Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Algoritmo de Evolução Diferencial em GPU aplicado ao Problema Quadrático de Alocação

Túlio Silva Jardim

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Prof. Dr. Fernando Bernardes de Oliveira

Abril, 2021
João Monlevade–MG

Túlio Silva Jardim

Algoritmo de Evolução Diferencial em GPU aplicado ao Problema Quadrático de Alocação

Orientador: Prof. Dr. Fernando Bernardes de Oliveira

Monografia apresentada ao curso de Sistemas de Informação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto
João Monlevade
Abril de 2021

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

J373a Jardim, Tulio Silva .
Algoritmo de evolução diferencial em GPU aplicado ao problema quadrático de alocação. [manuscrito] / Tulio Silva Jardim. - 2021.
45 f.: il.: color., gráf., tab..

Orientador: Prof. Dr. Fernando Bernardes de Oliveira.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Graduação em Sistemas de Informação .

1. Algorítmos . 2. CUDA (arquitetura de computador). 3. Otimização combinatória. 4. Unidades de processamento gráfico. I. Oliveira, Fernando Bernardes de. II. Universidade Federal de Ouro Preto. III. Título.

CDU 519.1

Bibliotecário(a) Responsável: Flavia Reis - CRB6-2431

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E APLICADAS
DEPARTAMENTO DE COMPUTACAO E SISTEMAS

FOLHA DE APROVAÇÃO

Túlio Silva Jardim

Algoritmo de Evolução Diferencial em GPU aplicado ao Problema Quadrático de Alocação

Monografia apresentada ao Curso de Sistemas de Informação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação

Aprovada em 28 de abril de 2021

Membros da banca

[Doutor] - Fernando Bernardes de Oliveira - Orientador(a) (Universidade Federal de Ouro Preto)
[Doutor] - Wendy Yadira Eras Herrera - (Universidade Federal de Ouro Preto)
[Doutor] - George Henrique Godim da Fonseca - (Universidade Federal de Ouro Preto)

Fernando Bernardes de Oliveira, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 18/06/2021

Documento assinado eletronicamente por **Fernando Bernardes de Oliveira**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 18/06/2021, às 17:54, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0184396** e o código CRC **D29FBCE7**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: - www.ufop.br

"Do the best you can until you know better.
Then when you know better, do better."

— Maya Angelou (1951 – 2014)

Resumo

A utilização de Unidades de Processamento Gráfico (GPUs) para solucionar problemas complexos tem recebido crescente atenção no ambiente acadêmico (e mais recentemente, mineradores de criptomoedas), obtendo resultados cada vez mais rápidos em relação às implementações que utilizam apenas processamento da Unidade Central de Processamento (CPU). Na literatura é comum encontrar relatos de acelerações de 30 vezes na execução de determinados algoritmos pela GPU, principalmente os algoritmos evolucionários. O objetivo deste estudo é verificar o desempenho da GPU para solucionar o Problema Quadrático de Alocação, a partir da utilização do Algoritmo de Evolução Diferencial. Para o desenvolvimento do algoritmo, emprega-se o *framework* ***C****ompute* ***U****nified* ***D****evice* ***A****rchitecture* (CUDA) e a técnica de representação *Relative Positioning Index* – Índice por Posição Relativa (RPI). Neste trabalho são realizados três experimentos baseados na comparação dos três algoritmos desenvolvidos, cada um com uma abordagem de paralelismo diferente, e nas técnicas de exploração de vizinhança implementadas. Por fim, são também comparados os resultados obtidos com os melhores resultados encontrados na literatura, que reforçam a importância de paralelizar algoritmos sempre que for válido, mesmo que estes algoritmos sejam executados apenas pela CPU. A partir de técnicas de paralelismo básicas foram obtidos resultados bem próximos aos que a literatura apresenta, sugerindo que o algoritmo é promissor, mesmo com a diferença entre as representações.

Palavras-chaves: Problema Quadrático de Alocação, Algoritmo de Evolução Diferencial, GPGPU, CUDA.

Abstract

The usage of Graphical Processing Units (GPUs) for solving complex problems gets more and more common between researchers (and, more recently, also criptocurrency miners), ending up in faster and faster results when compared to Central Processing Unit (CPU) only implementations. It is now common to find reports among the literature of 30 times faster execution on GPUs, mainly on evolutionary algorithms. The goal of this study is to verify the GPU's performance on solving the Quadratic Assignment Problem through the usage of the Differential Evolution Algorithm. For the algorithm's development, the framework ***C****ompute* ***U****nified* ***D****evice* ***A****rchitecture* (CUDA) and the representation Relative Positioning Index (RPI) are also needed. The experimentation is based on comparing the three different versions of the algorithm developed, as well as the local search approaches studied. At last, three results are also compared with the best results available on the literature, which reinforce the importance of parallelizing algorithms when it is valid to, even on CPU-only implementations. Through simple parallelism practices results similar to the literature were generated, and it suggests that the proposed algorithm is promising, even though the difference of representations exists.

Key-words: Quadratic Assignment Problem. Differential Evolution Algorithm. GPGPU. CUDA.

Lista de ilustrações

Figura 1 – Ilustração de uma instância do PQA 13
Figura 2 – Funcionamento geral do AED 18
Figura 3 – Representação da execução do algoritmo evolucionário híbrido de Luong, Melab e Talbi (2010) 22
Figura 4 – Diferença de *gaps* (%) entre as três implementações em relação ao tamanho das instâncias 34
Figura 5 – Duração média de execuções (em segundos) em relação ao tamanho das instâncias 35
Figura 6 – Execuções de funções objetivo por segundo em relação ao tamanho das instâncias 36
Figura 7 – *Gap* (%) em relação ao tamanho das instâncias 37
Figura 8 – Tempo de execução (em segundos) em relação ao tamanho das instâncias 38

Lista de tabelas

Tabela 1 – Parâmetros utilizados para experimentos 32
Tabela 2 – Instâncias selecionadas para experimentos 33
Tabela 3 – Média de resultados entre as implementações 34
Tabela 4 – Resultados médios dos três algoritmos de busca 37
Tabela 5 – Comparação dos resultados mínimo e médio produzidos com a literatura 39

Lista de abreviaturas e siglas

ACO *Ant Colony Optimization* – Otimização da Colônia de Formigas

AED Algoritmo de Evolução Diferencial

CPU Unidade Central de Processamento

CUDA ***C**ompute **U**nified **D**evice **A**rchitecture*

GPU Unidade de Processamento Gráfico

GPGPU Unidade de Processamento Gráfico de Propósito Geral

OpenCL ***Open** **C**omputing **L**anguage*

PQA Problema Quadrático de Alocação

QAPLIB *Quadratic Assignment Problem Library* – Biblioteca do PQA

RPI *Relative Positioning Index* – Índice por Posição Relativa

SIMD *Single instruction, multiple data* – Única instrução, múltiplos dados

Sumário

1 INTRODUÇÃO . . . 12
1.1 Descrição do PQA . . . 13
1.1.1 Formulação do problema . . . 14
1.2 Unidade de Processamento Gráfico de Propósito Geral . . . 14
1.3 Objetivos . . . 15
1.4 Metodologia . . . 15
1.5 Organização do trabalho . . . 16

2 REVISÃO BIBLIOGRÁFICA . . . 17
2.1 Aplicações do PQA . . . 17
2.2 Algoritmo de Evolução Diferencial . . . 17
2.2.1 Mutação . . . 19
2.2.2 Cruzamento . . . 19
2.2.3 Seleção . . . 20
2.2.4 Ajuste dos parâmetros . . . 20
2.3 *Relative Positioning Index* – Índice por Posição Relativa . . . 21
2.4 Paralelização por meio da GPU . . . 21
2.5 Trabalhos correlatos . . . 22
2.6 Considerações finais . . . 23

3 DESENVOLVIMENTO . . . 24
3.1 Representação do PQA . . . 24
3.2 AED proposto para o PQA . . . 25
3.2.1 Geração da população inicial . . . 26
3.2.2 Processo de Evolução . . . 26
3.2.3 Implementação do paralelismo . . . 27
3.3 Busca Local . . . 28
3.3.1 *Swap* comum . . . 29
3.3.2 *2-opt swap* . . . 29
3.3.3 Reinserção . . . 29
3.4 Considerações finais . . . 30

4 EXPERIMENTOS COMPUTACIONAIS . . . 31
4.1 Ambiente computacional . . . 31
4.2 Planejamento do experimento . . . 31
4.2.1 Parametrização . . . 32

4.3 Resultados e análises . . . 33
4.3.1 Diferenças entre as implementações . . . 33
4.3.2 Diferenças entre buscas locais . . . 37
4.3.3 Comparação dos resultados com a literatura . . . 38
4.4 Considerações finais . . . 40

5 CONCLUSÃO . . . 41
5.1 Propostas para trabalho futuros . . . 41

REFERÊNCIAS . . . 43

1 Introdução

A programação em sistemas com alto nível de paralelismo foi por um longo período o domínio de relativamente poucos pesquisadores e, portanto, poucas oportunidades de experimentação (RYOO et al., 2008). Por muito tempo, as Unidades de Processo Gráfico (do inglês, *Graphics Processing Unit* – GPU) eram feitas apenas para interpretar sequências binárias da Unidade Central de Processamento (do inglês, *Central Processing Unit* – CPU) e renderizar imagens 2D.

Com o passar dos anos, as GPUs começaram a renderizar também imagens 3D e a ter, cada vez menos, limitações de *hardware*. Após o lançamento dos *frameworks* ***C**ompute **U**nified **D**evice **A**rchitecture* (CUDA) em 2006 e ***O**pen **C**omputing **L**anguage* (OpenCL) em 2008, tornou-se popular o conceito de Unidade de Processamento Gráfico de Propósito Geral (GPGPU) (do inglês, *General-purpose Computing on Graphics Processing Units*) para resolução de diferentes tipos de problemas, os quais eram tradicionalmente solucionados por meio das CPUs (JOUBERT et al., 2012).

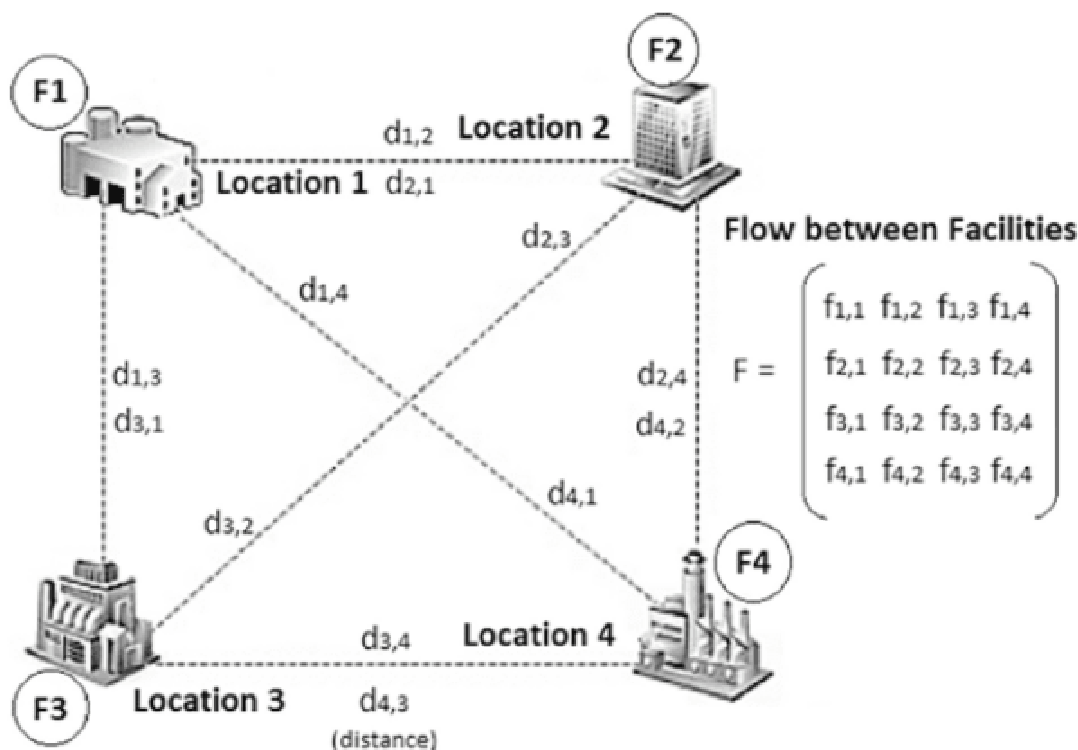
Dentre os diferentes problemas na área de otimização existe o Problema Quadrático de Alocação (PQA). Esse problema foi definido após a idealização de Koopmans e Beckmann (1957) ao estudar um problema de alocação de n facilidades em n localidades, no qual entre cada facilidade há um fluxo e entre cada localidade há uma distância. O objetivo é minimizar os custos relacionados ao posicionamento dessas facilidades, percorrendo distâncias menores e mantendo um alto fluxo.

