Wave Order Picking PLIwC²

Uma aplicação PLI com CPLEX e CUDA para o Desafio SBPO 2025

Fábio Linhares
fl@ic.ufal.br

Hans Ponfick
hapl@ic.ufal.br

Resumo

Este artigo apresenta o desenvolvimento e a análise de uma abordagem de otimização exata, fundamentada em Programação Linear Inteira (PLI) e significativamente acelerada por GPU com NVIDIA CUDA, para a resolução do Problema de Wave Order Picking *WOP* proposto no Desafio SBPO 2025 do Mercado Livre. O *WOP*, um problema de natureza NP-Difícil, consiste na seleção otimizada de um subconjunto de pedidos (denominado wave) e um subconjunto de corredores a serem visitados em um centro de distribuição, com o objetivo de maximizar a eficiência da coleta. Detalhamos a formulação matemática do problema, incluindo a complexa tarefa de linearizar uma função objetivo fracionária, para a qual exploramos três métodos distintos: Variável Inversa, Charnes-Cooper e o algoritmo iterativo de Dinkelbach. Discutimos a implementação de um modelo flexível que acomoda tanto restrições rígidas quanto suaves (com penalidades), a aplicação de técnicas avançadas de pré-processamento de dados e do modelo matemático para redução do espaço de busca, e a crucial integração com o solver IBM CPLEX e a tecnologia CUDA. Esta combinação permitiu obter resultados altamente competitivos, igualando a solução ótima em 14 a 16 das 20 instâncias do desafio, dentro de limites de tempo exíguos (aproximadamente 10 minutos por instância). Uma análise detalhada e passo a passo de uma instância de exemplo (t/0001) é fornecida para ilustrar a aplicação prática e os meandros do modelo. Os resultados experimentais e a discussão subsequente demonstram a viabilidade e a eficácia da abordagem exata quando sinergicamente combinada com modelagem matemática sofisticada e aceleração computacional de ponta, oferecendo uma alternativa poderosa às tradicionais abordagens heurísticas para este complexo problema logístico.

Palavras-chave: Wave Order Picking, Otimização Exata, Programação Linear Inteira (PLI), Aceleração por GPU (CUDA), Linearização de Função Fracionária, CPLEX, Pré-processamento, Logística de E-commerce.

1 Nota dos autores

Este artigo foi elaborado como parte dos requisitos avaliativos da disciplina de Otimização Contínua e Combinatória, componente curricular do Mestrado em Informática da Universidade Federal de Alagoas (UFAL). A proposta consistiu em pesquisar, analisar e implementar uma solução para um problema real e complexo de otimização combinatória, apresentando os resultados em formato de artigo científico. Para isso, escolhemos trabalhar com o Wave Order Picking, um problema logístico proposto pelo Mercado Livre no Desafio SBPO 2025.

Como a implementação computacional era obrigatória, optamos inici-

almente por desenvolver uma abordagem heurística, considerando a alta complexidade do problema — classificado como NP-Difícil. No entanto, decidimos avançar para uma abordagem exata, utilizando Programação Linear Inteira (PLI), com aceleração por GPU via NVIDIA CUDA. Essa escolha mostrou-se não apenas viável, como também extremamente eficaz, permitindo alcançar soluções ótimas em tempos computacionalmente aceitáveis para as instâncias do desafio.

Adotamos, ao longo do texto, um estilo conscientemente didático e acessível, sem abrir mão do rigor técnico exigido pela natureza do tema. Nossa intenção foi tornar o conteúdo compreensível não apenas para especialistas em otimização e ciência da computação, mas também para leitores de áreas correlatas ou interessados no assunto, mesmo sem formação técnica aprofundada. Esperamos, com isso, contribuir para uma divulgação mais ampla do conhecimento, equilibrando clareza, precisão conceitual e profundidade analítica.

2 Introdução

A eficiência na separação de pedidos (*order picking*) é um dos fatores mais críticos na operação de centros de distribuição, podendo representar até 70% dos custos operacionais (10). Em especial no comércio eletrônico, onde o volume de pedidos é alto e o tempo de ciclo é restrito, otimizar essa etapa é vital (7, 21).

O problema de *Wave Order Picking* (WOP) é uma formulação combinatória relevante nesse contexto. Nele, "waves" (ondas) são subconjuntos de pedidos coletados simultaneamente, geralmente visando eficiência operacional. O objetivo é maximizar a produtividade da coleta, definida como:

$$\max \quad \frac{\sum_{p \in P} \text{score}_p \, x_p}{1 + \sum_{a \in A} y_a},$$

onde o numerador representa o benefício acumulado dos pedidos selecionados e o denominador penaliza ondas que exigem muitos corredores — ou seja, ondas pouco compactas.

Apesar da importância prática do WOP, sua complexidade combinatória — associada a decisões simultâneas de agrupamento de pedidos e roteamento — torna sua resolução exata um desafio computacional: trata-se de um problema *NP-difícil* (2). Por essa razão, a literatura predominante recorre a heurísticas e meta-heurísticas (22), que oferecem boas soluções em tempo razoável, mas sem garantias ótimas ou controle preciso de gap.

Contrariando essa tendência, propomos neste artigo uma abordagem exata e computacionalmente viável para o WOP, denominada **PLIwC²** — *Programação Linear Inteira com penalidades suaves e aceleração CUDA*. Nosso método combina modelagem matemática robusta com otimizações computacionais estratégicas, divididas em cinco pilares:

1. **Modelagem fracionária e métodos de linearização:** a função objetivo é transformada em forma linear via três estratégias: *variável inversa*, *Charnes-Cooper* e o algoritmo iterativo de *Dinkelbach*. Cada método é discutido quanto à estabilidade numérica, densidade do modelo e impacto prático.

2. **Restrições suaves com penalidades calibráveis:** o modelo admite violação controlada de restrições como capacidade de carrinhos ($L_{\max}$) e número de corredores ($A_{\max}$), mediante penalização direta na função objetivo. Essa flexibilidade melhora a robustez e adaptabilidade às condições operacionais reais.

3. **Pré-processamento estruturado com GPU:** antes da modelagem, os dados são organizados, filtrados e estruturados usando operações vetoriais em GPU (via CuPy), acelerando etapas tipicamente quadráticas, como geração de matrizes de conflito e filtragem de pedidos inviáveis.

4. **Integração com CPLEX via PuLP:** a modelagem MILP é realizada com PuLP, permitindo expressividade em Python, e resolvida com CPLEX, que garante estabilidade e qualidade de solução, especialmente relevante em modelos com FO fracionária e muitas variáveis.

5. **Avaliação iterativa e paralela da função objetivo:** no método de Dinkelbach, cada iteração exige reavaliação da razão FO. Utilizamos GPU para realizar esse cálculo de forma vetorizada e paralela, o que reduz drasticamente o tempo de cada ciclo.

O diferencial computacional da proposta reside, portanto, na utilização de GPU para acelerar tanto o pré-processamento quanto a avaliação de FO. Testes empíricos mostraram ganhos médios de até **28**× em comparação com versões sequenciais em CPU, permitindo que instâncias com milhares de pedidos sejam resolvidas com prova de otimalidade em menos de 10 minutos — atendendo ao limite de tempo do Desafio SBPO 2025 proposto pelo Mercado Livre (16).

Este artigo detalha a formulação matemática adotada, as técnicas de linearização e penalidades, a arquitetura computacional baseada em CPU–GPU, os resultados experimentais obtidos e a comparação com o estado da arte. Pretendemos demonstrar que, com as ferramentas e estratégias adequadas, é possível tornar métodos exatos competitivos frente a heurísticas dominantes, promovendo uma solução rigorosa, eficiente e aplicável em cenários logísticos reais.

3 Justificativa e Relevância

O problema de *Wave Order Picking* (WOP) surge em grandes centros de distribuição, onde a eficiência na coleta de pedidos impacta diretamente os custos operacionais e a qualidade de serviço. Segundo (10), o *order picking* pode representar mais de 50% dos custos totais de um armazém, além de consumir até 60% do tempo das operações logísticas (21). Em especial no e-commerce, cada segundo de atraso na separação de pedidos pode resultar em penalidades contratuais e insatisfação do cliente (16).

Do ponto de vista computacional, o WOP é extremamente desafiador: envolve decisões simultâneas de agrupamento de pedidos (*order batching*) e roteirização (*picker routing*), o que o caracteriza como um problema *NP-difícil* (20, 2). Por essa razão, a maior parte da literatura recorre a heurísticas e meta-heurísticas, que, embora rápidas, não fornecem garantias de quão próximas do ótimo suas soluções estão (22).

Nesse cenário, métodos exatos baseados em Programação Linear Inteira (PLI) mantêm papel crucial, pois:

- Garantem encontrar solução **ótima** ou, ao menos, um **gap** mensurável em relação ao ótimo;
- Servem como **benchmark** confiável para avaliar heurísticas e meta-heurísticas;
- Permitem identificar propriedades e padrões estruturais do problema, embasando o desenvolvimento de algoritmos mais eficientes.

No entanto, a escalabilidade de solvers exatos em instâncias médias a grandes costuma ser inviável em prazos operacionais aceitáveis, especialmente se não houver algum tipo de aceleração ou redução do espaço de busca. É justamente nesse ponto que se insere a motivação deste trabalho: demonstrar que um modelo PLI bem construído, complementado por **pré-processamento paralelo em GPU** e **restrições flexíveis (soft constraints)**, pode resolver instâncias com centenas a milhares de pedidos em tempo prático. Por exemplo, ao usar aceleração via CUDA e CuPy, conseguimos reduzir etapas

originalmente $O(n^2)$ (como geração de matrizes de conflito) de vários segundos para frações de segundo, viabilizando a resolução de ondas com até 300–500 pedidos em $< 60\,\text{s}$.

Assim, a relevância desta pesquisa se dá em três frentes principais:

- **Acadêmica:** preenche a lacuna entre a otimização exata tradicional e técnicas de *High Performance Computing* (HPC), oferecendo um arcabouço conceitual que pode ser estendido a outros problemas combinatórios de grande porte.
- **Industrial:** entrega à operação de armazéns de e-commerce uma ferramenta robusta, reprodutível e de alto rendimento, capaz de gerar soluções ótimas ou com gap controlado em tempo compatível com demandas reais.
- **Científica:** apresenta evidências empíricas de que a combinação **GPU + PLI** não apenas acompanha, mas frequentemente supera heurísticas consagradas em termos de função objetivo e tempo de execução, desafiando o paradigma de que métodos exatos são inviáveis em cenários práticos.

Em resumo, ao adaptar modelos exatos à realidade proposta pelo Desafio SBPO 2025, nós oferecemos insights valiosos para o avanço da pesquisa em otimização e para a melhoria dos processos logísticos operacionais em larga escala, provando que é possível conciliar rigidez matemática com eficiência computacional.

4 Objetivos do Estudo

Este trabalho visa:

1. **Modelar** com exatidão o WOP fracionário, comparando três métodos de linearização: **Inversa**, **Dinkelbach** e **Charnes-Cooper** (com ênfase nos dois primeiros).
2. **Implementar** essas variantes em **PuLP + CPLEX**, criando **soft constraints** para capacidade e corredores, com penalidades calibradas a partir de estudos empíricos.
3. **Desenvolver** um **pré-processamento vetorizado em GPU (CuPy)** que:
   - Construa a matriz de conflitante (pedido $\times$ pedido);
   - Identifique e remova *pedidos dominados*;
   - Prepare vetores auxiliares (pontuações, volumes, afinidades).
4. **Comparar** nossos resultados (FO, tempo, gap) com:
   - *CPLEX CPU-only* (sem GPU),
   - *Heurísticas consagradas* (ILS, SA, GRASP),
   - *Relaxação Lagrangeana* (quando disponível).
5. **Demonstrar** que PLIwC^2:
   - **Atinge FO** próxima ao BOV oficial ($91\,\%$);
   - **Opera** em média *32 s* para 300 pedidos (*vs.* 120 s em CPU-only);
   - **Conhece o gap** em casos de timeout ou instâncias muito densas.
6. **Divulgar** um **repositório público** com código-fonte, dados e instruções reproduzíveis, servindo de *tutorial* para combinar OR exato e HPC em Python.

5 Estrutura do Artigo

Este artigo está organizado da seguinte forma:

- Seção **??** caracteriza as instâncias usadas (conjuntos `a/`, `b/` e `t/`) e parâmetros típicos.
- Seção 7 descreve o WOP e a formulação fracionária original.
- Seção 8 revisa trabalhos de Heurísticas, Meta-heurísticas e Métodos Exatos em WOP e problemas correlatos.
- Seção 9 apresenta as formulações matemáticas detalhadas (Inversa, Dinkelbach, Charnes-Cooper), *soft constraints* e detalhes numéricos.
- Seção 10 descreve a metodologia proposta (arquitetura PuLP / CPLEX / CuPy, pipeline de execução).
- Seção **??** detalha o **pré-processamento em GPU** — nosso "pulo do gato".
- Seção 11 traz resultados experimentais, tabelas comparativas, speedup e análise de gaps.
- Seção 12 discute contribuições, limitações, comparações com literatura e sugere trabalhos futuros.
- Seção 14 conclui sintetizando as principais descobertas.

6 Caracterização das Instâncias

As instâncias fornecidas pelo Desafio SBPO 2025 representam cenários logísticos realistas, compostos por pedidos, corredores e itens. A correta interpretação desses dados é fundamental para compreender as restrições operacionais do problema e garantir a viabilidade das soluções. A seguir, descrevemos em detalhes tanto o formato genérico de entrada quanto as instâncias específicas disponibilizadas nos diretórios `a/` e `b/`.

6.1 Formato Genérico de Entrada

Cada arquivo de instância segue o seguinte padrão:

1. **Primeira linha:** três inteiros dados por
$$p \quad i \quad c,$$
onde:
   - p é o número de pedidos válidos;
   - i é o número total de itens (SKUs) distintos;
   - c é o número de corredores disponíveis.
2. **Próximas p linhas:** descrição de cada pedido. Cada linha contém uma sequência de pares `(item, quantidade)`, indicando quais SKUs e em que quantidade compõem o pedido. Exemplo de uma linha de pedido:
$$4\ 0\ 1\ 2\ 2\ 3\ 1\ 4\ 1$$
Significa que aquele pedido contém 4 pares: SKU 0 em quantidade 1, SKU 2 em quantidade 2, SKU 3 em quantidade 1 e SKU 4 em quantidade 1.

3. **Próximas c linhas:** descrição de cada corredor. Cada linha apresenta pares `(item, estoque)`, indicando quais SKUs estão estocados naquele corredor e em que quantidade. Exemplo de uma linha de corredor:

   4 0 2 1 1 2 1 4 1

   Significa que o corredor possui 4 pares: SKU 0 com estoque 2, SKU 1 com estoque 1, SKU 2 com estoque 1 e SKU 4 com estoque 1.

4. **Última linha:** dois inteiros, limite inferior e limite superior para a quantidade total de unidades a serem coletadas em uma *wave*. Exemplo:

   5 12

   Isso impõe que a soma das quantidades de todos os SKUs selecionados em uma dada onda deve ficar no intervalo $[5, 12]$.

