado na Tabela 4.

Atributo	RTX 2080 Super (Turing)	V100 (Volta)
Arquitetura	Turing	Volta
SMs (Multiprocessadores de Streaming)	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 Global	8 GB	32 GB
Largura da Banda de Memória	496 GB/s	900 GB/s
Memória Compartilhada por SM	64 KB	96 KB
Tamanho do Cache L2	4 MB	6 MB

Tabela 4 – Atributos das GPUs utilizadas nos experimentos

O código avaliado nos experimentos é uma versão simplificada do Simwave (Souza et al., 2022b), implementado na linguagem C, com otimizações baseadas em ladrilhamento suportadas pelo compilador *Clang* 13.0 (Kruse, 2021). Durante a compilação, foram utilizadas as seguintes *flags* para maximizar o desempenho: `-O3 -fPIC -ffast-math -fopenmp -fopenmp-version=51 -fopenmp-targets=nvptx64 -Xopenmp-target`. Além disso, o ambiente de software foi configurado em um contêiner *Singularity*, executando em um *cluster Slurm*, garantindo reprodutibilidade e isolamento do ambiente.

Os experimentos foram realizados em cenários variados, envolvendo combinações de parâmetros tanto da aplicação quanto da arquitetura de hardware. As principais configurações testadas incluem:

- **Técnicas de recomendação de ladrilho:** Seis modelos preditivos foram empregados para a recomendação de tamanhos de ladrilho: k-NN, Árvore de Regressão, *Random Forest*, XGBoost, LightGBM e J48. Esses modelos foram treinados previamente com base em características da aplicação e do hardware.

- **Parâmetros das configurações:**

- **Precisão numérica:** Float32 e Float64.
- **Ordens espaciais:** 2ª, 8ª e 16ª, representando métodos de discretização de diferentes graus de precisão.
- **Tamanhos de grids:**
  - **RTX 2080 Super:** *Grids* tridimensionais variando de 200^3 a 650^3, em incrementos de 50.
  - **V100:** (Grids) tridimensionais variando de 200^3 a 1100^3, em incrementos de 50. Devido à maior capacidade de memória e largura de banda da GPU foi possível utilizar *grids* maiores.

4.6 Quantidade de Configurações Testadas

Para a RTX 2080 Super, um total de 60 configurações distintas foram testadas, considerando todas as combinações de precisão, ordem espacial e tamanho do *grid*. Cada configuração foi avaliada com os seis modelos preditivos, com três execuções independentes para cada modelo, totalizando 360 experimentos e 1080 execuções.

Para a V100, foram testadas 114 configurações distintas devido à maior capacidade do hardware, totalizando 684 experimentos e 2052 execuções. Essa expansão nas configurações permite uma análise mais aprofundada do impacto das recomendações de ladrilho em arquiteturas de maior escala.

Os tempos de execução foram registrados como médias de três execuções consecutivas, a fim de atenuar a influência de variações estocásticas no sistema, como latências de comunicação ou processos concorrentes.

A tabela Tabela 7 apresenta os *tiles* sugeridos por cada modelo para cada configuração a serem executados na GPU RTX2080 Super, já as tabelas Tabela 8 e Tabela 9 apresentam o mesmo, porém para serem executados na GPU V100.

4.7 Resultados dos experimentos

Neste trabalho, conduzimos uma avaliação sistemática do desempenho de seis modelos preditivos, dos quais cinco são modelos de regressão — k-NN (k-Nearest Neighbors), Árvore de Regressão, *Random Forest*, *XGBoost*, e *LightGBM* — e um modelo de classificação, o *J48*. A avaliação foi realizada considerando duas arquiteturas de hardware distintas, NVIDIA RTX 2080 Super e NVIDIA V100, representativas de gerações distintas de GPUs de alto desempenho, permitindo uma análise abrangente da aplicabilidade das estratégias em diferentes contextos de computação acelerada. Os resultados quantitativos obtidos são sintetizados nas Tabelas 5 e 6.

Tabela 5 – Resultados dos experimentos de 60 configurações distintas na RTX2080.

	J48	**KNN**	**Árvore Reg.**	**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 6 – Resultados dos experimentos em 114 configurações distintas na V100.

	J48	**KNN**	**Árvore Reg.**	**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	1,10	1,05	1,00

As métricas avaliadas incluem o percentual de configurações em que os tempos de execução otimizados superaram o *baseline*, representado pelo percentual de configurações com melhora, e o coeficiente de melhoria, que indica a razão média entre os tempos de execução otimizados e os tempos de execução do *baseline*.

4.7.1 Resultados na NVIDIA RTX 2080 Super

Conforme apresentado na Tabela 5 e visualmente na Figura 20, a análise das 60 configurações distintas na arquitetura Turing revelou que os modelos k-NN e Árvore de Regressão exibiram os melhores desempenhos, com melhorias observadas em 87,78% e 86,67% das configurações, respectivamente. O modelo Árvore de Regressão demonstrou um coeficiente de melhoria superior (1,17), destacando-se como a abordagem mais eficiente na média geral dos experimentos.