A Figura 1 apresenta um exemplo de quatro facilidades que devem ser alocadas da melhor maneira possível. Entre cada localidade há uma distância e entre cada facilidade há um fluxo.

Como o PQA é um problema da classe *NP-Difícil*, e ainda não existe um algoritmo capaz de solucioná-lo em tempo polinomial, uma meta-heurística baseada no Algoritmo de Evolução Diferencial (AED) é proposta. Criado por Storn e Price (1997), o AED é um algoritmo de otimização, e é considerado pelos autores como "robusto, fácil de utilizar e se emprega muito bem à computação paralela". De maneira geral, este trabalho tem como objetivo implementar uma versão do AED na CPU e outra na GPU, e avaliar o desempenho entre elas. A validação do modelo proposto será realizada a partir das instâncias disponibilizadas pela *Quadratic Assignment Problem Library* – Biblioteca do PQA (QAPLIB), proposta por Burkard, Karisch e Rendl (1997) e mantida atualmente por professores voluntários.

A seguir, a Seção 1.1 descreve o PQA. Uma introdução à GPGPU é apresentada na Seção 1.2. Os objetivos são descritos na Seção 1.3 e a metodologia abordada ao longo

Figura 1 – Ilustração de uma instância do PQA



Fonte: Retirado de Kilic e Yüzgeç (2018)

do trabalho é relatada na Seção 1.4. A organização do restante do trabalho é apresentada na Seção 1.5.

1.1 Descrição do PQA

O PQA é um problema NP-difícil, o que implica que o tempo para obter uma solução ótima é dado por uma função exponencial do tamanho do problema. Dentre os algoritmos que propõem solucionar o PQA, existem os métodos determinísticos e heurísticos. Os determinísticos são exatos e obtêm soluções ótimas. Entretanto, eles não são capazes de resolver problemas com conjuntos maiores do que 20 pontos, e quando existe esse possibilidade, não são computacionalmente econômicos (MOHAMMADI; MIRZAIE; DERHAMI, 2015).

Já os métodos heurísticos são capazes de encontrar soluções com boa qualidade em um tempo curto, porém sem garantia da otimalidade. Uma vez que a maioria das aplicações do PQA lidam com problemas grandes, a utilização de técnicas meta-heurísticas tornam-se necessárias. No caso dos algoritmos evolucionários, como o AED, são geradas soluções por meio de várias iterações, denominadas gerações. Os algoritmos evolucionários

guiam as soluções passadas entre gerações para explorar o espaço de busca de problemas genéricos.

1.1.1 Formulação do problema

O Problema Quadrático de Alocação pode ser definido como a alocação quadrática a partir de um conjunto de permutações S, na qual cada localidade aloca exatamente uma facilidade e todas as facilidades devem ser alocadas. Ou seja, S é sempre uma ordenação diferente do conjunto original $C = \{1; 2; ...; n-1; n\}$.

A função objetivo é dada pela minimização da soma do custo (o fluxo vezes a distância) entre cada facilidade do problema. Tal função é representada pela Equação 1.1.

$$\min_{S \in C} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{S[i]S[j]} \right) \tag{1.1}$$

Nesta equação, f_{ij} é o fluxo entre i e j e d é a matriz de distância. Para a representar a alocação, $S[i]$ é localidade onde i se insere na solução S. Dessa maneira, é possível verificar que o custo é a soma para cada relacionamento entre as facilidades i e j do fluxo entre facilidades multiplicado por distância entre as alocações selecionadas.

1.2 Unidade de Processamento Gráfico de Propósito Geral

A partir dos anos 2000, as GPUs deixaram de ser relacionadas apenas à renderização rápida de imagens e sua utilização em jogos e se tornaram o centro de atenção da comunidade científica. Isso se deve à adição de pontos flutuantes, o que significava na época que os *overflows* relacionados à aritmética de pontos fixos não era mais um problema. Uma das primeiras tentativas de utilização da GPU para aplicações que não são gráficas foi realizada por Larsen e McAllister (2001 apud DU et al., 2012), sendo calculada a multiplicação de matrizes.

Enquanto as CPUs são fabricadas com o objetivo de realizar diversos tipos de tarefa rapidamente, as GPUs são desenvolvidas para realizar tarefas de maneira concorrente. Hoje em dia, é comum a utilização de processamento pela GPU em trechos de código que se beneficiam de processamento paralelo. Assim, foram desenvolvidas várias tecnologias facilitadoras desse uso, como o CUDA, mantido pela empresa NVIDIA e exclusiva para Unidade de Processamento Gráficos (GPUs) feitas pela mesma, e o OpenCL, que realiza a execução de código em diversas plataformas heterogêneas. O OpenCL foi desenvolvido pela Apple e é mantido pelo Khronos Group[1], uma organização sem fins lucrativos americana.

[1] <https://www.khronos.org/>

1.3 Objetivos

O presente trabalho possui como objetivo geral o estudo das diferenças de processamento entre a CPU e a GPU, bem como identificar como é feita a implementação de algoritmos utilizando ambos dispositivos. Para avaliar o comportamento de tais dispositivos, este trabalho também possui como objetivo a solução de um problema comum da otimização combinatória ao paralelizar o AED.

Este trabalho possui os seguintes objetivos específicos:

1. Definir e implementar um método baseado no Algoritmo de Evolução Diferencial para gerar soluções eficientes para o Problema Quadrático de Alocação tanto na CPU quanto na GPU;
2. Estudar e aplicar mecanismos de busca local para refinamento de soluções;
3. Validar o algoritmo proposto por meio de experimentos computacionais utilizando instâncias de experimentos disponíveis na literatura.

1.4 Metodologia

A metodologia para a execução do projeto consiste em uma pesquisa bibliográfica, para a revisão acerca dos temas envolvidos, bem como uma pesquisa experimental, na qual é avaliada os modelos proposto por meio de instâncias de teste. As atividades deste projeto podem ser descritas como segue:

1. Revisar a literatura sobre o PQA acerca de trabalhos correlatos e demais conceitos, bem como identificar modelos e arquiteturas para execução em GPU;
2. Estudar métodos para representação do problema, identificando abordagens e estruturas de dados específicas para o problema;
3. Definir um modelo e implementar o AED para o problema com execução na CPU e na GPU;
4. Estudar e incorporar mecanismos de busca local aplicados ao problema;
5. Planejar e realizar experimentos para avaliar a performance da meta-heurística proposta;
6. Analisar e discutir os resultados obtidos, além de identificar possíveis melhorias e considerações gerais sobre o processo.

1.5 Organização do trabalho

No Capítulo 2 está contida a revisão bibliográfica acerca das aplicações do Problema Quadrático de Alocação e o funcionamento do Algoritmo de Evolução Diferencial. Trabalhos correlatos e como representar o contexto do PQA para o AED também são descritos. As técnicas utilizadas para o desenvolvimento do trabalho são apresentadas no Capítulo 3. O Capítulo 4 descreve a apresentação e discussão dos resultados obtidos. Por fim, o Capítulo 5 apresenta as principais observações, bem como sugere abordagens para trabalhos futuros.

2 Revisão bibliográfica

Este capítulo apresenta a revisão bibliográfica para o desenvolvimento do trabalho. Foram analisadas, principalmente, as aplicações do PQA (Seção 2.1), o funcionamento do AED (Seção 2.2), como é possível representar o PQA utilizando o Algoritmo de Evolução Diferencial (Seção 2.3), o uso de processamento paralelo por meio da GPU (Seção 2.4) e os trabalhos correlatos (Seção 2.5).

2.1 Aplicações do PQA

Em Jr. e Rahmann (2006), o PQA é utilizado para formulação de microarranjos de DNA, técnica utilizada para pesquisar genes simultaneamente. No artigo de Lin et al. (2018) são citados diversos problemas relacionados à área da educação que podem ser resolvidos por pelo PQA, como problemas de agendamento de horários das aulas e dos professores (*timetabling problems*) e alocação de alunos a projetos, dentre outros.

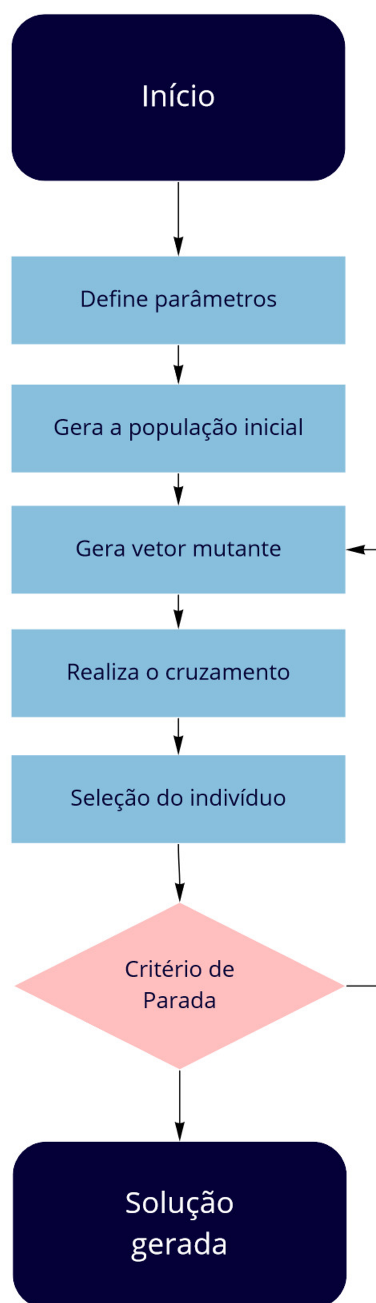
Outros exemplos são a alocação de setores em faculdades, hospitais ou empresas, adaptar dispositivos para terem um design mais ergonômico (por meio da organização de botões em uma central de controle por exemplo) ou até mesmo a distribuição de atletas em uma equipe de revezamento. (ZAIED; EL-FATAH, 2014)

Por possuir um conceito tão abrangente, aplicações para o Problema Quadrático de Alocação surgem em inúmeros processos, assim como o problema de alocação linear (no qual existe apenas uma dimensão de custo, ao invés das dimensões de distância e fluxo).

2.2 Algoritmo de Evolução Diferencial

O Algoritmo de Evolução Diferencial atua sobre a premissa de um algoritmo evolucionário. Nesse algoritmo, é gerada uma população com indivíduos no formato de solução do problema a ser abordado. Em seguida, o *loop* principal do algoritmo será repetido até o limite de gerações estipulado, conforme etapas descritas nas Seções 2.2.1, 2.2.2 e 2.2.3.

