UNIVERSIDADE ESTADUAL DO MARANHÃO – UEMA
ENGENHARIA DA COMPUTAÇÃO E SISTEMAS

GUSTAVO GUSMÃO ROCHA

AVALIAÇÃO DE DESEMPENHO DE METAHEURÍSTICAS USANDO O PARALELISMO DO TENSORFLOW E PYTORCH

SÃO LUÍS - MA
2025

Gustavo Gusmão Rocha

Avaliação de Desempenho de Metaheurísticas Usando o Paralelismo do TensorFlow e PyTorch

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia da Computação e Sistemas da Universidade Estadual do Maranhão (UEMA), como requisito parcial para a obtenção do título de Mestre em Engenharia da Computação e Sistemas.

Orientadores:
Prof Dr. Omar Andrés Carmona Cortes
Prof. Dr. Fábio Manoel França Lobato

Universidade Estadual do Maranhão - UEMA
Engenharia da Computação e Sistemas
Programa de Pós-Graduação em Engenharia da Computação e Sistemas (PECS)

Orientador: Prof Dr. Omar Andrés Carmona Cortes
Coorientador: Prof. Dr. Fábio Manoel França Lobato

São Luís - MA
2025

Rocha, Gustavo Gusmão
Avaliação de desempenho de metaheurísticas usando o paralelismo do tensorflow e pytorch / Gustavo Gusmão Rocha. – São Luis, MA, 2025.
68 f

Dissertação (Mestrado em Engenharia de Computação e Sistemas/PECS) - Universidade Estadual do Maranhão, 2025.

Orientador: Prof. Dr. Omar Andres Carmona Cortes
Coorientador: Prof. Dr. Fábio Manoel França Lobato

1.Processamento Paralelo. 2.Metaheurísticas. 3.GPUs. 4.Otimização. I.Titulo.

CDU: 004.023

Elaborado por Cássia Diniz - CRB 13/910

ATA Nº 07

Ao vigésimo quarto dia do mês de junho do ano de dois mil e vinte e cinco, às 09h30min, via videoconferência, instalou-se a banca examinadora de dissertação de mestrado do aluno GUSTAVO GUSMÃO ROCHA. A banca examinadora foi composta pelos professores Dr. OMAR ANDRES CARMONA CORTES, UEMA, orientador e presidente; Dr. FÁBIO MANOEL FRANÇA LOBATO, UEMA, coorientador; Dr. ANTONIO FERNANDO LAVAREDA JACOB JUNIOR, examinador interno, UEMA e Dr. LEANDRO COLOMBI RESENDO, examinador externo à IES, IFES. Deu-se início a abertura dos trabalhos, por parte do professor Dr. OMAR ANDRES CARMONA CORTES, orientador e presidente da banca, que, após apresentar os membros da banca examinadora e esclarecer a tramitação da defesa, solicitou o aluno que iniciasse a apresentação da dissertação, intitulado: **AVALIAÇÃO DE DESEMPENHO DE METAHEURISTICAS USANDO O PARALELISMO DO TENSORFLOW E PYTORCH EM GPU**, marcando um tempo de 45 minutos para a apresentação. Concluída a exposição, o Prof. Dr. OMAR ANDRES CARMONA CORTES, presidente e orientador, passou a palavra ao coorientador, Dr. FÁBIO MANOEL FRANÇA LOBATO, dando continuidade com o examinador externo à IES; Dr. LEANDRO COLOMBI RESENDO, seguido do examinador interno Dr. ANTONIO FERNANDO LAVAREDA JACOB JUNIOR, para que fizessem o mesmo; após o que fez suas considerações sobre o trabalho em julgamento; tendo sido **APROVADO** o aluno, conforme as normas vigentes na Universidade Estadual do Maranhão. A versão final da dissertação deverá ser entregue ao programa, no prazo de 30 dias; contendo as modificações sugeridas pela banca examinadora e constante na folha de correção anexa.

gov.br Documento assinado digitalmente
ANTONIO FERNANDO LAVAREDA JACOB JUNIOI
Data: 24/06/2025 11:50:54-0300
Verifique em https://validar.iti.gov.br

Dr. Antonio Fernando L. Jacob Junior, UEMA
Examinador Interno

gov.br Documento assinado digitalmente
OMAR ANDRES CARMONA CORTES
Data: 24/06/2025 11:40:38-0300
Verifique em https://validar.iti.gov.br

Dr. Omar Andres Carmona Cortes, UEMA
Presidente e Orientador

gov.br Documento assinado digitalmente
LEANDRO COLOMBI RESENDO
Data: 25/06/2025 19:33:13-0300
Verifique em https://validar.iti.gov.br

Dr. Leandro Colombi Resendo, IFES
Examinador Externo à IES

gov.br Documento assinado digitalmente
GUSTAVO GUSMAO ROCHA
Data: 24/06/2025 16:05:15-0300
Verifique em https://validar.iti.gov.br

Gustavo Gusmão Rocha
Mestrando

gov.br Documento assinado digitalmente
FABIO MANOEL FRANCA LOBATO
Data: 24/06/2025 14:36:36-0300
Verifique em https://validar.iti.gov.br

Dr. Fabio Manoel França Lobato, UEMA
Coorientador

Pois tudo o que Deus criou é bom,
e nada deve ser rejeitado,
se for recebido com ação de graças,
pois é santificado pela palavra de Deus e pela oração.
(1 Timóteo 4:4-5)

AGRADECIMENTOS

Toda honra e toda glória ao nosso Senhor Jesus Cristo. Minha esposa, filhos, sobrinhos e meus Pais, que são a base que me sustenta e me inspira a seguir em frente, sempre unidos como uma verdadeira família. Aos meus amigos da UNDB e UEMA, com quem compartilho uma amizade que durará para sempre. Ao meu orientador Omar Carmona e meu co-orientador Fábio Lobato, pela paciência e por me motivar a explorar todo o meu potencial. E um agradecimento especial a Bruno Seabra Nogueira Mendonça Lima, por seu apoio e incentivo ao inicio dessa jornada.

“Acima de tudo, adquire sabedoria e conhecimento,
ainda que te custem tudo o que possuis,
(Provérbios 4:7)

RESUMO

A otimização computacional tem se tornado um desafio crítico em engenharia e inteligência artificial, especialmente no contexto da computação paralela, devido à alta complexidade das soluções e à demanda intensiva por recursos computacionais. As metaheurísticas, como *Particle Swarm Optimization* (OEP), *Cuckoo Search* (BC), *Differential Evolution* (ED) e *Genetic Algorithms* (AG), são abordagens eficazes que combinam busca local e global para explorar grandes espaços de solução. No entanto, a execução dessas técnicas pode ser computacionalmente onerosa, exigindo estratégias eficientes para paralelização. Este trabalho avalia o desempenho de metaheurísticas paralelizadas utilizando *TensorFlow* e *PyTorch* em funções de otimização *benchmark*, analisando métricas como *speedup* e eficiência entre CPU e GPU. Foram empregadas técnicas como *tf.distribute.MirroredStrategy*, que replica tensores em múltiplos dispositivos dentro do *strategy.scope*, e, no *PyTorch*, a estratégia *device = torch.device('cuda')*, permitindo que operações como *torch.rand*, *torch.cat*, *torch.argsort* e *torch.where* sejam distribuídas eficientemente. Os experimentos demonstram o impacto do paralelismo na busca por soluções globais e locais, destacando o papel das GPUs e arquiteturas paralelas na otimização computacional. A análise técnica fornece subsídios para a escolha das tecnologias mais adequadas em aplicações críticas, onde o uso de paralelismo é justificado pela necessidade de resolver problemas de alta dimensionalidade e complexidade.

Palavras-chave: Processamento Paralelo. Metaheurísticas. *GPUs*. Otimização.

ABSTRACT

Computational optimization has become a critical challenge in engineering and artificial intelligence, especially in parallel computing, due to the high complexity of solutions and the intensive demand for computational resources. Metaheuristics, such as Particle Swarm Optimization (PSO), Cuckoo Search (CS), Differential Evolution (DE), and Genetic Algorithms (GA), are effective approaches that combine local and global search to explore large solution spaces. However, executing these techniques can be computationally expensive, requiring efficient parallelization strategies. This study evaluates the performance of parallelized metaheuristics using *TensorFlow* and *PyTorch* on *benchmark* optimization functions, analyzing metrics such as *speedup* and efficiency between the CPU and GPU. Techniques such as *tf.distribute.MirroredStrategy*, which replicates tensors across multiple devices within the *strategy.scope*, and, in *PyTorch*, the *device = torch.device('cuda')* strategy, allowing operations like *torch.rand*, *torch.cat*, *torch.argsort*, and *torch.where* to be efficiently distributed, were employed. The experiments demonstrate the impact of parallelism on the search for global and local solutions, highlighting the role of GPUs and parallel architectures in computational optimization. This technical analysis provides insight into the selection of the most suitable technologies for critical applications where parallelism is justified by the need to solve complex and high-dimensional problems.

Keywords: Parallel Processing. Metaheuristics. GPUs. Optimization.

LISTA DE ILUSTRAÇÕES

Figura 1 – Fluxograma de um metaheurísticas padrão. 19
Figura 2 – Fluxograma de uma metaheurísticas AG. 21
Figura 3 – Concorrência de task. 27
Figura 4 – Simultaneidade de task. 27
Figura 5 – Paralelismo 27
Figura 6 – Fluxograma *metaheurísticas* na *GPU* 30
Figura 7 – Função *Discus e Rastrigin 50D TensorFlow* 44
Figura 8 – Função *Rosenbrock e Cigar 50D TensorFlow* 44
Figura 9 – Função *Elliptic e Ackley 50D TensorFlow* 44
Figura 10 – Função *Griewank e Salomon 50D TensorFlow* 44
Figura 11 – Função *Discus e Rastrigin 100D TensorFlow* 46
Figura 12 – Função *Rosenbrock e Cigar 100D TensorFlow* 46
Figura 13 – Função *Elliptic e Ackley 100D TensorFlow* 46
Figura 14 – Função *Griewank e Salomon 100D TensorFlow* 46
Figura 15 – Função Discus e Rastrigin 500D TensorFlow 48
Figura 16 – Função Rosenbrock e Cigar 500D TensorFlow 48
Figura 17 – Função Elliptic e Ackley 500D TensorFlow 48
Figura 18 – Função Griewank e Salomon 500D TensorFlow 48
Figura 19 – Função *Discus e Rastrigin 50D PyTorch* 60
Figura 20 – Função *Rosenbrock e Cigar 50D PyTorch* 60
Figura 21 – Função *Elliptic e Ackley 50D PyTorch* 60
Figura 22 – Função *Griewank e Salomon 50D PyTorch* 60
Figura 23 – Função *Discus e Rastrigin 100D PyTorch* 62
Figura 24 – Função *Rosenbrock e Cigar 100D PyTorch* 62
Figura 25 – Função *Elliptic e Ackley 100D PyTorch* 62
Figura 26 – Função *Griewank e Salomon 100D PyTorch* 62
Figura 27 – Função *Discus e Rastrigin 500D PyTorch* 64
Figura 28 – Função *Rosenbrock e Cigar 500D PyTorch* 64
Figura 29 – Função *Elliptic e Ackley 500D PyTorch* 64
Figura 30 – Função *Griewank e Salomon 500D PyTorch* 64

LISTA DE TABELAS

Tabela 1 – Características das Funções Benchmark 17
Tabela 2 – Analogia evolução natural e algoritmo genético 20
Tabela 3 – Paradigmas de Paralelização 26
Tabela 4 – Funções de Otimização, Equações, Domínios e Ótimo Global 32
Tabela 5 – Desempenho de heurísticas em Funções *Benchmark Tensorflow* (50D, 10000 iterações) 35
Tabela 6 – Desempenho de heurísticas em Funções *Benchmark Tensorflow* (100D, 10000 iterações) . . . 36
Tabela 7 – Desempenho de heuristicas em Funções *Benchmark Tensorflow* (500D, 10000 iterações) . . . 36
Tabela 8 – Resultados de Speedup e Eficiência OEP *Tensorflow* (50D, 10000 iterações) 37
Tabela 9 – Resultados de *Speedup* e Eficiência AG *Tensorflow* (50D, 10000 iterações) 37
Tabela 10 – Resultados de *Speedup* e Eficiência ED *Tensorflow* (50D, 10000 iterações) 38
Tabela 11 – Resultados de *Speedup* e Eficiência BC *Tensorflow* (50D, 10000 iterações) 38
Tabela 12 – Resultados de *Speedup* e Eficiência OEP *Tensorflow* (100D, 10000 Iterações) 39
Tabela 13 – Resultados de *Speedup* e Eficiência AG *Tensorflow* (100D, 10000 Iterações) 39
Tabela 14 – Resultados de *Speedup* e Eficiência ED *Tensorflow* (100D, 10000 Iterações) 40
Tabela 15 – Resultados de *Speedup* e Eficiência BC *Tensorflow* (100D, 10000 Iterações) 40
Tabela 16 – Resultados de *Speedup* e Eficiência OEP *Tensorflow* (500D, 10000 Iterações) 41
Tabela 17 – Resultados de *Speedup* e Eficiência AG *Tensorflow* (500D, 10000 Iterações) 41
Tabela 18 – Resultados de *Speedup* e Eficiência ED *Tensorflow* (500D, 10000 Iterações) 42
Tabela 19 – Resultados de *Speedup* e Eficiência BC *Tensorflow* (500D, 10000 Iterações) 42
Tabela 20 – Desempenho de heurísticas em Funções *Benchmark Pytorch* (50D, 10000 iterações) 50
Tabela 21 – Desempenho de heurísticas em Funções *Benchmark Pytorch* (100D, 10000 iterações) 51
Tabela 22 – Desempenho de heurísticas em Funções *Benchmark Pytorch* (500D, 10000 iterações) 51
Tabela 23 – Resultados de Speedup e Eficiência OEP *Pytorch* (50D, 10000 iterações) 52
Tabela 24 – Resultados de Speedup e Eficiência AG *Pytorch* (50D, 10000 iterações) 53
Tabela 25 – Resultados de *Speedup* e Eficiência ED *Pytorch* (50D, 10000 Iterações) 53
Tabela 26 – Resultados de Speedup e Eficiência BC *Pytorch* (50D, 10000 Iterações) 54
Tabela 27 – Resultados de Speedup e Eficiência OEP *Pytorch* (100D, 10000 Iterações) 54
Tabela 28 – Resultados de Speedup e Eficiência AG *Pytorch* (100D, 10000 Iterações) 55
Tabela 29 – Resultados de Speedup e Eficiência ED *Pytorch* (100D, 10000 Iterações) 55
Tabela 30 – Resultados de Speedup e Eficiência BC *Pytorch* (100D, 10000 Iterações) 56
Tabela 31 – Resultados de Speedup e Eficiência OEP *Pytorch* (500D, 10000 Iterações) 56
Tabela 32 – Resultados de Speedup e Eficiência AG *Pytorch* (500D, 10000 Iterações) 57
Tabela 33 – Resultados de Speedup e Eficiência ED *Pytorch* (500D, 10000 Iterações) 57
Tabela 34 – Resultados de Speedup e Eficiência BC *Pytorch* (500D, 10000 Iterações) 58
Tabela 35 – Comparativo de Melhor Desempenho (Qualidade e Speedup) entre Frameworks (500D) 66