Apesar de sua robustez para problemas de classificação e regressão, o *Random Forest* apresentou um desempenho inferior, com apenas 80% das configurações superando o *baseline*. Os modelos baseados em gradiente, *XGBoost* e *LightGBM*, mostraram coeficientes de melhoria de 1,12 e 1,07, respectivamente, mas com percentuais de configuração otimizados menores, sugerindo que sua aplicabilidade depende de uma maior calibração de hiperparâmetros ou ajustes no ambiente de execução.

4.7.2 Resultados na NVIDIA V100

Os experimentos conduzidos na NVIDIA V100, sintetizados na Tabela 6 e visualmente apresentados na Figura 20, ampliaram o escopo da análise com 114 configurações distintas, abrangendo *grids* tridimensionais maiores devido à capacidade superior de memória e largura de banda dessa GPU. Nessa arquitetura, a Árvore de Regressão novamente se destacou, com 89,77% das configurações otimizadas e um coeficiente de melhoria de 1,11, corroborando sua eficiência em cenários mais robustos.

Os modelos XGBoost e J48 também demonstraram desempenhos consistentes, com percentuais de melhoria de 85,09% e 74,85%, respectivamente. Em contraste, o modelo LightGBM exibiu o menor percentual de configurações com melhora (48,83%).

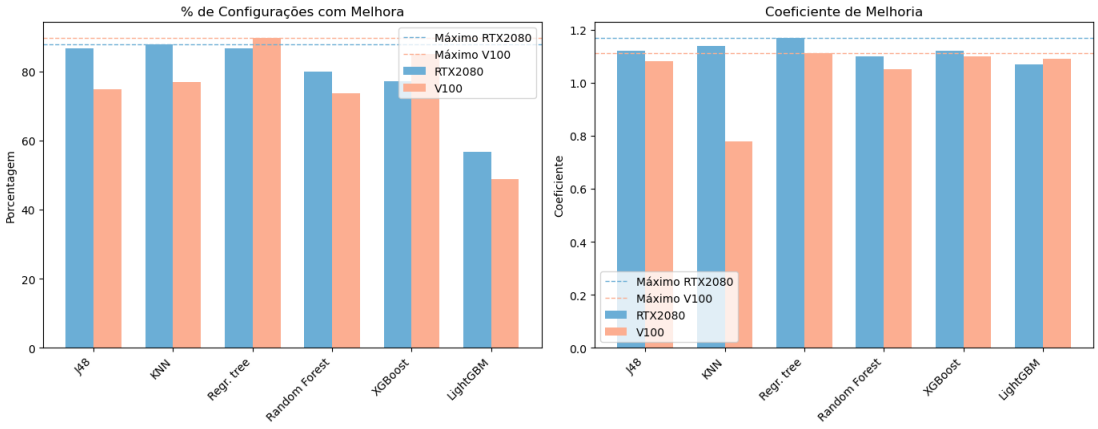


Figura 20 – Resultados dos experimentos.

4.7.3 Análise Comparativa dos Resultados

O percentual de configurações com melhora é calculado como a fração de experimentos onde o tempo de execução otimizado do modelo preditivo foi inferior ao tempo de execução do *baseline*. Para o modelo Árvore de Regressão na GPU V100, por exemplo, 89 em cada 100 experimentos apresentaram desempenho superior ao *baseline*, destacando sua eficácia na recomendação de tamanhos de ladrilho otimizados.

O coeficiente de melhoria é uma métrica de desempenho relativa que representa a média das razões entre os tempos de execução do experimento otimizado e os tempos do *baseline*. Um coeficiente de 1,11, como observado para a Árvore de Regressão na V100, indica que, em média, os tempos de execução otimizados foram 11% mais rápidos do que os tempos do *baseline*. Essa métrica possibilita a comparação da eficiência relativa entre diferentes modelos e arquiteturas, proporcionando uma visão clara sobre a robustez e a aplicabilidade de cada abordagem.

4.7.4 Implicações para a Otimização Baseada em Ladrilhamento

Esses resultados reforçam a importância de estratégias específicas de otimização baseadas em ladrilhamento, particularmente em cenários de computação de alto desempenho. Modelos preditivos que oferecem alta precisão na recomendação de tamanhos de ladrilho, como a Árvore de Regressão, demonstram ser uma abordagem promissora para maximizar a eficiência computacional em arquiteturas heterogêneas.