Figura 2 – Funcionamento geral do AED



Fonte: Elaborado pelo autor.

A implementação original do AED proposta por Storn e Price (1997) define algumas restrições. Obrigatoriamente, indivíduos (soluções) devem ser representados por um vetor de decimais. Também são definidos os seguintes parâmetros:

- Quantidade de gerações $G \geq 1$;
- Tamanho da população $Np \geq 4$;
- Taxa de cruzamento (decimal) $Cr \in [0; 1]$;
- Fator escalável ou peso diferencial (decimal) $F \in [0; 2]$.

A proposta do algoritmo é executar os procedimentos de **mutação** (Seção 2.2.1), **cruzamento** (Seção 2.2.2) e **seleção** (Seção 2.2.3) por G vezes (referente à cada geração).

2.2.1 Mutação

Os procedimentos de mutação, cruzamento e seleção são realizados para cada indivíduo na posição i em uma matriz representativa da população p. Considerando índices gerados aleatoriamente a, b e c, diferentes do índice i e diferentes entre si, assim como o peso diferencial F citado anteriormente e o tamanho dos indivíduos n, é gerado um novo vetor mutante $m[i]$. A mutação pode ser representada pela Equação 2.1. O índice j representa cada um dos valores no indivíduo i.

$$m[i][j] = p[a][j] + F \cdot (p[b][j] - p[c][j]) \quad \forall \quad 1 \leq j \leq n \tag{2.1}$$

2.2.2 Cruzamento

Imediatamente após a mutação, é necessária a realização do cruzamento para combinar os valores de diferentes indivíduos. Para isso, é criado um vetor de triagem $t[i]$ a partir de uma cópia de $p[i]$, um vetor de decimais aleatórios $r[j] \in [0, 1] \ \forall \ 1 \leq j \leq n$ e um índice também aleatório $R \in \{1, ..., n\}$.

Os índices de r que forem menores ou iguais em relação à Cr serão os índices a serem passados do vetor $m[i]$ para o $t[i]$. O índice R também será passado, como garantia de que ao menos um valor do conjunto mutante vá à triagem. Esse relacionamento é formulado da seguinte maneira:

$$t[i][j] = \begin{cases} m[i][j] & \text{se } (r[j] \leq Cr) \text{ ou } R = j \ , \\ p[i][j] & \text{senão} \end{cases} \quad \forall \quad 1 \leq j \leq n \tag{2.2}$$

2.2.3 Seleção

Antes de prosseguir para a próxima geração, é necessário verificar se realmente houve alguma evolução na geração atual. Para isso, durante o processo de seleção, compara-se os valores do vetor $p[i]$ e o vetor t perante a função objetivo. Caso t possua uma performance superior, ele substituirá $p[i]$ durante a próxima geração.

Como é possível observar na função objetivo representada pela Equação 1.1, soluções do PQA são apresentadas a partir de um conjunto de inteiros S, enquanto as soluções geradas pelo AED são os conjuntos de decimais $p[i]$. Portanto, para solucionar a função objetivo do PQA é necessário equivaler o conjunto $p[i]$ em uma solução π. Isso é realizado a partir de algoritmos de representação.

2.2.4 Ajuste dos parâmetros

Os parâmetros do AED podem ser definidos por abordagens de *parameter tuning* ou *parameter control*. Na primeira abordagem, os valores são constantes definidas antes da execução do algoritmo. Na segunda, existem valores iniciais para os parâmetros, mas eles são alterados durante a execução, através de métodos determinísticos, adaptativos ou auto-adaptativos (LIU; LAMPINEN, 2005).

Independente da abordagem, existem alguns critérios de escolha durante a parametrização que oferecem um alto retorno performático. Quanto maior o valor de F, por exemplo, maior a chance de se escapar de um ótimo local, mas esse valor deve ser ajustado conforme a função objetivo e os demais parâmetros (GÄMPERLE; MÜLLER; KOUMOUTSAKOS, 2002). De acordo com a análise de Wu et al. (2016), os valores comuns utilizados pela literatura são entre $F \in [0{,}4; 0{,}95]$.

Storn (2008) sugere valores iniciais para o Cr de $[0{,}8; 1)$, devido à uma alta convergência. Porém Gämperle, Müller e Koumoutsakos (2002) alertam que uma alta convergência pode convergir a população prematuramente, em torno de um mínimo local. Portanto, os autores indicam que os valores ideais geralmente se localizam entre $[0{,}3; 0{,}9]$. Em Piotrowski (2017), são explorados vários cenários em relação ao tamanho da população Np. Algumas das conclusões do autor são:

- $Np < 50$ é raramente recomendado até mesmo para problemas pequenos.
- $Np = 100$ é geralmente uma boa escolha para problemas de $n < 30$.
- Para problemas $n \geq 30$, a recomendação é experimentar em torno de $Np \in [3n; 10n]$.

Os valores sugeridos são ideais como valores iniciais apenas. Uma parametrização correta depende de muito estudo e experimentação, conforme analisado em Gämperle, Müller e Koumoutsakos (2002).

2.3 *Relative Positioning Index* – Índice por Posição Relativa

Como a versão canônica do AED utiliza representação real, alguma estratégia deveria ser utilizada para abordar problemas combinatórios, como o PQA. Em Alves (2016) são listados diversos destes algoritmos para representar o AED no contexto discreto. Dentre as possíveis abordagens existe o *Relative Positioning Index* – Índice por Posição Relativa (RPI) (também conhecido como *largest-order-value rule* (QIAN et al., 2008)). A representação RPI é uma função que transforma conjuntos de números reais em conjuntos de números inteiros. Para isto, o menor valor real de um conjunto (denominado neste exemplo como S_d) é transformado no menor inteiro pertencente à um outro conjunto, denominado S. Após esse relacionamento entre os dois menores valores, ele se repete no segundo menor valor de cada conjunto e assim em diante até o n-ésimo valor.

Por exemplo, caso $S = [1; 2; 3; 4]$ e se considere uma possível solução na qual $S_d = [0{,}37; 0{,}18; 0{,}98; 0{,}64]$, ao executar a função de RPI, o comportamento é: $f_{RPI}(S_d) \rightarrow [2; 1; 4; 3]$.

2.4 Paralelização por meio da GPU

Para a programação para GPUs é utilizado o conceito de *Single instruction, multiple data* – Única instrução, múltiplos dados (SIMD), o que é ideal para resolver a mesma instrução em dados similares. Para o AED, esse conceito pode ser aplicado para paralelizar as operações que se repetem para cada indivíduo de uma população, durante a geração da população e durante cada geração.

Na implementação de CUDA para C/C++, temos três tipos de funções (VERONESE; KROHLING, 2010):

- **Funções hospedeiras:** Chamadas e executadas pela CPU.
- **Funções *kernel*:** Chamadas pela CPU e executadas pelo dispositivo (GPU). Tais funções utilizam o modificador de acesso __*global*__ antes do tipo de retorno da função, obrigatoriamente *void*.
- **Funções de dispositivo:** Chamadas e executadas pela GPU. Utilizam o modificador de acesso __*device*__ e não possuem restrição de retorno.

Tais funções definem um padrão de desenvolvimento no qual uma função hospedeira pode invocar diversas funções *kernel* em paralelo. E as funções *kernel*, por sua vez, podem invocar apenas funções de dispositivo, que são executadas na mesma *thread* que a função invocadora.

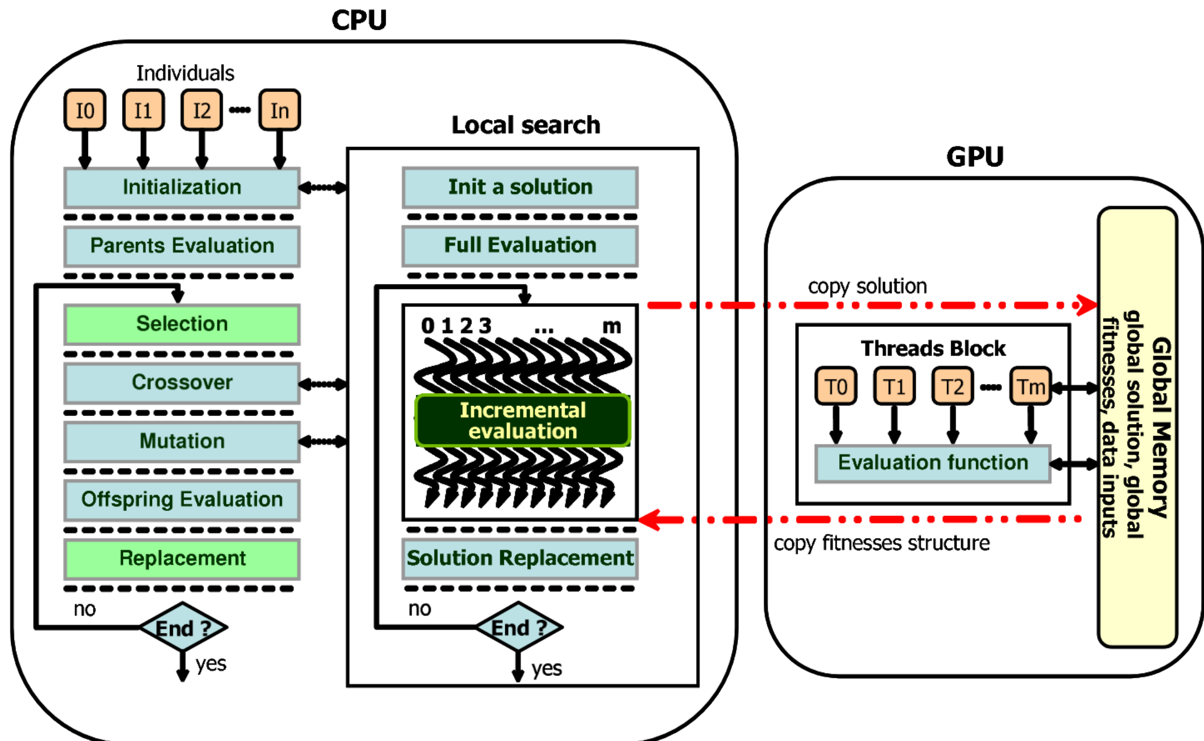
De acordo com Nvidia (2021), as *threads* são distribuídas em blocos de até 1.024 *threads* que comunicam entre si. Também é estabelecido o conceito de *grids*, sendo que cada *grid* possui vários blocos de *threads*.

Em CUDA, é possível invocar uma função *kernel* $fkernel()$ com bl blocos e th *threads* por bloco a partir da sintaxe $fkernel <<< bl, th >>> ()$.

2.5 Trabalhos correlatos

Meta-heurísticas híbridas são os algoritmos que combinam uma meta-heurística, como um algoritmo evolucionário,com um método de busca local (LUONG; MELAB; TALBI, 2010). Em Luong, Melab e Talbi (2010) é abordado um algoritmo evolucionário híbrido cuja função objetivo é executada pela GPU. O autor executa a função objetivo sempre dentro de uma busca local. Isso significa que a busca local é executada durante a inicialização de valores, durante o cruzamento e durante a mutação, conforme ilustrado na Figura 3.