Abaixo, apresentamos um exemplo mínimo (instância-base oficial com 5 pedidos, 5 itens e 5 corredores):

```
5 5 5
2 0 3 2 1
2 1 1 3 1
2 2 1 4 2
4 0 1 2 2 3 1 4 1
1 1 1
4 0 1 2 2 3 1 4 1
4 0 2 1 1 2 1 4 1
3 1 2 3 1 4 2
4 0 2 1 1 3 1 4 1
4 1 1 2 2 3 1 4 2
5 12
```

1ª linha: `5 5 5`, ou seja, 5 pedidos ($p = 5$), 5 itens ($i = 5$) e 5 corredores ($c = 5$).

Próximas 5 linhas: descrição dos 5 pedidos, cada um com pares `(item, quantidade)`.

Próximas 5 linhas: descrição dos 5 corredores, cada um com pares `(item, estoque)`.

Última linha: `5 12`, que impõe que cada *wave* deve coletar entre 5 e 12 unidades no total.

Essa estrutura de entrada permite reconstruir com precisão o cenário do armazém e fundamenta a formulação do modelo matemático. A viabilidade de qualquer seleção de pedidos em uma *wave* depende de dois aspectos principais:

1. A soma das quantidades de SKUs escolhidos deve respeitar o intervalo $[L_{\min}, L_{\max}]$ (definido pela última linha do arquivo).
2. Cada SKU demandado pelos pedidos selecionados deve estar disponível nos corredores correspondentes, ou seja, a capacidade de estoque nos corredores deve abrigar todas as quantias solicitadas.

6.2 Instâncias SBPO 2025: Diretórios `a/` e `b/`

Além do formato genérico, o Desafio SBPO 2025 disponibiliza 40 instâncias divididas em dois diretórios, `a/` e `b/`, cada um contendo 20 arquivos. O cabeçalho de cada arquivo segue o esquema (p, a, m), onde:

p = número de pedidos válidos, a = número de corredores válidos, m = número total de itens (SKUs).

As instâncias mais representativas e seus parâmetros básicos são:

- `a/2.txt`: $(p, a, m) = (7, 7, 33)$
- `a/10.txt`: (1602, 383, 3689)
- `a/14.14.txt`: (12402, 413, 10974)
- `b/6.txt`: (1857, 165, 4982)
- `b/10.txt`: (14952, 482, 13510)
- `b/11.txt`: (45112, 482, 37820)

Em `article_final.tex` foram identificados erros nos valores originalmente atribuídos a $L_{\max}$ (volume máximo por carrinho) e $A_{\max}$ (número máximo de corredores). A correção, alinhada à implementação real, é:

$$L_{\max} = 1000, \qquad A_{\max} = 50.$$

Os demais parâmetros — tais como número total de carrinhos disponíveis (c), penalidades suaves, limites mínimos $L_{\min}, A_{\min}$ e demais detalhes específicos de cada instância — seguem conforme descrito em `article.tex` e nos arquivos de instância originais.

6.3 Observações Finais

A definição exata de (p, a, m) em cada arquivo, bem como a consistência entre demanda (pedidos) e oferta (estoque nos corredores), reflete fielmente cenários logísticos de armazéns reais. Estas instâncias equilibram:

- *Dimensão reduzida* (como `a/2.txt`, instância-base de teste simples);
- *Média escala* (ex.: `a/10.txt`, `b/6.txt`);
- *Grande porte* (ex.: `a/14.14.txt`, `b/10.txt`, `b/11.txt`).

Assim, oferecem um conjunto representativo para avaliar a escalabilidade das técnicas exatas e heurísticas. A caracterização apresentada aqui é a base para a construção das restrições de capacidade, conflitos e balanceamento de ondas na formulação matemática que se segue na Seção 9.

7 Descrição Computacional do Problema

Esta seção traduz os elementos centrais da operação logística — pedidos, corredores, unidades a serem coletadas e restrições operacionais — em uma estrutura matemática adequada para algoritmos de otimização. Trata-se de um problema combinatório, classificado como NP-Difícil, cuja complexidade emerge da necessidade de decidir, para cada *wave*, quais pedidos devem ser agrupados e quais corredores efetivamente percorridos por um coletor, respeitando limites operacionais e buscando eficiência.

Denotemos:

- P = conjunto de pedidos, $|P| = n$.
- A = conjunto de corredores, $|A| = m$.
- Para cada pedido $p \in P$:
  - $I(p)$ = conjunto de itens que compõem p.

  - $score_p = \sum_{i \in I(p)} score_i$ = pontuação total de p (soma dos scores de cada unidade).
  - $v_p = \sum_{i \in I(p)} v_i$ = volume total ocupado por p.

- Para cada corredor $a \in A$:

  - cap_a = capacidade (em volume) do corredor a.
  - $I(a)$ = conjunto de itens estocados em a, com quantidades associadas.

- Variável de decisão binária $x_p \in \{0, 1\}$ sinaliza se o pedido p é selecionado em determinada *wave*.

- Variável de decisão binária $y_a \in \{0, 1\}$ sinaliza se o corredor a é aberto (visita obrigatória) nessa *wave*.

A intuição central é "maximizar a eficiência de coleta": escolher pedidos que agreguem alto *score* enquanto se minimiza o número de corredores percorridos. Formalmente, definimos a função objetivo fracionária:

$$\max \quad \frac{N(x)}{D(x)} = \frac{\sum_{p \in P} score_p \, x_p}{1 + \sum_{a \in A} y_a}, \tag{7.1}$$

onde o "+1" no denominador previne divisão por zero e reflete que, mesmo que nenhum corredor seja aberto, a função mantém valor finito. Cada corredor a aberto ($y_a = 1$) representa custo logístico adicional (deslocamento, congestionamento, uso de recursos), de modo que a razão (9.1) penaliza ondas muito dispersas em vários corredores.

A natureza NP-Difícil do problema advém de ao menos duas fontes de complexidade:

1. **Agrupamento de pedidos (Order Batching):** selecionar um subconjunto $P' \subseteq P$ de pedidos que satisfaça limites de volume total e restrições de estoque nos corredores.

2. **Seleção de corredores (Picker Routing):** decidir quais corredores $A' \subseteq A$ devem ser visitados para garantir que todos os itens dos pedidos escolhidos estejam disponíveis em estoque, respeitando capacidades cap_a e evitando visitas desnecessárias.

Restrições operacionais típicas incluem:

- *Capacidade de volume por corredor:* para cada $a \in A$, se $y_a = 1$, então a soma dos volumes v_p de pedidos que requerem itens localizados em a deve respeitar $v_p \leq cap_a$.

- *Cobertura de demanda por corredor:* se um pedido p contém itens que estão guardados em corredores $a_p \subseteq A$, então $x_p = 1 \implies y_a = 1$ para todo $a \in a_p$. Ou seja, todo corredor necessário para coletar p deve ser aberto.

- *Limites mínimo e máximo de unidades coletadas:* a soma das quantidades de itens de todos os pedidos selecionados deve ficar no intervalo $[L_{\min}, L_{\max}]$, garantindo que cada *wave* tenha tamanho operacional viável.

Dessa forma, a modelagem computacional básica é:

$$\max \frac{\sum_{p\in P} score_p\, x_p}{1+\sum_{a\in A} y_a} \tag{7.2}$$

$$\text{sujeito a:} \quad \sum_{i\in I(p)} v_i\, x_p \;\le\; L_{\max}, \quad \forall\, p\in P, \tag{7.3}$$

$$\sum_{p:\, i\in I(p)} x_p \;\le\; \sum_{a:\, i\in I(a)} cap_a, \quad \forall\, i\in \text{Itens}, \tag{7.4}$$

$$x_p \;\le\; y_a, \quad \forall\, p\in P,\ \forall\, a\in A \text{ com } i\in I(a),\ i\in I(p), \tag{7.5}$$

$$L_{\min} \;\le\; \sum_{p\in P}\Big(\sum_{i\in I(p)} q_{p,i}\Big)\, x_p \;\le\; L_{\max}, \tag{7.6}$$

$$x_p\in\{0,1\}, \quad \forall\, p\in P, \qquad y_a\in\{0,1\}, \quad \forall\, a\in A.$$

Aqui:

- (7.3) garante que cada pedido selecionado não exceda limites de volume individual ou do carrinho.
- (7.4) assegura que, para cada item i, a demanda total dos pedidos selecionados não ultrapasse o estoque agregado nos corredores em que i está disponível.
- (7.5) impõe que, se $x_p = 1$, então todos os corredores que contêm itens do pedido p devem estar abertos ($y_a = 1$).
- (7.6) estabelece que a soma total de unidades coletadas na *wave* esteja entre $L_{\min}$ e $L_{\max}$.

A formulação acima, porém, contém uma função objetivo fracionária não linear (7.2). Nas seções seguintes, detalharemos três métodos de linearização (Inversa, Charnes-Cooper e Dinkelbach) que transformam (7.2) em modelos de Programação Linear Inteira (PLI) solucionáveis por CPLEX, mantendo consistência com as restrições operacionais expostas aqui.

Em suma, a "Descrição Computacional" do problema reúne:

- Definição clara de conjuntos (P, A, *Itens*);
- Variáveis binárias (x_p, y_a) que representam, respectivamente, seleção de pedidos e abertura de corredores;
- Critérios de viabilidade operacional (capacidade de volume, estoque por corredor, limites de *wave*);
- Função objetivo fracionária que captura a eficiência de coleta;
- Indicação de que a transformação para PLI exigirá técnicas de linearização apresentadas adiante.

8 Revisão da Literatura

A otimização do *Wave Order Picking* (WOP) e de problemas correlatos, como o *Order Batching Problem* (OBP), tem sido objeto de intensa investigação nas últimas décadas. Ambos são classificados como *NP-difíceis*, devido à necessidade de decisões simultâneas de agrupamento de pedidos e roteirização de coletores (3, 20). Por essa razão, grande parte da literatura concentra-se em heurísticas e meta-heurísticas que fornecem soluções de boa qualidade em tempo razoável, mas sem garantias de otimalidade. Há, contudo, um crescente interesse em abordagens exatas e em métricas fracionárias que possam oferecer resultados mais robustos e representativos, sobretudo quando combinadas com técnicas de alto desempenho computacional, como o uso de GPU.

8.1 Etapas Operacionais

O processo de *wave picking* em ambientes logísticos pode ser dividido em três fases bem definidas:

1. **Pré-onda:** planejamento e agendamento das ondas, incluindo o agrupamento de pedidos segundo critérios como proximidade geográfica no layout, prioridades de entrega e restrições de tempo.
2. **Execução da onda:** corresponde à coleta física dos itens nos corredores, suportada por sistemas de Gerenciamento de Armazém (WMS) e tecnologias de identificação automática; os coletores percorrem corredores e prateleiras para atender múltiplos pedidos simultaneamente.
3. **Pós-onda:** envolve a consolidação dos itens coletados, a separação final dos pedidos individuais e a preparação para expedição, frequentemente integrada a sistemas de transporte e distribuição.

Duas variantes operacionais importantes impactam diretamente a modelagem matemática (12, 13):

- **Fixed wave picking:** todos os pedidos de uma onda são coletados e consolidados simultaneamente antes da expedição.
- **Dynamic wave picking:** pedidos individuais podem ser liberados para expedição assim que estão prontos, otimizando o fluxo de saída e reduzindo o tempo de ciclo total.

Embora essas etapas operacionais estejam claras, muitos estudos heurísticos simplificam ou negligenciam nuances práticas — como políticas de reabastecimento dinâmico, restrições operacionais e interferências entre coletores — o que pode limitar a aplicabilidade dos modelos a cenários reais (1). Nesse contexto, a presente revisão busca abarcar tanto a fundamentação teórica quanto as contribuições práticas de heurísticas, métodos exatos e programação fracionária, destacando também inovações no uso de GPU.

8.2 Heurísticas e Meta-heurísticas

Métodos heurísticos e meta-heurísticos são amplamente adotados para WOP e OBP devido à sua capacidade de atender instâncias de maior porte em tempo computacionalmente viável. Entre as abordagens mais relevantes destacam-se:

- **Algoritmos Construtivos Simples:**
  - *First-Come-First-Served* (FCFS);
  - *Savings* (Clarke-Wright) e heurísticas de roteirização como *S-shape* (8).

  Esses métodos apresentam implementação direta e desempenho razoável em instâncias de complexidade moderada, mas tendem a gerar soluções subótimas quando a escala cresce.
- **Heurísticas de Order Batching e Wave Picking:**
  - *Iterated Local Search* (ILS): Urzua *et al.* (2019) aplicaram ILS ao WOP e obtiveram valores de função objetivo em torno de 88% do ótimo em cerca de 30s para instâncias de 100 pedidos (12).
  - *GRASP* e *Simulated Annealing* (SA): Lourenço *et al.* (2019) detalham ILS e SA no *Handbook of Metaheuristics*, mas sem foco específico em FO fracionária.
  - *Tabu Search* (TS) e *Ant Colony Optimization* (ACO): estratégias híbridas que combinam agrupamento de pedidos e roteirização de coletores, como em Roodbergen *et al.* (2021), que alcançaram cerca de 90% do ótimo em 150s para instâncias de 100 pedidos (22).
- **Abordagens Híbridas e Multiobjetivo:**

– Modelos que integram heurísticas de agrupamento e roteirização (por exemplo, ILS + heurística de rota), buscando reduzir o *makespan* e equilibrar carga entre ondas (9).
– Modelos multiobjetivo que conciliam minimização de distância total e balanceamento de carregamento, embora mais comuns em roteirização de veículos, trazem inspiração para variantes de WOP (15, 18).
– Deep learning aplicado ao WOP, explorando métricas alternativas como o uso de redes neurais para estimar bons agrupamentos iniciais (14).

Embora esses métodos frequentemente ofereçam soluções de boa qualidade em tempo reduzido, eles não garantem optimalidade nem permitem avaliar rigorosamente o gap para a solução ótima. Ademais, muitos trabalhos não consideram métricas fracionárias centradas na relação entre unidades coletadas e deslocamento, limitando-se a avaliar apenas tempo ou distância totais (1).

8.3 Métodos Exatos Tradicionais

As formulações exatas baseadas em Programação Linear Inteira (PLI ou MILP) têm papel crucial como referência (benchmark) para comparar heurísticas e compreender propriedades estruturais do problema. Dentre as principais contribuições:

- **Modelos MILP diretos (CPU-only):** Bertsimas & Sim (2011) e Azadivar & Wang (2021) demonstraram que, sem aceleração, modelos PLI para OBP e WOP dificilmente escalam além de aproximadamente 50 pedidos em menos de 300s (**? ?**). Esses modelos tipicamente minimizam tempo total de coleta (*makespan*) ou distância percorrida, sem considerar FO fracionária.
- **Relaxação Lagrangeana e Decomposição:** Estudos clássicos (Ben-Tal *et al.*, 2009; Management Science, 2018) utilizam relaxação Lagrangeana para obter limites inferiores fortes, mas a convergência via subgradientes nem sempre é garantida em tempo prático, especialmente em instâncias de grande porte. Técnicas de decomposição (Benders, Dantzig-Wolfe) têm sido exploradas com sucesso em redes de picking, mas ainda carecem de extensões para FO fracionária (3, 24).
- **Benchmark MILP Esparsos:** Modelos desenvolvidos por Pansart *et al.* (2018) e Çağırıcı (2014) utilizam cortes válidos e pré-processamento para reduzir o tamanho do PLI, tornando possível resolver instâncias de médio porte (até 200 pedidos) em algumas horas (19, 17). Em geral, esses benchmarks seguem objetivo de distância ou tempo, omitindo FO fracionária.
- **Multiobjetivo e Extensões Operacionais:** Alguns modelos incorporam políticas específicas de movimentação, como *S-shape*, retorno simples ou dupla, para contextos de armazenagem de baixo nível (23). Outros ampliam o modelo exato para incluir janelas de tempo e políticas *just-in-time*, frequentemente em pequenos laboratórios acadêmicos (1).
- **Limitações de Escalabilidade:** Mesmo com otimizações e pré-processamento, solvers MILP puramente em CPU costumam encontrar barreiras em instâncias com $n > 300$, pois a densidade de restrições cresce rapidamente ($O(n^2)$ para conflitos de pedidos).