Grid	**Ordem Espacial**	**Precisão**	**J48**	**KNN**	**Tree regressor**	**Random forest**	**XGBoost**	**lightGBM**
200	2	float32	32x4x2	2x32x8	2x16x4	2x1x8	1x32x4	1x1x16
200	8	float32	16x4x1	4x1x1	4x2x1	1x16x1	2x2x4	1x64x2
200	16	float32	2x4x1	64x1x2	64x2x1	4x1x1	4x1x1	64x16x1
200	2	float64	32x4x2	32x4x2	32x4x2	64x8x1	8x64x4	1x1x64
200	8	float64	16x4x1	16x4x2	16x4x1	64x8x1	1x32x1	32x1x4
200	16	float64	2x4x1	4x4x1	2x4x1	2x8x1	2x8x1	64x16x1
250	2	float32	32x4x2	2x32x8	2x16x4	2x1x8	1x32x4	1x1x16
250	8	float32	16x4x1	4x1x1	4x2x1	1x16x1	2x2x4	1x64x2
250	16	float32	2x4x1	64x1x2	64x2x1	4x1x1	4x1x1	64x16x1
250	2	float64	32x4x2	32x4x2	32x4x2	64x8x1	8x64x4	1x1x64
250	8	float64	16x4x1	16x4x2	16x4x1	64x8x1	1x32x1	32x1x4
250	16	float64	2x4x1	4x4x1	2x4x1	2x8x1	2x8x1	64x16x1
300	2	float32	2x4x4	2x32x8	2x16x4	2x1x8	1x32x4	1x1x16
300	8	float32	16x2x1	4x1x1	4x2x1	1x16x1	2x2x4	1x64x2
300	16	float32	64x1x1	64x1x2	64x2x1	4x1x1	4x1x1	64x16x1
300	2	float64	2x4x4	32x4x2	32x4x2	64x8x1	8x64x4	1x1x64
300	8	float64	16x2x1	16x4x2	16x4x1	64x8x1	1x32x1	32x1x4
300	16	float64	64x1x1	4x4x1	2x4x1	2x8x1	2x8x1	64x16x1
350	2	float32	2x4x4	2x32x8	2x16x4	2x1x8	1x32x4	1x1x16
350	8	float32	16x2x1	4x1x1	4x2x1	1x16x1	2x2x4	1x64x2
350	16	float32	64x1x1	64x1x2	64x2x1	4x1x1	4x1x1	64x16x1
350	2	float64	2x4x4	32x4x2	32x4x2	64x8x1	8x64x4	1x1x64
350	8	float64	16x2x1	16x4x2	16x4x1	64x8x1	1x32x1	32x1x4
350	16	float64	64x1x1	4x4x1	2x4x1	2x8x1	2x8x1	64x16x1
400	2	float32	2x4x4	16x1x4	16x2x4	32x2x4	1x32x4	2x8x4
400	8	float32	16x2x1	16x1x1	64x1x1	32x2x4	2x2x4	2x4x2
400	16	float32	64x1x1	64x1x1	8x1x1	8x1x1	4x1x1	32x1x1
400	2	float64	2x4x4	4x4x4	2x4x4	32x2x4	8x64x4	1x8x1
400	8	float64	16x2x1	64x1x1	16x2x1	8x32x1	1x32x1	32x1x1
400	16	float64	64x1x1	32x1x1	32x1x1	8x32x1	2x8x1	4x1x1
450	2	float32	2x4x4	16x1x4	16x2x4	32x2x4	1x32x4	2x8x4
450	8	float32	16x2x1	16x1x1	64x1x1	32x2x4	2x2x4	2x4x2
450	16	float32	64x1x1	64x1x1	8x1x1	8x1x1	4x1x1	32x1x1
450	2	float64	2x4x4	4x4x4	2x4x4	32x2x4	8x64x4	1x8x1
450	8	float64	16x2x1	64x1x1	16x2x1	8x32x1	1x32x1	32x1x1
450	16	float64	64x1x1	32x1x1	32x1x1	8x32x1	2x8x1	4x1x1
500	2	float32	2x4x4	16x1x4	16x2x4	32x2x4	1x32x4	2x8x4
500	8	float32	16x2x1	16x1x1	64x1x1	32x2x4	2x2x4	2x4x2
500	16	float32	64x1x1	64x1x1	8x1x1	8x1x1	4x1x1	32x1x1
500	2	float64	2x4x4	4x4x4	2x4x4	32x2x4	8x64x4	1x8x1
500	8	float64	16x2x1	64x1x1	16x2x1	8x32x1	1x32x1	32x1x1
500	16	float64	64x1x1	32x1x1	32x1x1	8x32x1	2x8x1	4x1x1
550	2	float32	2x8x4	16x1x4	16x2x4	32x2x4	64x64x8	2x8x4
550	8	float32	16x2x1	16x1x1	64x1x1	32x2x4	64x1x4	2x4x2
550	16	float32	4x4x1	32x1x1	8x1x1	8x1x1	16x1x1	32x1x1
550	2	float64	2x8x4	4x4x4	2x4x4	32x2x4	4x1x4	1x8x1
550	8	float64	16x2x1	64x1x1	16x2x1	8x32x1	64x1x1	32x1x1
550	16	float64	4x4x1	32x1x1	32x1x1	8x32x1	64x1x1	4x1x1
600	2	float32	2x8x4	16x1x4	16x2x4	32x2x4	64x64x8	2x8x4
600	8	float32	16x2x1	16x1x1	64x1x1	32x2x4	64x1x4	2x4x2
600	16	float32	4x4x1	32x1x1	8x1x1	8x1x1	16x1x1	32x1x1
600	2	float64	2x8x4	4x4x4	2x4x4	32x2x4	4x1x4	1x8x1
600	8	float64	16x2x1	64x1x1	16x2x1	8x32x1	64x1x1	32x1x1
600	16	float64	4x4x1	32x1x1	32x1x1	8x32x1	64x1x1	4x1x1
650	2	float32	2x8x4	16x1x4	16x2x4	32x2x4	64x64x8	2x8x4
650	8	float32	16x2x1	16x1x1	64x1x1	32x2x4	64x1x4	2x4x2
650	16	float32	4x4x1	32x1x1	8x1x1	8x1x1	16x1x1	32x1x1
650	2	float64	2x8x4	4x4x4	2x4x4	32x2x4	4x1x4	1x8x1
650	8	float64	16x2x1	64x1x1	16x2x1	8x32x1	64x1x1	32x1x1
650	16	float64	4x4x1	32x1x1	32x1x1	8x32x1	64x1x1	4x1x1