Figura 3 – Representação da execução do algoritmo evolucionário híbrido de Luong, Melab e Talbi (2010)



Fonte: Luong, Melab e Talbi (2010, p. 5)

O AED é utilizado em conjunto com CUDA no trabalho de Veronese e Krohling (2010). Os autores do trabalho descrevem que os algoritmos evolucionários possuem

paralelismo inerentemente. Portanto, eles podem apresentar grande vantagem ao solucionar problemas custosos. No trabalho foram obtidas melhoras de $9,91$ até $35,48$ vezes no tempo de processamento. Veronese e Krohling (2010) não citam nenhuma técnica de busca local e não abordam nenhum problema de otimização específico. Os experimentos foram realizados sobre seis funções de objetivo escolhidas.

O algoritmo *Ant Colony Optimization* – Otimização da Colônia de Formigas (ACO), proposto no começo dos anos 1990 (DORIGO; CARO, 1999), faz uma referência a uma colônia de formigas, pois se comunicam ao deixar rastros de informações pelos caminhos visitados, interpretados por meio de grafos. O algoritmo é paralelizado desde 1998 (STÜTZLE, 1998) e é utilizado por Tsutsui (2012) para solucionar o PQA. No trabalho, é utilizado um computador com 4 GPUs do modelo GTX 480 para solucionar instâncias no tamanho $n \in [40; 150]$. Os resultados sugerem, em média, uma aceleração de $23,8$ vezes em relação ao cálculo utilizando apenas a CPU (TSUTSUI, 2012).

Um novo algoritmo genético é proposto por Mohammadi, Mirzaie e Derhami (2015), no qual metade da população é selecionada como os melhores resultados até então e a outra metade é selecionada a partir de indivíduos aleatórios, evitando assim a convergência prematura. Tal algoritmo é adaptado para operar paralelamente por meio da GPU, no qual os autores reportaram resultados 8 vezes mais rápidos em um problema de $n = 12$ e até $31,5$ vezes mais rápidos em um problema $n = 100$ em relação ao algoritmo genético convencional. A melhora na velocidade aumentou proporcionalmente com o tamanho do problema. Aqueles autores ainda propõem como trabalhos futuros a utilização de busca local no algoritmo genético proposto para melhorar os resultados.

2.6 Considerações finais

Apesar do conceito do PQA ser simples, observa-se pela revisão bibliográfica que o solução do problema não é trivial. Várias características foram avaliadas sobre o AED, como a parametrização, a representação necessária, bem como sobre os conceitos de paralelismo por GPU. Com os trabalhos correlatos (Seção 2.5), foram avaliadas algumas implementações similares para o problema. De modo geral, o PQA é amplamente explorado na literatura por meio de diferentes abordagens.

3 Desenvolvimento

A ideia principal deste trabalho é estudar o GPGPU ao se implementar algoritmos em CPU e GPU, bem como e avaliar as diferenças de comportamento perante a um problema clássico de otimização. Nesse caso, será utilizado o PQA.

A Seção 3.1 define o modelo das instâncias do PQA e o método de representação. Nas Seção 3.2 estão detalhes da implementação do AED e na Seção 3.3 são explorados três implementações de busca local a serem avaliados Capítulo 4.

3.1 Representação do PQA

A representação do Problema Quadrático de Alocação selecionada se baseia nos dados disponibilizados pela QAPLIB. A base de dados conta com 138 instâncias do problema. Para este trabalho, foi desenvolvida uma função capaz de ler determinada instância e alocar junto ao código utilizando a seguinte estrutura de dados:

$$estrutura \quad instancia \leftarrow \left\{ \begin{array}{ll} inteiro & n \\ inteiro & distancia[n][n] \\ inteiro & fluxo[n][n] \\ inteiro & solucaoLiteratura[n] \\ inteiro & custoLiteratura \end{array} \right\} \tag{3.1}$$

Na estrutura, a variável n representa o tamanho do problema. Em seguida são populadas as matrizes de distância e de fluxo, ambas com dimensões de $n \times n$. A base do QAPLIB também disponibiliza a melhor solução descoberta pela literatura para comparação.

Como a operação de mutação do AED gera soluções em decimais (conforme Seção 2.2.1) e o PQA possui soluções limitadas a inteiros entre $[1; n]$, para que o problema seja adaptado ao AED é utilizada a representação RPI, conforme exemplo apresentado na Seção 2.3. Assim, cada indivíduo da população é um conjunto de números decimais, denominado por S_d.

As soluções do PQA são representadas por permutações do conjunto $S = \{1; 2; ...; n-1; n\}$, no qual $S[i]$ é a facilidade atribuída a i. Para este trabalho, o conjunto S é utilizado apenas durante o cálculo da função objetivo, portanto o RPI ocorre somente na execução da $funcaoObjetivo(S_d, Inst)$, conforme o Algoritmo 1.

Algoritmo 1 Cálculo da Função Objetivo

```
1: função FUNCAOOBJETIVO(S_d, Inst)
2:     S ← RPI(S_d, Inst.n)
3:     custo ← 0
4:     para i ← 1 até Inst.n faça
5:         para j ← 1 até Inst.n faça
6:             custo ← custo + Inst.fluxo[i][j] × Inst.distancia[S[i]][S[j]]
7:         fim para
8:     fim para
9:     S ← ∅
10:    devolve custo
11: fim função
```

Nota-se que S é alocado apenas durante o cálculo da Equação 1.1. O pseudocódigo para o RPI, o qual transforma de valores decimais para inteiros, é apresentado no Algoritmo 2. A função recebe como parâmetros o conjunto da solução em decimal S_d e o tamanho do problema n. A variável $maiorQueEu$ inicia com o valor de n e diminui em um para cada índice maior que o índice atual. Caso S_d possua decimais iguais entre si, o segundo *loop* altera o valor dos elementos duplicados, para garantir os princípios citados na Seção 1.1.1.

Algoritmo 2 *Relative Positioning Index*

```
1: função RPI(S_d, n)
2:     para i ← 1 até n faça
3:         maiorQueEu ← n
4:         para j ← 1 até n faça
5:             se i ≠ j e S_d[i] ≤ S_d[j] então
6:                 maiorQueEu ← maiorQueEu − 1
7:             fim se
8:         fim para
9:         S[i] ← maiorQueEu
10:    fim para
11:    para i ← 1 até n faça
12:        para j ← 1 até n faça
13:            se S[i] = S[j] então
14:                S[j] ← S[j] − 1                    ▷ Corrige índices iguais
15:            fim se
16:        fim para
17:    fim para
18:    devolve S
19: fim função
```

3.2 AED proposto para o PQA

Nesta seção é apresentada a abordagem proposta para o PQA. O algoritmo é executado logo após a declaração da instância escolhida, e se inicia pela geração da população inicial (Seção 3.2.1). Em seguida, para cada indivíduo em cada uma das G

gerações definidas é executado o processo de evolução (Seção 3.2.2). Por fim, é retornada a solução que gerou o custo mínimo dentre a última evolução.

3.2.1 Geração da população inicial

A partir da representação decimal, os valores para cada um dos N_p indivíduos são gerados aleatoriamente. Para isso, é considerado um intervalo inicial $[inf, sup]$ para cálculo. Esse intervalo é utilizado apenas na geração da população inicial. No decorrer do processo, esse intervalo não é verificado.

A escolha desse intervalo não afeta significativamente o algoritmo. Com o passar das gerações e pelo peso diferencial (parâmetro F), esse valor é incrementado, e, caso o intervalo inicial seja muito esparso, eles podem ultrapassar os limites inferiores e superiores conforme o tipo de dado utilizado. Assim, é importante observar algum mecanismo de correção para os valores, se necessário. Neste projeto não observou-se a necessidade de implementar isso em virtude do intervalo adotado.

3.2.2 Processo de Evolução

A função de evolução implementa as operações de mutação, cruzamento e seleção (Seção 2.2). Para refinamento dos resultados obtidos em cada geração também é comum a utilização de mecanismos de busca local entre as operações de cruzamento e seleção (Seção 3.3).

No Algoritmo 3, Pop se refere à uma matriz na qual cada conjunto é um indivíduo (ou solução) S_d. Também é utilizada a matriz Pop_2, que se refere à população da geração $G+1$ (Seção 3.2.3). N_p é o tamanho da população, enquanto i é o identificador do indivíduo S_d em Pop. Também é passado o objeto de dados da instância $Inst$, os parâmetros de peso diferencial F, taxa de cruzamento Cr e a taxa de ocorrência de busca local Br. Os custos obtidos por cada indivíduo são armazenados no conjunto $Custos$.

Primeiro, são selecionados aleatoriamente os índices mutantes a, b e c, que devem ser valores diferentes de i e diferentes entre si. Em seguida, para cada índice do vetor de triagem t, caso sejam satisfeitos os critérios de cruzamento, $t[k]$ recebe o resultado da mutação. A busca local também possui um critério de ocorrência. Caso um decimal aleatório $U\ [0,0;\ 1,0]$ seja menor que Br, $custo$ e t recebem os valores calculados por $bestFirst(t, Inst)$ (Seção 3.3). Caso contrário, o custo é calculado diretamente por $funcaoObjetivo(t, Inst)$. Caso o $custo$ seja menor que o valor anterior de $Custos[i]$, t substitui o $Pop[i]$ na próxima geração e o $Custos[i]$ é atualizado.

Algoritmo 3 Processo de Evolução

```
1: função EVOLUTION(Pop, N_p, i, Inst, F, Cr, Br, Custos, Pop_2)
2:     repita
3:         a ← randInt(1, N_p)
4:         b ← randInt(1, N_p)
5:         c ← randInt(1, N_p)
6:     até que i ≠ a ≠ b ≠ c      ▷ Escolha de índices mutantes, diferentes entre si e entre i
7:     R ← randInt(1, Inst.n)
8:     t ←Ø
9:     para k ← 1 até Inst.n faça
10:        se randDec(0,0; 1,0) < Cr ou R = k então      ▷ Mutaciona se necessário
11:            t[k] ← Pop[a][k] + (F × (Pop[b][k] − Pop[c][k]))
12:        senão
13:            t[k] ← Pop[i][k]
14:        fim se
15:    fim para
16:    se randDec(0.0, 1.0) < Br então
17:        custo, t ← bestFirst(t, Inst)      ▷ Caso ocorra busca local
18:    senão
19:        custo ← funcaoObjetivo(t, Inst)      ▷ Caso não ocorra busca local
20:    fim se
21:    se custo < Custos[i] então
22:        Pop_2[i] ← t
23:        Custos[i] ← custo
24:    senão
25:        Pop_2[i] ← Pop[i]
26:    fim se
27:    devolve Pop_2
28: fim função
```

3.2.3 Implementação do paralelismo

O paralelismo é obtido ao aplicar o conceito de SIMD. Isto é realizado ao manipular as *threads* da CPU ou da GPU neste trabalho. Portanto, o trabalho possui duas versões de implementação, uma para cada tipo de manipulação. A diferença entre as duas versões está em como a função de geração da população e a função de evolução estão implementadas.

Em ambas versões, são alocadas N_p *threads* para realizar essas funções, e em seguida elas são sincronizadas. Para evitar conflitos entre leitura e escrita das *threads*, no Algoritmo 3 *Pop* é atualizado com os valores de Pop_2 apenas após o sincronismo das *threads*.

A implementação deste trabalho foi feita com a Linguagem C++. A manipulação das *threads* da CPU é feita a partir da classe nativa `std::thread`, enquanto a implementação via GPU é feita a partir do CUDA. Como referência para o desenvolvimento foram utilizados os trabalhos Ran (2017) e Veronese e Krohling (2010). Na primeira referência, os *kernels* são chamados utilizando sempre 1 bloco de N_p *threads*. Na segunda, são utilizados diversos blocos. Com base nas duas abordagens foram criadas para este trabalho duas

implementações. A versão denominada apenas como implementação por GPU invoca sempre um bloco. A segunda versão utiliza n^2 blocos de N_p *threads* para calcular a função objetivo e será descrita como GPU-P. Para esta segunda versão, a função objetivo recebe o vetor de solução após o cálculo de RPI e o endereço de memória da variável *custo*. Portanto, nessa versão, a função realiza apenas a operação atômica referente à linha 6 do Algoritmo 1. Como são n^2 blocos invocados e cada bloco possui um índice $blockIdx.x$, o índice i recebe $blockIdx.x$ / n, enquanto j recebe $blockIdx.x$ % n.