LISTA DE ABREVIATURAS E SIGLAS

AE	Algoritmos Evolutivos
AG	Algoritmo Genético
API	*Application Programming Interface*
BC	Busca Cuco
CE	Computação Evolucionária
CPU	*Central Processing Unit*
CS	*Cuckoo Search*
ED	*Evolução Diferencial*
GA	*Genetic Algorithms*
GPU	*Graphics Processing Unit*
IA	Inteligência Artificial
ICGA91	Primeira Organização International *Conference on Genetic Algorithms*
WCCI	*World Conference on Computational Intelligence*
OEP	Otimização por Enxame de Partículas
PPSN	*Parallel Problem Solving from Nature*
PSO	*Particle Swarm Optimization*
ROCm	*Radeon Open Compute*
TF	*Tensorflow*

SUMÁRIO

1 INTRODUÇÃO . . . 13
1.1 Justificativa . . . 13
1.2 Pergunta da pesquisa . . . 14
1.3 Hipótese . . . 14
1.4 Objetivo Geral . . . 14
1.5 Objetivos Específicos . . . 14
1.6 Estrutura da Dissertação . . . 14

2 REFERENCIAL TEÓRICO . . . 15
2.1 Trabalhos Correlatos . . . 15
2.2 Frameworks . . . 15
2.3 Funções Benchmark . . . 16
2.4 Speedup e Eficiência de Paralelização . . . 17
2.5 Metaheurísticas . . . 18
2.5.1 Algoritmo Genético . . . 20
2.5.2 Otimização por Enxame de Partículas . . . 22
2.5.3 Algoritmo Evolução Diferencial . . . 23
2.5.4 Algoritmo *Busca Cuco* . . . 24
2.6 Paralelismo . . . 26
2.7 Considerações finais do Referencial teórico . . . 28

3 METODOLOGIA . . . 29
3.1 Experimentos . . . 29
3.1.1 Implementação das *metaheurísticas* . . . 29
3.1.2 Metodologia de parametrização em *TensorFlow* . . . 30
3.1.3 Metodologia de parametrização em *PyTorch* . . . 31
3.1.4 Funções benchmark de otimização . . . 32
3.2 Considerações Finais da Metodologia . . . 33

4 RESULTADOS . . . 35
4.1 Resultados obtidos para paralelização usando o *Tensorflow* . . . 35
4.1.1 Resultados de *speedup* e eficiência em *Tensorflow* . . . 37
4.1.2 Resultados Gráficos Tempo *GPU* Por Algoritimos *Tensorflow* . . . 44
4.1.3 Discussão dos Resultados para *Tensorflow* . . . 49
4.2 Resultados obtidos para paralelização usando o *Pytorch* . . . 50
4.2.1 Resultados de *speedup* e eficiência em *Pytorch* . . . 52
4.2.2 Resultados Gráficos Tempo *GPU* Por Algoritimos *PyTorch* . . . 60
4.2.3 Discussão dos Resultados para *PyTorch* . . . 65
4.3 Análise Comparativa: TensorFlow e PyTorch . . . 66

5 CONCLUSÃO . . . 67

Referências . . . 69

1 INTRODUÇÃO

Os problemas de otimização têm sido um dos principais desafios em diversas áreas, desde a engenharia até a inteligência artificial, devido à complexidade de suas soluções e à alta demanda por recursos computacionais. Metaheurísticas surgiram como ferramentas eficazes para abordar esses problemas, pois combinam busca local e global na exploração de grandes espaços de solução. Contudo, a aplicação dessas técnicas em problemas reais enfrenta obstáculos como o aumento exponencial do tempo de execução e a necessidade de maior eficiência no uso dos recursos computacionais.

Dessa forma, a computação paralela se apresenta como uma solução fundamental para superar essas limitações. Ela permite dividir tarefas complexas em subtarefas menores, executadas simultaneamente em múltiplas unidades de processamento, como *CPUs* e *GPUs*. De acordo com (Kumar 2023), o avanço das arquiteturas paralelas e o uso de *GPUs* têm impulsionado a eficiência em áreas que exigem processamento intensivo, como aprendizado de máquina e simulações científicas. As *GPUs* modernas, como as da série *NVIDIA RTX* e *AMD Radeon*, possuem milhares de núcleos que permitem executar operações matemáticas massivamente paralelas, tornando-as ideais para otimização na computação científica.

Neste contexto, *TensorFlow* e *PyTorch* são *frameworks* amplamente utilizados devido às suas capacidades de programação paralela e treinamento de modelos em grande escala. O *PyTorch*, por exemplo, oferece flexibilidade no controle de operações paralelas, enquanto o *TensorFlow* se destaca por sua integração com arquiteturas distribuídas (Abadi ,2016). Estudos recentes indicam que o paralelismo implementado nesses *frameworks* não apenas reduz o tempo de execução, mas também melhora a escalabilidade e a capacidade de resolver problemas maiores e mais complexos (Smith 2023).

Diante desse cenário, este trabalho investiga o desempenho de metaheurísticas paralelizadas utilizando *TensorFlow* e *PyTorch* em funções de otimização *benchmark*. Para isso, foram utilizadas quatro meta-heurísticas: Otimização por Enxame de Partículas (OEP), Busca do Cuco (BC), Evolução Diferencial (ED) e Algoritmo Genético (AG). Foram analisadas métricas como *speedup*, tempo de execução e eficiência na utilização de *CPUs* e *GPUs*. Técnicas específicas foram aplicadas, como *tf.distribute.MirroredStrategy*, que replica tensores em múltiplos dispositivos dentro do *strategy.scope*, e, no *PyTorch*, a estratégia *device = torch.device('cuda')*, permitindo que operações como *torch.rand*, *torch.cat*, *torch.argsort* e *torch.where* sejam distribuídas de forma eficiente.

Além disso, a pesquisa explora o impacto do paralelismo na busca por soluções globais e locais, destacando como essas tecnologias podem ser aplicadas a problemas reais de otimização. A crescente adoção de supercomputadores e arquiteturas paralelas, como as plataformas baseadas em *CUDA* e *ROCm*, reforça a relevância deste estudo, fornecendo subsídios técnicos para a escolha de tecnologias adequadas em aplicações críticas. Com isso, este trabalho contribui para a compreensão das vantagens do paralelismo na otimização computacional, auxiliando no desenvolvimento de soluções mais escaláveis e eficientes para problemas de alta complexidade.

1.1 Justificativa

O uso de metaheurísticas em problemas de otimização é amplamente difundido devido à sua capacidade de explorar e explotar grandes espaços de busca de forma eficaz, especialmente em situações onde métodos tradicionais falham devido à alta dimensionalidade ou complexidade das funções objetivo. No entanto, a aplicação prática de metaheurísticas frequentemente enfrenta desafios significativos, como o aumento exponencial do tempo de execução e a demanda crescente por recursos computacionais à medida que o problema escala. Nesse cenário, o paralelismo surge como uma abordagem indispensável, permitindo dividir e executar tarefas simultaneamente em múltiplas unidades de processamento, como *CPUs* e *GPUs*, reduzindo drasticamente os tempos de execução e possibilitando a resolução de problemas mais desafiadores.

Além disso, o desenvolvimento de arquiteturas paralelas modernas, como *GPUs* com milhares de núcleos e plataformas como *CUDA* e *ROCm*, juntamente com *frameworks* de programação paralela como *TensorFlow* e *PyTorch*, oferece oportunidades para implementar metaheurísticas de maneira mais eficiente e escalável. Essas tecnologias possibilitam não apenas acelerar o processamento, mas também explorar simultaneamente diferentes regiões do espaço de busca, promovendo uma convergência mais rápida e, potencialmente, soluções de maior qualidade. Assim, investigar o uso do paralelismo em metaheurísticas é uma contribuição relevante, pois oferece um caminho para resolver problemas complexos em áreas como aprendizado de máquina, engenharia e inteligência artificial com maior eficiência e menor custo computacional.

1.2 Pergunta da pesquisa

De que forma a implementação de paralelismo com metaheurísticas em *frameworks* como *TensorFlow* e *PyTorch* influencia o desempenho computacional e a qualidade das soluções obtidas em problemas de otimização de alta dimensionalidade?

1.3 Hipótese

A implementação de paralelismo com metaheurísticas em *frameworks* como *TensorFlow* e *PyTorch* proporciona um impacto positivo no desempenho computacional, reduzindo significativamente o tempo de execução, e melhora ou mantém a qualidade das soluções obtidas em problemas de otimização de alta dimensionalidade.

1.4 Objetivo Geral

i. Avaliar o desempenho do paralelismo das metaheurísticas com *frameworks TensorFlow* e *PyTorch*.

1.5 Objetivos Específicos

i. Implementar metaheurísticas representativas, considerando diferentes cenários de otimização.

ii. Aplicar paralelismo às metaheurísticas utilizando o *framework TensorFlow*, explorando suas capacidades de execução em *CPU*s e *GPU*s.

iii. Implementar a paralelização das mesmas metaheurísticas utilizando o *framework PyTorch*, avaliando seus diferenciais em relação ao *TensorFlow*.

iv. Realizar testes de desempenho em problemas de otimização de grandes dimensões, utilizando funções de *benchmark* para analisar e comparar os resultados obtidos.

1.6 Estrutura da Dissertação

O Capítulo 1 apresenta a introdução do trabalho, abordando a justificativa, a pergunta de pesquisa e hipótese, além dos objetivos geral e específicos. O Capítulo 2 trata do referencial teórico, apresentando os conceitos gerais e trabalhos correlatos. São abordadas funções *benchmark*, técnicas de paralelização e metaheurísticas utilizadas, bem como o *speedup* e eficiência de paralelização. O Capítulo 3 detalha a metodologia empregada, incluindo os experimentos realizados e as técnicas de paralelização utilizando *Torch* e *TensorFlow*. O Capítulo 4 apresenta os resultados obtidos, comparando o desempenho das metaheurísticas em diferentes cenários de execução, tanto na *CPU* quanto na *GPU*. O Capítulo 5 traz a conclusão do estudo, discutindo as principais contribuições do trabalho e sugerindo direções para pesquisas futuras. Por fim, são apresentadas as referências utilizadas ao longo do trabalho.

2 REFERENCIAL TEÓRICO

2.1 Trabalhos Correlatos

Em seu trabalho, (Li *et al.* 2020) apresentam estratégias avançadas para melhorar o desempenho de treinamentos paralelos de redes neurais profundas utilizando *PyTorch*. Eles detalham o uso da biblioteca *DistributedDataParallel* (*DDP*), que otimiza o treinamento distribuído em *clusters* de *GPUs*. São explorados desafios como comunicação eficiente entre nós, escalabilidade em larga escala e balanceamento de carga, além de propor soluções práticas para reduzir a sobrecarga associada à sincronização de gradientes e movimentação de dados.

Também são apresentados estudos de caso e experimentos realizados em cenários de treinamento distribuído, mostrando como as melhorias implementadas no *PyTorch* podem aumentar significativamente a velocidade e eficiência. Os autores compartilham *insights* práticos para desenvolvedores e pesquisadores enfrentarem desafios de *Deep Learning*, com foco em maximizar o uso de recursos computacionais. Este trabalho contribui para aplicações que demandam escalabilidade e alto desempenho em treinamentos massivos.

Em (Dorneles 2021), é apresentada uma comparação de duas formas distintas de paralelizar Algoritmo Genético em *GPUs* com *CUDA*. Uma utiliza a memória compartilhada da *GPU* para cooperação na computação paralela, enquanto a outra utiliza as funções de *Warp-Shuffle* do *CUDA*. Avaliam cada uma dessas implementações em termos de desempenho, além de realizar diversos experimentos variando parâmetros como granularidade do paralelismo, tamanho da população e número de dimensões do problema, usando *speedup* e funções *benchmark* para medir o desempenho das *GPUs* com *CUDA*.

Em (Chien *et al.* 2019), é explorado o uso do *TensorFlow* e *PyTorch* com supercomputadores, mostrando que o *TensorFlow* e *PyTorch* reduzem a dificuldade de programação com aceleradores em ambientes distribuídos devido à sua facilidade de programação. A pesquisa visa entender o impacto geral do *TensorFlow* e *PyTorch* em supercomputadores de alto desempenho, implementando funções *benchmark* com o uso de *kernels* amplamente utilizados pela comunidade de supercomputadores, como *STREAM*, *Conjugate Gradient* e *Fast Fourier Transform*.

A pesquisa segue uma abordagem orientada a dados, semelhante à de um *pipeline* de *Machine Learning*, onde as matrizes de entrada são divididas em blocos menores para acelerar o cálculo com o uso do *TensorFlow* e *PyTorch*. O trabalho demonstra um desempenho escalável à medida que o número de *GPUs* aumenta. No geral, o trabalho destaca o potencial do *TensorFlow* e *PyTorch* na comunidade de supercomputadores de alto desempenho, especialmente considerando a dificuldade atual de programação em supercomputadores com aceleradores. Nesta pesquisa, utilizaremos a plataforma *Google Colaboratory*, conhecida como *Google Colab*, uma ferramenta em nuvem que permite criar e executar códigos em *Python*, além de configurar máquinas virtuais.

O *Google Colaboratory* oferece um ambiente de desenvolvimento integrado que elimina a necessidade de configuração de ambiente local, permitindo aos usuários começarem a trabalhar imediatamente. Com suporte a *notebooks Jupyter*, é possível combinar código executável, texto explicativo, gráficos e outras mídias em um único documento interativo, permitindo outras parametrizações, que é a proposta deste trabalho.

2.2 Frameworks

O *PyTorch*, desenvolvido inicialmente pela equipe de pesquisa de IA do Facebook (agora Meta), é uma biblioteca de *Machine Learning* altamente flexível e fácil de usar. Segundo a documentação (PyTorch 2020) foi projetado para suportar programação dinâmica, permitindo alterações durante a execução do código, o que é uma grande vantagem em pesquisas de IA. Em termos de paralelismo, o *PyTorch* oferece suporte a técnicas como *DataParallel* e *DistributedDataParallel* para treinamento em múltiplas *GPUs*, permitindo que grandes volumes de dados sejam processados de maneira eficiente e escalável.

Além do *dataparallelism*, o documento (PyTorch 2020) suporta o *model parallelism*, onde partes do modelo são distribuídas entre diferentes dispositivos. Isso é útil para modelos que são grandes demais para caber em uma única *GPU*. A biblioteca também implementa o *tensor parallelism*, que divide tensores grandes entre *GPUs* para realizar operações paralelamente. A utilização do *torch.multiprocessing* permite paralelismo em nível de *CPU*, além do uso de *streams* e *eventos* para otimizar a execução de operações em *GPUs*.

O *TensorFlow*, criado pelo *Google Brain Team* e lançado em 2015, é uma biblioteca de *Machine Learning* com um modelo de gráfico computacional estático. Esse design permite otimizações de desempenho antes da execução do código, mas pode tornar a depuração mais difícil. Segundo (Abadi ,2016) o *TensorFlow* oferece várias estratégias de paralelismo, como *MirroredStrategy*, que replica o modelo em múltiplas *GPUs*, distribuindo os dados entre elas para treinamento paralelo, e o *pipeline parallelism*, que distribui diferentes partes do processamento de dados em várias *GPUs* ou *CPUs*.