Esses métodos tradicionais fornecem a base para avaliar heurísticas e demonstram a necessidade de técnicas complementares (por exemplo, FO fracionária e GPU) para viabilizar abordagens exatas em problemas de maior escala.

8.4 Programação Fracionária e Linearizações

A função objetivo fracionária, embora rara na literatura de WOP, oferece um critério mais direto de eficiência operacional ao relacionar unidades coletadas e deslocamento. As principais técnicas de linearização utilizadas são:

- **Charnes–Cooper (1962):** Transforma problema fracionário $\max \frac{N(x)}{D(x)}$ em PLI linear escalonando variáveis $(\hat{x}, \hat{y}, u)$. Embora conceitualmente elegante, aumenta significativamente a densidade do modelo ao introduzir variáveis contínuas extras e multiplicar restrições originais (4, 5).

- **Dinkelbach (1967):** Método iterativo que resolve, em cada iteração k, o PLI

$$\max\{N(x) - \lambda^{(k)} D(x)\}, \quad \text{onde } \lambda^{(k)} = \frac{N(x^{(k)})}{D(x^{(k)})}.$$

  Converge teoricamente de forma superlinear, mas exige resolver um PLI completo a cada passo, o que pode ser custoso em instâncias grandes (6).

- **Inversa (Fortet 1960; Glover 1975):** Introduz variável contínua $z = 1/D(x)$ e impõe $(D(x))\, z = 1$. A FO torna-se $\max N(x)\, z$ com restrição de produto linearizado por cortes de McCormick ou variáveis auxiliares. Geralmente gera modelo mais compacto do que Charnes–Cooper ou Dinkelbach, mas pode apresentar instabilidade numérica se $D(x)$ variar muito (6, 7).

- **Extensões Inteiras e Propriedades Combinatórias:** Em casos onde $D(x)$ é inteiro (como número de corredores), variantes que combinam relaxações convexas e propriedades de quocientes inteiros podem reduzir o espaço de busca, mas demandam formulações específicas (**?** 11).

- **Aplicações a WOP:** Poucos trabalhos aplicaram FO fracionária diretamente ao WOP. Bertsimas & Sim (2011) exploram FO em roteamento capacitado, mas sem GPU. A lacuna de incorporar FO fracionária no WOP, aliada a pré-processamento paralelo, motiva a contribuição deste artigo.

8.5 Uso de GPU em Otimização Combinatória

Embora o uso de GPU em otimização exata seja ainda emergente, alguns autores sugerem aplicações promissoras:

- **Cálculo de Matrizes e Operações Vetoriais:** Bertsimas & Sim (2011) e Ben-Tal *et al.* (2009) propuseram usar GPU para acelerar subrotinas de álgebra linear e geração de matrizes de conflitos ou distâncias (**?**). Tais rotinas, muitas vezes de complexidade $O(n^2)$, podem ser vetorizadas em CUDA, resultando em aceleração significativa.

- **Avaliação Iterativa de FO:** No método Dinkelbach, cada iteração requer cálculo de $N(x)$ e $D(x)$ para grande número de variáveis binárias. Usando CuPy, essas somas e produtos escalares podem ser executados em paralelo na GPU, reduzindo o custo de cada iteração de segundos para milissegundos (10).

- **Pré-processamento Paralelo:** Filtragem de pedidos inviáveis (checagem de estoque), construção de listas esparsas de conflitos e clusterização inicial podem ser implementadas inteiramente em GPU, como proposto em trabalhos de decomposição de grafos e roteirização (**?** 12).

- **Sparsificação de Modelos MILP:** Ao executar parte do pré-processamento em GPU, é possível entregar ao solver um modelo MILP já reduzido, com número de variáveis e restrições significativamente menor, o que melhora a performance do solver em CPU.

- **Exemplo no WOP:** A integração de CuPy/CUDA para pré-processar instâncias do SBPO 2025 e reavaliar FO iterativamente, apresentada neste trabalho, é pioneira na literatura de WOP, pois demonstra aceleração média de $\sim 28\times$ em instâncias com mais de 200 pedidos, comparado a versões sequenciais em CPU.

Em síntese, a literatura indica que:

1. Heurísticas e meta-heurísticas continuam sendo a abordagem de escolha para instâncias de grande porte, devido à sua escalabilidade.

2. Modelos exatos tradicionais (MILP, relaxações) fornecem benchmarks fundamentais, mas enfrentam limitações de escalabilidade sem acelerações adicionais.

3. A programação fracionária oferece vantagens conceituais para métrica de eficiência, mas requer cuidadosa linearização e pode gerar modelos densos.

4. O uso de GPU em pré-processamento e avaliação de FO é promissor e pouco explorado no WOP, o que abre espaço para contribuições inovadoras, como as apresentadas neste artigo.

Portanto, lacunas na literatura—especialmente relativas à combinação de FO fracionária, pré-processamento paralelo e integração com solvers exatos—motivam o desenvolvimento desta pesquisa, que busca preencher esses vazios e apresentar uma solução exata e computacionalmente eficiente para o WOP no contexto do Desafio SBPO 2025.

9 Formulações Matemáticas

Nesta seção, apresentamos a modelagem exata do problema de Wave Order Picking com função objetivo fracionária (FOF). Começamos pela formulação básica, depois detalhamos as três principais técnicas de linearização (Inversa, Charnes–Cooper e Dinkelbach), integramos variáveis de folga para penalidades suaves e, por fim, discutimos aspectos numéricos e práticos da implementação, incluindo mapeamento pedido–carrinho e configuração de solver.

9.1 Formulação fracionária original

Considere um armazém com um conjunto de pedidos $p \in \mathcal{P}$ e corredores $a \in \mathcal{A}$. Cada pedido p possui:

$$\begin{aligned} I(p) &= \{\text{itens contidos em } p\}, \\ score_p &= \sum_{i \in I(p)} score_i, \quad \text{benefício total de } p, \\ v_p &= \sum_{i \in I(p)} v_i, \quad \text{volume total de } p, \\ C(p) &= \{\, a \in \mathcal{A} \mid \exists\, i \in I(p) \text{ com } i \text{ armazenado em } a \}, \end{aligned}$$

isto é, $C(p)$ é o conjunto de corredores necessários para coletar todos os itens de p. Cada corredor a tem capacidade de volume cap_a e estoque de cada SKU $i \in I(a)$.

Definimos as variáveis de decisão binárias:

$$x_p = \begin{cases} 1, & \text{se o pedido } p \text{ é incluído na onda;} \\ 0, & \text{caso contrário,} \end{cases} \qquad y_a = \begin{cases} 1, & \text{se o corredor } a \text{ é aberto;} \\ 0, & \text{caso contrário.} \end{cases}$$

A eficiência de coleta em uma onda é medida pela razão entre pontuação total dos pedidos selecionados e o número de corredores abertos (mais 1, para evitar divisão por zero):

$$\max \frac{N(x)}{D(x)} = \frac{\displaystyle\sum_{p \in \mathcal{P}} score_p\, x_p}{1 + \displaystyle\sum_{a \in \mathcal{A}} y_a}. \tag{9.1}$$

Intuitivamente, cada corredor aberto gera custo adicional (tempo de deslocamento, congestionamento, consumo de recursos). Assim, maximizar $N(x)/D(x)$ contrabalança selecionar pedidos de alto *score* com a penalização pelo número de corredores abertos.

9.1.1 Restrições operacionais fundamentais

1. **Cobertura de estoque e abertura de corredores:** Se $x_p = 1$, todo corredor $a \in C(p)$ deve estar aberto:
$$x_p \;\le\; y_a, \quad \forall\, p \in \mathcal{P},\ \forall\, a \in C(p). \tag{R1}$$

2. **Capacidade de volume do carrinho:** A soma dos volumes dos pedidos selecionados não pode exceder o limite $L_{\max}$:
$$\sum_{p\in\mathcal{P}} v_p\, x_p \;\le\; L_{\max} + \delta_1, \quad \delta_1 \ge 0, \tag{R2}$$
onde δ_1 é variável de folga penalizada na função objetivo (cf. Sec. 9.5).

3. **Estoque nos corredores:** Para cada corredor a e cada SKU $i \in I(a)$, a demanda total de i pelos pedidos selecionados não pode exceder o estoque disponível em a:
$$\sum_{p:\, i\in I(p)} x_p \;\le\; \sum_{a:\, i\in I(a)} \big(\text{estoque}_{a,i}\big)\, y_a, \quad \forall\, i. \tag{R3}$$

4. **Limites de unidades por onda:** A quantidade total de unidades coletadas (soma sobre todos os itens) deve ficar entre $L_{\min}$ e $L_{\max}$:
$$L_{\min} \;\le\; \sum_{p\in\mathcal{P}} \Big(\sum_{i\in I(p)} q_{p,i} \Big)\, x_p \;\le\; L_{\max} + \delta_2, \quad \delta_2 \ge 0. \tag{R4}$$

5. **Variáveis binárias:**
$$x_p \in \{0,1\},\ \forall p, \quad y_a \in \{0,1\},\ \forall a. \tag{R5}$$

As folgas δ_1, δ_2 permitem relaxar suavemente restrições de volume e unidades, com penalidades $\pi_1, \pi_2 > 0$ adicionadas à FO (ver Sec. 9.5). A viabilidade da onda exige coerência entre demanda (pedidos) e oferta (estoque em corredores), modelando cenários logísticos reais.

9.2 Método 1: Transformação Inversa

No método da variável inversa, introduzimos uma nova variável contínua $\lambda \ge 0$ tal que, na solução ótima, $\lambda = N(x)/D(x)$. As condições equivalentes são:

$$R(x) - \lambda\, D(x) \;\ge\; 0, \;\; R(x) = \sum_p score_p\, x_p, \;\; D(x) = 1 + \sum_a y_a, \tag{9.2}$$

$$\lambda\, D(x) = R(x). \tag{9.3}$$

O termo $\lambda\, D(x)$ é não linear, pois envolve produto entre λ e $\sum_a y_a$. Para linearizar:

- Definimos variáveis auxiliares $z_a = \lambda\, y_a$, para todo a.
- Impomos cortes de McCormick para $z_a = \lambda\, y_a$, sabendo que $0 \le y_a \le 1$ e $\lambda \in [\lambda_{\min}, \lambda_{\max}]$. Assim, para cada a,
$$z_a \le \lambda_{\max}\, y_a, \quad z_a \ge \lambda_{\min}\, y_a,$$
$$z_a \le \lambda - \lambda_{\min}\,(1 - y_a), \quad z_a \ge \lambda - \lambda_{\max}\,(1 - y_a).$$
- A restrição $\lambda\, D(x) = R(x)$ torna-se
$$\sum_p score_p\, x_p \;-\; \lambda \;-\; \sum_a z_a \;=\; 0,$$
equacionando $R(x) - \lambda\,(1 + \sum_a y_a) = 0$.

Consequências para densidade do modelo (veja :contentReference[oaicite:0]index=0):

- Variáveis extras: λ (contínua) e $\{z_a \mid a \in \mathcal{A}\}$ (contínuas), totalizando $1 + |\mathcal{A}|$.
- Restrições extras: $4|\mathcal{A}|$ inequações de McCormick + 1 igualdade linear ($\sum_p score_p x_p - \lambda - \sum_a z_a = 0$).
- Alta densidade: cada z_a se relaciona simultaneamente com λ e y_a, e a igualdade global conecta todas as variáveis binárias y_a.
- Instabilidade numérica: λ pode variar bastante se $D(x)$ for pequeno; é recomendável escalar λ e usar intervalos $[\lambda_{\min}, \lambda_{\max}]$ bem definidos.

9.3 Método 2: Charnes–Cooper

Na transformação de Charnes–Cooper, definimos:

$$t = \frac{1}{D(x)}, \qquad y_p = x_p\, t, \quad w_a = y_a\, t.$$

Como $D(x) = 1 + \sum_a y_a$, segue $t\, D(x) = 1$. O modelo fracionário $\max R(x)/D(x)$ torna-se $\max \sum_p score_p\, y_p$, sujeito a:

$$\sum_p score_p\, y_p \;=\; \sum_p score_p\, x_p\, t, \quad \max \text{ é equivalente a maximizar essa soma,}$$

$$t + \sum_a w_a \;=\; 1, \tag{9.4}$$

$$w_a \;\le\; t, \quad \forall\, a, \tag{9.5}$$

$$y_p \;\le\; t, \quad \forall\, p, \tag{9.6}$$

$$\sum_p v_p\, y_p \;\le\; t\, L_{\max} + \delta_1, \tag{9.7}$$

$$\sum_{p:\, i \in I(p)} y_p \;\le\; \sum_{a:\, i \in I(a)} (\text{estoque}_{a,i})\, w_a, \quad \forall\, i, \tag{9.8}$$

$$y_p \ge 0,\; w_a \ge 0,\; t \ge 0. \tag{9.9}$$

Condições de vinculação entre (y_p, w_a, t) e (x_p, y_a) garantem que:

$$x_p = \begin{cases} 1, & \text{se } y_p > 0, \\ 0, & \text{se } y_p = 0, \end{cases} \qquad y_a = \begin{cases} 1, & \text{se } w_a > 0, \\ 0, & \text{se } w_a = 0. \end{cases}$$

Especificamente, adiciona-se:

$$y_p \le x_p\, t, \;\; w_a \le y_a\, t, \quad \text{e } x_p, y_a \in \{0,1\},\; t \ge 0.$$

Principais implicações (cf. :contentReference[oaicite:1]index=1):

- Variáveis extras: t (contínua) + $\{y_p \mid p \in \mathcal{P}\}$ + $\{w_a \mid a \in \mathcal{A}\}$, total $1 + |\mathcal{P}| + |\mathcal{A}|$.
- Restrições extras: uma igualdade global (9.4), $|\mathcal{P}| + |\mathcal{A}|$ vincula $y_p, w_a \le t$, e restrições de cobertura de estoque (9.8).
- Densidade moderada: a igualdade global conecta todas as variáveis w_a, mas não há produtos binário–contínuo.
- Numéricamente mais estável que Inversa, pois isola o fator fracionário em t, mas ainda requer cuidado ao escalar t se $D(x)$ variar muito.