3.3 Busca Local

Algoritmos de busca local possuem o objetivo de refinar a solução obtida após cada iteração. Para este trabalho foram analisados três movimentos de busca local. Todos os três foram formulados a partir da estratégia *best-first* (DAVENDRA; BIALIC-DAVENDRA; METLICKA, 2020), na qual a função termina de executar a partir do primeiro resultado de melhora, ou após explorar todas as alternativas. Tais movimentos foram escolhidos com base na representação escolhida e em trabalhos correlatos.

O pseudocódigo para o processo de busca local é apresentado no Algoritmo 4, onde as funções de $troca(S_d, i, j)$ e $destroca(S_d, i, j)$ se referem ao movimento analisado em questão. Tais movimentos serão analisados nas Seções 3.3.1, 3.3.2 e 3.3.3. A função $troca(S_d, i, j)$ tem o objetivo de avaliar o custo vizinho, enquanto $destroca(S_d, i, j)$ retorna à solução inicial, caso o custo vizinho não seja favorável.

Algoritmo 4 Algoritmo *best-first*

```
1:  função BESTFIRST(S_d, n)
2:      S ← RPI(S_d, Inst → n)
3:      custo ← funcaoObjetivo(S_d, Inst)
4:      para i ← 1 até n − 1 faça
5:          para j ← i + 1 até n faça
6:              S_d ← troca(S_d, i, j)
7:              S ← RPI(S_d, Inst → n)
8:              custoV ← funcaoObjetivo(S_d, Inst)
9:              se custoV < custo então
10:                 devolve custoV, S_d
11:             fim se
12:             S_d ← destroca(S_d, i, j)
13:         fim para
14:     fim para
15:     devolve custo, S_d
16: fim função
```

Os parâmetros S_d e $Inst$ são respectivamente a solução a ser avaliada e a instância definida (Equação 3.1). As variáveis *custo* e $custoV$ se referem ao custo da solução inicial

e o custo da solução vizinha explorada, de acordo com o Algoritmo 2 e a função objetivo (Algoritmo 1).

3.3.1 *Swap* comum

O movimento de *swap* comum é a implementação mais básica de um algoritmo de busca local. Neste movimento, apenas são trocadas as facilidades i e j. Essa mesma função é capaz de substituir as duas funções $troca(S_d, i, j)$ e $destroca(S_d, i, j)$ do Algoritmo 4.

Algoritmo 5 Algoritmo de *Swap* comum

```
1: função swapComum(S_d, i, j)
2:     aux ← S_d[i]
3:     S_d[i] ← S_d[j]
4:     S_d[j] ← aux
5:     devolve S_d
6: fim função
```

3.3.2 *2-opt swap*

O movimento de *2-opt swap* é uma das mais famosas implementações de busca local. O algoritmo foi proposto por Croes (1958) e consiste em realizar um *swap* comum entre duas posições i e j e inverter a posição dos elementos entre eles. O Algoritmo 6 realiza esse procedimento ao invocar o Algoritmo 5 para elementos entre i e j. Esse algoritmo também é capaz de substituir as duas funções $troca(S_d, i, j)$ e $destroca(S_d, i, j)$ no algoritmo *best-first*.

Algoritmo 6 Algoritmo *2-opt swap*

```
1: função 2optSwap(S_d, i, j)
2:     a ← i
3:     b ← j
4:     para k ← 1 até (j − i + 1)/2 faça
5:         S_d ← swapComum(S_d, a, b)
6:         a ← a + 1
7:         b ← b − 1
8:     fim para
9:     devolve S_d
10: fim função
```

3.3.3 Reinserção

O movimento de *reinserção* remove o elemento na posição i e o reinsere no índice j, após reorganizar os elementos entre os dois índices. Na busca local, o Algoritmo 7 substitui a função $troca(S_d, i, j)$.

Algoritmo 7 Algoritmo de reinserção

```
1: função swapReinsercao(S_d, i, j)
2:     aux ← S_d[i]
3:     para k ← i + 1 até j faça
4:         S_d[k − 1] ← S_d[k]
5:         se k = j então
6:             S_d[k] ← aux
7:         fim se
8:     fim para
9:     devolve S_d
10: fim função
```

O processo que desfaz o movimento de reinserção ao substituir a função $destroca(S_d, i, j)$ é apresentado no Algoritmo 8.

Algoritmo 8 Algoritmo para inverter reinserção

```
1: função swapReinsercaoInverso(S_d, i, j)
2:     aux ← S_d[j]
3:     para k ← j − 1 até i faça
4:         S_d[k + 1] ← S_d[k]
5:         se k = i então
6:             S_d[k] ← aux
7:         fim se
8:     fim para
9:     devolve S_d
10: fim função
```

3.4 Considerações finais

Este capítulo apresentou a abordagem proposta para o PQA tem como base o AED. A representação para o problema foi estabelecida, considerando o contexto combinatório do problema. Além disso, foi definido a estrutura principal do algoritmo, com as estratégias de busca local para refinamento das soluções.

O capítulo seguinte apresentará os experimentos realizados para validar o algoritmo proposto, bem como as devidas análises e considerações.

4 Experimentos Computacionais

Este capítulo descreve os experimentos do trabalho. Na Seção 4.1, as especificações do ambiente computacional utilizado para os experimentos são relatadas. A Seção 4.2 descreve a seleção das técnicas utilizadas para os experimentos e a Seção 4.3 discute os resultados obtidos.

4.1 Ambiente computacional

Os experimentos do algoritmo foram realizados por meio de um computador com as seguintes especificações:

- **Sistema Operacional:** Ubuntu 20.04.2 LTS (64 bits)
- **Processador:** AMD® Ryzen 5 3600x 6-core processor × 12
- **Memória RAM:** 16 GB
- **Placa gráfica:** NVIDIA Corporation TU104 [GeForce RTX 2060], com 1.920 CUDA Cores e 6 GB de memória RAM GDDR6.

4.2 Planejamento do experimento

Ambas as versões do algoritmo foram projetadas a fim de facilitar os experimentos. A função inicial do código recebe todos os parâmetros necessários e a quantidade de experimentos a serem realizados. Em seguida, as instâncias são executadas de maneira intercalada e os resultados são armazenados em arquivos *.tsv* (*tab-separated values*).

Os arquivos gerados possuem como informações o nome da instância, o *gap* (calculado pela Equação 4.1) dos resultados, a quantidade de vezes que a função objetivo é chamada, o tempo de execução, a melhor solução e custo encontrados, o tipo de implementação selecionada, o horário de início da execução e os demais parâmetros.

$$GAP(\%) = \frac{\text{Solução Obtida} \times \text{Solução da Literatura}}{\text{Solução da Literatura}} \times 100 \tag{4.1}$$

Os experimentos realizados foram planejados a fim de cumprir os três objetivos do trabalho:

- **Definir e implementar um método baseado no AED para gerar soluções eficientes para o PQA tanto na CPU quanto na GPU**: As implementações em ambos os ambientes são comparadas na Seção 4.3.1;
- **Estudar e aplicar mecanismos de busca local para refinamento de soluções**: São avaliados na Seção 4.3.2 três algoritmos de busca local escolhidos;
- **Validar o algoritmo proposto por meio de experimentos computacionais utilizando instâncias de experimentos disponíveis na literatura**: A eficácia do algoritmo é avaliada na Seção 4.3.3.

4.2.1 Parametrização

Como o foco do trabalho é o comportamento geral dos algoritmos estudados, não foram testados os parâmetros ideais para cada instância selecionada. Portanto, a parametrização se baseia nos padrões recomendados pela literatura (Seção 2.2.4). Para o tamanho da população optou-se pelo mesmo valor que o limite de *threads* por bloco estabelecida pelo CUDA, até mesmo nas execuções via CPU, uma vez que o trabalho lida com instâncias de n até 256 e todos os indivíduos da população irão atuar em paralelo.

Os parâmetros da Tabela 1 serão utilizados durante todas as análises. Conforme o contexto, as demais peculiaridades serão abordadas em cada parte da Seção 4.3.

Tabela 1 – Parâmetros utilizados para experimentos

Parâmetros	
Cr	0,9
F	0,7
N_p	1.024
Limite de execuções da função objetivo	150.000.000
Intervalo de valores para elementos inicializados	[-3,0; 3,0]

4.3 Resultados e análises

Nas seções a seguir são apresentados os dados relacionados às análises citadas. Cada experimento realizado nas instâncias selecionadas se baseiam em uma amostra de 10 execuções, sempre intercaladas com outras instâncias. Nas mesmas seções também estão contidas as análises dos resultados obtidos.

4.3.1 Diferenças entre as implementações

Para este experimento, foram selecionadas 14 instâncias das 138 disponíveis no QAPLIB. Na seleção foi priorizada a diversidade de tamanhos, inclusive as duas maiores (*tai256c* e *tho150*). Para facilitar a visualização dos dados, os gráficos e tabelas referentes à esta análise exibem apenas o tamanho das instâncias.

Tabela 2 – Instâncias selecionadas para experimentos

Nome	**Tamanho** n
chr12c	12
els19	19
bur26a	26
ste36a	36
tho40	40
sko56	56
lipa60a	60
tai64c	64
sko72	72
lipa90a	90
wil100	100
esc128	128
tho150	150
tai256c	256

Fonte: QAPLIB

Para esta análise, foram selecionadas as implementações de CPU, GPU e GPU-P (definidas na Seção 3.2.3). Em GPU-P, a função *kernel* de evolução é dividida. A primeira função, que utiliza o mesmo bloco, realiza a mutação, crossover e gera também S através do RPI. Em seguida é necessário realizar o sincronismo das *threads*, pois a função de custo se torna outra função *kernel* com n^2 blocos de N_p *threads*. É sincronizado novamente e processo de seleção então é realizado em outra função *kernel*, de apenas 1 bloco.

A Tabela 3 contém os principais indicadores obtidos para essa análise após execuções com o critério de parada de 1.000 gerações. Para melhor visualização, os resultados também são representados por meio das Figuras 4, 5 e 6.