Além disso, o *TensorFlow* oferece suporte ao uso de *Tensor Processing Units* (*TPUs*), aceleradores de hardware desenvolvidos pelo Google, que são otimizados para operações de *Deep Learning*. A integração com *CUDA* e *TPUs* permite que o *TensorFlow* aproveite a computação distribuída de maneira eficiente, sendo uma das razões pelas quais a biblioteca é amplamente adotada em ambientes de produção que requerem treinamento de modelos em larga escala.

Segundo (Roveda e Ugo 2021), *Python* é uma linguagem de programação de alto nível, dinâmica, interpretada, modular, multiplataforma e orientada a objetos, conhecida por sua simplicidade e flexibilidade. Seu modelo orientado a objetos facilita o desenvolvimento e manutenção de grandes projetos, proporcionando controle e reutilização de código. Além disso, *Python* é amplamente utilizada em áreas como *Machine Learning*, inteligência artificial e computação científica. No que diz respeito ao paralelismo, *Python* oferece bibliotecas como *multiprocessing* e *concurrent.futures*, que permitem a execução de tarefas simultâneas, aproveitando múltiplos núcleos de *CPU* e *GPUs*, melhorando o desempenho em cálculos intensivos, especialmente em tarefas de *Machine Learning* e análise de grandes volumes de dados.

O (Numpy 2021) é uma biblioteca *Python* voltada para cálculos com *arrays* multidimensionais, essencial em tarefas como *Machine Learning*, onde é utilizada para realizar operações como multiplicação, adição e transposição de *arrays*. Ela oferece uma maneira fácil e eficiente de realizar cálculos e armazenar dados de treinamento e parâmetros de modelos. Além disso, o *NumPy* é útil em várias operações matemáticas, incluindo integração numérica, diferenciação, interpolação, extrapolação e álgebra linear, sendo frequentemente usada em conjunto com outras bibliotecas como *SciPy* e *Matplotlib*. O *NumPy* também suporta paralelismo em nível de operações, utilizando otimizações internas que permitem a execução de cálculos em múltiplos núcleos de processador, o que acelera o desempenho em tarefas de grande escala, como aquelas típicas em *Machine Learning*.

2.3 Funções Benchmark

Segundo (Alba e Tomassini 2002), as funções *benchmark* de otimização são ferramentas fundamentais para a avaliação comparativa de algoritmos. Elas são compostas por expressões matemáticas desenhadas para simular uma variedade de cenários de otimização — incluindo diferentes níveis de complexidade, presença de múltiplos ótimos locais, comportamentos não lineares e problemas de alta dimensionalidade. O uso dessas funções permite a análise objetiva de características como eficácia, eficiência computacional e precisão na convergência dos métodos, servindo como referência para validar o desempenho das abordagens propostas.

As funções apresentadas na Tabela 1 são amplamente utilizadas na literatura justamente por sua diversidade estrutural. Todas são funções contínuas, mas diferem significativamente quanto à *separabilidade*, *multimodalidade*, *convexidade* e *complexidade computacional*, características que impactam diretamente a dificuldade do processo de otimização.

Tabela 1 – Características das Funções Benchmark

Função	Separável	Multimodal	Convexa	Complexidade
Discus	Não	Não	Sim	Alta
Rastrigin	Sim	Sim	Não	Alta
Rosenbrock	Não	Não	Não	Alta
Cigar	Não	Não	Sim	Alta
Elliptic	Não	Não	Sim	Alta
Ackley	Sim	Sim	Não	Média
Griewank	Sim	Sim	Não	Média
Salomon	Sim	Sim	Não	Média

Fonte: Autor

Funções como *Rastrigin*, *Ackley*, *Griewank* e *Salomon* são classificadas como *multimodais* e *separáveis*, desafiando os algoritmos a escapar de ótimos locais e a realizar uma busca eficaz mesmo com variáveis independentes. Por outro lado, *Rosenbrock*, *Cigar*, *Discus* e *Elliptic* são funções não *separáveis*, o que significa que há forte dependência entre variáveis, exigindo maior robustez na exploração do espaço de busca.

As funções *convexas* — como *Discus*, *Cigar* e *Elliptic* — apresentam uma estrutura que facilita a convergência global, porém sua condição numérica desfavorável pode dificultar a otimização prática. Já as funções não *convexas*, como *Rosenbrock* e *Rastrigin*, aumentam a *complexidade* da paisagem de busca, exigindo algoritmos mais adaptativos. Por fim, a diversidade de características dessas funções *benchmark* permite uma avaliação abrangente dos algoritmos, não apenas em termos de desempenho absoluto, mas também quanto à sua capacidade de generalização e adaptação frente a diferentes topologias e condições do espaço de busca.

2.4 Speedup e Eficiência de Paralelização

Speedup e eficiência de paralelização são métricas amplamente utilizadas para avaliar o desempenho de algoritmos paralelizados em comparação com suas versões sequenciais. O *speedup* é definido como a razão entre o tempo de computação gasto pelo algoritmo serial e o tempo de computação gasto pelo algoritmo paralelo, conforme a Equação 2.1:

$$S(P) = \frac{T(1)}{T(P)} \tag{2.1}$$

Onde:

- $E(P)$ é a eficiência de paralelização.
- $S(P)$ é o *speedup*.
- P é o número de processadores usados no sistema paralelo.

No qual $T(1)$ é o tempo computacional gasto pelo algoritmo em um único processador mais eficiente para resolver o problema, e $T(P)$ é o tempo computacional gasto pelo algoritmo paralelo em P processadores. Para medir $T(1)$, o algoritmo é executado isoladamente em um único processo nos nós computacionais, utilizando-se eficientemente os recursos computacionais disponíveis, conforme equação 2.2:

$$E(P) = \frac{S(P)}{P} = \frac{T(1)}{P \cdot T(P)} \tag{2.2}$$

Para contextualizar os limites teóricos do ganho de desempenho em sistemas paralelos, é fundamental apresentar a **Lei de Amdahl**. Formulada por Gene Amdahl em 1967 (Amdahl 1967), esta lei afirma que a aceleração máxima (*speedup*) de um programa é limitada pela sua fração de código que deve ser executada sequencialmente. Portanto,

gargalos como a transferência de dados e a lógica de controle do algoritmo determinam o teto da aceleração possível, independentemente do número de processadores adicionados.

O *ganho percentual* é uma métrica complementar ao *speedup*, empregada para expressar a redução percentual no tempo de execução quando se utiliza paralelização, especialmente com suporte de *GPU*. Essa métrica fornece uma perspectiva direta sobre a economia de tempo computacional alcançada com a execução paralela. Sua formulação é apresentada na Equação 2.3:

$$\text{Ganho } (\%) = \left(1 - \frac{T_{\text{GPU}}}{T_{\text{Nucleos}}}\right) \times 100 \tag{2.3}$$

Na Equação 2.3, T_{Nucleos} representa o tempo de execução em múltiplos núcleos, enquanto T_{GPU} refere-se ao tempo de execução em uma *GPU*. Valores mais elevados de ganho indicam maior redução no tempo de processamento, refletindo um melhor aproveitamento dos recursos paralelos disponíveis (Barney 2021).

Por fim, o *speedup* pode ser analisado sob duas perspectivas: *speedup forte* e *speedup fraco*. O *speedup forte* avalia o ganho de desempenho ao aumentar o número de processadores enquanto o tamanho do problema permanece constante. Esse tipo de análise é útil para verificar a escalabilidade de um algoritmo em cenários onde o problema original não pode ser particionado ou aumentado. Já o *speedup fraco* considera o desempenho à medida que o tamanho do problema aumenta proporcionalmente ao número de processadores, sendo mais apropriado para problemas que exigem maior escalabilidade computacional. Em geral, o *speedup fraco* tende a mostrar ganhos mais estáveis à medida que novos recursos computacionais são adicionados, enquanto o *speedup forte* pode apresentar limitações relacionadas a *overheads* de comunicação ou sincronização entre os processadores (Barney 2021).

2.5 Metaheurísticas

A Computação Evolucionária (CE) é um ramo emergente da Inteligência Artificial que propõe um novo paradigma para a solução de problemas, inspirado na Seleção Natural descrita por Darwin em 1859. As metaheurísticas envolvem metodologias de busca em espaços de soluções ótimas candidatas, capazes de gerenciar operadores de busca local e global, promovendo robustez e eficácia na procura por soluções (Attux *et al.* 2021).

Os algoritmos de computação evolutiva são usados para resolver problemas complexos de otimização e busca. Eles operam com uma população inicial de soluções candidatas, que são manipuladas ao longo de várias gerações. O processo evolutivo é guiado por três passos principais: reprodução com herança genética, variação aleatória na população e aplicação da "seleção natural"para determinar quais soluções são mais aptas e devem ser preservadas. Essa abordagem mimetiza os princípios da evolução biológica, onde indivíduos mais adaptados ao ambiente têm maior probabilidade de sobreviver e passar suas características para a próxima geração.

> A Computação Evolucionária surgiu em encontros de pesquisadores de metaheurísticas na conferência *Parallel Problem Solving from Nature* (*PPSN*), realizada em Dortmund em 1990. Em 1991, a primeira *International Conference on Genetic Algorithms* (*ICGA91*) foi organizada, estabelecendo o nome da área. Em 1993, a *MIT Press* lançou o jornal homônimo, e a área foi incluída na primeira *World Conference on Computational Intelligence* (*WCCI*) em 1994, abordando temas como redes neurais artificiais, sistemas nebulosos e metaheurísticas (Attux *et al.* 2021).

A variação introduz diversidade na população, transmitida por herança, enquanto a seleção tende a eliminar os indivíduos inaptos. Segundo (Attux *et al.* 2021), quatro operações fundamentais são realizadas junto à população: Reprodução, Variação, Atribuição de um valor de aptidão (*fitness*) e Seleção dos indivíduos que comporão a próxima geração. Essas operações estão ilustradas no fluxograma de uma metaheurística padrão na Figura 1 e no pseudocódigo da metaheurística, apresentado no *Algorithm* 1.

Figura 1 – Fluxograma de um metaheurísticas padrão.

Fonte: - (Attux *et al.* 2021)

Algorithm 1 Pseudocódigo da metaheurísticas Padrão

Entrada : Número de indivíduos N, Probabilidade de crossover PC, Probabilidade de mutação PM
Saída : População final $P(t+1)$
Início
$[P] = (N, PC, PM)$
Inicialize N aleatoriamente
Avaliar P
$t \leftarrow 1$
while *(até que critério de parada não seja satisfeito)* **do**
  for $i \leftarrow 1$ *até* $N/2$ **do**
    Selecionar (P, fit)
    Reproduzir (P, fit, PC)
    Variar (P, PM)
    Avaliar $P(t+1)$
    $t \leftarrow t+1$
  end
end
Fim

Dentre os modelos paralelos existentes, destaca-se o modelo de ilhas (*island model*), no qual a população é dividida em subpopulações menores que evoluem de forma relativamente independente. Cada uma dessas subpopulações, denominadas ilhas, realiza sua própria evolução local, utilizando operadores clássicos como seleção, *crossover* e mutação, com trocas ocasionais de indivíduos entre elas. Essa abordagem favorece a manutenção da diversidade genética e reduz o risco de convergência prematura.

Segundo Tomassini e Alba (Alba e Tomassini 2002), os modelos de ilhas são uma das principais formas de estruturar algoritmos evolucionários paralelos, sendo também conhecidos como algoritmos distribuídos evolutivos paralelo (ADEP) ou de *coarse-grain*. A migração de indivíduos entre ilhas ocorre de maneira controlada, seguindo parâmetros como frequência de migração, número de migrantes e política de seleção dos indivíduos a serem enviados. Esses aspectos são fundamentais para o desempenho do algoritmo, influenciando tanto a qualidade das soluções quanto a velocidade de convergência.

Diversas topologias de conexão entre as ilhas são discutidas na literatura, incluindo estruturas em anel (*ring*),

malha (*mesh*) e hipercubo (*hypercube*). Cada uma dessas topologias define diferentes formas de comunicação entre as subpopulações, afetando o fluxo de informação e a dinâmica evolutiva. Por exemplo, a topologia em anel favorece uma comunicação local e controlada, enquanto o *hypercube* permite uma conectividade mais ampla e rápida. Tomassini destaca que não há uma topologia universalmente superior; a escolha depende da natureza do problema e dos recursos computacionais disponíveis (Alba e Tomassini 2002).

Por fim, (Alba e Tomassini 2002) ressalta que os modelos de ilhas, além de promoverem paralelismo natural, também permitem ganhos significativos de desempenho, inclusive com a possibilidade de *speedup superlinear*. Isso ocorre porque diferentes ilhas podem explorar regiões distintas do espaço de busca simultaneamente, aumentando a probabilidade de encontrar soluções de alta qualidade em menos tempo. Assim, os modelos distribuídos representam uma alternativa robusta e eficiente para a resolução de problemas de otimização em ambientes de computação paralela e distribuída.

2.5.1 Algoritmo Genético

Algoritmo Genético constitui uma técnica de busca e otimização, altamente paralela, inspirada no princípio Darwiniano de seleção natural e reprodução genética (GOLDBERG, 1989 Apud (Pacheco e AL 2021)). Esta metaheurística é baseada na evolução biológica e/ou no comportamento de seres vivos, tais como formigas e abelhas. O AG é um método de otimização bio-inspirado, desenvolvido por John Henry Holland em 1975.

> "Segundo a teoria evolucionária de Charles Darwin, os seres sofrem mutações e os mais aptos (melhores) vão sobreviver. Então investigar como ocorre tal seleção natural e efetuar simulação permite gerar dados com características desejadas." (Massago 2013).

Segundo (Goldschmidt 2015), os princípios imitados do algoritmo genético formam um sistema natural; os cromossomos são então submetidos a um processo evolucionário que envolve avaliação, seleção, recombinação sexual (*crossover*) e mutação. Podemos caracterizar através dos componentes: problema a ser otimizado; representação das soluções de problema; decodificação do cromossoma; avaliação; seleção; operadores genéticos; inicialização da população, como mostra a Tabela 2

Tabela 2 – Analogia evolução natural e algoritmo genético

Evolução natural	**Algoritmo Genético**
Meio Ambiente	Problema
Indivíduo	Solução
Cromossomo	Representação (binária, vetor, inteiro)
Gene	Característica do problema
Alelo	Valor da Característica
Reprodução Sexual	Operador de Cruzamento
Mutação	Operador de Mutação
População	Conjunto de soluções
Gerações	Ciclos

Fonte: (Goldschmidt 2015)

Segundo (Goldschmidt 2015), os componentes do algoritmo genético são aplicados a problemas de otimização de diversas complexidades, além de diversos parâmetros em busca da melhor solução, com restrições ou condições como: otimização de funções matemáticas; otimização combinatorial; otimização de planejamento; problema do caixeiro viajante; problema de otimização de rota de veículos; otimização de layout de circuitos; otimização de distribuição; otimização em negócios e síntese de circuitos eletrônicos.

Em geral, as técnicas, os parâmetros e os tipos de operadores genéticos afetam significativamente o desempenho dos AG. A escolha de técnicas, parâmetros e operadores é empírica, porém em sintonia com o tipo de problema envolvido. A inicialização da população determina o processo de criação dos indivíduos para o primeiro ciclo do algoritmo, sendo comum a formação da população inicial a partir de indivíduos aleatoriamente criados.