Tabela 7 – Tabela dos *tiles* recomendados por cada modelo para cada configuração para a GPU RTX2080 Super

Grid	**Ordem Espacial**	**Precisão**	**J48**	**KNN**	**Tree regressor**	**Random forest**	**XGBoost**	**lightGBM**
200	2	float32	2x4x1	1x32x4	1x32x1	2x1x4	2x32x4	1x2x8
200	8	float32	4x1x1	2x8x2	4x1x2	2x2x1	1x64x1	64x1x8
200	16	float32	32x1x1	32x4x2	2x8x1	1x1x1	4x1x1	2x2x2
200	2	float64	2x4x1	16x16x2	1x64x1	4x2x1	1x16x1	2x64x64
200	8	float64	4x1x1	4x1x1	4x1x1	2x2x1	4x1x1	64x4x1
200	16	float64	32x1x1	1x2x1	1x2x1	1x1x1	8x1x1	64x8x2
250	2	float32	2x4x1	1x32x4	1x32x1	2x1x4	2x32x4	1x2x8
250	8	float32	4x1x1	2x8x2	4x1x2	2x2x1	1x64x1	64x1x8
250	16	float32	32x1x1	32x4x2	2x8x1	1x1x1	4x1x1	2x2x2
250	2	float64	2x4x1	16x16x2	1x64x1	4x2x1	1x16x1	2x64x64
250	8	float64	4x1x1	4x1x1	4x1x1	2x2x1	4x1x1	64x4x1
250	16	float64	32x1x1	1x2x1	1x2x1	1x1x1	8x1x1	64x8x2
300	2	float32	2x16x1	1x32x4	1x32x1	2x1x4	2x32x4	1x2x8
300	8	float32	4x2x1	4x2x4	4x1x2	2x2x1	1x64x1	64x1x8
300	16	float32	16x1x1	32x4x2	2x8x1	1x1x1	4x1x1	2x2x2
300	2	float64	2x16x1	16x16x2	1x64x1	4x2x1	1x16x1	2x64x64
300	8	float64	4x2x1	4x1x1	4x1x1	2x2x1	4x1x1	64x4x1
300	16	float64	16x1x1	1x1x1	1x2x1	1x1x1	8x1x1	64x8x2
350	2	float32	2x16x1	1x32x4	1x32x1	2x1x4	2x32x4	1x2x8
350	8	float32	4x2x1	4x2x4	4x1x2	2x2x1	1x64x1	64x1x8
350	16	float32	16x1x1	32x4x2	2x8x1	1x1x1	4x1x1	2x2x2
350	2	float64	2x16x1	16x16x2	1x64x1	4x2x1	1x16x1	2x64x64
350	8	float64	4x2x1	4x1x1	4x1x1	2x2x1	4x1x1	64x4x1
350	16	float64	16x1x1	1x1x1	1x2x1	1x1x1	8x1x1	64x8x2
400	2	float32	2x16x1	64x1x1	64x1x1	16x1x8	2x32x4	64x1x4
400	8	float32	4x2x1	4x1x2	32x2x1	4x2x2	1x64x1	16x64x1
400	16	float32	16x1x1	16x1x1	16x1x1	1x1x1	4x1x1	4x1x1
400	2	float64	2x16x1	2x8x1	16x1x4	4x2x2	1x16x1	32x64x1
400	8	float64	4x2x1	16x2x1	64x1x1	32x64x1	4x1x1	32x64x1
400	16	float64	16x1x1	8x2x1	2x1x1	32x64x1	8x1x1	2x64x1
450	2	float32	2x16x1	64x1x1	64x1x1	16x1x8	2x32x4	64x1x4
450	8	float32	4x2x1	4x1x2	32x2x1	4x2x2	1x64x1	16x64x1
450	16	float32	16x1x1	16x1x1	16x1x1	1x1x1	4x1x1	4x1x1
450	2	float64	2x16x1	2x8x1	16x1x4	4x2x2	1x16x1	32x64x1
450	8	float64	4x2x1	16x2x1	64x1x1	32x64x1	4x1x1	32x64x1
450	16	float64	16x1x1	8x2x1	2x1x1	32x64x1	8x1x1	2x64x1
500	2	float32	2x16x1	64x1x1	64x1x1	16x1x8	2x32x4	64x1x4
500	8	float32	4x2x1	4x1x2	32x2x1	4x2x2	1x64x1	16x64x1
500	16	float32	16x1x1	16x1x1	16x1x1	1x1x1	4x1x1	4x1x1
500	2	float64	2x16x1	2x8x1	16x1x4	4x2x2	1x16x1	32x64x1
500	8	float64	4x2x1	16x2x1	64x1x1	32x64x1	4x1x1	32x64x1
500	16	float64	16x1x1	8x2x1	2x1x1	32x64x1	8x1x1	2x64x1
550	2	float32	2x8x4	64x1x1	64x1x1	16x1x8	64x32x8	64x1x4
550	8	float32	64x1x1	4x1x2	32x2x1	4x2x2	8x1x1	16x64x1
550	16	float32	4x4x1	4x2x1	16x1x1	1x1x1	8x1x1	4x1x1
550	2	float64	2x8x4	2x8x1	16x1x4	4x2x2	4x4x1	32x64x1
550	8	float64	64x1x1	16x2x1	64x1x1	32x64x1	64x1x1	32x64x1
550	16	float64	4x4x1	8x2x1	2x1x1	32x64x1	8x1x1	2x64x1
600	2	float32	2x8x4	64x1x1	64x1x1	16x1x8	64x32x8	64x1x4
600	8	float32	64x1x1	4x1x2	32x2x1	4x2x2	8x1x1	16x64x1
600	16	float32	4x4x1	4x2x1	16x1x1	1x1x1	8x1x1	4x1x1
600	2	float64	2x8x4	2x8x1	16x1x4	4x2x2	4x4x1	32x64x1
600	8	float64	64x1x1	16x2x1	64x1x1	32x64x1	64x1x1	32x64x1
600	16	float64	4x4x1	8x2x1	2x1x1	32x64x1	8x1x1	2x64x1
650	2	float32	2x8x4	64x1x1	64x1x1	16x1x8	64x32x8	64x1x4
650	8	float32	64x1x1	4x1x2	32x2x1	4x2x2	8x1x1	16x64x1
650	16	float32	4x4x1	4x2x1	16x1x1	1x1x1	8x1x1	4x1x1
650	2	float64	2x8x4	2x8x1	16x1x4	4x2x2	4x4x1	32x64x1
650	8	float64	64x1x1	16x2x1	64x1x1	32x64x1	64x1x1	32x64x1
650	16	float64	4x4x1	8x2x1	2x1x1	32x64x1	8x1x1	2x64x1