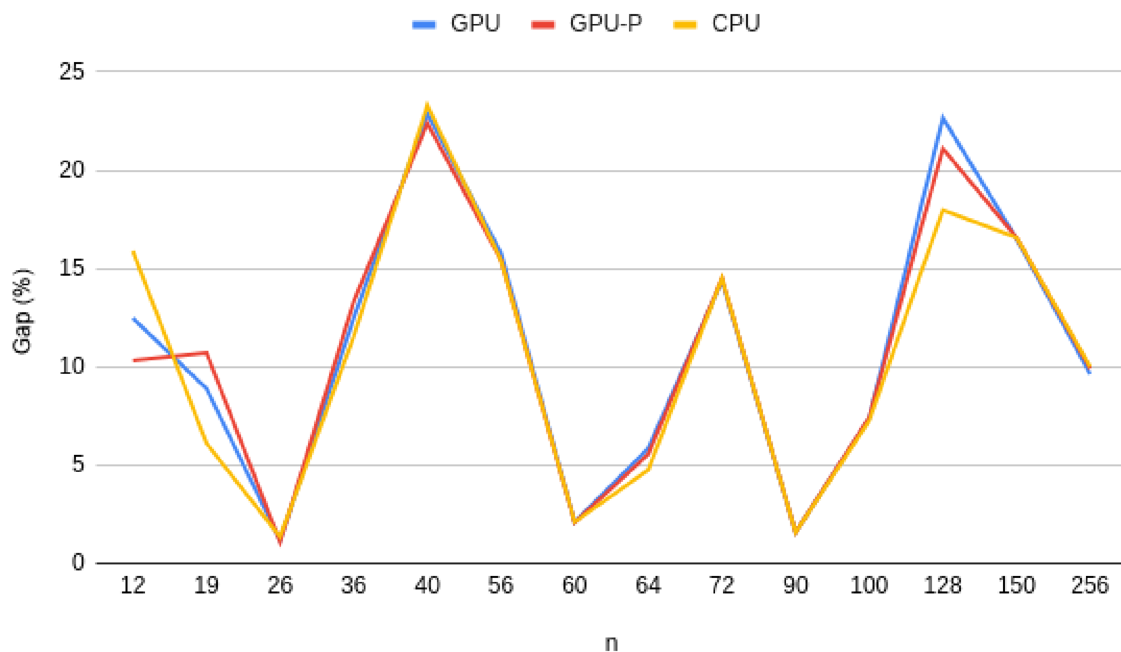
Tabela 3 – Média de resultados entre as implementações

n	Gap (%)			Tempo (s)			Chamadas de FO (1000/s)		
	GPU	**GPU-P**	**CPU**	**GPU**	**GPU-P**	**CPU**	**GPU**	**GPU-P**	**CPU**
12	12,46	**10,32**	15,88	**1,13**	1,80	16,39	910,75	1.263,18	**62,53**
19	8,90	10,71	**6,11**	1,87	**1,20**	16,24	547,40	856,31	**63,12**
26	1,15	**1,08**	1,35	2,71	**1,95**	16,24	378,08	526,55	**63,13**
36	12,48	13,32	**11,54**	4,31	**3,35**	16,42	237,68	306,42	**62,44**
40	22,86	**22,38**	23,29	5,54	**4,09**	16,70	185,17	250,87	**61,36**
56	15,79	15,43	**15,42**	9,71	**7,34**	16,60	105,57	139,72	**61,76**
60	2,09	**2,08**	2,10	13,19	**8,77**	16,71	77,71	116,84	**61,36**
64	5,84	5,55	**4,76**	14,77	**11,24**	16,82	69,39	91,24	**60,97**
72	**14,37**	14,47	14,48	18,15	**12,31**	17,33	**56,47**	83,29	59,18
90	**1,57**	1,58	1,58	27,85	18,73	**18,10**	**36,81**	54,73	56,64
100	7,38	7,43	**7,26**	33,18	22,80	**18,48**	**30,89**	44,96	55,46
128	22,66	21,09	**17,97**	56,05	41,99	**18,83**	**18,29**	24,41	54,43
150	**16,50**	16,58	16,57	89,85	54,61	**20,72**	**11,41**	18,77	49,49
256	**9,62**	9,90	9,99	299,33	106,70	**36,44**	**3,42**	9,61	28,13
Méd.	**10,97**	**10,85**	**10,59**	**41,57**	**21,20**	**18,71**	**190,64**	**270,49**	**57,14**

Fonte: Elaborado pelo autor.

A Figura 4 foi elaborada para avaliar a eficácia das implementações, uma vez que as três versões devem seguir o mesmo algoritmo.

Figura 4 – Diferença de *gaps* (%) entre as três implementações em relação ao tamanho das instâncias

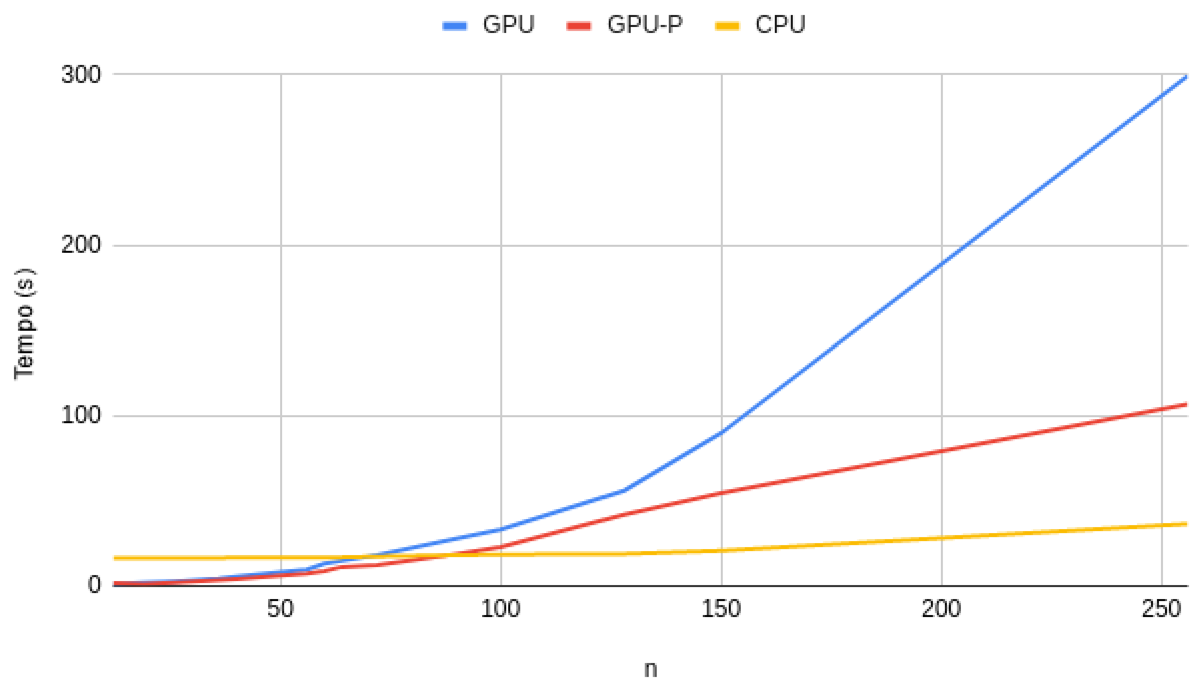


Fonte: Elaborado pelo autor.

Considerando o contexto experimental estabelecido, os resultados sugerem que existe uma similaridade entre os *gaps* médios, com exceção das instâncias *chr12c*, *els19* e *esc128*. Ao analisar cada resultado gerado por essas instâncias, verifica-se que há pouca diversidade nos valores gerados. Nesta análise, isto pode indicar que o algoritmo tende a convergir nos mesmos ótimos locais, o que poderia explicar o distanciamento entre o *gap* médio de cada implementação.

As Figuras 5 e 6 apresentam duas visualizações sobre o mesmo comportamento. Na Figura 5 é representada a média de tempo por cada execução. Para a CPU, os resultados sugerem uma pequena variação no tempo de execução. Entre as instâncias de tamanho 12 e 128, esse tempo está entre 16 e 19 segundos.

Figura 5 – Duração média de execuções (em segundos) em relação ao tamanho das instâncias



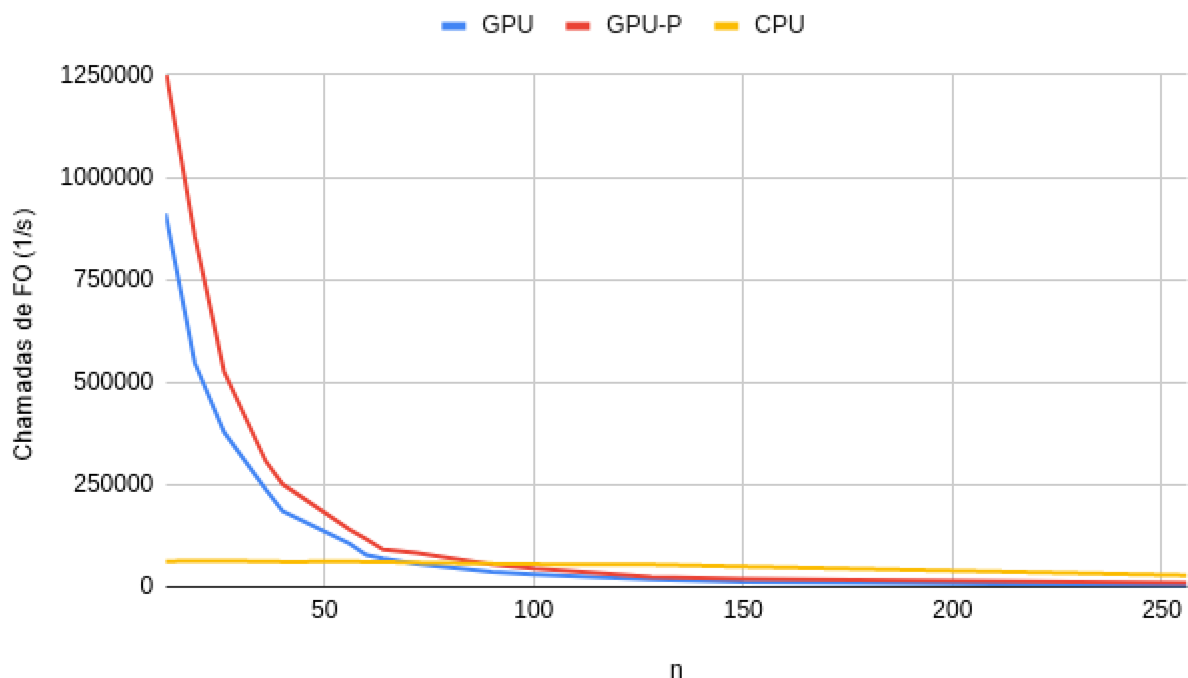
Fonte: Elaborado pelo autor.

A função objetivo é o maior gargalo do algoritmo. Ela é executada N_p vezes em cada geração G e possui complexidade $O(n^2)$. Como a vantagem de processamento de uma GPU está no paralelismo, executar tal função pode ser um desafio na implementação GPU, que utiliza a mesma *thread* durante toda a evolução. Já a implementação GPU-P conta com a complexidade de $O(1)$ no cálculo da função objetivo, pois ela é chamada para n^2 blocos de N_p threads em cada geração G. Apesar de se comportar melhor que a implementação inicial por GPU, essa implementação também sofre atrasos em problemas maiores. Isso

pode ser justificado devido à maior demanda de sincronismo das *threads*. A granularidade do paralelismo pode influenciar no processamento. Assim, esse comportamento pode ser avaliado em trabalhos futuros.

A Figura 6 ilustra a quantidade de chamadas na função objetivo por segundo entre as instâncias. As retas apresentam o mesmo comportamento que a figura anterior, já que todas as execuções continham 1.025.024 chamadas à função objetivo, referente à N_p chamadas durante a geração da população somadas das $N_p \times G$ chamadas durante a evolução. Ainda naquela figura, os resultados sugerem que a GPU é mais eficiente que a CPU entre as instâncias de tamanho 12 e 64. Já para a implementação GPU-P, a instância de tamanho 72 também é favorável sobre a CPU.

Figura 6 – Execuções de funções objetivo por segundo em relação ao tamanho das instâncias



Fonte: Elaborado pelo autor.

Em todos os casos, os resultados sugerem que a implementação GPU-P possui maior performance sobre a implementação inicial da GPU, com exceção do primeiro resultado a ser gerado. A cada execução gerada foi percebido um tempo de espera maior sempre na primeira instância listada. Tal diferença provavelmente se atribui à alocação inicial dos blocos, dado que a única diferença entre as implementações é a quantidade de blocos da função objetivo. Isso também se refere à granularidade e será investigado posteriormente.