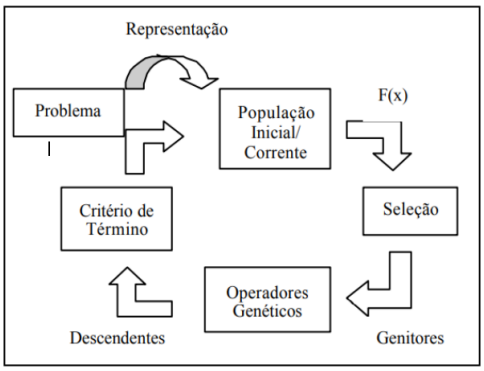
Tipicamente, a população inicial é formada a partir de indivíduos aleatoriamente criados. Populações iniciais aleatórias podem ser semeadas com bons cromossomos para uma evolução mais rápida, quando se conhece, a priori, o valor de boas "sementes".

Para um Algoritmo Genético, são vários parâmetros que controlam a realização do processo evolucionário:

- **Tamanho da população** - número de pontos do espaço de busca sendo considerados em paralelo a cada ciclo;
- **Taxa de Crossover** - probabilidade de um indivíduo ser recombinado com outro;
- **Taxa de Mutação** - probabilidade do conteúdo de cada gene do cromossoma ser alterado;
- **Número de Gerações** - quantidade ciclos do AG. Pode ser deduzido caso se tenha a informação do tamanho da população e total de indivíduos (total de indivíduos / tamanho da população);
- **Total de indivíduos** - total de tentativas em um experimento (tamanho da população x número de gerações);

Os dois últimos parâmetros são, em geral, empregados como critérios de parada em um algoritmo genético. Esse algoritmo pode ser descrito como um processo contínuo que repete ciclos de evolução controlados por um critério de parada, conforme apresentado pela Figura 2.

Figura 2 – Fluxograma de uma metaheurísticas AG.



Fonte: - (Attux *et al.* 2021)

As técnicas de reprodução determinam o critério de substituição dos indivíduos de uma população para a geração seguinte.

- **Troca de toda a População:** A cada ciclo, N novos indivíduos são criados, substituindo a população anterior. Para tanto, $N/2$ pares são selecionados para acasalamento, gerando N descendentes.
- **Elitismo:** Todos os cromossomos são substituídos, sendo o cromossomo mais apto da população corrente copiado para a população seguinte.
- ***Steady State*:** Gera M indivíduos $(M < N)$, que substituem os piores indivíduos da população corrente. Pode haver duplicação de indivíduos na população resultante.
- ***Steady State sem Duplicados*:** Similar à técnica de *Steady State*, não permitindo a presença de indivíduos duplicados na população.

Logo, as técnicas de aptidão influenciam na forma com que as aptidões dos cromossomos em uma população são numericamente atribuídas. A seguir encontram-se citados alguns exemplos:

- **Método Clássico:** Atribuir como aptidão de um cromossomo o valor numérico do resultado da avaliação. Este método, embora utilizado em muitos problemas, pode apresentar duas situações que precisam ser tratadas:
  - Competição próxima – indivíduos cujas aptidões são numericamente muito próximas, dificultando a distinção entre eles quanto à qualidade da solução proporcionada;
  - Super-indivíduos – são indivíduos com avaliação muito superior à média, que podem dominar o processo de seleção, impedindo que o AG obtenha novas soluções, potencialmente melhores do que as representadas pelos super-indivíduos.

- **Normalização Linear:** Esta técnica re-escala a aptidão dos cromossomos com base em sua posição (ranking) em uma ordenação. Primeiro, os indivíduos são ordenados da pior para a melhor avaliação. Ao pior indivíduo (ranking $i = 1$) é atribuída uma aptidão mínima (min), e ao melhor (ranking $i = N$), uma aptidão máxima (max). Estes valores são definidos pelo usuário, embora a formulação original do método sugira que as condições $min = 0$ e $max = 2$ sejam atendidas.

  A aptidão para os indivíduos intermediários é então distribuída linearmente. A lógica é que o intervalo total de aptidão ($max - min$) é dividido em incrementos iguais. Como há $N - 1$ "degraus"entre os N indivíduos, o incremento por degrau é de $\frac{max-min}{N-1}$. Assim, a nova aptidão, A_i, de um indivíduo é a aptidão base (min) somada a este incremento, multiplicado por quantos "degraus"($i - 1$) ele está acima do pior, conforme a Equação 2.4.

$$A_i = \min + \left\lfloor \frac{\max - \min}{N - 1} \right\rfloor \cdot (i - 1) \tag{2.4}$$

Segundo (Goldschmidt 2015), pode-se intuitivamente perceber que a taxa de aplicação do operador de *crossover* deve ser maior nas primeiras gerações, quando a população se apresenta dispersa pelo espaço de busca. Após várias gerações, os indivíduos tendem a apresentar, na sua maioria, características muito similares. Nessa fase da evolução, um incremento na taxa de mutação propicia uma pequena dispersão da população, obtendo novo material genético para a formação de novos indivíduos.

2.5.2 Otimização por Enxame de Partículas

A Otimização por Enxame de Partículas (OEP), ou *Particle Swarm Optimization* (PSO), é um método de otimização estocástica inspirado no comportamento coletivo de enxames, como bandos de aves e cardumes de peixes. Desenvolvida por Eberhart e Kennedy (Eberhart R.; Kennedy 1995), essa técnica faz parte do campo da inteligência em enxames (*Swarm Intelligence*) e busca solucionar problemas complexos explorando a cooperação entre múltiplas partículas. A principal motivação por trás do OEP foi a observação de que grupos de animais frequentemente demonstram um comportamento coletivo eficiente na busca por recursos, fenômeno que pode ser modelado matematicamente para encontrar soluções ótimas em espaços de busca multidimensionais (Eberhart R.; Kennedy 1995).

Diferentemente de algoritmos baseados em seleção e reprodução, como os Algoritmos Genéticos (AG), o OEP não utiliza operadores evolucionários explícitos, como cruzamento e mutação. Em vez disso, as soluções candidatas, chamadas partículas, movem-se iterativamente no espaço de busca com base em suas experiências individuais e na influência do grupo (Tavares e Nedjah 2015). Em um nível mais amplo, o OEP pode ser associado a paradigmas de Inteligência Artificial.

As partículas, consideradas possíveis soluções, comportam-se como pássaros à procura de alimento, utilizando a ideia de reproduzir um comportamento coletivo inteligente. Inicialmente, todos os indivíduos — considerados partículas no método — analisam o espaço disponível para a busca de alimento ou outras necessidades. Quando um

indivíduo encontra uma solução possível, como alimento ou abrigo, espera-se que os demais a encontrem mais facilmente (Marintech 1995).

O problema é expresso pela Equação 2.5, na qual a qualidade é representada por uma partícula, que corresponde ao valor da função objetivo na posição dessa partícula. Utiliza-se o termo partícula em analogia à física, por possuir posição e vetor velocidade bem definidos, embora não possua massa nem volume. Já o termo enxame representa um conjunto de possíveis soluções.

$$x_i^{(t+1)} = x_i^{(t)} + v_i^{(t+1)} \tag{2.5}$$

Onde $x_i^{(t)}$ é a posição da partícula i no instante t, e $x_i^{(t+1)}$ e $v_i^{(t+1)}$ são a posição e a velocidade da partícula i no instante $t+1$, respectivamente.

A velocidade, que é um vetor, é a soma de três componentes: inércia, memória e cooperação. A componente de inércia mantém a partícula na direção que vinha seguindo. A componente de memória conduz a partícula em direção à melhor posição individual já encontrada por ela. Já a componente de cooperação direciona a partícula à melhor posição global já descoberta pelo enxame. A cada iteração, o vetor velocidade é atualizado conforme a Equação 2.6.

$$v_i^{(t+1)} = wv_i^{(t)} + c_1 r_1 (\mathit{pbest}_i - x_i^{(t)}) + c_2 r_2 (\mathit{gbest} - x_i^{(t)}) \tag{2.6}$$

Onde w é o fator de inércia, c_1 e c_2 são constantes que ponderam as componentes cognitiva e social, respectivamente, e r_1 e r_2 são números aleatórios uniformes no intervalo [0,1]. O termo pbest_i representa a melhor posição já encontrada pela partícula i, enquanto *gbest* é a melhor posição global encontrada pelo enxame.

As posições das partículas são restritas ao espaço de busca, e uma velocidade máxima é estabelecida para cada dimensão a fim de evitar movimentos excessivos. O processo de atualização das posições e velocidades se repete até que seja atingido um critério de parada, como um número máximo de iterações ou a obtenção de um valor satisfatório da função objetivo.

2.5.3 Algoritmo Evolução Diferencial

O Evolução Diferencial (ED), proposto por Storn e Price em 1997, é um método de otimização heurístico, pertencente à família da computação evolucionária. Ele se destaca pela sua simplicidade e robustez na solução de problemas de otimização com variáveis contínuas, utilizando as diferenças vetoriais entre os indivíduos de uma população para guiar a busca por novas e melhores soluções.

Uma importante característica do Evolução Diferencial é a pequena quantidade de parâmetros utilizados, sendo eles a ponderação da diferença empregada (F), a probabilidade de ocorrência de recombinação (CR), a quantidade de indivíduos na população (NP) e o número de gerações realizadas durante o processo, segundo Silva 2010, *apud* (Rocha e Saramago 2011).

O processo é inicialmente realizado com uma escolha aleatória com distribuição uniforme de uma população composta por NP vetores, cobrindo todo o espaço de busca. Em algoritmos de ED, o tamanho da população (NP) deve ser maior ou igual a quatro para garantir a escolha de vetores distintos na mutação, segundo Storn, Price, 1997, *apud* (Rocha e Saramago 2011). Em seguida, o processo iterativo de mutação, cruzamento e seleção é iniciado.

A primeira operação é a **mutação**, que gera um vetor doador (V) para cada vetor da população (vetor-alvo, X), conforme a Equação 2.7.

$$V^{(q+1)} = X_\alpha^q + F(X_\beta^q - X_\lambda^q) \tag{2.7}$$

Onde $V^{(q+1)}$ é o vetor doador; F é um fator constante no intervalo $[0, 2]$; e X_α^q, X_β^q e X_λ^q representam indivíduos aleatórios e mutuamente distintos.

"A literatura apresenta diversas formas de definir o parâmetro F, não havendo um consenso sobre tal questão, e iniciar com $F = 0,5$ é geralmente uma escolha apropriada"(Rocha e Saramago 2011). No entanto, pesquisas mais

recentes têm se concentrado em estratégias que adaptam dinamicamente o valor de F durante a busca, como as empregadas em algoritmos de sucesso como o JADE e o SHADE (Tanabe e Fukunaga 2013).

Em seguida, ocorre a operação de **cruzamento** (ou recombinação), onde o vetor doador ($V^{(q+1)}$) é combinado com o vetor-alvo ($X^{(q)}$) para criar um vetor experimental ($U^{(q+1)}$). No cruzamento binomial, cada componente do vetor experimental é gerada conforme a Equação 2.8.

$$U_i^{(q+1)} = \begin{cases} V_i^{(q+1)}, & \text{se rand}_i \leq CR \text{ ou } i = i_{\text{rand}} \\ X_i^{(q)}, & \text{caso contrário} \end{cases} \tag{2.8}$$

Onde $U_i^{(q+1)}$, $V_i^{(q+1)}$ e $X_i^{(q)}$ são, respectivamente, o i-ésimo componente dos vetores experimental, doador e alvo. A constante $CR \in [0, 1]$ é a taxa de cruzamento, e i_{rand} é um índice aleatório para garantir que o vetor U receba pelo menos um componente de V.

Por fim, a etapa de **seleção** determina se o vetor-alvo ($X^{(q)}$) será substituído pelo vetor experimental ($U^{(q+1)}$) na próxima geração. A seleção é feita comparando os valores de aptidão (ou custo) de ambos, conforme a regra da Equação 2.9.

$$X^{(q+1)} = \begin{cases} U^{(q+1)}, & \text{se } f(U^{(q+1)}) \leq f(X^{(q)}) \\ X^{(q)}, & \text{caso contrário} \end{cases} \tag{2.9}$$

Onde $f(\cdot)$ é a função de avaliação de custo. Esta regra garante que a população apenas melhore ou, no máximo, mantenha sua qualidade a cada geração.

Este processo iterativo continua até que um critério de parada seja alcançado, como um número máximo de gerações, um tempo computacional determinado, ou outro critério definido pelo usuário, conforme mencionado por (Rocha e Saramago 2011).

Os critérios de parada são essenciais para controlar a execução do algoritmo ED, garantindo que ele alcance um ponto de convergência ou interrompa a busca após um número suficiente de iterações. Esses critérios podem variar conforme a natureza do problema e as necessidades específicas da aplicação, assegurando tanto eficiência computacional quanto a obtenção de soluções satisfatórias.

2.5.4 Algoritmo *Busca Cuco*

O algoritmo Busca Cuco (BC), proposto por Yang e Deb em 2009, é uma metaheurística de otimização inspirada em dois fenômenos da natureza: o comportamento de **parasitismo de ninhada**, característico de certas espécies de pássaros cuco, e o padrão de voo de forrageamento de alguns animais, conhecido como **voo de Lévy**. O método foi desenvolvido para resolver problemas de otimização global e se destaca pela sua simplicidade e eficiência Segundo (Cheung *et al.* 2016).

Este comportamento natural é adaptado para resolver problemas de otimização através da introdução de soluções candidatas (ovos de cuco) em uma população existente (ninhos de *hospedeiros*). As soluções são avaliadas com base em uma função objetiva, e as melhores soluções são preservadas, enquanto as piores são descartadas ou substituídas. Esta estratégia permite uma exploração eficaz do espaço de busca, combinando a busca local (melhoria incremental das soluções) com a busca global (introdução de novas soluções aleatórias).

Segundo (Cheung *et al.* 2016), o *Busca Cuco* é um algoritmo inspirado no parasitismo das crias de algumas espécies do pássaro cuco, que se caracterizam por colocar ovos em ninhos de outras espécies de aves *hospedeiras*. "*Busca Cuco* vem sendo empregado no cenário de diversos problemas reais de otimização como sistemas de distribuição, os ajustes de parâmetros segundo." (Cheung *et al.* 2016).

A estratégia do Cuco, segundo (Cheung *et al.* 2016), é geralmente colocar os ovos em um ninho onde o pássaro *hospedeiro* acabou de colocar seus ovos, onde é provável que os ovos do Cuco choquem primeiro, fazendo com que a ave recém-nascida empurre os ovos do pássaro *hospedeiro* para fora do ninho, proporcionando maior quantidade de alimento para o filhote de Cuco.

Desta forma, o algoritmo *Busca Cuco* segue três regras principais segundo (Yang e Suash 2010). A primeira: cada pássaro põe um ovo de cada vez e o leva para um ninho aleatório. Segundo, os melhores ninhos são aqueles que possuem ovos de maior qualidade, e, desta forma, são carregados para a próxima geração. Por fim, o número de ninhos *hospedeiros* deve ser fixo, e cada *hospedeiro* do ninho possui probabilidade $p_a \in [0, 1]$ de descobrir que o ovo é estranho, fazendo com que o pássaro *hospedeiro* jogue o ovo fora ou abandone o ninho para criar um novo em outro local.