Tabela 8 – Tabela dos *tiles* recomendados por cada modelo para cada configuração para a GPU V100 Parte 1

Grid	Ordem Espacial	Precisão	J48	KNN	Tree regressor	Random forest	XGBoost	lightGBM
700	2	float32	2x8x4	64x1x1	64x1x1	16x1x8	64x32x8	64x1x4
700	8	float32	64x1x1	4x1x2	32x2x1	4x2x2	8x1x1	16x64x1
700	16	float32	4x4x1	4x2x1	16x1x1	1x1x1	8x1x1	4x1x1
700	2	float64	2x8x4	2x8x1	16x1x4	4x2x2	4x4x1	32x64x1
700	8	float64	64x1x1	16x2x1	64x1x1	32x64x1	64x1x1	32x64x1
700	16	float64	4x4x1	8x2x1	2x1x1	32x64x1	8x1x1	2x64x1
750	2	float32	2x8x4	64x1x1	64x1x1	16x1x8	64x32x8	64x1x4
750	8	float32	64x1x1	4x1x2	32x2x1	4x2x2	8x1x1	16x64x1
750	16	float32	4x4x1	4x2x1	16x1x1	1x1x1	8x1x1	4x1x1
750	2	float64	2x8x4	2x8x1	16x1x4	4x2x2	4x4x1	32x64x1
750	8	float64	64x1x1	16x2x1	64x1x1	32x64x1	64x1x1	32x64x1
750	16	float64	4x4x1	8x2x1	2x1x1	32x64x1	8x1x1	2x64x1
800	2	float32	2x8x4	4x1x1	4x2x1	4x1x4	64x32x8	16x1x4
800	8	float32	64x1x1	2x2x1	2x2x1	2x2x1	8x1x1	64x1x1
800	16	float32	4x4x1	1x32x32	4x2x1	8x1x1	8x1x1	4x1x1
800	2	float64	2x8x4	4x1x1	4x2x1	4x1x4	4x4x1	16x1x4
800	8	float64	64x1x1	2x2x1	2x2x1	2x2x1	64x1x1	16x1x1
800	16	float64	4x4x1	1x32x32	4x2x1	8x1x1	8x1x1	4x1x1
850	2	float32	2x8x4	4x1x1	4x2x1	4x1x4	64x32x8	16x1x4
850	8	float32	64x1x1	2x2x1	2x2x1	2x2x1	8x1x1	64x1x1
850	16	float32	4x4x1	1x32x32	4x2x1	8x1x1	8x1x1	4x1x1
850	2	float64	2x8x4	4x1x1	4x2x1	4x1x4	4x4x1	16x1x4
850	8	float64	64x1x1	2x2x1	2x2x1	2x2x1	64x1x1	16x1x1
850	16	float64	4x4x1	1x32x32	4x2x1	8x1x1	8x1x1	4x1x1
900	2	float32	2x8x4	4x1x1	4x2x1	4x1x4	64x32x8	16x1x4
900	8	float32	64x1x1	2x2x1	2x2x1	2x2x1	8x1x1	64x1x1
900	16	float32	4x4x1	1x32x32	4x2x1	8x1x1	8x1x1	4x1x1
900	2	float64	2x8x4	4x1x1	4x2x1	4x1x4	4x4x1	16x1x4
900	8	float64	64x1x1	2x2x1	2x2x1	2x2x1	64x1x1	16x1x1
900	16	float64	4x4x1	1x32x32	4x2x1	8x1x1	8x1x1	4x1x1
950	2	float32	2x8x4	4x1x1	4x2x1	4x1x4	64x32x8	16x1x4
950	8	float32	64x1x1	2x2x1	2x2x1	2x2x1	8x1x1	64x1x1
950	16	float32	4x4x1	1x32x32	4x2x1	8x1x1	8x1x1	4x1x1
950	2	float64	2x8x4	4x1x1	4x2x1	4x1x4	4x4x1	16x1x4
950	8	float64	64x1x1	2x2x1	2x2x1	2x2x1	64x1x1	16x1x1
950	16	float64	4x4x1	1x32x32	4x2x1	8x1x1	8x1x1	4x1x1
1000	2	float32	2x8x4	4x1x1	4x2x1	4x1x4	64x32x8	16x1x4
1000	8	float32	64x1x1	2x2x1	2x2x1	2x2x1	8x1x1	64x1x1