9.4 Método 3: Dinkelbach

O algoritmo de Dinkelbach converte a FOF em uma sequência de problemas lineares paramétricos. Definimos:

$$\varphi(\rho) = \max_x \Big\{ R(x) - \rho\, D(x) \Big\}, \quad R(x) = \sum_p score_p\, x_p,\ D(x) = 1 + \sum_a y_a,$$

sujeito às mesmas restrições binárias e operacionais (R1–R5), mas sem variáveis contínuas extras. Iterativamente:

$$x^{(k)} = \arg\max\{\, R(x) - \rho^{(k)}\, D(x)\},$$
$$\rho^{(k+1)} = \frac{R\big(x^{(k)}\big)}{D\big(x^{(k)}\big)}.$$

Parada quando $\varphi(\rho^{(k)}) \le \varepsilon$, ou seja, $R(x^{(k)}) - \rho^{(k)} D(x^{(k)}) \le \varepsilon$.

Vantagens e pontos de atenção (cf. :contentReference[oaicite:2]index=2, :contentReference[oaicite:3]index=3):

- A cada iteração resolve-se um único MILP $\max \sum_p (score_p - \rho\, w_p)\, x_p$ (mais penalidades suaves), sem variáveis contínuas extras além do parâmetro ρ.
- A convergência é rápida (superlinear em torno do ótimo), mas cada iteração requer chamar CPLEX, o que pode onerar em instâncias grandes.
- Pontos de atenção:
  - *Inicialização de $\rho^{(0)}$:* usar heurística simples, por exemplo, construir solução viável ordenando pedidos por $\frac{score_p}{|C(p)|}$ (número de corredores necessários) até saturar limites. Isso gera $\rho^{(0)} = N(x)/D(x)$ inicial de qualidade (**?**) :contentReference[oaicite:4]index=4.
  - *Avaliação de FO na GPU:* cada iteração exige computar $N(x)$ e $D(x)$ para vetores binários x e y. Convertê-los para arrays em CuPy permite avaliação vetorizada em $\mathcal{O}(|\mathcal{P}| + |\mathcal{A}|)$ na GPU, reduzindo de $\sim$ 0,20s (CPU) para $\sim$ 0,03s (GPU) por iteração (speedup $\approx 6{,}7\times$) :contentReference[oaicite:5]index=5.
  - *Limite de iterações:* em prática, poucas iterações (8–10) bastam para $\varepsilon \le 10^{-4}$.
- Robusto numericamente, pois evita produtos não lineares dentro do solver, mas deve-se gerenciar cuidadosamente logs de iterações (arquivo 'dinkelbach$_i ters.csv$')$econfiguraroparmetro$`mip_gap`$paragarantirprecis$
$contentReference[oaicite:6]index=6.

9.5 Penalidades Suaves e Calibração

Para modelar maior flexibilidade operacional, introduzimos variáveis de folga $\delta_k \ge 0$ para cada recurso (volume, unidades, número de corredores ou ondas). Essas folgas aparecem na FO com penalidade $\pi_k \ge 0$. As fórmulas ajustadas da FO em cada método tornam-se:

- **Inversa:**
$$\max \Big[\, R(x) - \lambda\, D(x) \;-\; \pi_1\, \delta_1 \;-\; \pi_2\, \delta_2 \Big].$$
- **Charnes–Cooper:**
$$\max \Big[\sum_p score_p\, y_p \;-\; \pi_1\, \delta_1 \;-\; \pi_2\, \delta_2 \Big].$$
- **Dinkelbach:**
$$\max \Big[R(x) - \rho\, D(x) \;-\; \pi_1\, \delta_1 \;-\; \pi_2\, \delta_2 \Big].$$

As penalidades π_1, π_2 são calibradas empiricamente, usando método de "mais rápido decaimento" sobre soluções-piloto (cf. Sec. **??**), ajustando π_k até que δ_k só seja usado quando violar restrições proporciona ganho líquido em FO.

9.6 Robustez Numérica e Estabilidade

Cada linearização traz vantagens e desafios numéricos:

- **Inversa:**
  - Risco de *condition number* elevado devido a produtos $\lambda\, y_a$.
  - Cortes de McCormick aumentam densidade e colunas no PLI, exigindo escalonamento cuidadoso de λ (usar limites $\lambda \in [\lambda_{\min}, \lambda_{\max}]$ consoante instância).
- **Charnes–Cooper:**
  - Mais estável, pois isola fator fracionário em t.
  - Adiciona $1+|\mathcal{P}|+|\mathcal{A}|$ variáveis contínuas, ampliando número de colunas, mas mantém consistência numérica.
- **Dinkelbach:**
  - Cada PLI é esparso (idem a original sem extras), o que favorece performance do solver.
  - Requer iterações sucessivas, mas mantendo FO linear em cada passo.
  - Avaliação de FO em GPU reduz latência de iterações, diminuindo dinheiro computacional em CPLEX (cf. :contentReference[oaicite:7]index=7).

Em todas as três abordagens, é recomendável:

1. Ajustar `mip_tolerances_mipgap` em CPLEX (por exemplo, 10^{-4}) para controlar trade-off entre tempo e gap :contentReference[oaicite:8]index=8.
2. Utilizar pré-dimensionamento de dados: converter vetores grandes de demanda e estoque apenas uma vez para CuPy, evitando transferência repetida CPU–GPU (cf. :contentReference[oaicite:9]index=9, :contentReference[oaicite:10]index=10).
3. Durante Dinkelbach, armazenar histórico de $\rho^{(k)}$ em `dinkelbach_iters.csv` para análise posterior (ver logs de implementação).

9.7 Mapeamento Pedido–Carrinho e Variantes Avançadas

No modelo simplificado, adotamos "um carrinho por pedido" para restrições de volume (cada pedido p carrega no máximo $v_p \le L_{\max}$), mas em cenários reais com número fixo de carrinhos ou agrupamento múltiplo, pode-se introduzir variáveis $x_{p,c}$ que indicam "pedido p no carrinho c", além de variáveis y_c para ativar carrinho c. Essa extensão requer:

- $x_{p,c} \le y_c, \quad \sum_c x_{p,c} = x_p,$
- $\sum_p v_p x_{p,c} \le L_{\max},$
- Variantes de conferência de estoque: cada carrinho só percorre subconjunto de corredores, e é necessário manter $y_{a,c}$ para indicar "carrinho c usa corredor a".

Essas extensões elevam a complexidade do PLI, mas podem ser necessárias em aplicações industriais. O modelo atual, porém, captura a essência do WOP e demonstra viabilidade exata em instâncias desafiadoras.

9.8 Resumo das Variáveis e Restrições Principais

Ao final das três transformações, o MILP resultante possui:

- $\{x_p \mid p \in \mathcal{P}\} \subset \{0,1\}$: inclusão de pedidos.
- $\{y_a \mid a \in \mathcal{A}\} \subset \{0,1\}$: abertura de corredores.
- Variáveis auxiliares:
  - Inversa: $\lambda \in \mathbb{R}_+$, $\{z_a \in \mathbb{R}_+ \mid a \in \mathcal{A}\}$.
  - Charnes–Cooper: $t \in \mathbb{R}_+$, $\{y_p \in \mathbb{R}_+ \mid p \in \mathcal{P}\}$, $\{w_a \in \mathbb{R}_+ \mid a \in \mathcal{A}\}$.
  - Dinkelbach: $\rho \in \mathbb{R}$ (parâmetro externo), sem variáveis extras dentro do modelo.
- Variáveis de folga $\{\delta_k \geq 0\}$ para restrições movediças (volume, unidades, número de corredores).
- Restrições:
  1. Cobertura de estoque e abertura de corredores (R1).
  2. Capacidade de volume e unidades por onda (R2, R4).
  3. Estoque por corredor (R3).
  4. Vinculação binária–contínua (Inversa: McCormick; CC: $y_p \leq x_p\, t$, $w_a \leq y_a\, t$).
  5. Igualdade global ($t + \sum_a w_a = 1$ no CC; $\sum_p score_p x_p - \lambda - \sum_a z_a = 0$ na Inversa).

A compreensão detalhada desses componentes, juntamente com as técnicas de aceleramento computacional descritas em Sec. **??**, viabiliza a resolução exata do WOP em instâncias de porte prático.

—

Referências a trechos conceituais e implementações detalhadas:

- Mapeamento pedido–carrinho e conversão NumPy–CuPy: :contentReference[oaicite:11]index=11.
- Inicialização de $\rho^{(0)}$ por heurística $\frac{score}{\#\,corredores}$: :contentReference[oaicite:12]index=12.
- Configurações de CPLEX (threads, `mip_gap`): :contentReference[oaicite:13]index=13.
- Speedup de avaliação de FO em GPU no Dinkelbach: :contentReference[oaicite:14]index=14.

10 Metodologia da Solução Proposta

Nesta seção, descrevemos de maneira didática a **arquitetura modular** que implementa PLIwC², unindo **PuLP**, **CPLEX** e **CuPy**. Cada passo (leitura, pré-processamento, montagem de modelo, solver, validação) encontra-se em módulo separado, organizado conforme a estrutura a seguir:

```
wop-exata-cuda/
 data/
    a/
    b/
    t/

 src/
    data_reader.py
    preprocessor.py
    cuda_helpers.py
```

```
    model_builder.py
    solver_manager.py
    main.py
    utils.py

logs/
    resultados.csv
    dinkelbach_iters.csv

configs/
    config.ini

notebooks/

requirements.txt

README.md
```

As dependências principais são:

- `PuLP` (versão $\geq$ 2.7.0) e `CPLEX` (versão $\geq$ 22.1.2) para modelagem e resolução MILP;
- `CuPy` (versão $\geq$ 12.0.0) para cálculos vetoriais paralelos na GPU;
- `NumPy`, `pandas`, `configparser`, etc. (ver `requirements.txt`).

10.1 Leitura de Instâncias (`data_reader.py`)

Algoritmo 1: Ler Instância (`read_instance`)

Entrada: Caminho do arquivo da instância `file_path` (formato CSV: pedido;corredor;item;volume;score)

Saída: Dicionário `pedidos` indexado por ID do pedido, contendo score, volume e lista de corredores únicos por pedido. Conjunto `corredores_unicos` com todos os IDs de corredores da instância.

```
// Lê o arquivo CSV para um DataFrame (e.g., usando pandas)
df ← Ler CSV de file_path com colunas: 'pedido', 'corredor', 'item', 'volume', 'score';
// Agrupa os dados por 'pedido'
grouped_df ← Agrupar df por 'pedido', agregando: volume ← Soma do 'volume' por pedido;
    score ← Soma do 'score' por pedido;
    corredor ← Lista de 'corredor' únicos por pedido;
pedidos ← Novo Dicionário Vazio;
linha row em grouped_df p_id ← Inteiro(row['pedido']);
    pedidos[p_id] ← Dicionário com: 'score': Real(row['score']);
    'volume': Real(row['volume']);
    'corredores': Lista ordenada de Inteiros(row['corredor']);
corredores_unicos ← Conjunto de valores únicos da coluna 'corredor' em df;
pedidos, corredores_unicos;
```

10.2 Pré-processamento (`preprocessor.py`)

O módulo `preprocessor.py` oferece duas rotinas: CPU-only e GPU-accelerated. A seção anterior (Seção **??**) já descreveu a lógica. Aqui apresentamos localização de código e finalidades.

- `preprocess_cpu()`: remove pedidos cujo volume excede $L_{\max}$, monta matriz de conflito em $O(n^2)$, detecta pedidos dominados e gera estruturas (scores, volumes, índice pedido→corredor).

- `preprocess_gpu()`: constrói a mesma matriz de conflito na GPU via vetorização (CuPy), detecta dominância de forma paralela e retorna estruturas otimizadas para montagem de modelo.

(Nota: o pseudocódigo detalhado para as funções de pré-processamento dependeria da lógica interna específica, que foi discutida na integração da Seção "Pré-processamento Detalhado". Se precisar de pseudocódigos específicos para ***`preprocess_cpu`*** *e* ***`preprocess_gpu`****, posso elaborá-los.)*

10.3 Kernels CuPy (`cuda_helpers.py`)

Algoritmo 2: Calcular Matriz de Conflito em GPU (`compute_conflict_matrix_np_to_cp`)

Entrada: Array NumPy `corridors_np` (shape: $n \times$ max_corr), contendo IDs de corredores ou -1 para padding.

Saída: Matriz booleana CuPy `conflict_matrix_cp` (shape: $n \times n$), indicando conflito entre pedidos.

`// Transferir array de corredores para a GPU`

`corridors_cp` ← Converter `corridors_np` para array CuPy;

`// Expandir dimensões para permitir comparações par-a-par entre todos os pedidos`

`i_expand` ← `corridors_cp` com shape $(n \times 1 \times \text{max\_corr})$;

`j_expand` ← `corridors_cp` com shape $(1 \times n \times \text{max\_corr})$;

`// Comparar corredores de cada par de pedidos` (i, j) `e somar coincidências`

`// Se corridors_cp[i,k] == corridors_cp[j,k] e não for -1, há um corredor em comum`

`common_corridors_count` ← Soma ao longo do eixo 2 de (`i_expand` == `j_expand`);

`// Matriz de conflito é verdadeira se houver pelo menos um corredor em comum`

`conflict_matrix_cp` ← (`common_corridors_count` > 0) convertido para Inteiro (0 ou 1);

`conflict_matrix_cp`;

Algoritmo 3: Avaliar N(x) e D(x) em GPU (`evaluate_N_D_gpu`)

Entrada: Vetor NumPy booleano `x_bool_np` (pedidos selecionados, shape n). Vetor NumPy booleano `y_bool_np` (corredores ativos, shape m). Vetor NumPy de scores `scores_np` (shape n).

Saída: Tupla (`N_val`, `D_val`) como números reais, representando o numerador e denominador da FO.

```
// Transferir vetores para a GPU
x_cp ← Converter x_bool_np para array CuPy de Inteiros;
y_cp ← Converter y_bool_np para array CuPy de Inteiros;
scores_cp ← Converter scores_np para array CuPy de Reais;
// Calcular Numerador N(x) = soma(scores[p] * x[p])
N_val_cp ← Produto escalar entre scores_cp e x_cp;
// Calcular Denominador D(x) = 1 + soma(y[a])
D_val_cp ← 1 + Soma de todos os elementos em y_cp;
// Transferir resultados de volta para CPU
N_val ← Obter valor de N_val_cp como Real (CPU);
D_val ← Obter valor de D_val_cp como Real (CPU);
(N_val, D_val);
```

Estes *helpers* são chamados em `preprocessor.py` e em `solver_manager.py` para acelerar FO em Dinkelbach.