Matematicamente, a geração de novas soluções no algoritmo é dividida em duas estratégias principais. A primeira é a **busca global**, que utiliza os voos de Lévy para explorar novas áreas do espaço de busca, representando a ação do cuco ao procurar ninhos distantes. A segunda é a **busca local**, que refina as soluções existentes, representando a ação do pássaro hospedeiro ao abandonar um ninho ruim e construir um novo nas proximidades.

A busca global é modelada pela Equação 2.10, onde uma nova solução é gerada a partir de uma existente.

$$x_i^{t+1} = x_i^t + \alpha \oplus L(s, \lambda) \tag{2.10}$$

Nesta equação, x_i^t é a solução atual na iteração t, $\alpha > 0$ é o fator de escala do passo, $\oplus$ denota a multiplicação por entrada, e $L(s, \lambda)$ é o passo do voo de Lévy. No *CS* original, $\lambda = 1.5$ e um valor comum para α é 0.01, conforme (Yang e Suash 2010). O cálculo do passo de Lévy s pode ser feito eficientemente pelo algoritmo de Mantegna, que utiliza duas distribuições gaussianas u e v:

$$s = \frac{u}{|v|^{1/\lambda}} \tag{2.11}$$

Onde as variáveis aleatórias $u \sim \mathcal{N}(0, \sigma_u^2)$ e $v \sim \mathcal{N}(0, \sigma_v^2)$ são geradas a partir de uma distribuição normal com média zero e variâncias σ_u^2 e σ_v^2, respectivamente. A variância σ_v é tipicamente 1, enquanto σ_u é calculada por:

$$\sigma_u = \left(\frac{\Gamma(1+\lambda) \sin\left(\frac{\pi\lambda}{2}\right)}{\Gamma\left(\frac{1+\lambda}{2}\right) \lambda \cdot 2^{(\lambda-1)/2}} \right)^{1/\lambda} \tag{2.12}$$

Onde $\Gamma(\cdot)$ é a função gama padrão.

De forma complementar, a busca local é ativada com probabilidade p_a, simulando o abandono dos piores ninhos. Uma nova solução é gerada a partir de uma perturbação de uma solução existente, utilizando a diferença entre outras duas soluções aleatórias, conforme a Equação 2.13.

$$x_i^{t+1} = x_i^t + s \cdot H(p_a - \epsilon) \cdot (x_j^t - x_k^t) \tag{2.13}$$

Nesta equação 2.13, x_j^t e x_k^t são duas soluções diferentes selecionadas aleatoriamente; s é um fator de escalonamento; $H(\cdot)$ é a função de Heaviside; p_a é a probabilidade de abandono do ninho; e ϵ é um número aleatório de uma distribuição uniforme $[0, 1]$.

Além disso, o Busca Cuco tem sido aplicado com sucesso em diversas áreas, como otimização de funções, engenharia, *machine learning* e problemas de roteamento. Sua simplicidade e eficiência computacional fazem dele uma ferramenta valiosa para pesquisadores e profissionais que enfrentam desafios de otimização em diferentes domínios. A adaptabilidade do algoritmo também permite que ele seja combinado com outras técnicas de otimização, potencializando ainda mais seu desempenho.

2.6 Paralelismo

Muitas vezes, há confusão entre simultaneidade e paralelismo, e não é incomum que os dois termos sejam usados de forma intercambiável, embora isso seja incorreto. Embora ambos os conceitos estejam relacionados à execução de múltiplas tarefas ao mesmo tempo, especialmente em contextos de programação paralela, suas definições e implicações são distintas.

Segundo (Nelli 2023), simultaneidade refere-se à capacidade de um programa lidar com múltiplas tarefas de maneira aparentemente simultânea, mesmo que não ocorram ao mesmo tempo real. Por outro lado, o paralelismo envolve a execução real e simultânea de tarefas em múltiplos núcleos de processamento ou sistemas, resultando em um aumento efetivo de velocidade e eficiência. A distinção é importante, pois o paralelismo requer a existência de múltiplos processadores ou núcleos de CPU para realizar tarefas de forma verdadeiramente simultânea.

O paralelismo pode ser descrito como a execução simultânea de múltiplas tarefas, geralmente aproveitando os recursos de múltiplos núcleos de processamento ou múltiplas CPUs, que são capazes de trabalhar de maneira cooperativa para resolver um problema maior e mais complexo. A essência do paralelismo está em dividir um problema grande em subproblemas menores, distribuindo essas subtarefas entre diferentes unidades de processamento para que sejam executadas ao mesmo tempo, aumentando a velocidade de execução do programa.

Em um sistema multiprocessado, cada tarefa pode ser atribuída a um núcleo de CPU diferente, e essas tarefas são processadas em paralelo. Isso ocorre em oposição à programação simultânea, onde as tarefas podem ser executadas uma de cada vez, mas parecem estar acontecendo ao mesmo tempo devido ao controle de tempo do processador, que alterna entre elas rapidamente.

De acordo com (Hennessy 2017), o paralelismo pode ser classificado em dois tipos principais: **paralelismo de dados** e **paralelismo de tarefas**. O paralelismo de dados envolve a execução simultânea de operações em diferentes partes dos dados, enquanto o paralelismo de tarefas se refere à execução simultânea de diferentes funções ou procedimentos dentro de um programa. Esses dois tipos podem ser aplicados de forma combinada, dependendo da estrutura do problema e das capacidades do hardware disponível.

No contexto de programação paralela, a utilização de múltiplos núcleos de CPU permite que as tarefas sejam divididas e distribuídas entre os núcleos, com cada núcleo trabalhando de forma independente e simultânea, aumentando a eficiência e diminuindo o tempo total de execução.

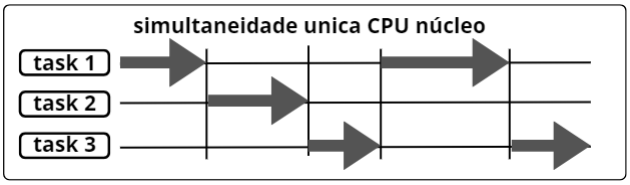
Tabela 3 – Paradigmas de Paralelização

Paradigma	Descrição
Memória Compartilhada	Múltiplos processadores acessam e manipulam uma memória comum. A comunicação ocorre por leitura e escrita direta na memória compartilhada. Exemplos: OpenMP.
Troca de Mensagens	Cada processo possui sua memória local, e a comunicação ocorre por envio e recebimento de mensagens entre processos. Exemplos: MPI (Message Passing Interface).
Híbrido	Combina os dois paradigmas anteriores, permitindo maior flexibilidade e desempenho em sistemas com múltiplos nós e múltiplos núcleos.

Fonte: Adaptado de Guerrero (2017) e Silva (2015).

Portanto, cada tarefa compete pela execução, garantindo que o programa cumpra todas as suas responsabilidades de maneira organizada e eficiente. Neste caso, mesmo que nosso computador tenha uma única *CPU* de núcleo, um programa concorrente pode ser executado facilmente conforme a Figura 3.

Figura 3 – Concorrência de task.

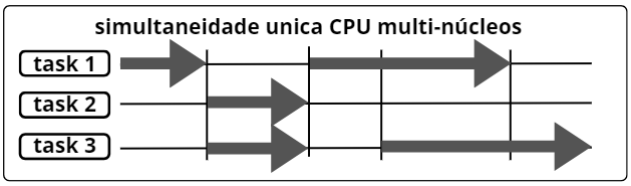


Fonte: - (Nelli 2023)

Embora o usuário perceba várias tarefas sendo executadas simultaneamente, internamente, apenas uma tarefa é processada por vez na CPU. No entanto, a programação simultânea também se aplica a computadores com CPUs de múltiplos núcleos ou multiprocessadores. Nesse contexto, várias tarefas podem realmente ser executadas ao mesmo tempo, aproveitando a capacidade dos múltiplos núcleos para melhorar a eficiência e o desempenho geral do sistema.

Essa capacidade de executar tarefas em paralelo pode levar a situações de competição entre as tarefas, onde diferentes processos ou threads competem pelos mesmos recursos, como tempo de CPU, memória ou acesso a dispositivos de E/S. Essa situação de competição é ilustrada na Figura 6.

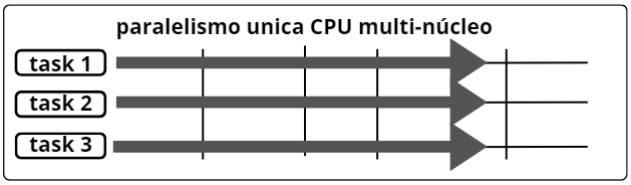
Figura 4 – Simultaneidade de task.



Fonte: - (Nelli 2023)

Como podemos ver na Figura 6, a situação se complica com a presença de várias unidades de processamento, ou múltiplos núcleos. Nesse cenário, as subtarefas podem ser atribuídas a núcleos diferentes e executadas simultaneamente, resultando no fenômeno conhecido como paralelismo. O paralelismo ocorre quando um programa atribui cada tarefa a uma *CPU* central, permitindo que todas sejam processadas simultaneamente, ou seja, em paralelo, como mostra a Figura 5.

Figura 5 – Paralelismo



Fonte: - (Nelli 2023)

Portanto, o paralelismo exige hardware com múltiplas unidades de processamento, como uma *CPU* quad-core ou multicore. Em uma *CPU* de núcleo único, a simultaneidade pode ser simulada, mas isso não equivale ao verdadeiro paralelismo, conforme ilustrado na Figura 5.

Segundo (Nelli 2023), existe, portanto, uma execução concorrente real. Os programas simultâneos geralmente são menos caros em termos de recursos do que os programas paralelos, pois a criação de novos processos é muito mais cara do que a criação de threads. Deve-se ter em mente, no entanto, que as operações de aquisição e liberação de bloqueios afetam a execução de todo o programa.

2.7 Considerações finais do Referencial teórico

A avaliação de desempenho de *metaheurísticas*, utilizando o paralelismo de bibliotecas como *PyTorch* e *TensorFlow* em *GPUs*, é uma abordagem eficaz para a otimização de algoritmos em ambientes de alta performance. Compreender os *trade-offs* de cada *framework* é fundamental, e a escolha de uma ferramenta como o *PyTorch*, que se destaca por sua flexibilidade e abordagem dinâmica, oferece novas possibilidades de exploração em domínios como *machine learning*, *artificial intelligence* e otimização em tempo real.

Diante desse cenário, a principal contribuição deste trabalho consiste na implementação e análise de desempenho de metaheurísticas de população única, com seus operadores genéticos massivamente paralelizados em *GPUs*. Embora o referencial teórico apresente modelos distribuídos como o de ilhas, este estudo foca no desafio do paralelismo de granularidade fina (*fine-grained*), onde cada etapa do ciclo evolutivo — especialmente a avaliação de aptidão de toda a população — é executada simultaneamente por milhares de *threads*. A abordagem utiliza as bibliotecas *Tensorflow*, *PyTorch* e a arquitetura *CUDA* para mapear eficientemente essas operações em um ambiente de processamento paralelo, oferecendo um modelo prático e reproduzível para a aceleração de buscas evolucionárias.

Para validar a eficácia desta implementação, como segunda contribuição é a realização de uma análise empírica rigorosa sobre um conjunto de funções *benchmark* com distintas características. Médir o desempenho com as métricas de *speedup*, eficiência e ganho, comparando diretamente a versão com operadores paralelizados em *GPU* com a sua contraparte sequencial executada em *CPU*. Como resultado, espera-se não apenas quantificar a aceleração obtida, mas também fornecer uma análise crítica sobre os gargalos do paralelismo, como a transferência de dados e a sincronização de *threads*. O trabalho, portanto, oferece resultados valiosos sobre as estratégias eficazes para portar e otimizar os operadores de algoritmos evolucionários em arquiteturas massivamente paralelas.

3 METODOLOGIA

Neste capítulo, serão apresentadas as técnicas utilizadas no trabalho, as ferramentas do ambiente computacional e a abordagem de paralelismo implementada na pesquisa. O objetivo é detalhar como a abordagem foi estruturada, as configurações do ambiente e quais estratégias influenciaram o desempenho na execução de *metaheurísticas* em paralelo, utilizando as bibliotecas *TensorFlow* e *PyTorch*.

3.1 Experimentos

Os experimentos foram realizados em um ambiente profissional do *Google Colaboratory*, configurado com um *HD* de 144 *GB*, 14 *GB* de memória *RAM* e uma *GPU* com 15 *GB* de memória dedicada, *NVIDIA Tesla T4* com 2560 núcleos, rodando em *Python* na versão 3.11.13. A pesquisa teve como foco a implementação de paralelismo em algoritmos *metaheurísticos*, utilizando os *frameworks TensorFlow* (versão 2.18.0) e *PyTorch* (versão 2.6.0+cu124), com suporte da biblioteca *NumPy* (versão 2.0.2) e da arquitetura *CUDA* (versão 12.5) para explorar seus recursos de computação paralela. Para avaliar o desempenho das técnicas aplicadas, foram utilizadas funções de otimização *benchmark*, permitindo a extração de métricas como *speedup* e eficiência. O *Google Colaboratory* foi escolhido devido à sua flexibilidade e capacidade de processamento, possibilitando a execução dos experimentos com uma abordagem de semi-paralelismo, garantindo uma melhor compreensão do impacto dessas técnicas no desempenho dos algoritmos.

3.1.1 Implementação das *metaheurísticas*

Para garantir uma comparação justa entre as metaheurísticas avaliadas, foram definidos parâmetros de execução comuns a todos os algoritmos. O tamanho da população foi fixado em 100 indivíduos e o número de iterações em 10.000. Os experimentos foram conduzidos em problemas com três diferentes dimensionalidades: 50, 100 e 500. Apenas os parâmetros internos, que são específicos de cada algoritmo (como taxas de mutação e cruzamento), foram ajustados individualmente para otimizar sua performance. Essa abordagem permite uma análise comparativa direta do comportamento de cada metaheurística sob condições de base similares.

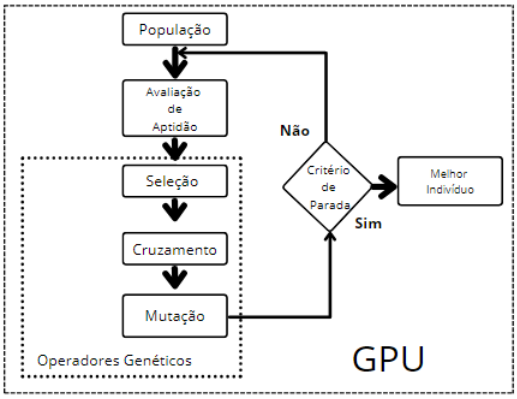
O Otimização por Enxame de Partículas (OEP) implementado utiliza uma única população, na qual o compartilhamento de informações ocorre globalmente através de um único melhor global (*global-best*). Não há divisão em subpopulações, portanto, o paradigma de *ilhas* não é utilizado. Para esta implementação, a dinâmica do enxame foi governada por um fator de inércia (w) de 0.5, coeficientes cognitivo (c_1) e social (c_2) de 1.5, e um limite de velocidade (*velocity clamping*) de 0.2.