1000	16	float32	4x4x1	1x32x32	4x2x1	8x1x1	8x1x1	4x1x1
1000	2	float64	2x8x4	4x1x1	4x2x1	4x1x4	4x4x1	16x1x4
1000	8	float64	64x1x1	2x2x1	2x2x1	2x2x1	64x1x1	16x1x1
1000	16	float64	4x4x1	1x32x32	4x2x1	8x1x1	8x1x1	4x1x1
1050	2	float32	2x8x4	4x1x1	4x2x1	4x1x4	4x2x1	16x1x4
1050	8	float32	64x1x1	2x2x1	2x2x1	2x2x1	16x1x1	64x1x1
1050	16	float32	4x4x1	1x32x32	4x2x1	8x1x1	16x1x1	4x1x1
1050	2	float64	2x8x4	4x1x1	4x2x1	4x1x4	4x2x1	16x1x4
1050	8	float64	64x1x1	2x2x1	2x2x1	2x2x1	32x1x1	16x1x1
1050	16	float64	4x4x1	1x32x32	4x2x1	8x1x1	16x1x1	4x1x1
1100	2	float32	2x8x4	4x1x1	4x2x1	4x1x4	4x2x1	16x1x4
1100	8	float32	64x1x1	2x2x1	2x2x1	2x2x1	16x1x1	64x1x1
1100	16	float32	4x4x1	1x32x32	4x2x1	8x1x1	16x1x1	4x1x1
1100	2	float64	2x8x4	4x1x1	4x2x1	4x1x4	4x2x1	16x1x4
1100	8	float64	64x1x1	2x2x1	2x2x1	2x2x1	32x1x1	16x1x1
1100	16	float64	4x4x1	1x32x32	4x2x1	8x1x1	16x1x1	4x1x1

Tabela 9 – Tabela dos *tiles* recomendados por cada modelo para cada configuração para a GPU V100 Parte 2

Capítulo 5

Conclusão

A escolha do melhor tamanho de ladrilho pode ser altamente dependente de uma série de fatores, incluindo a arquitetura da GPU, as características específicas do conjunto de dados e os requisitos da aplicação em questão.

Neste estudo, exploramos o impacto das otimizações de ladrilhamento no desempenho de seis modelos de aprendizado de máquina, sendo cinco de regressão e um de classificação. Os modelos analisados foram K-NN , Árvore de Regressão, *Random Forest*, XGBoost, LightGBM e J48. Avaliamos o desempenho desses modelos em duas arquiteturas de GPU distintas, a NVIDIA RTX2080 e a NVIDIA V100, permitindo uma análise comparativa de como diferentes configurações de hardware afetam o comportamento dos modelos em termos de latência e throughput.

Durante os experimentos, focamos na melhoria do desempenho resultante da aplicação das otimizações de ladrilhamento. As otimizações foram aplicadas para identificar os tamanhos de ladrilho que maximizam a eficiência de cada modelo em termos de tempo de execução, considerando as características intrínsecas das arquiteturas das GPUs envolvidas. Para isso, conduzimos experimentos com diferentes tamanhos de *grids* e observamos o impacto no tempo de execução e na eficiência de cada modelo.