4.3.2 Diferenças entre buscas locais

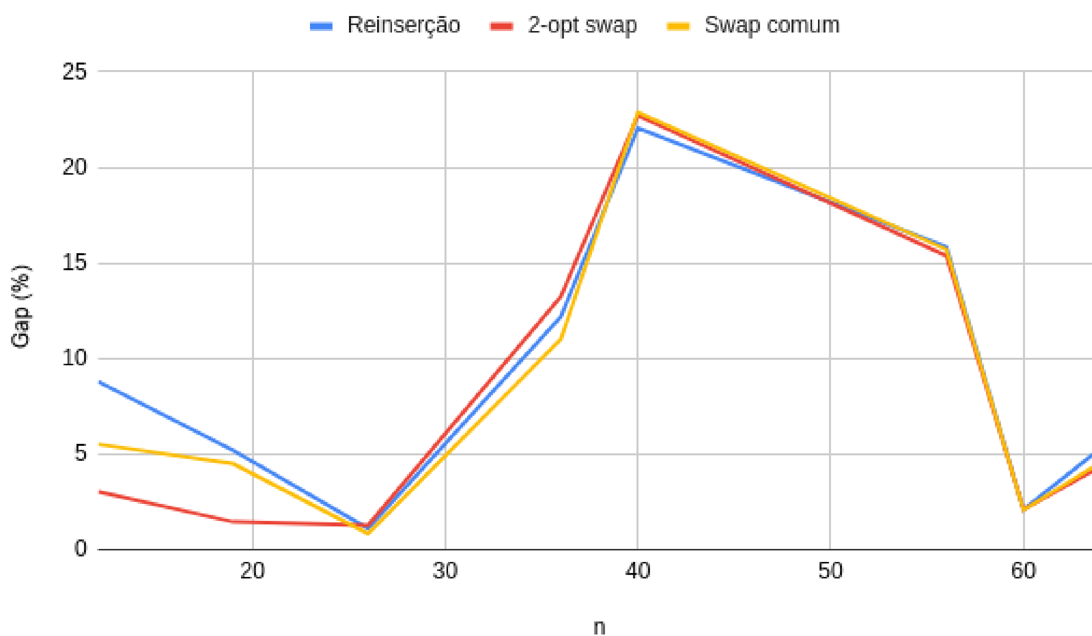
Para esse experimento, foram selecionadas oito das instâncias descritas na Tabela 2. A execução do experimento foi na implementação por GPU original, e os valores avaliados foram a média de *gap*, média de chamadas da função objetivo e média de tempo dentre as instâncias. O número máximo de gerações foi definido como $G = 1.000$ e as buscas locais foram executadas durante 100% das seleções.

Na Tabela 4, as colunas Swap, 2opt e Reins se referem aos Algoritmos 5, 6 e 7, respectivamente. Os resultados são apresentados nas Figuras 7 e 8.

Tabela 4 – Resultados médios dos três algoritmos de busca

	Gap (%)			**Tempo (s)**			**Chamadas de FO (/1000)**		
n	**Swap**	**2opt**	**Reins**	**Swap**	**2opt**	**Reins**	**Swap**	**2opt**	**Reins**
12	5,51	**3,02**	8,79	**5,72**	5,88	7,04	**4.128,75**	4.264,60	5.428,12
19	4,50	**1,45**	5,20	15,91	**13,99**	20,62	7.748,27	**5.682,49**	10.072,73
26	**0,83**	1,27	1,10	**4,39**	6,62	21,91	**1.547,81**	1.607,65	2.697,49
36	**11,00**	13,23	12,18	**37,20**	94,96	116,89	**4.648,47**	10.873,85	15.341,52
40	22,89	22,70	**22,06**	**11,95**	13,63	88,62	**1.707,36**	1.732,36	3.729,61
56	15,71	**15,38**	15,85	**23,75**	25,21	313,68	**1.715,84**	1.724,06	5.418,90
60	**2,08**	2,10	2,09	**25,35**	39,20	311,74	**1.595,11**	1725,56	4.835,86
64	4,55	**4,33**	5,30	**82,91**	224,30	1.279,88	**4.570,79**	5.889,19	45.518,16
Méd.	**8,38**	**7,93**	**9,07**	**25,89**	**57,92**	**270,04**	**3.462,30**	**4.187,47**	**11.630,29**

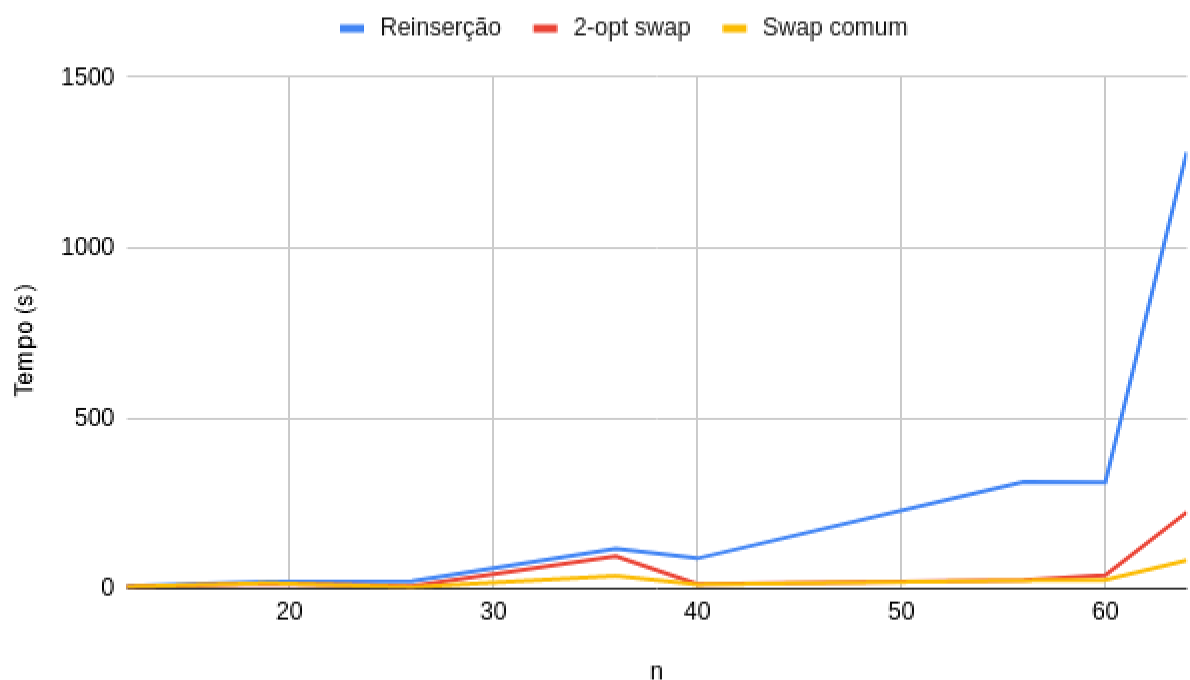
Figura 7 – *Gap* (%) em relação ao tamanho das instâncias



Fonte: Elaborado pelo autor.

Dentre as instâncias analisadas, considerando o contexto experimental estabelecido, poucas geraram divergência nos *gaps* analisados. Os resultados sugerem que a diferença ocorre de maneira significativa apenas nas instâncias *chr12c* e *els19*. Isto pode indicar uma vantagem do *2-opt swap* ao abordar problemas pequenos, e também uma desvantagem para o algoritmo de reinserção. Portanto, estas duas instâncias também apresentaram divergência no *gap* médio da análise anterior (Seção 4.3.1). Logo, é interessante para trabalhos futuros experimentar com mais instâncias de tamanho similar.

Figura 8 – Tempo de execução (em segundos) em relação ao tamanho das instâncias



Fonte: Elaborado pelo autor.

Ao analisar as demais colunas, observando a quantidade de chamadas da função objetivo, os resultados sugerem mais execuções para o algoritmo de reinserção. Isso pode estar associado à exploração da vizinhança, o que implica no aumento do tempo de execução. A média de execução da instância *tai64c* chegou a 21 minutos. Considerando as definições utilizadas, o o algoritmo pode demandar mais tempo de execução para instâncias maiores. Dependendo da janela de tempo disponível para encontrar uma solução, os resultados podem não ser satisfatórios.

4.3.3 Comparação dos resultados com a literatura

Para a verificação do algoritmo, serão avaliados os resultados obtidos com os valores disponíveis na literatura. Os experimentos para essa comparação foram realizados por meio

da implementação por CPU. Essa versão foi escolhida pelo fato de ser a implementação que menos varia o tempo de execução entre cada instância.

Para este experimento, o número máximo de gerações é de $G = 15.000$. Os experimentos também contam com a busca local *best-first 2-opt swap* (Seção 3.3) com taxa de ocorrência de 20% das seleções (se o decimal aleatório $B \in [0, 1]$ for menor ou igual a $0,2$, ocorre a busca local antes da seleção). Os resultados estão disponíveis na Tabela 5, com o valor obtido, *gap* em % e a quantidade de chamadas da função objetivo (em milhões) nas colunas *FO*.

No geral, cada execução teve sua duração entre 1 a 7 minutos, com exceção da instância *esc128*, que obteve uma média de 23 minutos de execução. Na Tabela 5 existem instâncias que chegaram no limite de 150 milhões de execuções da função objetivo. Estas mesmas instâncias são as que também obtiveram o menor *gap*.

Tabela 5 – Comparação dos resultados mínimo e médio produzidos com a literatura

	Literatura	**Mínimo encontrado**			**Média encontrada**		
Instância	**Valor**	**Valor**	**Gap**	**FO**	**Valor**	**Gap**	**FO**
chr12c	11.156	11.156	**0**	96,84	11.412	2,29	118,41
esc16a	68	68	**0**	150,00	68	**0**	150,01
had16	3.720	3.720	**0**	150,00	3.724	0,10	150,02
had18	5.358	5.360	0,04	150,00	5.393	0,65	150,02
nug18	1.930	1.964	1,76	150,00	1.993	3,24	150,01
els19	17.212.548	17.212.548	**0**	150,00	17.352.715	0,81	150,01
had20	6.922	6.924	0,03	150,00	6.963	0,59	150,03
nug20	2.570	2.628	2,26	150,00	2.653	3,24	150,03
bur26a	5.426.670	5.437.625	0,20	150,01	5.444.438	0,33	150,02
bur26h	7.098.658	7.101.179	0,04	68,35	7.122.652	0,34	141,87
esc32g	6	6	**0**	150,02	6	**0**	150,08
ste36a	9.526	10.192	6,99	150,00	10.511	10,34	150,03
tho40	240.516	258.962	7,67	31,33	286.917	19,29	32,91
sko56	34.458	39.344	14,18	31,03	39.497	14,62	31,07
lipa60a	107.218	109.368	2,01	30,95	109.389	2,03	31,02
tai64c	1.855.928	1.872.872	0,91	124,57	1.893.344	2,02	125,22
sko72	66.256	75.250	13,57	31,35	75.341	13,71	31,37
sko81	90.998	102.582	12,73	31,48	102.902	13,08	31,61
lipa90a	360.630	366.060	1,51	30,51	366.116	1,52	30,54
wil100	273.038	292.062	6,97	31,68	292.374	7,08	31,81
esc128	64	64	**0**	150,00	66	3,13	150,06
Média	**1.562.297**	**1.567.140**	**3,37**	**102,77**	**1.577.546**	**4,69**	**107,43**

Considerando o contexto experimental estabelecido, os resultados sugerem que a quantidade de funções objetivo executadas é relativa à facilidade de encontrar bons vizinhos durante a busca local. Isso pode ser indicado a partir de instâncias com menos ótimos locais não finalizarem a busca *best-first* antecipadamente, o que pode resultar em mais execuções da função objetivo. Esse é o caso das instâncias *sko56*, *sko72* e *sko81*, de

Skorin-Kapov (1990). Tais instâncias apresentaram os *gaps* mais distantes e as menores execuções das buscas locais. Isso pode ser justificado pela quantidade de ótimos locais dos problemas. É importante observar também que esses resultados se referem à estrutura de vizinhança adotada. Diferentes estratégias podem encontrar resultados diferentes. Por isso, essa é uma questão que também deve ser investigada posteriormente.