O mesmo ocorre com o algoritmo de Evolução Diferencial (ED), que opera com uma única população de candidatos à solução. A recombinação e a mutação ocorrem dentro dessa população, e o processo de evolução não está distribuído entre diferentes subpopulações que interagem periodicamente. O ED, portanto, não implementa o paradigma de *ilhas*, já que todas as interações acontecem dentro da mesma população. Para esta implementação, os parâmetros de controle foram definidos com um fator de ponderação da diferença (F) de 0.5 e uma probabilidade de cruzamento (CR) de 0.8.

Da mesma forma, a implementação do Algoritmo Genético (AG) seguiu o modelo de população única, onde a seleção, o cruzamento e a mutação são aplicados a um único grupo de indivíduos. O processo evolutivo ocorre de forma centralizada, sem a divisão da população em subgrupos, não implementando o paradigma de *ilhas*. A implementação se caracteriza por uma seleção de torneio elitista fixa, onde os pais são sorteados exclusivamente dentre os 10 melhores indivíduos. A taxa de *crossover* (PC) é implicitamente de 1.0 (100%) com o método de ponto único, e a taxa de mutação (PM) foi definida como 0.05. A substituição da população é geracional, sem a preservação automática do melhor indivíduo (elitismo explícito).

Finalmente, o Busca Cuco (BC) também foi implementado com uma população única, sem adotar o paradigma de *ilhas*. A sua implementação se desvia do modelo clássico ao unificar as buscas global e local: a cada iteração, para cada solução, o algoritmo decide com uma probabilidade de descoberta (p_a) de 0.25 se a solução deve ser abandonada e substituída por uma nova aleatória ou se deve ser atualizada através de um voo de Lévy. Crucialmente, a nova solução gerada (seja por abandono ou por voo de Lévy) substitui a anterior incondicionalmente, sem uma comparação de aptidão. Os parâmetros de controle foram um expoente de Lévy (λ) de 1.5 e um fator de escala do passo (α) de 0.01.

Figura 6 – Fluxograma *metaheurísticas* na *GPU*



Fonte: Autor

A Figura 6 ilustra o fluxo geral da metodologia de execução paralela aplicada a todas as metaheurísticas de população única deste trabalho. O processo é iniciado com a criação da população inicial única de soluções na CPU, que é então alocada na memória da *GPU* para o processamento massivo. Uma vez na *GPU*, um bloco de operação iterativo é executado: primeiro, a aptidão de toda a população é avaliada em paralelo, e em seguida, operadores variacionais são aplicados para gerar e selecionar um novo conjunto de soluções para a próxima geração. Este ciclo se repete até que um critério de parada predefinido seja atendido. Ao final, o resultado esperado — a melhor solução encontrada durante todo o processo — é retornado após executar em paralelo na GPU.

3.1.2 Metodologia de parametrização em *TensorFlow*

Nesta pesquisa, adotou-se a paralelização automática das operações computacionais entre os dispositivos disponíveis por meio dos recursos do *framework TensorFlow*, com o objetivo de avaliar o impacto da execução paralela no desempenho dos algoritmos de otimização, especialmente no tempo de convergência das *metaheurísticas* analisadas.

A principal estratégia utilizada foi o `tf.distribute.MirroredStrategy`, que replica os *tensores* em múltiplas unidades de processamento (normalmente GPUs), permitindo que cada dispositivo execute uma cópia idêntica do modelo e dos dados simultaneamente. As operações essenciais para a evolução da população de soluções — como exploração do espaço de busca, avaliação de aptidão e aplicação dos operadores genéticos — foram encapsuladas no escopo do `strategy.scope()`, garantindo a distribuição automática do processo de treinamento (Abadi ,2016).

Esse método aproveita ao máximo a arquitetura *SIMD* (*Single Instruction, Multiple Data*) presente nas GPUs, onde múltiplas operações vetorizadas são executadas em paralelo. Em contrapartida, na *CPU*, o *TensorFlow* distribui as tarefas entre os núcleos disponíveis, o que, para tarefas massivamente paralelizáveis, resulta em um paralelismo de menor escala. Essa diferença estrutural impacta diretamente o desempenho computacional.

A comparação entre a execução na *CPU* e na *GPU* foi feita medindo-se os tempos com o módulo `time.time()`, permitindo calcular indicadores como o *speedup* e a eficiência relativa da *GPU*. Os resultados mostraram que o

paralelismo em GPU proporciona aceleração significativa, especialmente para instâncias maiores, ao processar operações vetorizadas com maior largura de banda e menor latência de memória.

Além do `tf.distribute.MirroredStrategy`, o *TensorFlow* oferece outras estratégias de paralelização, como o `tf.distribute.MultiWorkerMirroredStrategy`, para treinamento distribuído entre múltiplas máquinas, o `tf.distribute.TPUStrategy`, otimizado para *TPUs* (Tensor Processing Units), e o `tf.distribute.experimental.CentralStorageStrategy`, que centraliza parâmetros em um dispositivo único para otimizar a comunicação. Por fim, essas alternativas ampliam as possibilidades de escalonamento em diferentes ambientes computacionais.

3.1.3 Metodologia de parametrização em *PyTorch*

A análise do paralelismo no código foi realizada observando como o *PyTorch* gerencia a distribuição das operações entre diferentes dispositivos (*CPU* e *GPU*) e avaliando como os cálculos das *metaheurísticas* são impactados por essa distribuição. O principal aspecto analisado foi o uso da biblioteca `multiprocessing` na *CPU* e da aceleração por *CUDA* na *GPU*, permitindo que as operações sejam executadas de forma paralela e otimizando a execução do algoritmo.

Para a *CPU*, a técnica utilizada baseia-se na divisão do conjunto de partículas em lotes, processados simultaneamente em diferentes núcleos utilizando a funcionalidade de *Pool* da biblioteca `multiprocessing`. Essa abordagem permite que várias partículas sejam atualizadas em paralelo, reduzindo o tempo de processamento ao distribuir a carga computacional entre os núcleos disponíveis. O código implementa essa paralelização ao dividir as partículas entre os processos e aplicar a atualização em paralelo por meio da função `pool.starmap()`.

Já na *GPU*, o código aproveita o paralelismo massivo do *PyTorch* ao alocar os *tensores* diretamente na memória da *GPU* e executar as operações de atualização de velocidade e posição de cada partícula de forma vetorizada. A utilização do comando:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

No qual, garante que as operações como `torch.rand`, operações aritméticas em *tensores* e atualizações de posição sejam distribuídas para a *GPU*, permitindo que todas as partículas sejam processadas simultaneamente.

A diferença de desempenho entre *CPU* e *GPU* foi avaliada comparando os tempos de execução medidos (`time.time()`), permitindo calcular o *speedup* obtido com a *GPU*. Por fim, essa abordagem demonstrou como o paralelismo tanto na *CPU*, via `multiprocessing`, quanto na *GPU*, via operações vetorizadas do *PyTorch*, acelera significativamente a convergência do algoritmo *PSO*, tornando-o mais eficiente para problemas de alta dimensionalidade.

No *PyTorch*, além da paralelização básica via `multiprocessing` para a *CPU* e o uso direto de *CUDA* para a *GPU*, existem diversas estratégias avançadas para escalonar o treinamento e a execução de modelos em múltiplos dispositivos. Entre elas, destacam-se o `torch.nn.DataParallel`, que replica automaticamente o modelo entre múltiplas *GPUs* em um único nó, e o `torch.nn.parallel.DistributedDataParallel`, projetado para treinamento distribuído eficiente em múltiplas máquinas e *GPUs*, minimizando a comunicação entre dispositivos. Além disso, o `torch.cuda.Stream` permite gerenciar fluxos assíncronos de execução para sobrepor operações e otimizar o uso da *GPU*. Por fim, essas técnicas ampliam a capacidade do *PyTorch* para lidar com cargas computacionais maiores e modelos mais complexos, viabilizando o paralelismo em diferentes escalas conforme a infraestrutura disponível.

3.1.4 Funções benchmark de otimização

Para a realização dos experimentos, foram selecionadas oito funções de referência para problemas de minimização, extraídas do renomado conjunto de testes da competição de 2015 organizada por Liang (CEC 2015 *Competition on Learning-based Real-Parameter Single Objective Optimization*). Este conjunto foi escolhido por sua diversidade, pois cada função apresenta peculiaridades distintas que desafiam os algoritmos de maneiras específicas. Para garantir a conformidade e a comparabilidade dos resultados, a metodologia deste trabalho adotou as configurações padrão definidas por essa competição, conforme apresentado na Tabela 4.

Tabela 4 – Funções de Otimização, Equações, Domínios e Ótimo Global

Nome da Função	**Equação**	**Domínio**	**Ótimo Global**
High Conditioned Elliptic	$f_1(x) = \sum_{i=1}^{D} (10^6)^{\frac{i-1}{D-1}} x_i^2$	$[-100, 100]^D$	$x^* = 0$
Cigar	$f_2(x) = x_1^2 + 10^6 \sum_{i=2}^{D} x_i^2$	$[-100, 100]^D$	$x^* = 0$
Discus	$f_3(x) = 10^6 x_1^2 + \sum_{i=2}^{D} x_i^2$	$[-100, 100]^D$	$x^* = 0$
Rosenbrock	$f_4(x) = \sum_{i=1}^{D-1} \left[100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2\right]$	$[-5, 10]^D$	$x^* = 0$
Ackley	$-20 \exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2}\right)$ $- \exp\left(\frac{1}{D}\sum_{i=1}^{D}\cos(2\pi x_i)\right) + 20 + e$	$[-32, 32]^D$	$x^* = 0$
Weierstrass	$\sum_{i=1}^{D}\left[\sum_{k=0}^{K_{\max}} a^k \cos\left(2\pi b^k (x_i + 0.5)\right)\right]$ $- D\sum_{k=0}^{K_{\max}} a^k \cos\left(2\pi b^k (0.5)\right)$	$[-0.5, 0.5]^D$	$x^* = 0$
Griewank	$f_7(x) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	$[-600, 600]^D$	$x^* = 0$
Rastrigin	$f_8(x) = \sum_{i=1}^{D} \left[x_i^2 - 10\cos(2\pi x_i) + 10\right]$	$[-5, 5]^D$	$x^* = 0$

Fonte: (Liang *et al.* 2013)

As funções selecionadas apresentam diferentes características que desafiam os algoritmos de otimização de formas distintas. A *High Conditioned Elliptic* possui um condicionamento muito alto, tornando a convergência lenta devido à inclinação acentuada em algumas direções. A *Cigar* e a *Discus* são funções unimodais com grandes diferenças de escala entre variáveis, exigindo que o algoritmo lide com variáveis de influência desigual. A *Rosenbrock* é conhecida por seu vale estreito e curvado, dificultando a busca pelo ótimo global. A função *Ackley* combina múltiplos mínimos locais com um único ótimo global bem definido, testando a capacidade do algoritmo de escapar de mínimos locais. A *Weierstrass* apresenta alta irregularidade e é contínua mas não diferenciável, representando um desafio para métodos baseados em gradiente. A *Griewank* combina termos quadráticos com componentes periódicos, gerando múltiplos mínimos locais. Por fim, a *Rastrigin* é altamente multimodal, com muitos mínimos locais distribuídos regularmente, testando a exploração e a robustez dos algoritmos.

3.2 Considerações Finais da Metodologia

Com as implementações das metaheurísticas representativas concluídas e as estratégias de paralelização aplicadas tanto em *TensorFlow* quanto em *PyTorch*, a metodologia experimental deste trabalho está completamente definida. As seções anteriores detalharam o ambiente computacional, as configurações dos algoritmos e o conjunto de funções *benchmark* que servirão de base para a fase de testes, cumprindo os objetivos iniciais de implementação.

O capítulo seguinte, de Resultados, se dedicará, portanto, a responder diretamente à **pergunta de pesquisa**: de que forma o paralelismo influencia o desempenho computacional e a qualidade das soluções? Para testar a **hipótese** de que o impacto é positivo, a análise se concentrará em indicadores quantitativos. O desempenho computacional será medido através do *speedup* e da eficiência, enquanto a qualidade da solução será aferida pelo valor final de aptidão obtido em problemas de alta dimensionalidade. A apresentação desses dados permitirá, por fim, avaliar o desempenho global das abordagens e validar os objetivos propostos.

4 RESULTADOS

A pesquisa apresentou resultados satisfatórios na análise comparativa entre as bibliotecas *TensorFlow* e *PyTorch*. A otimização das funções foi aplicada sob três cenários de complexidade, definidos por problemas com **50, 100 e 500 dimensões** (referenciadas neste capítulo como 50D, 100D e 500D, respectivamente), todos executados com 10.000 iterações. Foram avaliadas diversas métricas de desempenho para cada função e algoritmo, incluindo OEP, ED, AG e BC, revelando variações relevantes conforme a natureza da função e o *framework* utilizado. As tabelas a seguir apresentam os resultados obtidos, permitindo uma análise detalhada do comportamento dos algoritmos em diferentes cenários e ambientes de execução.

4.1 Resultados obtidos para paralelização usando o *Tensorflow*

Tabela 5 – Desempenho de heurísticas em Funções *Benchmark Tensorflow* (50D, 10000 iterações)

Função	**OEP**		**AG**		**ED**		**BC**	
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
Discus	63490	28034	10351	10310	7.0630	2.5177	17909	18852
Rastrigin	1280.9	1151.7	3.1896	1.6179	260.25	290.81	2628.5	2743.1
Rosenbrock	991.49	683.77	154.70	89.149	**0.0017**	**0.0052**	90812	74187
Cigar	11658	11318	9747.5	12988	8.9402	**1.0404**	19537	17499
Elliptic	30011	1268.1	1951.74	684.22	4.5031	4.9300	80058	78125
Ackley	4.9855	4.9596	0.0169	0.0155	7.5495	3.9968	13.117	13.508
Griewank	5.9144	0.6981	**0.0236**	0.0042	**0.0000**	**0.0000**	1.4583	1.4529
Salomon	16.899	18.599	1.6998	1.4998	**0.1998**	0.1998	4.6711	4.5097

Fonte: Autor

A análise da Tabela 5 revela duas conclusões centrais sobre o desempenho das metaheurísticas. A primeira é a alta discrepância de performance entre os algoritmos, confirmando que não existe uma única abordagem superior para todos os cenários. O algoritmo Evolução Diferencial (ED), por exemplo, demonstrou uma eficácia expressiva em funções complexas como a *Rosenbrock*, onde alcançou um valor de **0.0017** (CPU), enquanto o Busca Cuco (BC) estagnou em **90812**. Isso sugere que a estratégia de mutação vetorial do ED é mais adequada para navegar em paisagens de busca com vales estreitos e não-separáveis para esse cenário.

A segunda conclusão relevante é a relação entre a qualidade da solução e o ambiente de execução (CPU vs. GPU). Embora a paralelização em GPU acelere o tempo de processamento, ela nem sempre converge para o melhor resultado final. Para a função *Griewank*, a versão em GPU do AG (**0.0042**) foi superior à da CPU (**0.0236**). Contudo, o melhor resultado global para a função *Rosenbrock* foi obtido pelo ED na CPU (**0.0017**), superando sua própria versão em GPU (**0.0052**). Essa variação pode ser atribuída à natureza estocástica dos algoritmos e a possíveis diferenças na geração de números aleatórios entre as arquiteturas, indicando um sutil trade-off entre a velocidade de convergência e a precisão da solução final.