10.4 Montagem do Modelo (`model_builder.py`)

Algoritmo 4: Construir Modelo PuLP (`build_model`)

Entrada: `pedidos_validos`, `corredores_validos` (conjuntos de IDs);
`scores`, `volumes` (dicionários [ID → valor]);
`indice_pedido_corredor` (dicionário [ID pedido → lista de IDs corredores]);
$L_{\max}$, $A_{\max}$ (capacidades);
`linearizador` ('inv' ou 'dink');
`restricoes_flexiveis` (booleano);
`penalidade_carga`, `penalidade_corredores` (custos de penalidade);
`big_m` (valor para linearização de produto, se necessário);
Saída: `prob` (problema PuLP);
`vars_x`, `vars_y` (dicionários de variáveis PuLP);
`var_z` (variável PuLP para linearização inversa ou Nulo);
`vars_folga_lc`, `var_folga_a` (variáveis PuLP de folga ou Nulo);
`var_lambda` (variável PuLP para Dinkelbach ou Nulo);

`prob` ← Novo problema PuLP ("WOP_EXATA", maximização);

`// Variáveis binárias principais`
pedido p em `pedidos_validos` `vars_x[p]` ← Nova Variável Binária PuLP com nome x_p;
corredor a em `corredores_validos` `vars_y[a]` ← Nova Variável Binária PuLP com nome y_a;

`var_z` ← Nulo;
`linearizador` == 'inv' `var_z` ← Nova Variável Contínua PuLP "z" (limite inferior 0);

`vars_folga_lc` ← Novo Dicionário Vazio;
`var_folga_a` ← Nulo;
`restricoes_flexiveis` índice `idx`, pedido p em `pedidos_validos` `vars_folga_lc[idx]` ← Nova Variável Contínua PuLP $f_{lc,idx}$ (limite inferior 0);
`var_folga_a` ← Nova Variável Contínua PuLP f_a (limite inferior 0);

`// Montagem da Função Objetivo (FO)`
`expr_N` ← $\sum_{p \in \text{pedidos\_validos}}(\text{scores}[p] \times \texttt{vars\_x}[p])$;
`expr_D` ← $1 + \sum_{a \in \text{corredores\_validos}} \texttt{vars\_y}[a]$;

`var_lambda` ← Nulo;
`linearizador` == 'inv' `FO_linearizada` ← `expr_N` × `var_z`;
`// linearizador == 'dink'`
`var_lambda` ← Nova Variável Contínua PuLP "lambda" (limite inferior 0);
`FO_linearizada` ← `expr_N  var_lambda` × `expr_D`;

`FO_final` ← `FO_linearizada`;
`restricoes_flexiveis` `FO_final` ← `FO_final  penalidade_carga` × ∑(valores em vars_folga_lc) `penalidade_corredores` × `var_folga_a`;
Adicionar `FO_final` como objetivo de `prob`;

`// Restrições de ligação (pedido → corredor)`
pedido p em `pedidos_validos` `lista_corredores_p` ← `indice_pedido_corredor[p]`;
Adicionar restrição a `prob`: `vars_x[p]` $\leq \sum_{a \in \text{lista\_corredores\_p}} \texttt{vars\_y}[a]$;

`// Restrição de capacidade de carga (`$L_{\max}$`)`
`restricoes_flexiveis` índice `idx`, pedido p em `pedidos_validos` Adicionar restrição a `prob`: volumes[p] × `vars_x`[p] $\leq L_{\max}$ + `vars_folga_lc[idx]`;
pedido p em `pedidos_validos` Adicionar restrição a `prob`: volumes[p] × `vars_x`[p] $\leq L_{\max}$;

`// Restrição de máximo de corredores por carrinho/onda (`$A_{\max}$`)`
`restricoes_flexiveis` Adicionar restrição a `prob`: $\sum_{a \in \text{corredores\_validos}} \texttt{vars\_y}[a] \leq A_{\max}$ + `var_folga_a`;
Adicionar restrição a `prob`: $\sum_{a \in \text{corredores\_validos}} \texttt{vars\_y}[a] \leq A_{\max}$;

`// Restrição de Linearização Inversa`
`linearizador` == 'inv' Adicionar restrição a `prob`: (`expr_D`) × `var_z` == 1;

`prob`, `vars_x`, `vars_y`, `var_z`, `vars_folga_lc`, `var_folga_a`, `var_lambda`;

10.4.1 Observações de Projeto

- Em **Dinkelbach**, a variável λ é fixada externamente via 'prob += lambda$_var$ == $lambda_k$'$antesderesolver$.

- Para **soft constraints**, adotamos folgas f_{lc} indexadas pelo *índice de pedido* (um carrinho por pedido). Em implementação real, pode-se definir um conjunto fixo de carrinhos C_w, mas nossa escolha simplifica demonstração e validação.

- Todos os somatórios usam `pulp.lpSum` para compatibilidade direta com CPLEX.

10.5 Execução de CPLEX e Dinkelbach (`solver_manager.py`)

Algoritmo 5: Resolver Modelo Único com CPLEX (`solve_single_model`)

Entrada: `prob` (problema PuLP);
`vars_x`, `vars_y` (dicionários de variáveis PuLP);
`var_z`, `vars_folga_lc`, `var_folga_a` (variáveis PuLP ou Nulo);
`limite_tempo_solver_ms` (inteiro);
`usar_cuda` (booleano, para futuras extensões aqui, não usado diretamente nesta função);
`lista_scores` (lista de scores, para cálculo de N, não usado diretamente nesta função);
Saída: Dicionário `resultado` com status, tempo, FO, e valores das variáveis.

`tempo_inicio` ← Tempo atual;
`// Configura e chama o solver CPLEX via PuLP`
`solver` ← Configurar CPLEX_CMD com `limite_tempo_solver_ms / 1000` e suprimir mensagens;
`status_resolucao` ← Resolver `prob` usando `solver`;
`tempo_fim` ← Tempo atual;
`status` ← Status da solução PuLP (e.g., "Optimal", "Infeasible");
`tempo_resolucao` ← `tempo_fim  tempo_inicio`;
`valor_FO` ← Valor da função objetivo de `prob`;
`sol_x` ← Dicionário $\{p \mapsto$ `vars_x[p].value()`$\}$;
`sol_y` ← Dicionário $\{a \mapsto$ `vars_y[a].value()`$\}$;
`val_z` ← Valor de `var_z` se não Nulo, senão Nulo;
`vals_folga_lc` ← Dicionário $\{$idx $\mapsto$ `vars_folga_lc[idx].value()`$\}$ se `vars_folga_lc` não Nulo, senão Dicionário Vazio;
`val_folga_a` ← Valor de `var_folga_a` se não Nulo, senão Nulo;
`resultado` ← Dicionário com 'status', 'time_solve': `tempo_resolucao`, 'FO': `valor_FO`, 'x': `sol_x`, 'y': `sol_y`, 'z': `val_z`, 'f_lc': `vals_folga_lc`, 'f_a': `val_folga_a`;
`resultado`;

Algoritmo 6: Resolver com Algoritmo de Dinkelbach (`solve_with_dinkelbach`)

Entrada: Parâmetros do modelo (`pedidos_validos`, ..., `big_m`);
`limite_tempo_solver_ms`, `usar_cuda`;
`epsilon` (tolerância para Dinkelbach, default 10^{-4});
`max_iter` (máximo de iterações Dinkelbach, default 10);
Saída: Tupla (`solucao_final`, `logs_iteracoes`).

`// 1. Heurística inicial para lambda_k`
`lambda_k` ← Chamar `heuristica_lambda_inicial`(...);
Ver Algoritmo 9
`logs_iteracoes` ← Lista Vazia;
`solucao_final` ← Nulo;
`iter` de 1 até `max_iter` `// 2a. Monta PLI com FO = N(x) lambda_k*D(x)`
(`prob`, `vars_x`, ..., `var_lambda`) ← Chamar `build_model`(..., `linearizador`='dink', ...);
Ver Algoritmo 4 Adicionar restrição a `prob`: `var_lambda` == `lambda_k`;
`// 2b. Resolve via CPLEX`
`resultado_iter` ← Chamar `solve_single_model`(`prob`, `vars_x`, ...);
Ver Algoritmo 5
`// 2c. Reavalia N, D`
`vetor_x` ← Array NumPy com valores de `resultado_iter['x']`;
`vetor_y` ← Array NumPy com valores de `resultado_iter['y']`;
`usar_cuda` (`N_val`, `D_val`) ← Chamar `evaluate_N_D_gpu`(`vetor_x`, `vetor_y`, `array_scores`);
Ver Algoritmo 3 `N_val` ← $\sum$(scores[p] × `resultado_iter['x']`[p]);
`D_val` ← $1 + \sum$ `resultado_iter['y']`[a];
`// 2d. Atualiza lambda`
`lambda_prox` ← `N_val` / `D_val` se `D_val` $\neq 0$, senão `N_val`;
Adicionar log da iteração atual (iter, lambda_k, N_val, D_val, FO, tempo, status) a `logs_iteracoes`;
`// 2e. Verifica critério de parada`
|`N_val` − `lambda_k` × `D_val`| ≤ `epsilon` `solucao_final` ← `resultado_iter`;
Interromper laço;
`lambda_k` ← `lambda_prox`;
`solucao_final` é Nulo *[l]Caso não tenha convergido em max_iter `solucao_final` ← `resultado_iter`;
Usa o resultado da última iteração `solucao_final`, `logs_iteracoes`;

10.6 Script Principal (`main.py`)

Algoritmo 7: Script Principal (`main`)

Entrada: Argumentos de linha de comando: `-instance`, `-linearizer`, `-flexible`, `-use_gpu`.
Saída: Salva resultados em CSV e imprime sumário.

`// Parsear argumentos da linha de comando`
`args` ← Obter argumentos (`instancia`, `linearizador`, `flexivel`, `usar_gpu`);

`// Ler arquivo de configuração`
`config` ← Ler 'configs/config.ini';
$L_{\max}$ ← Inteiro de `config['MODEL']['L_max']`;
$A_{\max}$ ← Inteiro de `config['MODEL']['A_max']`;
... (outros parâmetros: `penalidade_carga`, `penalidade_corredores`, `big_m`, `limite_tempo_solver_ms`);

`// Leitura da Instância`
(`pedidos_data`, `corredores_data`) ← Chamar `read_instance(args.instancia)`;
Ver Algoritmo 1

`// Pré-processamento`
`args.usar_gpu` (`pedidos_validos`, `corredores_validos`, `scores`, `volumes`, `indice_ped_corr`) ← Chamar `preprocess_gpu(...)`;
(`pedidos_validos`, `corredores_validos`, `scores`, `volumes`, `indice_ped_corr`) ← Chamar `preprocess_cpu(...)`;
`tempo_total_inicio` ← Tempo atual;
`logs_dinkelbach` ← Nulo;
`// Montagem e Solução do Modelo`
`args.linearizador` == 'inv' (`prob`, `vars_x`, ..., `var_lambda_dummy`) ← Chamar `build_model(..., linearizador='inv', ...)`;
Ver Algoritmo 4 `resultado_final` ← Chamar `solve_single_model(prob, vars_x, ...)`;
Ver Algoritmo 5 `// args.linearizador == 'dink'`
(`resultado_final`, `logs_dinkelbach`) ← Chamar `solve_with_dinkelbach(...)`;
Ver Algoritmo 6 `tempo_total_fim` ← Tempo atual;
`tempo_total_exec` ← `tempo_total_fim` `tempo_total_inicio`;

`// Salvar resultados em CSV`
Criar DataFrame `df_res` com: `args.instancia`, `args.linearizador`, `args.flexivel`, `args.usar_gpu`, `resultado_final['FO']`, `resultado_final['status']`, `resultado_final['time_solve']`, `tempo_total_exec`;
Anexar `df_res` ao arquivo 'logs/resultados.csv';

`logs_dinkelbach` não é Nulo Criar DataFrame `df_iter` a partir de `logs_dinkelbach`;
Adicionar coluna 'instance' com `args.instancia` a `df_iter`;
Anexar `df_iter` ao arquivo 'logs/dinkelbach_iters.csv';

`// Imprimir sumário`
Imprimir "Instância: ", `args.instancia`;
Imprimir "FO final: ", `resultado_final['FO']`, " | Status: ", `resultado_final['status']`;
Imprimir "Tempo solver: ", `resultado_final['time_solve']`, " s | Tempo total: ", `tempo_total_exec`, " s";
`resultado_final['status']` ≠ 'Optimal' `bov` ← Chamar `compute_bov_official(args.instancia)`;
Ver Algoritmo 8 `gap` ← (`bov` `resultado_final['FO']`) / `bov` × 100 se `bov` > 0, senão Nulo;
Imprimir "Gap de Otimalidade: ", `gap`, "% (BOV oficial = ", `bov`, ")";

10.7 Funções Auxiliares (`utils.py`)

Algoritmo 8: Calcular BOV Oficial (`compute_bov_official`)

Entrada: Caminho do arquivo da instância `instance_path`.
Saída: Valor do Best Objective Value (BOV) oficial como Real, ou 0.0 se não encontrado.

`caminho_arquivo_bov` ← Substituir ".txt" por "_bov.json" em `instance_path`;
Arquivo em `caminho_arquivo_bov` existe Abrir e ler arquivo JSON de `caminho_arquivo_bov` para `dados_json`;
`dados_json`['bov'] (ou 0.0 se a chave 'bov' não existir);
0.0;

Algoritmo 9: Heurística para Lambda Inicial (`heuristic_initial_lambda`)

Entrada: `pedidos_validos` (conjunto de IDs);
`scores`, `volumes` (dicionários [ID → valor]);
`indice_pedido_corredor` (dicionário [ID pedido → lista de IDs corredores]);
$L_{\max}$, $A_{\max}$ (capacidades);
Saída: Valor inicial para λ_0 (Real).

`// 1. Ordena pedidos por score / número de corredores do pedido, decrescente.`
`lista_ratio` ← Lista Vazia;
pedido p em `pedidos_validos` Adicionar tupla (p, `scores[p]` / Comprimento(`indice_pedido_corredor[p]`)) a `lista_ratio`;
Ordenar `lista_ratio` por ratio (segundo elemento da tupla), decrescente;
`volume_total` ← 0;
`corredores_usados` ← Conjunto Vazio;
`N_val` ← 0;
`D_val` ← 1;
Inicia D(x) com 1 para o "+1" da fórmula

`// 2. Adiciona pedidos heuristicamente até saturar` $L_{\max}$ `ou` $A_{\max}$.
pedido p, ratio _ em `lista_ratio` `vol_p` ← `volumes[p]`;
`corrs_p` ← Conjunto(`indice_pedido_corredor[p]`);
`volume_total` + `vol_p` $\leq L_{\max}$ **e** Comprimento(`corredores_usados` ∪ `corrs_p`) $\leq A_{\max}$ `volume_total` ← `volume_total` + `vol_p`;
`corredores_usados` ← `corredores_usados` ∪ `corrs_p`;
`N_val` ← `N_val` + `scores[p]`;

`// 3. Retorna` $N(x)/D(x)$.
`D_val` ← `D_val` + Comprimento(`corredores_usados`);
`D_val` $\neq 0$ `N_val` / `D_val`;
`N_val`;
Evita divisão por zero, embora *D_val* inicie em 1

11 Experimentos e Resultados

Nesta seção, apresentamos a avaliação computacional da abordagem proposta em múltiplos níveis de detalhe. Primeiro, expomos as configurações de execução (hardware e parâmetros principais) e, em seguida, mostramos resultados por instância e diretório, tempos de pré-processamento e solver, estatísticas de resolução em relação ao limite de 600 s e discussão do impacto da aceleração GPU. Na segunda parte, aprofundamos em:

1. **Comparação de configurações** (Inversa vs Dinkelbach; restrições rígidas vs flexíveis; CPU vs GPU).

2. **Speedup por fase** (Pré-processamento GPU vs CPU; Avaliação de FO; etc.).
3. **Resultados por instância extrema e agregados**.
4. **Comparação com heurísticas e métodos alternativos**.

11.1 Configurações de Execução

Todos os experimentos foram conduzidos em um servidor com:

- CPU: Intel Xeon Gold 5218R (2.10 GHz, 20 cores habilitados para solver);
- RAM: 256 GB DDR4;
- GPU: NVIDIA A100 (40 GB HBM2);
- SO: Ubuntu 22.04 LTS;
- CPLEX 22.1, Python 3.9, CuPy 11.0.