4.4 Considerações finais

Durante o planejamento do experimento, foram consideradas diversas abordagens para o algoritmo e parametrização dos experimentos, citados na Seção 5.1. Para análises corretas é essencial seguir os objetivos propostos durante o planejamento dos experimentos. A organização dos resultados em arquivos *.tsv* durante todas as execuções também é importante, pois registra os resultados dos mais simples experimentos, os quais podem complementar diversas análises futuras.

Os experimentos da Seção 4.3.1 indicam grande potencial na GPGPU, uma vez que a maior parte do código desenvolvido possui execução de apenas um bloco. O CUDA possui limitação de 65.535 blocos por dimensão de um *grid* (NVIDIA, 2021). Logo, o paralelismo tem muito a ser explorado com diferentes granularidades. Ainda assim, considerando o contexto experimental estabelecidos, os resultados sugerem que ambas as implementações por GPU reportam tempos menores de execução quanto à CPU na maioria das instâncias conhecidas ($n <= 64$).

Na Seção 4.3.2, os resultados sugerem que a estrutura de vizinhança de reinserção é inviável para o algoritmo proposto, enquanto as operações de *swap* comum e *2-opt swap* são similares. Apesar do *2-opt swap* ficar mais lento conforme o tamanho do problema, ele pode ser mais eficaz em problemas menores.

Na Seção 4.3.3, os resultados sugerem que o AED é promissor ao lidar com o PQA, apesar do contexto discreto do problema. Diferentes representações para o problema podem ser investigadas em outros projetos.

O capítulo seguinte descreve as conclusões acerca do trabalho, bem como apresenta propostas de trabalhos futuros para a continuidade do projeto.

5 Conclusão

Este trabalho teve como objetivo principal a utilização do Algoritmo de Evolução Diferencial para solucionar o Problema Quadrático de Alocação ao explorar algoritmos de execução paralela na CPU e GPU. Durante o desenvolvimento, o maior desafio foi selecionar o *framework* ideal para execução do código na GPU de maneira prática e funcional.

Foram introduzidos diversos conceitos acerca do tema escolhido, incluindo as aplicações do PQA, um estudo sobre o AED, como comportam seus parâmetros e as possíveis representações do PQA no algoritmo. Posteriormente descrevem-se os conceitos básicos de paralelismo ao utilizar CUDA e foram avaliados trabalhos com objetivos similares ao tema.

Os resultados sugerem que o algoritmo proposto tem desempenho satisfatório e promissor. Os valores reportados na Tabela 5 indicam resultados próximos da literatura e tempo de execução relativamente pequeno. Diferentes granularidades do paralelismo podem ser avaliadas, o que pode aumentar a performance do algoritmo proposto. Além disso, diferentes implementações que utilizam de vários blocos de *threads* podem ser verificadas.

5.1 Propostas para trabalho futuros

Durante o decorrer deste trabalho foram analisadas diversas abordagens diferentes que cumprem o mesmo propósito geral. Por limitações de tempo e escopo os seguintes itens não foram desenvolvidos e podem dar sequência ao trabalho.

- Realizar as implementações também em OpenCL;
- Implementar a paralelização de buscas locais para o problema;
- Experimentar com outras representações do PQA ou outros movimentos de busca local;
- Explorar a utilização de múltiplas *threads* de GPU e CPU em distintos trechos durante o mesmo algoritmo;
- Estudar o comportamento das demais instâncias disponibilizadas da QAPLIB;
- Otimizar o AED e seus parâmetros para facilitar a convergência a mínimos da literatura, com a implementação de métodos de *parameter control*, como o *iRace*.
- Planejar e realizar diferentes experimentos com outras configurações.

Dada a importância de continuidade do trabalho, o código-fonte utilizado será organizado e mantido em <https://github.com/tuliosj/CUDADifferentialEvolutionQAP> dentro dos limites da plataforma. Isso possibilita a contribuições de terceiros, bem como garante a reprodutibilidade do código e dos experimentos.

Referências

ALVES, Â. B. M. d. C. Análise de representações para o algoritmo de evolução diferencial no contexto discreto. *Instituto de Ciências Exatas e Aplicadas, Universidade Federal de Ouro Preto, João Monlevade*, Monografia (Graduação em Sistemas de Informação), 2016. Citado na página 21.

BURKARD, R. E.; KARISCH, S. E.; RENDL, F. Qaplib – a quadratic assignment problem library. *Journal of Global optimization*, Springer, v. 10, n. 4, p. 391–403, 1997. Citado na página 12.

CROES, G. A. A method for solving traveling-salesman problems. *Operations research*, INFORMS, v. 6, n. 6, p. 791–812, 1958. Citado na página 29.

DAVENDRA, D.; BIALIC-DAVENDRA, M.; METLICKA, M. Cuda accelerated 2-opt local search for the traveling salesman problem. In: *Novel Trends in the Traveling Salesman Problem*. [s.n.], 2020. ISBN 978-1-83962-453-7. Disponível em: <https://www.intechopen.com/books/novel-trends-in-the-traveling-salesman-problem/cuda-accelerated-2-opt-local-search-for-the-traveling-salesman-problem>. Citado na página 28.

DORIGO, M.; CARO, G. D. Ant colony optimization: a new meta-heuristic. In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. [S.l.: s.n.], 1999. v. 2, p. 1470–1477 Vol. 2. Citado na página 23.

DU, P. et al. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, Elsevier, v. 38, n. 8, p. 391–407, 2012. Citado na página 14.

GÄMPERLE, R.; MÜLLER, S. D.; KOUMOUTSAKOS, P. A parameter study for differential evolution. *Advances in intelligent systems, fuzzy systems, evolutionary computation*, Citeseer, v. 10, n. 10, p. 293–298, 2002. Citado na página 20.

JOUBERT, G. R. et al. *Advances in parallel computing*. [S.l.]: North-Holland, 2012. Citado na página 12.

JR., S. A. d. C.; RAHMANN, S. Microarray layout as quadratic assignment problem. In: HUSON, D. et al. (Ed.). *German Conference on Bioinformatics*. Bonn: Gesellschaft für Informatik e.V., 2006. p. 11–20. Citado na página 17.

KILIC, H.; Yüzgeç, U. Tournament selection based antlion optimization algorithm for solving quadratic assignment problem. *Engineering Science and Technology, an International Journal*, v. 22, 12 2018. Citado na página 13.

KOOPMANS, T. C.; BECKMANN, M. Assignment problems and the location of economic activities. *Econometrica*, [Wiley, Econometric Society], v. 25, n. 1, p. 53–76, 1957. ISSN 00129682, 14680262. Disponível em: <http://www.jstor.org/stable/1907742>. Citado na página 12.

LARSEN, E. S.; MCALLISTER, D. Fast matrix multiplies using graphics hardware. In: ACM. *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. [S.l.], 2001. p. 55–55. Citado na página 14.

LIN, Y.-K. et al. An assignment problem and its application in education domain: A review and potential path. *Advances in Operations Research*, Hindawi, v. 2018, p. 8958393, 2018. ISSN 1687-9147. Disponível em: <https://doi.org/10.1155/2018/8958393>. Citado na página 17.

LIU, J.; LAMPINEN, J. A fuzzy adaptive differential evolution algorithm. *Soft Computing*, v. 9, n. 6, p. 448–462, 2005. ISSN 1433-7479. Disponível em: <https://doi.org/10.1007/s00500-004-0363-x>. Citado na página 20.

LUONG, T. V.; MELAB, N.; TALBI, E. Parallel hybrid evolutionary algorithms on gpu. In: *IEEE Congress on Evolutionary Computation*. [S.l.: s.n.], 2010. p. 1–8. Citado 2 vezes nas páginas 7 e 22.

MOHAMMADI, J.; MIRZAIE, K.; DERHAMI, V. Parallel genetic algorithm based on gpu for solving quadratic assignment problem. In: *2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI)*. [S.l.: s.n.], 2015. p. 569–572. Citado 2 vezes nas páginas 13 e 23.

NVIDIA. *Features and Technical Specifications*. [S.l.]: CUDA Toolkit Documentation, 2021. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications>. Citado 2 vezes nas páginas 22 e 40.

PIOTROWSKI, A. P. Review of differential evolution population size. *Swarm and Evolutionary Computation*, v. 32, p. 1–24, 2017. ISSN 2210-6502. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2210650216300268>. Citado na página 20.

QIAN, B. et al. A hybrid differential evolution method for permutation flow-shop scheduling. *The International Journal of Advanced Manufacturing Technology*, Springer, v. 38, n. 7-8, p. 757–777, 2008. Citado na página 21.

RAN, I. *CUDADifferentialEvolution*. [S.l.]: GitHub, 2017. <https://github.com/ianran/CUDADifferentialEvolution>. Citado na página 27.

RYOO, S. et al. Program optimization carving for gpu computing. *Journal of Parallel and Distributed Computing*, v. 68, n. 10, p. 1389 – 1401, 2008. ISSN 0743-7315. General-Purpose Processing using Graphics Processing Units. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731508000968>. Citado na página 12.

SKORIN-KAPOV, J. Tabu search applied to the quadratic assignment problem. *ORSA Journal on computing*, INFORMS, v. 2, n. 1, p. 33–45, 1990. Citado na página 40.

STORN, R. Differential evolution research – trends and open questions. In: CHAKRABORTY, U. K. (Ed.). *Advances in Differential Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 1–31. ISBN 978-3-540-68830-3. Disponível em: <https://doi.org/10.1007/978-3-540-68830-3_1>. Citado na página 20.

STORN, R.; PRICE, K. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, Springer, v. 11, n. 4, p. 341–359, 1997. Citado 2 vezes nas páginas 12 e 19.

STÜTZLE, T. Parallelization strategies for ant colony optimization. In: EIBEN, A. E. et al. (Ed.). *Parallel Problem Solving from Nature — PPSN V*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 722–731. ISBN 978-3-540-49672-4. Citado na página 23.

TSUTSUI, S. Aco on multiple gpus with cuda for faster solution of qaps. In: *Proceedings of the 12th International Conference on Parallel Problem Solving from Nature - Volume Part II*. Berlin, Heidelberg: Springer-Verlag, 2012. (PPSN'12), p. 174–184. ISBN 978-3-642-32963-0. Disponível em: <http://dx.doi.org/10.1007/978-3-642-32964-7_18>. Citado na página 23.

VERONESE, L. de P.; KROHLING, R. A. Differential evolution algorithm on the gpu with c-cuda. In: *IEEE Congress on Evolutionary Computation*. [S.l.: s.n.], 2010. p. 1–7. Citado 4 vezes nas páginas 21, 22, 23 e 27.

WU, G. et al. Differential evolution with multi-population based ensemble of mutation strategies. *Information Sciences*, v. 329, p. 329–345, 2016. ISSN 0020-0255. Special issue on Discovery Science. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0020025515006635>. Citado na página 20.

ZAIED, A. N.; EL-FATAH, L. A. A survey of quadratic assignment problem. *International Journal of Computer Applications*, v. 101, 09 2014. Citado na página 17.