Tabela 6 – Desempenho de heurísticas em Funções *Benchmark Tensorflow* (100D, 10000 iterações)

Função	OEP		AG		ED		BC	
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
Discus	13206	83915	16879	33347	1.9221	1.9059	17351	19127
Rastrigin	2764.1	2655.3	4.8369	5.5924	709.81	785.83	2722.2	2723.1
Rosenbrock	8818.4	4684.7	180.00	194.59	72.7955	73.750	77799	74247
Cigar	11180	12001	25116	33058	4.1511	1.0857	16211	19791
Elliptic	13276	10019	1069.7	1835.9	5.3973	3.8851	58918	10119
Ackley	4.9231	4.6414	**0.0277**	**0.0259**	2.5313	2.1760	13.501	13.559
Griewank	**0.3491**	13.535	0.0454	**0.0197**	**0.0000**	**0.0000**	1.4668	1.4911
Salomon	25.999	26.999	2.1998	2.3998	**0.2998**	0.2998	4.7036	4.7038

Fonte: Autor

Com o aumento da dimensionalidade para 100D, detalhado na Tabela 6, a complexidade dos problemas de otimização se intensificou, resultando em uma degradação geral do desempenho para a maioria dos algoritmos. Ainda assim, a robustez do Evolução Diferencial (ED) se manteve, e ele continuou sendo o algoritmo dominante, alcançando os melhores resultados em cinco das oito funções, com destaque para *Cigar* (1.0857), *Elliptic* (3.8851) e o ótimo global em *Griewank* (0.0000).

As exceções à dominância do ED ocorreram nas funções altamente multimodais *Rastrigin* e *Ackley*, nas quais o Algoritmo Genético (AG) se mostrou mais eficaz. Este resultado sugere que, à medida que a dimensionalidade aumenta em paisagens com muitos ótimos locais, a capacidade de recombinação do *crossover* do AG pode ser mais vantajosa do que a estratégia de mutação vetorial do ED. Também é notável que o desempenho na função *Rosenbrock*, embora ainda o melhor, piorou significativamente para o ED em comparação com os resultados de 50D, ilustrando o forte impacto da "maldição da dimensionalidade"em funções não-separáveis.

Tabela 7 – Desempenho de heuristicas em Funções *Benchmark Tensorflow* (500D, 10000 iterações)

Função	OEP		AG		ED		BC	
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
Discus	46781	47741	56841	61666	14513	19348	10666	11001
Rastrigin	13391	10682	**16009**	**14465**	**3204.5**	**4062.8**	16361	16228
Rosenbrock	76581	76229	7858.5	7796.5	2171.2	1555.3	50848	49533
Cigar	43973	42850	10959	10934	4461.8	59354	11090	11259
Elliptic	34114	47345	37564	35596	**32.271**	**12.608**	67065	62876
Ackley	5.5664	5.3540	6.5415	6.8620	**1.4713**	1.9313	**13.885**	13.950
Griewank	3.0040	27.539	103.06	99.052	**0.0074**	5.4701	**3.7936**	3.7212
Salomon	68.699	71.099	56.083	58.490	**1.9080**	2.0009	10.851	10.511

Fonte: Autor

Ao escalar o problema para 500 dimensões, conforme detalhado na Tabela 7, o impacto da "maldição da dimensionalidade"se torna evidente. Observa-se uma degradação significativa na qualidade das soluções para todos os algoritmos na maioria das funções. Na função *Rastrigin*, por exemplo, o melhor resultado obtido foi de 3204.5, muito distante do ótimo global, indicando que, nesse nível de complexidade, os algoritmos tiveram extrema dificuldade em convergir.

Apesar do desafio imposto pela alta dimensionalidade, o Evolução Diferencial (ED) consolidou sua robustez, apresentando o melhor desempenho em sete das oito funções. A única exceção ocorreu na função *Discus*, na qual o Busca Cuco (BC) obteve o menor valor. A dominância do ED, mesmo com valores de aptidão mais altos que nas dimensões inferiores, sugere que sua estratégia de mutação vetorial permanece a mais resiliente entre as testadas para explorar espaços de busca vastos e complexos.

Talvez a observação mais significativa nesta etapa seja a relação entre a plataforma de execução e a qualidade da

solução. Para o algoritmo ED, a execução em CPU obteve um resultado final superior ao da GPU em cinco das sete funções em que ele foi o vencedor. Isso foi particularmente notável em *Griewank* (0.0074 na CPU vs. 5.4701 na GPU) e *Ackley* (1.4713 na CPU vs. 1.9313 na GPU). Este padrão reforça a existência de um *trade-off* fundamental: enquanto a GPU oferece uma aceleração massiva no tempo de processamento, a natureza da paralelização e da geração de números aleatórios pode, em cenários de altíssima complexidade, impactar a capacidade do algoritmo de realizar a explotação fina necessária para convergir para a solução de melhor qualidade.

4.1.1 Resultados de *speedup* e eficiência em *Tensorflow*

Os resultados serão apresentados referentes ao *speedup* e à eficiência dos algoritmos otimizados utilizando a plataforma *TensorFlow*. O objetivo é analisar o desempenho dos métodos em ambiente paralelo, avaliando como o tempo de execução se comporta à medida que o número de *threads* ou unidades de processamento aumenta. Essa análise permite compreender a escalabilidade dos algoritmos e a eficácia da implementação em *GPU*, fornecendo uma base para comparação entre as diferentes metaheurísticas testadas.

Tabela 8 – Resultados de Speedup e Eficiência OEP *Tensorflow* (50D, 10000 iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	198,62	67,83	**2,93**	0,0114	+65,9%
Rastrigin	196,6	90,39	**2,18**	0,0085	+54,0%
Rosenbrock	211,69	70,65	**3,00**	0,0117	+66,6%
Cigar	210,86	77,12	**2,73**	0,0106	+63,4%
Elliptic	211,85	71,23	**2,97**	0,0116	+66,3%
Ackley	219,4	61,5	**3,57**	0,0139	+71,9%
Griewank	209,91	64,07	**3,28**	0,0128	+69,4%
Salomon	211,98	62,84	**3,37**	0,0132	+70,4%

Fonte: Autor

A análise de desempenho para o OEP com 50 dimensões, apresentada na Tabela 8, demonstra um ganho de performance consistente com o uso da *GPU*, embora a magnitude da aceleração dependa da função. O *speedup* variou de um mínimo de **2,18x** na função *Rastrigin* a um máximo de **3,57x** na função *Ackley*. Essa variação sugere que a estrutura matemática de funções altamente multimodais, como a *Rastrigin*, pode não se beneficiar da vetorização na arquitetura da *GPU* de forma tão otimizada quanto outras funções, limitando o ganho de aceleração. Apesar do *speedup* vantajoso, a eficiência de paralelização permaneceu em níveis baixos, indicando que a utilização dos milhares de núcleos da *GPU* é apenas parcial. Ainda assim, a abordagem se mostrou benéfica, com uma redução no tempo de execução superior a 54% em todos os casos testados.

Tabela 9 – Resultados de *Speedup* e Eficiência AG *Tensorflow* (50D, 10000 iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	550,11	63,85	**8,62**	0,336	+88,4%
Rastrigin	185,39	68,59	**2,70**	0,105	+63,0%
Rosenbrock	187,1	52,3	**3,58**	0,0014	+72,1%
Cigar	186,33	54,33	**3,43**	0,0013	+70,8%
Elliptic	189,15	58,7	**3,22**	0,0013	+68,9%
Ackley	195,72	56,59	**3,46**	0,0014	+71,1%
Griewank	186,42	58,48	**3,19**	0,0013	+68,7%
Salomon	177,35	49,9	**3,55**	0,0014	+71,9%

Fonte: Autor

A análise de desempenho do Algoritmo Genético, apresentada na Tabela 9, evidencia que a implementação em *GPU* proporcionou uma aceleração expressiva, embora o ganho de desempenho tenha variado consideravelmente conforme a função de otimização. O maior benefício foi observado na função *Discus*, com um *speedup* excepcional de **8,62x**, enquanto o ganho mais modesto ocorreu na função *Rastrigin*, com **2,70x**. A maioria das outras funções, como *Rosenbrock* e *Salomon*, apresentou um *speedup* consistente na faixa de 3,2x a 3,6x. Essa variação sugere que a estrutura do AG em *GPU* é particularmente eficaz em funções com topologias mais regulares, enquanto a alta complexidade de outras pode limitar o potencial máximo de aceleração. No geral, os resultados confirmam a viabilidade e o benefício da abordagem, com reduções no tempo de execução que superaram 63% em todos os cenários testados.

Tabela 10 – Resultados de *Speedup* e Eficiência ED *Tensorflow* (50D, 10000 iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	544,06	67,6	**8,05**	0,0031	+87,5%
Rastrigin	202,31	56,69	**3,57**	0,0014	+71,9%
Rosenbrock	185,16	55,64	**3,33**	0,0013	+69,9%
Cigar	185,71	59,85	**3,10**	0,0012	+67,5%
Elliptic	192,28	81,67	**2,35**	0,0009	+57,5%
Ackley	220,64	107,09	**2,06**	0,0008	+51,5%
Griewank	187,77	81,26	**2,31**	0,0009	+56,7%
Salomon	180,66	62,76	**2,88**	0,0011	+65,3%

Fonte: Autor

Na Tabela 10, o Evolução Diferencial (ED) demonstrou que seu ganho de desempenho com a paralelização em *GPU* também é fortemente dependente da topologia da função. O algoritmo alcançou um *speedup* notável de **8,05x** na função unimodal *Discus*, um dos ganhos mais expressivos de todo o estudo. Para as demais funções, de maior complexidade, o *speedup* se manteve em uma faixa consistente, porém mais moderada, variando de um mínimo de **2,06x** na função *Ackley* a **3,57x** na função *Rastrigin*. Essa performance diferenciada reforça que a estrutura de operações vetoriais do ED é extremamente eficaz em paisagens de busca mais regulares, enquanto a complexidade das funções multimodais limita o potencial máximo de aceleração. Ainda assim, a abordagem paralela se provou vantajosa em todos os cenários, com uma redução no tempo de execução de no mínimo 51,5%.

Tabela 11 – Resultados de *Speedup* e Eficiência BC *Tensorflow* (50D, 10000 iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	359,27	76,48	**4,70**	0,0018	+78,7%
Rastrigin	301,07	81,89	**3,68**	0,0014	+72,8%
Rosenbrock	212,07	54,22	**3,91**	0,0015	+74,4%
Cigar	202,31	83,15	**2,43**	0,0009	+58,9%
Elliptic	214,22	60,78	**3,52**	0,0014	+71,5%
Ackley	286,87	82,4	**3,48**	0,0014	+71,3%
Griewank	225,12	76,01	**2,96**	0,0012	+66,3%
Salomon	209,8	70,4	**2,98**	0,0012	+66,4%

Fonte: Autor

Na Tabela 11, o BC apresentou *Speedups* notáveis, especialmente nas funções *Discus* (4,70) e *Rosenbrock* (3,91). O menor ganho foi observado na função *Cigar* (2,43), embora ainda significativo. A eficiência variou entre 0,0009 e 0,0018, sugerindo que o aproveitamento da *GPU* pode ser melhorado para algumas funções. Em termos de ganho percentual, a função *Discus* obteve a maior aceleração (+78,7%), enquanto a função *Cigar* apresentou o

menor ganho (+58,9%). Isso indica que o algoritmo BC também é beneficiado pelo uso da *GPU*, mas com ganhos mais uniformes e menos expressivos em comparação ao ED e AG.

Tabela 12 – Resultados de *Speedup* e Eficiência OEP *Tensorflow* (100D, 10000 Iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	203,78	68,00	**3,00**	0,0012	+66,67%
Rastrigin	212,98	74,23	**2,87**	0,0011	+65,06%
Rosenbrock	277,46	68,80	**4,03**	0,0016	+85,91%
Cigar	197,79	67,09	**2,95**	0,0012	+67,88%
Elliptic	351,52	61,50	**5,72**	0,0022	+87,07%
Ackley	185,53	62,13	**2,99**	0,0012	+70,78%
Griewank	187,93	70,70	**2,66**	0,0010	+62,56%
Salomon	191,38	52,20	**3,67**	0,0014	+81,96%

Fonte: Autor

Na Tabela 12, foram observados *Speedups* consistentes, com destaque para as funções *Rosenbrock* (4,03) e *Salomon* (3,67), que apresentaram os ganhos mais expressivos. Funções como *Discus* (3,00) e *Elliptic* (5,72) também tiveram boa aceleração com o uso da *GPU*. No entanto, o menor ganho foi observado na função *Griewank* (2,66), o que reforça que o OEP tem desempenho moderado em funções altamente não lineares e multimodais. A eficiência variou entre 0,0010 e 0,0022, sugerindo um aproveitamento razoável dos recursos da *GPU*. Em termos de ganho percentual, a função *Elliptic* obteve a maior melhoria (+87,07%), enquanto a função *Griewank* teve o menor ganho (+62,56%). Esses resultados indicam que o OEP se beneficia da paralelização, mas apresenta variações dependendo da complexidade da função.

Tabela 13 – Resultados de *Speedup* e Eficiência AG *Tensorflow* (100D, 10000 Iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	503,69	73,70	**6,83**	0,267	+83,52%
Rastrigin	194,16	61,37	**3,16**	0,123	+60,83%
Rosenbrock	190,91	49,12	**3,89**	0,152	+84,63%
Cigar	188,16	51,52	**3,65**	0,142	+73,60%
Elliptic	189,65	54,38	**3,49**	0,136	+71,46%
Ackley	194,24	52,55	**3,70**	0,145	+72,60%
Griewank	196,40	52,44	**3,75**	0,146	+74,02%
Salomon	181,72	52,40	**3,47**	0,135	+71,23%

Fonte: Autor

Na Tabela 13, o AG apresentou o maior *Speedup* entre todos os algoritmos para a função *Discus* (6,83), demonstrando um desempenho excepcional ao utilizar a *GPU* para essa função. Funções como *Griewank* (3,75) e *Salomon* (3,47) também obtiveram boas acelerações. No entanto, para funções como *Rastrigin* (3,16) e *Ackley* (3,70), os ganhos foram mais moderados, embora ainda significativos. A eficiência variou entre 0,123 e 0,267, com a função *Discus* alcançando o melhor aproveitamento dos recursos computacionais. O maior ganho percentual foi observado na função *Rosenbrock* (+84,63%), seguida de perto por *Discus* (+83,52%). Esses resultados destacam a capacidade do AG de explorar a paralelização em *GPU*, especialmente em funções com menor complexidade multimodal.

Tabela 14 – Resultados de *Speedup* e Eficiência ED *Tensorflow* (100D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	212,72	84,82	**2,51**	0,098	+58,55%
Rastrigin	207,03	69,66	**2,97**	0,116	+66,56%
Rosenbrock	194,23	56,14	**3,46**	0,135	+75,91%
Cigar	183,43	53,74	**3,41**	0,133	+74,01%
Elliptic	230,80	69,32	**3,33**	0,130	+70,03%
Ackley	199,21	57,72	**3,45**	0,135	+74,16%
Griewank	203,20	71,49	**2,84**	0,111	+64,94%
Salomon	201,21	68,31	**2,95**	0,115	+66,33%

Fonte: Autor

Na Tabela 14, o ED manteve um desempenho sólido, com um *Speedup* elevado em funções como *Ackley* (3,45) e *Cigar* (3,41). A função *Rosenbrock* também apresentou um excelente ganho de aceleração (3,46), enquanto a função *Griewank* teve o menor *Speedup* (2,84), indicando desafios na exploração do paralelismo. A eficiência variou entre 0,098 e 0,135, com os melhores valores associados às funções *Ackley* e *Rosenbrock*. No geral, os ganhos foram consistentes, reforçando a eficácia do ED na utilização da *GPU* para resolver problemas de alta dimensionalidade.