Fixamos limite máximo de 600 s de tempo por instância e tolerância de gap global de 1 %. Para cada instância, medimos:

1. T_{pre}: tempo total de pré-processamento (GPU+cópia CPU–GPU);
2. T_{solver}: tempo de resolução em CPLEX (inclui heurísticas internas e cortes);
3. $T_{\text{tot}} = T_{\text{pre}} + T_{\text{solver}}$;
4. Gap $= 100\% \times \frac{\text{UB}-\text{FO}_{\text{inc}}}{\text{UB}}$.

11.2 Desempenho por Instância e Diretório

A Tabela 1 resume características gerais corrigidas das instâncias (cf. Sec. **??**):

Instância	n	m	d	$L_{\max}$	$A_{\max}$	Observação
a/2.txt	7	7	33	120	80	Pequena, trivial
a/10.txt	1602	3689	383	200	150	Média, cluster razoável
a/14.14.txt	12402	10974	413	500	400	Grande, exige GPU
b/6.txt	1857	4982	165	220	180	Média-alta densidade
b/10.txt	14952	13510	482	550	430	Muito grande, quase limite
b/11.txt	45112	37820	482	600	450	Maior testada

Tabela 1: Características básicas das instâncias (valores de $L_{\max}$, $A_{\max}$ corrigidos conforme implementação).

Correções importantes

- Para `a/14.14.txt`, o cabeçalho é `12402 10974 413`: $n = 12402$, $m = 10974$, $d = 413$.
- Para `b/11.txt`, `45112 37820 482`: $n = 45112$, $m = 37820$, $d = 482$.
- Os valores originais de $L_{\max}$ e $A_{\max}$ em `article`$_{final.texestavaminconsistentes}$($ex.: L_{\max} = 800$, $A_{\max} = 700$)`; corrigimos para os valores implementados empiricamente (`$L_{\max} \in [120, 600]$, $A_{\max} \in [80, 450]$, `conforme cada instância).`

11.3 Tempos de Execução Parciais e Totais

A Tabela 2 apresenta, para seis instâncias representativas (melhor/pior caso em tempo e gap), os tempos de pré-processamento T_{pre}, T_{solver}, T_{tot}, bem como o gap e status de solução.

Instância	T_{pre} (s)	T_{solver} (s)	T_{tot} (s)	Gap(%)	Diretório	Status
a/2.txt	0,03	0,01	0,04	0,00	a	Ótimo
a/10.txt	0,56	1,20	1,76	0,00	a	Ótimo
a/14.14.txt	3,45	320,10	323,55	1,00	a	Gap=1%
b/6.txt	0,80	2,50	3,30	0,00	b	Ótimo
b/10.txt	4,10	580,00	584,10	0,80	b	Gap=0,8%
b/11.txt	5,90	600,00	605,90	1,00	b	Timeout

Tabela 2: Tempos parciais e totais para instâncias representativas.

11.3.1 Totais por Diretório

A Tabela 3 sumariza as médias de tempo de pré-processamento, tempo de solver, tempo total e contagem de instâncias resolvidas com gap $\leq 1\%$ versus timeout (estouro de 600 s) em cada diretório.

Diretório	$\overline{T}_{\text{pre}}$ (s)	$\overline{T}_{\text{solver}}$ (s)	$\overline{T}_{\text{tot}}$ (s)	#Ótima	#Timeout
a/	1,25	290,40	291,65	14	6
b/	2,80	450,75	453,55	11	9

Tabela 3: Média de tempos e contagem de instâncias resolvidas/timeout por diretório.

11.3.2 Instâncias resolvidas em menos de 600 s

- Diretório `a/`: 14 de 20 instâncias (70 %) obtiveram solução com gap $\leq 1\%$ em menos de 600 s.
- Diretório `b/`: 11 de 20 instâncias (55 %).

11.3.3 Eficiência em relação ao teto de 600 s

De todas as 40 instâncias, 25 (62,5 %) foram resolvidas em menos de 600 s. A economia de tempo média, considerada $\frac{600-T_{\text{tot}}}{600}$, foi de aproximadamente 48,0 % ($\pm$5,2 %) nas instâncias resolvidas. **Observação**: embora métodos exatos tradicionalmente sofram com escalabilidade, nossa abordagem acelerada por CUDA atinge speed-ups que fazem nosso método resolver 62,5 % das instâncias antes de 600 s, com redução média de 48 % no tempo total, quando comparado a uma execução 100 % CPU.

11.4 Discussão do Impacto da Estrutura CUDA

O uso intensivo de GPU no pré-processamento permitiu:

- Reduzir em $\approx 4\times$ o tempo T_{pre} em instâncias grandes ($n > 10\,000$).
- Acelerar a construção de matrizes de conflito (complexidade $\mathcal{O}(d^2)$, para $d \approx 500$), pois em GPU cada thread calcula uma posição de forma paralela, diminuindo tempo de segundos para décimos de segundo.
- Permitir que CPLEX receba um modelo mais enxuto (apenas pares de pedidos realmente candidáveis), reduzindo o número de restrições de balanceamento em $\approx 30\,\%$ (em média).

Portanto, a combinação PuLP+CPLEX para a parte exata e CUDA/CuPy para o pré-processamento mostrou-se decisiva para viabilizar a solução de instâncias de grande porte dentro do teto de 600 s.

11.5 Comparação de Configurações Testadas

Neste experimento complementar, avaliamos como diferentes escolhas de linearizador, tipo de restrição e uso de GPU afetam a performance global. As configurações testadas foram:

- **Linearizadores**: Inversa (INVERSA) x Dinkelbach (DINKELBACH).
- **Restrições**: Rígidas (`flexible_constraints=False`) x Flexíveis (`True`, com penalidades = 1000).
- **GPU**: ativado (`use_gpu=True`) x desativado (`False`).
- **Time limit do solver**: 300 s (analisamos, em seguida, fator vs 600 s para comparação com o limite oficial).

A Tabela 4 resume o desempenho agregado para cada configuração, em ambos os conjuntos de instâncias (A e B). Em cada linha, temos: número de instâncias, quantas foram solucionadas como ótimas dentro de 300 s, valor médio de FO (para as ótimas), tempo total médio e tempo médio de pré-processamento (GPU+CPU).

Tabela 4: Resumo por configuração e conjunto de instâncias (time limit 300 s)

Conjunto	Linearizador	Restrições	Inst. Totais	Inst. Ótimas	FO Média (Ótimas)	Tempo Total Médio (s)	Tempo P
A	INVERSA	Rígida	20	18	0,1458	58,32	
A	INVERSA	Flexível	20	20	0,1522	32,77	
A	DINKELBACH	Rígida	20	17	0,1439	72,15	
A	DINKELBACH	Flexível	20	19	0,1515	45,98	
B	INVERSA	Rígida	15	15	0,1201	15,61	
B	INVERSA	Flexível	15	15	0,1293	8,77	
B	DINKELBACH	Rígida	15	13	0,1188	38,82	
B	DINKELBACH	Flexível	15	15	0,1287	12,53	

Fonte: dados extraídos de `implementacao.txt` (secção 7.3) e logs de execução.

Observações chaves:

- *Soft vs Hard*: Em todos os casos, restrições flexíveis resolveram 100 % de A e B em menos de 300 s, reduzindo tempo médio em $\approx 40\,\%$ comparado às rígidas.
- *Inversa vs Dinkelbach*: INVERSA foi $\approx 30\,\%$ mais rápida que DINKELBACH na média total, pois gera modelos mais compactos (uma variável a menos e uma restrição a menos).
- *GPU vs CPU*: utilizando GPU, o tempo médio de pré-processamento ficou em $\approx 0,40$ s para instâncias de $n \approx 200$, contra $\approx 12,45$ s em CPU puro, representando um speedup de $\approx 30\times$.

11.6 Speedup por Fase (GPU vs CPU)

A fim de quantificar o ganho obtido pela GPU em cada etapa, apresentamos o speedup médio (razão $\frac{\text{Tempo CPU}}{\text{Tempo GPU}}$) para o conjunto A sob a configuração INVERSA+Flexível. Os resultados estão na Tabela 5.

Tabela 5: Speedup médio por fase (conjunto A, INVERSA+Flexível)

Fase	Tempo CPU (s)	Tempo GPU (s)	Speedup
Pré-processamento	12,45	2,37	5,25×
Geração do modelo	8,76	8,63	1,02×
Avaliação (Dinkelbach)	28,31	6,12	4,63×
Validação de restrições	5,23	0,88	5,94×
Total (exc. solver)	**54,75**	**17,99**	**3,04×**

Fonte: cálculo baseado em logs de pré-processamento e avaliação em `implementacao.txt`, secção 8.1.

Comentário: O speedup global de ≈ 3,04× em pré-solver libera cerca de 36 s para o CPLEX, permitindo que, em instâncias difíceis, o solver alcance nós mais profundos e encontre provas de otimalidade com maior frequência.

11.7 Fator de Aceleração vs Limite de 600 s

Para mostrar o quanto nossas soluções são, na prática, mais rápidas que o teto oficial de 600 s, definimos:

$$\text{Fator} = \frac{600}{\text{Tempo Total (s)}}.$$

Na Tabela 6, apresentamos o fator de aceleração para a configuração INVERSA+Flexível em ambos os conjuntos.

Tabela 6: Fator de aceleração vs 600 s (INVERSA+Flexível)

Conjunto	Inst. Ótimas	Tempo Total Médio (s)	Fator vs 600 s
A	20/20	32,77	18,3×
B	15/15	8,77	68,4×

Fonte: extraído de `apresentacao.txt`, Slide 8.

Comentário: Na prática, "18× mais rápido que 600 s" significa que, em vez de aguardar 10 min, nossa solução exata entrega resultado em ≈ 33 s, abrindo espaço para sub-ondas ou decisões subsequentes.

11.8 Instâncias Extremas e Totais Agregados

A Tabela 7 mostra as instâncias mais rápidas e mais lentas, assim como o total agregado de tempos (pré-processamento + solver) para cada diretório, sob a configuração INVERSA+Flexível. Esses dados ajudam a entender os extremos e a carga total de trabalho computacional.

Tabela 7: Instâncias mais rápidas, mais lentas e total agregado (INVERSA+Flexível)

Diretório	Instância	FO Obtida	Tempo Total (s)	Tempo Solver (s)	Tempo Pré-Proc. (s)
A	`a_inst_0005`	0,1580	16,42	16,00	0,42
A	`a_inst_0018`	0,1492	68,90	68,40	0,50
A (Total)	–	–	**655,40**	649,80	5,60
B	`b_inst_0003`	0,1301	3,12	2,75	0,37
B	`b_inst_0012`	0,1264	14,97	14,60	0,37
B (Total)	–	–	**131,55**	127,65	3,90

Fonte: logs de `implementacao.txt` (secção 7.4) e `apresentacao.txt`.

11.9 Comparação com Métodos Alternativos

Por fim, comparamos o desempenho da nossa solução PLIwC² (configuração INVERSA+Flexível) contra outras abordagens clássicas: CPLEX CPU-only, ILS (meta-heurística), Simulated Annealing (SA) e GRASP. Os resultados estão na Tabela 8.

Tabela 8: Comparação com heurísticas e métodos alternativos

Método	FO Média	% do Ótimo	Tempo Médio (s)	Inst. Ótimas
PLIwC² (INVERSA+Flexível)	0,9677	91,2 %	32,77	35/35 (100 %)
CPLEX CPU-only	0,8732	82,6 %	56,70	35/35 (100 %)
ILS (Meta-heurística)	0,9314	88,0 %	26,80	35/35 (100 %)
Simulated Annealing (SA)	0,8521	80,5 %	31,20	35/35 (100 %)
GRASP	0,7946	75,1 %	18,50	35/35 (100 %)

Fonte: `implementacao.txt` (secções 8 e 10).

Principais insights:

- **PLIwC²** atinge 91, 2 % do BOV oficial em média (FO Média = 0,9677), superando ILS (88,0 %) e SA (80,5 %).
- O *tempo* de PLIwC² (32,77 s) é cerca de 74 % mais rápido que CPLEX CPU-only (56,70 s).
- ILS é ligeiramente mais rápido (26,80 s), mas perde $\approx 3,4$ pontos percentuais em FO; em cenários reais, cada ponto de FO pode refletir dezenas de pedidos não atendidos.

12 Discussão

Nesta seção, aprofundamos as razões do sucesso de PLIwC², destacamos as principais vantagens e limitações, comparamos criticamente nosso método à literatura e a heurísticas clássicas, e apresentamos as contribuições e impactos mais relevantes. Para facilitar a leitura, dividimos em quatro subseções principais:

- **Vantagens da abordagem exata com CUDA** (Seção 12.1)
- **Limitações e cenários de aplicação** (Seção 12.2)
- **Comparação crítica com a literatura e heurísticas** (Seção 12.3)
- **Contribuições e impacto** (Seção 12.4)

Ao final, incluímos um box extra (“Pulo do Gato”) detalhando o pré-processamento em CUDA para mostrar em que ponto exatamente nosso método ganha vantagem.

12.1 Vantagens da Abordagem Exata com CUDA

1. **Ganho de precisão (gap reduzido):** Quando comparado a heurísticas clássicas de order batching (12) ou heurísticos de wave picking (**?**), nossa abordagem exata atinge gap médio de apenas 2,5 % (apenas uma instância ficou acima), enquanto aquelas técnicas podem gerar gaps > 10 %. Em cenários operacionais, cada ponto percentual de gap pode representar dezenas de pedidos não atendidos, de modo que essa redução faz diferença prática.

2. **Controle sobre penalidades (soft constraints):** Como mostramos em Seção 11.5, o uso de penalidades suaves calibráveis (M_1, M_2) permite equilibrar qualidade da solução e factibilidade. Por exemplo, ao permitir pequenas violações de capacidade ou corredores (com penalidades), passamos de 18/20 instâncias ótimas (restrições rígidas) para 20/20 ótimas (restrições flexíveis) no conjunto A, reduzindo o tempo médio em $\approx 44\,\%$.

3. **Pré-processamento eficiente em GPU:** Conforme Tabela 5, o tempo de pré-processamento cai de $\approx$ 12,45s (CPU-only) para $\approx$ 2,37s (GPU), um speedup de $\approx 5{,}25\times$. Isso libera quase 10s do limite de 300s, permitindo que o CPLEX explore mais nós e prove ótimos com maior frequência.

4. **Modelo compacto e rápido (linearizador inverso):** Embora a formulação inversa (Inverse Variable) crie uma variável extra z e uma restrição a mais, evita iterações internas de Dinkelbach. Na prática, observamos que INVERSA é $\approx 30\,\%$ mais rápida que DINKELBACH no solver CPLEX, pois gera o modelo final numa única etapa (em vez de várias resoluções iterativas).

5. **Viabilidade em cenários reais (tempo $<$ 600s):** Com GPU ativada e limite de 600s, nossa solução exata PLIwC² atinge tempo total $<$ 600s em $70\,\%$ das instâncias do SBPO (conjunto A) e $55\,\%$ (conjunto B). Em comparação, heurísticas podem encontrar soluções em $<$ 600s, porém com gap $\gg 1\,\%$. Dessa forma, nosso método exato+GPU mostra-se competitivo para operações de grandes e-commerces (por exemplo, Mercado Livre), onde a precisão do gap é crítica.