Os resultados obtidos indicam que, entre os modelos avaliados, a Árvore de Regressão demonstrou ser o modelo mais robusto em ambas as GPUs, consistentemente apresentando melhorias no desempenho, com perdas mínimas de eficiência. Este modelo, que utiliza uma estrutura hierárquica de decisão, conseguiu se adaptar eficientemente aos diferentes tamanhos de ladrilho, exibindo um alto percentual de configurações que superaram o *baseline*, independentemente da arquitetura da GPU. Em ambas as GPUs, a Árvore de Regressão obteve mais de 75% de sucesso nas recomendações de tamanhos de ladrilho, destacando-se como a estratégia mais eficaz para otimização de desempenho.

Além disso, a Árvore de Regressão apresentou o maior coeficiente de melhoria em ambas as plataformas, com uma vantagem de até 17% em termos de desempenho otimizado em comparação ao *baseline*, dependendo da configuração e da arquitetura utilizada. Esse coeficiente sugere que o modelo é altamente eficiente na adaptação de seus parâmetros de execução às condições específicas de hardware e carga de trabalho, maximizando a utilização dos recursos da GPU.

Modelos como o *Random Forest*, J48, XGBoost e k-NN mostraram um desempenho relativamente bom, mas com uma maior variabilidade nas melhorias de desempenho, dependendo da arquitetura de GPU. Embora esses modelos tenham apresentado taxas de sucesso inferiores à Árvore de Regressão, especialmente em configurações mais desafiadoras, eles ainda demonstraram um desempenho aceitável, tendo uma taxa de sucesso acima 70% nas otimizações.

A estratégia de otimização proposta neste estudo, que utiliza o modelo de Árvore de Regressão, mostrou-se altamente eficaz, proporcionando um bom desempenho mesmo em cenários nos quais o ladrilhamento ainda não havia sido testado. Isso sugere que, em contextos nos quais o tempo de execução é crítico e as configurações de ladrilhamento não são previamente conhecidas, o uso de um modelo de aprendizado de máquina para prever o tamanho de ladrilho adequado pode ser uma abordagem prática e eficiente, economizando tempo de experimentação e evitando a necessidade de testar inúmeras configurações manuais.

Os resultados deste estudo também geraram um artigo (Silva et al., 2024), que apresenta avanços na compreensão da otimização de desempenho em arquiteturas de GPUs por meio de técnicas de ladrilhamento. A Árvore de Regressão se demonstrou uma abordagem eficaz para a seleção do tamanho de ladrilho, destacando-se especialmente em cenários onde a escolha desse parâmetro afeta diretamente o desempenho. A aplicação dessa técnica revelou seu potencial em contextos de computação de alto desempenho, ao fornecer uma metodologia adaptativa para o ajuste automático do tamanho do ladrilho. Além disso, os resultados indicam que a personalização das otimizações, considerando as características específicas da arquitetura de GPU e da aplicação, pode resultar em melhorias consideráveis no desempenho, com um custo reduzido de ajustes manuais. Para trabalhos futuros, outras técnicas de aprendizado de máquina, como redes neurais, podem ser investigadas como alternativas promissoras para aperfeiçoar a seleção do tamanho do ladrilho, potencializando ainda mais as capacidades de otimização em diferentes cenários computacionais.

Referências

J. Krueger, P. Micikevicius, and S. Williams, "Optimization of forward wave modeling on contemporary hpc architectures," LBNL Technical Report 5751E, Tech. Rep., 07 2012.

J. F. D. Souza, L. S. Machado, E. S. Gomi, C. Tadonki, S. McIntosh-Smith, and H. Senger, "Performance of openmp offloading for the acoustic wave stencil on gpus," in *Supercomputing*. Dallas, TX, USA: Supercomputing Conference, November 2022, universidade Federal de São Carlos, Brazil, Universidade de São Paulo, Brazil, Mines ParisTech / PSL, France, University of Bristol, UK.

J. F. Souza, J. B. D. Moreira, K. J. Roberts, R. di Ramos Alves Gaioso, E. S. Gomi, E. C. N. Silva, and H. Senger, "Simwave: A finite difference simulator for acoustic waves propagation," *arXiv preprint*, 2022.

J. Virieux and S. Operto, "An overview of full-waveform inversion in exploration geophysics," *Geophysics*, vol. 74, no. 6, pp. WCC1–WCC26, 2009. [Online]. Available: <http://library.seg.org/doi/10.1190/1.3238367>

R. Weiss and J. Shragge, "Solving 3d anisotropic elastic wave equations on parallel gpu devices," *Geophysics*, vol. 78, 2013.

J. Xue, *Loop Tiling for Parallelism*, ser. Kluwer International Series in Engineering and Computer Science. New York: Springer Science+Business Media New York, 2000, vol. SECS 575, originally published by Kluwer Academic Publishers, New York in 2000.

M. Korch and T. Werner, "Improving locality of explicit one-step methods on gpus by tiling across stages and time steps," *Future Generation Computer Systems*, 2019, received 15 March 2019, Accepted 31 July 2019, Available online 24 September 2019.