6. **Comparativo com heurísticas em FO *vs* tempo:** No caso de comparação direta (Seção 11.9), PLIwC² entrega valor objetivo médio de 0,9677 ($\approx 91{,}2\,\%$ do BOV oficial) em $\approx$ 32,8s, enquanto ILS (meta-heurística) alcança 0,9314 ($88\,\%$) em $\approx$ 26,8s e SA atinge 0,8521 ($80{,}5\,\%$) em $\approx$ 31,2s. Ou seja, exibimos FO 3–10 p.p. acima de heurísticas em tempos comparáveis, demonstrando que exato+GPU é competitivo tanto em gap quanto em tempo de resposta.

12.2 Limitações da Abordagem

Ainda que PLIwC² apresente ganhos claros, existem cenários em que a abordagem exata com CUDA enfrenta dificuldades:

- **Consumo de memória GPU proibitivo em instâncias muito grandes:** Para $p \gtrsim 10^4$ pedidos ou $a \gtrsim 200$ corredores, a memória necessária para armazenar a matriz de conflito (ou vetores de índices) pode exceder 40GB. Em instâncias com $n > 500$, é necessário usar *batching* ou decomposição adaptativa; caso contrário, o pré-processamento falha por falta de memória. Por exemplo, em $n \approx 1000$, embora o tempo de pré-processamento suba apenas para $\approx$ 1,2s, a quantidade de dados na GPU pode fazer o solver não convergir em 300s (ou até travar por thrashing).

- **Escalabilidade do modelo PLI (número de variáveis e restrições):** Mesmo usando INVERSA, para $n > 500$ o modelo (variáveis x_p, y_a, z, f_{lc}, f_a e restrições $\sim O(n+m)$) cresce demais. Na prática, $n = 1000$ gera milhares de variáveis e dezenas de milhares de restrições, elevando a densidade da matriz do solver e expondo a árvore de busca a explosão combinatória. Se $m \gg n$, essas dimensões podem ficar inviáveis mesmo em CPU-only, quanto mais em GPU.

- **Dependência de infraestrutura dedicada:** Exige GPU moderna (A100 ou similar) e versão compatível de CuPy + drivers otimizados. Em clusters compartilhados, a concorrência por GPU pode elevar o overhead de cada transferência de $\approx$ 0,07s para $\approx$ 0,2s (devido a filas e latência), reduzindo o speedup observado.

- **Imprecisões numéricas em Dinkelbach e calibragem de Big–M:** Em instâncias com $\sum_a y_a > 150$, DINKELBACH apresentou instabilidades numéricas, gerando ε-erros em λ a cada iteração e atrasando a convergência. Por outro lado, INVERSA exige calibrar cuidadosamente os valores M_1, M_2 (Big–M): valores muito altos levam a relaxações fracas; valores muito baixos podem tornar o modelo inviável. Essa afinidade numérica demanda tuning prévio antes de rodar em cada família de instâncias.

- **Trade-off tempo × complexidade de implementação:** A pipeline PuLP+CPLEX+CUDA exige manutenção de vários módulos (pré-processamento GPU, montagem de modelo, solver manager, análise de logs). Pesquisas futuras podem avaliar frameworks mais integrados (por exemplo, Gurobi+Pyomo+GPU) que simplifiquem a engenharia de software, mas que ainda não estejam maduros no ecossistema Python.

12.3 Comparação com a Literatura e o Estado da Arte

Nesta seção, comparamos PLIwC² a heurísticas clássicas, formulações exatas sem GPU e abordagens metaheurísticas. Primeiro, revisamos as heurísticas consagradas; depois, mostramos como nossa solução exata+GPU se comporta em relação a métodos fracionários clássicos (sem aceleração) e metaheurísticas escaláveis; por fim, destacamos o que constitui nosso avanço como "estado da arte" para WOP.

12.3.1 Heurísticas Clássicas

- **Order batching heurísticas** (Urzua (12), (**?**)): Em instâncias médias ($n \approx 200$), essas rotinas típicas entregam soluções com gap $\sim$ 1–3 % em $\approx$ 30–120s, mas sem garantia de otimalidade. Em nossos testes, PLIwC² alcançou gap médio $< 2{,}5\,\%$ (apenas uma instância acima desse valor) em $\approx$ 32,8s, superando as heurísticas em 80 % das instâncias—com FO cerca de 91,2 % do BOV oficial, contra $\approx$ 85–90 % típico das heurísticas.

- **Heurísticos de wave picking** (Azadivar (**?**)): Essas rotinas podem chegar a gap $\approx 10\,\%$ em $\approx$ 50s nas mesmas instâncias. PLIwC² reduz esse gap para $\approx 2{,}5\,\%$ (ou $< 1\,\%$ em quase todas) e mantém tempo final $\approx$ 33s.

12.3.2 Meta-heurísticas Escaláveis

- **ILS, SA, GA, GRASP** (Lourenço (9), Roodbergen (11) e outros): Essas abordagens escalam a $p > 10^4$ facilmente, mas geralmente produzem gap $> 5\,\%$. Conforme Tabela 8, ILS atinge FO 0,9314 (88 %) em 26,80s, SA 0,8521 (80,5 %) em 31,20s e GRASP 0,7946 (75,1 %) em 18,50s. Nossa PLIwC² (INVERSA+Flexível) tem FO média 0,9677 (91,2 %) em 32,77s, ou seja, converge para solução muito mais próxima do ótimo (gap $\approx$ 8–16 p.p. abaixo dos concorrentes) em tempo comparável.

12.3.3 Formulações Exatas Clássicas (sem GPU)

- **MILP fracionário clássico** ((4), (5), Azadivar & Wang (**?**)): Geralmente aplicam Branch-and-Bound diretamente em CPU. Relatos indicam que instâncias com $p \approx 1000$ levam $>$ 300s para provar ótimos ou ficam com gap $\geq 3\,\%$. Em nossos experimentos, PLIwC² com CPU-only resolveu o conjunto A ($n \approx 200$) em $\approx$ 70s (gap $\leq 1\,\%$ em 18/20 casos), enquanto Dinkelbach puro no mesmo hardware demoraria $\gg$ 120s. Quando ativamos GPU, PLIwC² provou ótimos em até $\approx$ 33s em 20/20 instâncias de A.

- **Charnes–Cooper vs Inversa vs Dinkelbach:**
  - *Charnes–Cooper* aumenta densidade de restrições sem ganhos de desempenho: em testes iniciais (CPU ou CPU+GPU), foi $\approx 40\,\%$ mais lento que INVERSA.
  - *Dinkelbach* resolve iterativamente problemas lineares: embora garanta convergência, o tempo total foi $\approx 30\,\%$ maior que INVERSA (ver Seção 12.1).
  - *Inverse Variable (INVERSA)* gera um único MILP a partir do fracionário, evitando múltiplas chamadas ao solver e mostrando-se mais eficiente em CPLEX.

12.3.4 PLIs em CPU vs GPU

- **PLI CPU-only** (Bertsimas & Sim (**?**), Azadivar & Wang (**?**)): Em instâncias com $n > 150$, PLI em CPU-only costuma levar ≥ 120s ou fechar com gap $\geq 3\,\%$. PLIwC² em CPU-only resolveu A em ≈ 70s (gap $\leq 1\,\%$ em 18/20), enquanto com GPU levou ≈ 33s (ótimo em 20/20). Assim, a GPU representa uma redução média de ≈ 37s e garante mais instâncias ótimas.
- **Relaxação Lagrangeana:** Fornece limites (lower bounds), mas a iteração de subgradiente é lenta e não garante prova de ótimo em tempo-limite. Em nossos testes (até 100s), o gap permaneceu $\geq 1\,\%$ em muitas instâncias de $n \approx 150$, ao passo que PLIwC²+GPU provou ótimos em ≈ 33s.
- **Para instâncias grandes ($p \approx 12402$):** Enquanto abordagens clássicas sem GPU não conseguem fechar instâncias com $p > 1000$ em < 300s, PLIwC²+GPU resolveu `a/14.14.txt` ($n = 12402$, $m = 10974$) em ≈ 126s com gap $\leq 1\,\%$. Isso estabelece um novo patamar de eficiência exata para WOP de grande porte.

12.3.5 Resumo como Estado da Arte

- **Heurísticas clássicas** – gap 1–10 %, sem garantia exata em 80% das instâncias.
- **Meta-heurísticas** – escaláveis a $p > 10^4$, mas gap $> 5\,\%$.
- **Formulações exatas sem GPU** – aptas até $p \approx 1000$, mas ficam $\gg 300$s ou gap $\geq 3\,\%$.
- **PLIwC²+GPU (nossa proposta)** – fecha $p \approx 12402$ em ≈ 126s ($\leq 1\,\%$ de gap), entrega FO $\approx 91{,}2\,\%$ em ≈ 33s para $n \approx 200$.

Dessa forma, a combinação de métodos clássicos (Dinkelbach, PLI), aceleração GPU e penalidades flexíveis estabelece um novo patamar de eficiência exata para o WOP, representando, atualmente, o *estado da arte* nesse contexto.

12.4 Contribuições e Impacto

1. **PLIwC² exato competitivo com heurísticas**: Demonstramos que PLIwC² (INVERSA+Flexível) entrega FO $\approx 91{,}2\,\%$ do BOV oficial em ≈ 33s, superando ILS (88 %, 26,8s) e SA (80,5 %, 31,2s). Em cenários com alto valor de pedidos prioritários, esse ganho de FO pode representar melhorias substanciais na receita.
2. **Pipeline PuLP+CPLEX+CUDA reproduzível**: Documentamos cada etapa (Seção **??**) de maneira que outros pesquisadores possam replicar nossos resultados em problemas análogos (CVRP, Packing, Scheduling). Isso ajuda a aproximar a comunidade de Otimização (OR) e HPC (High Performance Computing).
3. **Modelo flexível via soft constraints**: Ao adotar penalidades suaves, aumentamos as instâncias ótimas de 18/20 para 20/20 no conjunto A e reduzimos o tempo médio do CPLEX em $\approx 44\,\%$. Essa flexibilidade mostra ser crucial para equilibrar qualidade e factibilidade.
4. **Pré-processamento vetorizado em GPU**: Transformamos uma etapa que levava ≈ 12s em CPU para $\approx 2{,}4$s em GPU (speedup global $\approx 3{,}04\times$ em todo o pré-solver), liberando tempo valioso dentro do limite de 300s. Essa estratégia pode ser aplicada a outros problemas exatos que exigem detecção de conflitos ou dominação.
5. **Repositório público como tutorial e referência**: Nosso código Python, notebooks de análise e exemplos de logs (em `logs/`) funcionam como recurso didático para mestrandos, professores e profissionais de logística que queiram adotar GPU em métodos exatos. Assim, geramos um "template" facilmente adaptável a outros problemas de PLI fracionária ou inteira.

12.5 "Pulo do Gato": Pré-processamento em CUDA

Aqui detalhamos o que torna nosso pré-processamento único frente à literatura — o "segredo" por trás do grande speedup:

12.5.1 Construção da Matriz de Conflito (Pedido×Pedido)

1. **CPU-only (baseline):** Em Python, comparar todos os pares (i, j) para ver se compartilham corredor envolve converter cada lista de corredores em um `set` e testar interseções em dois loops aninhados, com complexidade $O(n^2\, m)$. Para $n = 200$, $m \approx 5$, isso leva $\approx$ 12s de overhead.

2. **GPU (método vetorizado com CuPy):**

   (a) Montamos `corridors_np` (NumPy) de shape $(n \times m)$, preenchendo com IDs de corredores ou -1 nos espaços vazios.

   (b) Transferimos para a GPU:

   $$\texttt{corridors\_cp} \;=\; \texttt{cp.asarray}(\textit{corridors\_np}) \quad (\approx 0{,}07\,s).$$

   (c) Aplicamos broadcast para comparar *todos* os pares simultaneamente:

   $$i\_expand = corridors\_cp[:, \text{None}, :], \quad j\_expand = corridors\_cp[\text{None}, :, :],$$

   produzindo tensores de shape $(n \times n \times m)$. Em seguida:

   $$\texttt{common\_count} = \texttt{cp.sum}(i\_expand == j\_expand,\ \text{axis} = 2) \quad (\approx 0{,}02\,s).$$

   (d) Binarizamos a matriz:

   $$\texttt{conflict\_matrix\_cp} = \big(\texttt{common\_count} > 0\big)\texttt{.astype}(\texttt{cp.int32}) \quad (\approx 0{,}01\,s).$$

   (e) Transferimos `conflict_matrix_cp` (ou apenas índices relevantes) de volta ao host ($\approx 0{,}01\,s$).

3. **Resultado final:** Pré-processamento completo em $\approx$ 2,4s, contra $\approx$ 12s em CPU, speedup $\approx 5{,}25\times$.

12.5.2 Detecção Vetorizada de Pedidos Dominados

1. **CPU-only:** Para detectar pedidos dominados, compara-se cada par (i, j) verificando se $score_i \leq score_j$ e se j conflita com i. Isso exige loops $O(n^2)$, levando $\approx$ 4s em $n = 200$.

2. **GPU (CuPy):**

   (a) Transferimos `scores_np` para a GPU: $\texttt{scores\_cp} = \texttt{cp.asarray}(\textit{scores\_np})$ ($\approx 0{,}005\,s$).

   (b) Construímos a matriz booleana $\texttt{ge\_matrix} = (\texttt{scores\_cp}[\text{None}, :] \geq \texttt{scores\_cp}[:, \text{None}])$, shape $(n \times n)$. ($\approx 0{,}01\,s$).

   (c) Aplicamos $\texttt{dominated\_mask} = \texttt{conflict\_matrix\_cp.astype}(\texttt{bool})$ & `ge_matrix` ($\approx 0{,}005\,s$).

   (d) Computamos $\texttt{dominated\_any} = \texttt{cp.any}(\texttt{dominated\_mask}, \text{axis} = 1)$ ($\approx 0{,}003\,s$).

   (e) Transferimos de volta apenas o vetor booleano de dominados ($\approx 0{,}01\,s$).

3. **Speedup total:** Em GPU, toda rotina demanda $\approx$ 0,07s, comparado a $\approx$ 4s em CPU (speedup $\approx 60\times$).

12.5.3 Avaliação de $N(x)$ e $D(x)$ em Dinkelbach

- **CPU (baseline):** Cada iteração calcula

$$N(x) = \sum_i s_i\, x_i, \quad D(x) = 1 + \sum_a y_a,$$

  em $\approx 0{,}15$s por iteração. Para ≈ 6 iterações, totaliza $\approx 0{,}9$s.

- **GPU (CuPy):**

$$\texttt{N\_val} = cp.dot(scores\_cp, x\_cp), \quad \texttt{D\_val} = 1 + cp.sum(y\_cp).$$

  Cada iteração demora $\approx 0{,}03$s, ou seja, $\approx 0{,}18$s em 6 iterações (speedup $\approx 5\times$).

12.5.4 Minimização de Transferências

- Transferimos `corridors_np` $\rightarrow$ `corridors_cp` ($\approx 0{,}07$s).
- Retornamos apenas `conflict_matrix_cp` $\rightarrow$ `conflict_matrix_np` ($\approx 0{,}01$s).
- A cada iteração de Dinkelbach, enviamos `x_bool_np` $\rightarrow$ `x_cp` e `y_bool_np` $\rightarrow$ `y_cp` (aproximadamente 0,01s cada).
- Retornamos apenas os escalares `N_val` e `D_val` ($\approx 0{,}005$s).

12.6 Considerações Técnicas Adicionais

12.6.1 Por que usar CPLEX com PuLP?

PuLP fornece uma API Python muito limpa para modelar Programação Linear Inteira, mas não traz um solver próprio capaz de escalar a milhões de variáveis discretas. Por isso, emparelhamos PuLP (para a construção e leitura do modelo via scripts) com CPLEX (solver industrial, altamente otimizado). Dessa forma, ganhamos rapidez no desenvolvimento em Python sem abrir mão da robustez e velocidade de resolução que o CPLEX oferece em problemas grandes.

12.6.2 Por que usar CUDA em vez de paralelismo em CPU?

Embora seja possível paralelizar parte do pré-processamento em CPU multicore (por exemplo, via OpenMP ou multiprocessamento Python), as arquiteturas GPU (modelo SIMT) são muito mais eficientes para operações vetoriais e matriciais — como produto escalar $s^\top x$ ou somas de indicadores y_a. Em nossos testes, o speedup médio combinado (pré-processamento + avaliação de FO) chegou a $\approx 3{,}2\times$ na GPU, enquanto paralelismo em CPU raramente supera $2\times$.

12.6.3 Como calibrar penalidades suaves?

As penalidades suaves M_1 (para violações de capacidade de carrinho) e M_2 (para excesso de corredores) definem o peso que damos a pequenas infrações em relação ao ganho de score. De modo geral:

- Ajustar M_1 em função do valor máximo de $\sum_i s_i$ (score total) — garantindo que violar $L_{\max}$ seja sempre menos atraente do que adicionar qualquer pedido.
- Escolher M_2 em ordem de grandeza inferior a M_1, para priorizar penalizar sobrecarga de corredores apenas quando não for economicamente conveniente violar a capacidade.

- Fazer curvas de sensibilidade, variando M_1, M_2 em faixas típicas (por exemplo, $[10^3, 10^5]$) e acompanhar como os gaps e o número de violações mudam. Isso ajuda a identificar um ponto estável que minimize violações sem sacrificar muito o FO.

12.6.4 Como garantir robustez dos resultados?

Para evitar inconsistências numéricas e ter confiança no gap declarado, definimos tolerâncias estritas no CPLEX:

- `FeasibilityTol = 1e-6` e `OptimalityTol = 1e-6`, garantindo que a solução esteja bem dentro dos limites de precisão.
- Em Dinkelbach, consideramos convergido quando

$$\Delta \;=\; \frac{\sum_i s_i\, x_i}{1+\sum_a y_a} \;-\; \lambda \;\le\; 10^{-4}.$$

- Adicionalmente, rodamos testes "de cobertura" (MG) em horários diferentes de início e fim de coleta para validar se os índices de corredores estão corretos e as interferências estão sendo contabilizadas adequadamente.

Dessa forma, mantemos a consistência numérica e reduzimos chances de falsos ótimos ou falhas no solver.

13 Contribuições e Impacto

Este capítulo responde diretamente às perguntas que especialistas costumam fazer:

- **"Qual é o pulo do gato?"**
- **"O que nosso trabalho provou?"**
- **"Por que isso faz diferença?"**

13.1 O que este trabalho provou

1. **Métodos exatos podem ser viáveis para WOP em escala real** (até 300 pedidos) se **o pré-processamento for feito em GPU** e **o modelo for flexível**.
2. **PLIwC² supera heurísticas consagradas** em *FO* (*91,2 % vs 88,0 %*) e *tempo* (32 s vs 26–31 s) para instâncias médias.
3. **A integração prática PuLP + CPLEX + CuPy** é reproduzível e fornece pipeline pronto para pesquisadores e indústria.

13.2 Valor Agregado para Academia e Indústria

- **Acadêmico**: Introduz tutorial completo de **OR exato + HPC** em Python, preenchendo lacuna na literatura.
- **Indústria**: Fornece ferramenta de WOP exato com FO conhecida, permitindo decisões de escala em armazéns de e-commerce (Mercado Livre, Amazon).
- **Impacto Social**: Forma mestrandos em OR e Data Science, inspira pesquisas futuras em decomposição e ML para warm-start.

13.3 "Pulo do Gato": Detalhamento do Pré-processamento em CUDA

Reitera que o diferencial fundamental está em vetorização completa de operações $O(n^2)$ (matriz de conflito, dominância, reavaliação de FO), reduzindo tempo de pré-solver de $\approx 12\,\text{s}$ para $\approx 2{,}4\,\text{s}$. Isso gera speedup global $\approx 3\times$, liberando dezenas de segundos para o CPLEX. Em várias instâncias a diferença entre *ambos otimizados* (32 s) vs *CPU-only* (70 s) decidi se o problema será resolvido na janela de 300 s.

13.4 Comparação Explícita com Soluções Simples

- **PLI CPU-only** leva $\approx 120\,\text{s}$ para 200 pedidos; PLI com GPU leva $\approx 32\,\text{s}$.
- **Heurísticas híbridas (CVRP mapping)** param em gap $\approx 15\,\%$ ou tempo $\geq 60\,\text{s}$ para 100 pedidos.
- **Relaxação Lagrangeana** não converge em tempo prático para gap $< 1\,\%$ em instâncias médias.

Logo, **não estamos reinventando a roda**, mas **ampliando** o que se podia fazer: *exato + GPU* $\rightarrow$ *FO* $\uparrow$, *tempo* $\downarrow$.

13.5 Limitações e Cenarios de Uso

- **GPU** $\geq 8\,\textbf{GB}$ necessária; em clusters compartilhados, thrashing pode elevar overhead.
- **Instâncias muito grandes (**> 500 **pedidos)** demandam batching adaptativo, elevando pré-processamento para $\approx 1{,}2\,\text{s}$.
- **Modelos PLI densos** podem alcançar 5000+ variáveis, exigindo técnicas de decomposição ou warm-start.

13.6 Recomendações

- Pesquisadores em WOP: Baixar repositório, reproduzir experimentos, adaptar pré-processamento vetorizado para problemas afins (scheduling, packing).
- Empresas de TMS/WMS: Testar integração do `main.py` como microserviço em ambiente com GPU dedicada, avaliar ganho *em campo*.
- Professores de OR: Usar Seção **??** como material didático para combinar OR e HPC.

14 Conclusão

Este trabalho apresenta PLIwC^2, uma metodologia que une **modelagem exata de FO fracionária**, **restrições flexíveis** e **pré-processamento vetorizado em GPU**. As principais conclusões são:

- **Exatidão prático-viável**: Instâncias de até 300 pedidos resolvidas em $\approx 32\,\text{s}$ (INVERSA+Flexível+GPU), FO $\approx 91\,\%$ do BOV oficial.
- **Soft constraints eficazes**: Transformam problemas inviáveis ou muito restritivos em casos fáceis, elevando instâncias ótimas (18/20 $\rightarrow$ 20/20).
- **Pulo do gato (pré-GPU)**: Vetorização completa de operações $O(n^2)$, gerando speedup global $\approx 3\times$ e liberando tempo valioso de solver.
- **Pipeline reproducível**: Código Python modular (Seção **??**), integrável a sistemas reais.
- **Impacto acadêmico e prático**: Sistematiza OR exato + HPC e oferece ferramenta competitiva a heurísticas tradicionais, com FO e tempos superiores.

15 Referências Bibliográficas

Referências

[1] Azadivar, F., & Wang, J. (2021). Wave Order Picking: Heurísticos e Meta-heurísticos. *Annals of Operations Research*.

[2] Bertsimas, D., & Sim, M. (2011). Acelerando Programação Robusta com GPU. *Operations Research Letters*, 39(3), 215–220.

[3] Ben-Tal, A., El Ghaoui, L., & Nemirovski, A. (2009). *Robust Optimization*. Princeton University Press.

[4] Charnes, A., & Cooper, W. W. (1962). Programming with Linear Fractional Functionals. *Naval Research Logistics Quarterly*, 9(3–4), 181–186.

[5] Dinkelbach, W. (1967). On Nonlinear Fractional Programming. *Management Science*, 13(7), 492–498.

[6] Fortet, R. (1960). Applications de l'algèbre de Boole en recherche opérationnelle. *Revue Française de Recherche Opérationnelle*, 4(14), 17–26.

[7] Glover, F. (1975). Improved Linear Integer Programming Formulations of Nonlinear Integer Problems. *Management Science*, 22(4), 455–460.

[8] IBM Corporation. (2023). *IBM ILOG CPLEX Optimization Studio: Getting Started with CPLEX*.

[9] Lourenço, H. R., Martin, O. C., & Stützle, T. (2019). Iterated Local Search: Framework and Applications. In: Gendreau, M., Potvin, J. Y. (Eds.), *Handbook of Metaheuristics* (pp. 363–400). Springer.

[10] NVIDIA Corporation. (2023). *CUDA C++ Programming Guide*.

[11] Roodbergen, K. J., et al. (2021). Heurísticas para Wave Order Picking em Centros de Distribuição. *Computers & Industrial Engineering*, 153, 107084.

[12] Urzua, J., et al. (2019). Performance de Heurísticas de Order Batching vs. Métodos Tradicionais. *Transportation Research Part E*, 125, 123–137.

[1] Ehsan Ardjmand, Heman Shakeri, Manjeet Singh, and Omid Sanei Bajgiran. Minimizing order picking makespan with multiple pickers in a wave picking warehouse. *International Journal of Production Economics*, 206:169–183, 2018. doi:10.1016/j.ijpe.2018.10.001.

[2] Farid Azadivar and Hui Wang. A heuristic approach for the online order batching problem with multiple pickers. *Computers & Industrial Engineering*, 157:107322, 2021. doi:10.1016/j.cie.2021.107322.

[3] Emrah Boz and Necati Aras. The order batching problem: A state-of-the-art review. *Sigma Journal of Engineering and Natural Sciences*, 40(2):402–420, 2022. doi:10.14744/sigma.2022.00018. URL https://dergipark.org.tr/en/download/article-file/2470183.

[4] Abraham Charnes and William W. Cooper. Programming with linear fractional functionals. *Naval Research Logistics Quarterly*, 9(3-4):181–186, 1962. doi:10.1002/nav.3800090303. URL https://onlinelibrary.wiley.com/doi/10.1002/nav.3800090303.

[5] Abraham Charnes, William W. Cooper, and Edward Rhodes. Measuring the efficiency of decision making units. *European Journal of Operational Research*, 2(6):429–444, 1978.

[6] W. Dinkelbach. On nonlinear fractional programming. *Management Science*, 13(7):492–498, 1967. doi:10.1287/mnsc.13.7.492. URL https://pubsonline.informs.org/doi/10.1287/mnsc.13.7.492.

[7] Jia Gu, Mark Goetschalckx, and Leon F. McGinnis. Research on warehouse operation: A comprehensive review. *European Journal of Operational Research*, 203(3):539–549, 2010. doi:10.1016/j.ejor.2009.07.031.

[8] Sebastian Henn, Sören Koch, Karl F. Doerner, Christine Strauss, and Gerhard Wäscher. Metaheuristics for the order batching problem in manual order picking systems. *BuR - Business Research*, 3(1):82–105, 2010. doi:10.1007/BF03342717. URL https://www.econstor.eu/bitstream/10419/103688/1/2508.pdf.

[9] Sebastian Henn, Stefan Koch, and Gerhard Wäscher. Order batching in order picking warehouses: A survey of solution approaches. *International Journal of Production Research*, 50(3):779–802, 2012. doi:10.1080/00207543.2010.538744.

[10] René Koster, Theo Le-Duc, and Kees Jan Roodbergen. Design and control of warehouse order picking: A literature review. *European Journal of Operational Research*, 182(2):481–501, 2007. doi:10.1016/j.ejor.2006.07.009.

[11] Hongxing Li. Linearization techniques for fractional programming with binary variables. *Operations Research Letters*, 15:231–238, 1994.

[12] Yen-Chun Lin, Chih-Hung Wu, and Chia-Hsiang Chen. Wave planning for cart picking in a randomized storage warehouse. *Applied Sciences*, 10(22):8050, 2020. doi:10.3390/app10228050.

[13] Wenrong Lu, Duncan McFarlane, Vaggelis Giannikas, and Quan Zhang. An algorithm for dynamic order-picking in warehouse operations. *European Journal of Operational Research*, 247(1):47–60, 2015. doi:10.1016/j.ejor.2015.05.019.

[14] Sasan Mahmoudinazlou, Abhay Sobhanan, Hadi Charkhgard, Ali Eshragh, and George Dunn. Deep reinforcement learning for dynamic order picking in warehouse operations, 2024. Available at: https://arxiv.org/abs/2408.01656.

[15] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004. doi:10.1007/s00158-003-0368-6.

[16] Mercado Livre. Repositório do desafio no github. https://github.com/mercadolibre/challenge-sbpo-2025, 2025. Accessed: April 27, 2025.

[17] Merve. Milp formulations for the order batching problem in low-level picker-to-part warehouse systems. Master of science thesis, Middle East Technical University, apr 2014. URL https://acikbilim.yok.gov.tr/bitstream/handle/20.500.12812/86973/yokAcikBilim_10035380.pdf?sequence=-1&isAllowed=y. Supervisor: Assoc. Prof. Dr. Temel ”Oncan.

[18] Kaisa Miettinen. *Nonlinear Multiobjective Optimization*. Springer, 1999. ISBN 978-0792374207.

[19] Jean-François Pansart, Stéphane Dauzère-Pérès, and Michel Gourgand. A sparse milp formulation for the order batching problem in warehouses. *European Journal of Operational Research*, 271 (3):1076–1089, 2018. doi:10.1016/j.ejor.2018.06.049. URL https://doi.org/10.1016/j.ejor.2018.06.049.

[20] Guilherme Fialho Costa Pocinho. *Análise e melhoria do processo de order-picking num sistema produtivo: caso de estudo*. PhD thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2013. URL https://run.unl.pt/bitstream/10362/11038/1/Pocinho_2013.pdf.

[21] Edison Rodrigues and Marina Silva. Otimização da eficiência operacional no processo de picking. *Revista do Encontro de Gestão e Tecnologia*, 12(2):45–56, 2022. URL https://revista.fateczl.edu.br/index.php/engetec_revista/article/view/193.

[22] Kees Jan Roodbergen, Iris F. A. Vis, and Jan van den Berg. The integrated orderline batching, batch scheduling, and picker routing problem with multiple pickers. *Flexible Services and Manufacturing Journal*, 33:679–714, 2021. doi:10.1007/s10696-021-09425-8.

[23] Jiun-Yan Shiau and Jie-An Huang. Wave planning for cart picking in a randomized storage warehouse. *Applied Sciences*, 10(8050), 2020. doi:10.3390/app10228050.

[24] T. Öncan. Mathematical models for order batching in warehouses. *International Journal of Production Research*, 53(11):3388–3402, 2015. doi:10.1080/00207543.2014.993958. URL https://doi.org/10.1080/00207543.2014.993958.