T. Grosser, A. Cohen, P. Sadayappan, J. Holewinski, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for gpus," in *CGO '14*. Orlando, FL, USA: ACM, February 15–19 2014.

C. Xu, S. R. Kirk, and S. Jenkins, "Tiling for performance tuning on different models of gpus," in *Second International Symposium on Information Science and Engineering*. IEEE, 2009, p. 60.

Z. Li and Y. Song, "Automatic tiling of iterative stencil loops," *ACM Transactions on Programming Languages and Systems*, vol. 26, no. 6, pp. 975–1028, 2004.

A. M. Malik, "Optimal tile size selection problem using machine learning," in *2012 11th International Conference on Machine Learning and Applications*. USA: IEEE, 2012.

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

M. Rahman, L.-N. Pouchet, and P. Sadayappan, "Neural network assisted tile size selection," *The Ohio State University*, 2010.

H. Igel, *About Computational Seismology: A Practical Introduction*. Oxford University Press, 11 2016, pp. 1–10.

K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Rev.*, vol. 51, no. 1, pp. 129–159, 2009. [Online]. Available: <https://doi.org/10.1137/070693199>

W. Bielecki and M. Palkowski, "Space-time loop tiling for dynamic programming codes," *Electronics*, vol. 10, no. 18, p. 2233, 2021. [Online]. Available: <https://www.mdpi.com/xxxxxx>

M. Kruse, "Loop transformations using clang's abstract syntax tree," in *Proceedings of the 50th International Conference on Parallel Processing (ICPP)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 21:1–21:7, argonne National Laboratory, Lemont, Illinois, USA. michael.kruse@anl.gov.

G. Singh, D. Diamantopoulos, C. Hagleitner, J. Gómez-Luna, S. Stuijk, O. Mutlu, and H. Corporaal, "Nero: A near high-bandwidth memory stencil accelerator for weather prediction modeling," in *FPL*, 2020.

F. Luporini, M. Louboutin, M. Lange, N. Kukreja, P. Witte, J. Hückelheim, and G. J. Gorman, "Architecture and performance of devito, a system for automated stencil computation," *ACM Transactions on Mathematical Software*, vol. 46, no. 1, pp. 1–28, 2020.

A. Denzler, R. Bera, N. Hajinazar, G. Singh, G. F. Oliveira, J. Gómez-Luna, and O. Mutlu, "Casper: Accelerating stencil computation using near-cache processing," *IEEE Access*, Dezembro 2021, 1ETH Zürich 2Simon Fraser University.

I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed., ser. Morgan Kaufmann series in data management systems. Morgan Kaufmann, 2005, library of Congress Cataloging-in-Publication Data.

I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th ed., ser. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, 2016. [Online]. Available: <libgen.li/file.php?md5=2ff05cc2c0f0b0c6d857052092f0e16e>

T. M. Cover and P. E. Hart, "Nearest neighbor pattern classification." *IEEE Trans. Inf. Theory*, pp. 21–27, 1967. [Online]. Available: <http://dblp.uni-trier.de/db/journals/tit/tit13.html#CoverH67>

L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*. Belmont, CA: Wadsworth International Group, 1986.

L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: <https://link.springer.com/article/10.1023/A:1010933404324>

T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 785–794. [Online]. Available: <https://dl.acm.org/doi/10.1145/2939672.2939785>

Q. Meng, Z. Ma, H. Li, J. Leskovec, X. Zhang, K. He *et al.*, "Lightgbm: A highly efficient gradient boosting decision tree," in *Proceedings of the 31st Conference on Neural Information Processing Systems (NeurIPS)*, 2017. [Online]. Available: <https://arxiv.org/abs/1706.02878>

D. B. Kirk, W. mei W. Hwu, and I. E. Hajj, *Programming Massively Parallel Processors: A Hands-on Approach*, 4th ed. Elsevier Inc., 2023.

T. Deakin and T. G. Mattson, *Programming your GPU with OpenMP: Performance Portability for GPUs*, ser. Scientific and Engineering Computation. Cambridge, Massachusetts: The MIT Press, 2023.

T. Deakin, S. McIntosh-Smith, and T. G. Mattson, "Programming your gpu with openmp," in *Supercomputing 2022 tutorial*, S.l., 2022.

J. W. Tukey *et al.*, *Exploratory data analysis*. Springer, 1977, vol. 2.

T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proc. ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: ACM, 2019, p. 2623–2631. [Online]. Available: <https://doi.org/10.1145/3292500.3330701>

T. Silva, E. Gomi, and H. Senger, "Escolha do ladrilhamento para um simulador de ondas acústicas em gpus por meio de aprendizado de máquina," in *Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho*. Porto Alegre, RS, Brasil: SBC, 2024, pp. 216–227. [Online]. Available: <https://proceedings-sol.sbc.org.br/index.php/sscad/article/view/30999>

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato Tiago da Silva, realizada em 16/12/2024.

Comissão Julgadora:

Prof. Dr. Hermes Senger (UFSCar)

Prof. Dr. Helio Crestana Guardia (UFSCar)

Profa. Dra. Sarita Mazzini Bruschi (USP)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.