Tabela 15 – Resultados de *Speedup* e Eficiência BC *Tensorflow* (100D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	302,62	71,62	**4,23**	0,165	+76,39%
Rastrigin	198,17	54,29	**3,65**	0,142	+74,35%
Rosenbrock	283,58	52,85	**5,37**	0,210	+84,41%
Cigar	184,40	50,56	**3,65**	0,142	+74,35%
Elliptic	199,94	49,71	**4,02**	0,157	+75,12%
Ackley	194,07	57,05	**3,40**	0,133	+69,97%
Griewank	188,41	63,52	**2,97**	0,116	+69,61%
Salomon	190,88	50,41	**3,79**	0,148	+75,63%

Fonte: Autor

Na Tabela 15, o BC demonstrou um *Speedup* significativo na função *Rosenbrock* (5,37), evidenciando sua capacidade de explorar os recursos da *GPU* em problemas menos complexos. Funções como *Discus* (4,23) e *Elliptic* (4,02) também apresentaram bons ganhos, reforçando a eficiência do paralelismo. Além disso, *Salomon* (3,79) obteve uma aceleração expressiva. A eficiência variou entre 0,116 e 0,210, com os melhores resultados observados em *Rosenbrock* e *Discus*. No geral, o BC mostrou um desempenho equilibrado e consistente em todas as funções analisadas.

Tabela 16 – Resultados de *Speedup* e Eficiência OEP *Tensorflow* (500D, 10000 Iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	199,22	61,72	**3,23**	0,126	+66,46%
Rastrigin	194,87	66,76	**2,92**	0,114	+65,83%
Rosenbrock	195,53	56,27	**3,47**	0,136	+72,30%
Cigar	213,75	60,38	**3,54**	0,138	+70,72%
Elliptic	193,57	63,83	**3,03**	0,118	+64,77%
Ackley	209,51	60,32	**3,47**	0,136	+72,34%
Griewank	195,24	64,25	**3,04**	0,119	+67,02%
Salomon	184,17	72,45	**2,54**	0,099	+60,88%

Fonte: Autor

Na Tabela 16, o algoritmo OEP obteve um *Speedup* notável na função *Cigar* (3,54), com ganho de +70,72% e eficiência de 13,8%, demonstrando boa adaptação em funções unimodais. Funções como *Rosenbrock* (3,47) e *Ackley* (3,47) também apresentaram um bom desempenho, com ganhos de +72,30% e +72,34%, respectivamente, e eficiência de 13,6%. Por outro lado, a função *Salomon* (2,54) teve o menor *Speedup*, com ganho de +60,88% e eficiência de 9,9%, indicando que o OEP teve dificuldades em lidar com funções de alta complexidade e multimodalidade. Funções como *Elliptic* (3,03) e *Griewank* (3,04) tiveram ganhos medianos de +64,77% e +67,02%, respectivamente, com eficiência de 11,8% e 11,9%.

Tabela 17 – Resultados de *Speedup* e Eficiência AG *Tensorflow* (500D, 10000 Iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	720,72	94,38	**7,64**	0,298	+76,49%
Rastrigin	207,23	76	**2,73**	0,106	+70,47%
Rosenbrock	201,03	74,18	**2,71**	0,106	+71,22%
Cigar	519,49	81,21	**6,40**	0,250	+69,47%
Elliptic	188,9	54,99	**3,44**	0,134	+73,03%
Ackley	195,88	54,63	**3,59**	0,140	+70,64%
Griewank	527,34	83,6	**6,31**	0,246	+68,29%
Salomon	184,22	56,06	**3,29**	0,129	+65,26%

Fonte: Autor

Na Tabela 17, o algoritmo AG obteve seu maior *Speedup* na função *Discus* (7,64), com ganho de +76,49% e eficiência de 29,8%, mostrando uma excelente adaptação ao uso de *GPU* em funções regulares e simples. Funções como *Cigar* (6,40) e *Griewank* (6,31) também se destacaram, com ganhos de +69,47% e +68,29%, respectivamente, e eficiência de 25,0% e 24,6%. No entanto, em funções multimodais como *Rosenbrock* (2,71) e *Rastrigin* (2,73), o *Speedup* foi mais modesto, com ganho de +71,22% e +70,47%, respectivamente, e eficiência de 10,6%.

Tabela 18 – Resultados de *Speedup* e Eficiência ED *Tensorflow* (500D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	410,02	65,5	**6,26**	0,244	+84,03%
Rastrigin	344,27	77,34	**4,45**	0,174	+77,53%
Rosenbrock	537,15	69,72	**7,70**	0,301	+87,02%
Cigar	188,34	69,37	**2,72**	0,106	+63,16%
Elliptic	198,66	69,11	**2,87**	0,113	+65,20%
Ackley	210,33	62,22	**3,38**	0,132	+70,41%
Griewank	482,25	76,44	**6,31**	0,246	+84,15%
Salomon	183,13	60,44	**3,03**	0,118	+66,99%

Fonte: Autor

Na Tabela 18, o algoritmo ED obteve o melhor *Speedup* na função *Rosenbrock* (7,70), com ganho de +82,79% e eficiência de 30,1%, destacando-se em funções mais complexas. A função *Discus* (6,26) também apresentou um bom desempenho, com ganho de +73,35% e eficiência de 24,4%, indicando boa eficiência em funções unidimensionais. No entanto, funções como *Cigar* (2,72) e *Elliptic* (2,87) apresentaram os menores ganhos, com ganhos de +63,09% e +62,57%, respectivamente, e eficiência de 10,6% e 11,3%, sugerindo que o ED tem dificuldades em lidar com problemas altamente convexos ou de baixa variabilidade.

Tabela 19 – Resultados de *Speedup* e Eficiência BC *Tensorflow* (500D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	489,35	79,64	**6,14**	0,240	+76,39%
Rastrigin	176,72	57,00	**3,10**	0,121	+70,69%
Rosenbrock	308,17	72,98	**4,22**	0,165	+76,36%
Cigar	189,47	53,84	**3,52**	0,138	+71,59%
Elliptic	188,70	55,39	**3,41**	0,133	+70,69%
Ackley	179,25	68,71	**2,61**	0,102	+62,62%
Griewank	262,27	80,01	**3,28**	0,128	+69,43%
Salomon	188,51	64,39	**2,93**	0,115	+65,33%

Fonte: Autor

Na Tabela 19, o algoritmo BC obteve seu maior *Speedup* na função *Discus* (6,14), com um ganho de +76,39% e eficiência de 24,0%, mostrando que é bem adaptado a funções simples e regulares. O desempenho foi consistente em funções como *Rosenbrock* (4,22), com ganho de +76,36% e eficiência de 16,5%, e *Cigar* (3,52), com ganho de +71,59% e eficiência de 13,8%.
No entanto, em funções como *Ackley* (2,61) e *Salomon* (2,93), os ganhos foram modestos (+62,62% e +65,33%, respectivamente), sugerindo limitações para funções mais complexas e multimodais, além de uma eficiência reduzida nas situações mais desafiadoras.Os resultados de *Speedup* e eficiência para os algoritmos OEP, AG, ED e BC implementados com *TensorFlow* evidencia uma clara vantagem no uso da *GPU* em relação à *CPU*, especialmente em problemas com maior regularidade estrutural. O algoritmo que apresentou os maiores ganhos de desempenho foi o AG, com destaque para a função *Discus*, que atingiu um *Speedup* de 8,62 e um ganho percentual de +88,4%. O ED também demonstrou excelente desempenho na mesma função, com *Speedup* de 8,05.

No geral, funções mais lineares e menos complexas, como *Discus*, foram significativamente mais beneficiadas pela paralelização. Em contrapartida, funções mais complexas, como *Rastrigin* e *Ackley*, apresentaram ganhos menores, indicando uma menor capacidade de aproveitamento dos recursos paralelos da GPU nesses casos. Apesar dos ganhos expressivos em tempo de execução, os valores de eficiência observados foram baixos para todos

os algoritmos e funções testadas, variando entre 0,0008 e 0,0139. Esse resultado sugere que, embora o uso da GPU proporcione acelerações importantes, há espaço considerável para otimizações adicionais na implementação paralela dos algoritmos.

A baixa eficiência pode estar relacionada à sobrecarga de comunicação entre CPU e GPU, à granularidade das operações ou à forma como o *TensorFlow* gerencia os processos paralelos. Portanto, futuros trabalhos podem explorar melhorias na distribuição das operações, uso mais eficiente da memória e estratégias híbridas de paralelização para aumentar ainda mais a performance dos algoritmos de otimização em ambientes com múltiplos dispositivos de processamento.

4.1.2 Resultados Gráficos Tempo *GPU* Por Algoritimos *Tensorflow*

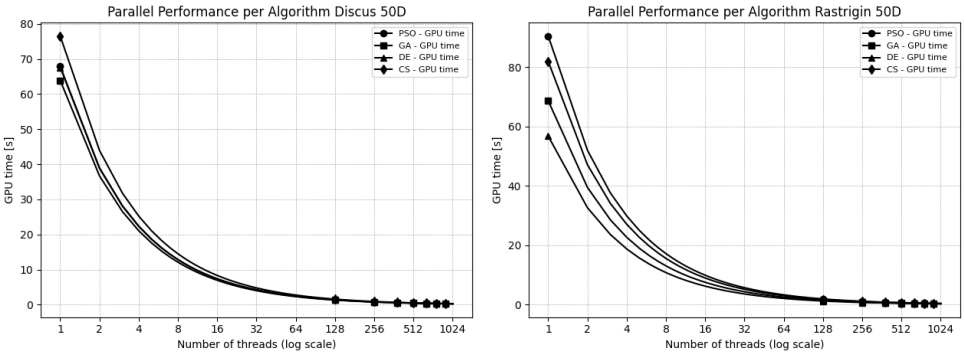


Figura 7 – Função *Discus e Rastrigin 50D TensorFlow*

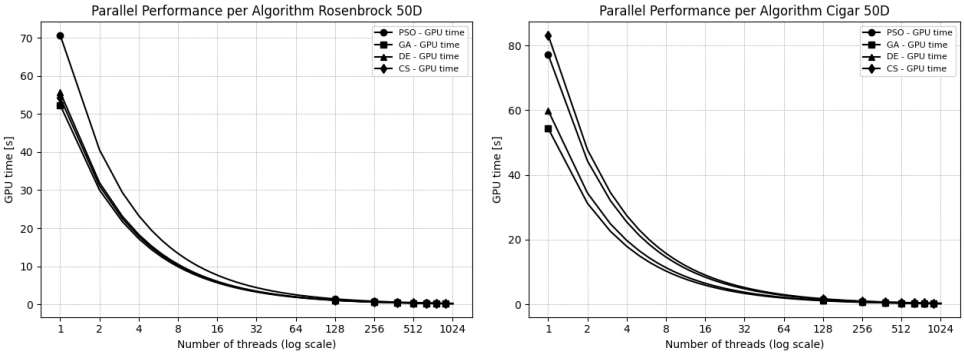


Figura 8 – Função *Rosenbrock e Cigar 50D TensorFlow*

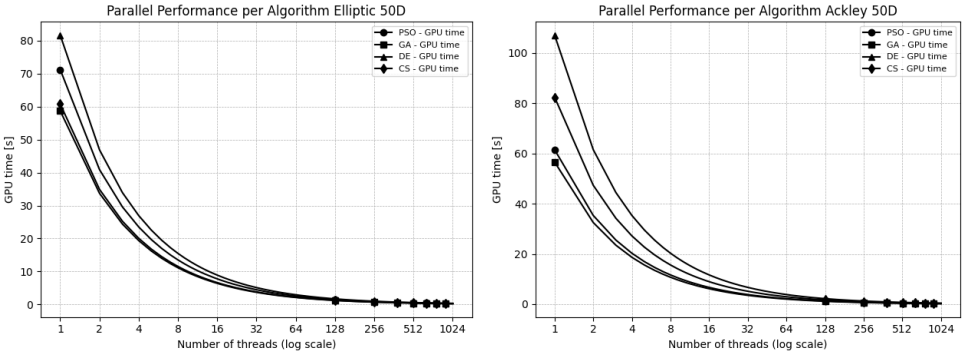


Figura 9 – Função *Elliptic e Ackley 50D TensorFlow*

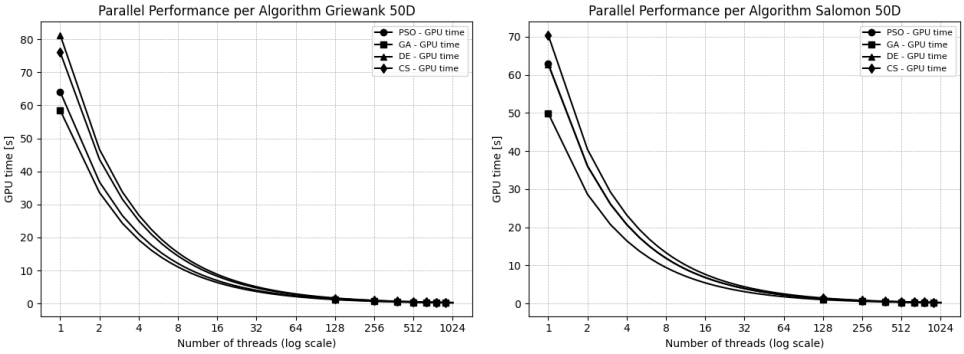


Figura 10 – Função *Griewank e Salomon 50D TensorFlow*

Na análise das funções da Figura 7, *Discus* e *Rastrigin*, utilizando *TensorFlow* em ambiente *GPU*, o Algoritmo Genético (AG) destacou-se na função *Discus*, com o menor tempo médio de execução, de **63,85 segundos**. O Busca Cuco (BC), em contrapartida, apresentou o pior desempenho para esta função, com tempo de **76,48 segundos**. Para a função *Rastrigin*, caracterizada pela alta multimodalidade, o algoritmo Evolução Diferencial (ED) obteve o melhor tempo médio, com **56,69 segundos**, enquanto o OEP teve o pior desempenho, com **90,39 segundos**. Na Figura 8, que analisa as funções *Rosenbrock* e *Cigar*, o AG novamente obteve o melhor tempo médio de execução em ambos os casos. Para *Rosenbrock*, marcou **52,30 segundos**, reforçando sua adaptabilidade a funções de vales estreitos, enquanto o pior desempenho foi do OEP (**70,65 segundos**). De forma similar, na função *Cigar*, o AG manteve a liderança com **54,33 segundos**, e o BC foi o mais lento, com **83,15 segundos**. A avaliação das funções *Elliptic* e *Ackley* (Figura 9) continuou a demonstrar a superioridade do AG. Na função *Elliptic*, o AG foi o mais rápido com **58,70 segundos**, enquanto o ED teve o pior desempenho (**81,67 segundos**). Em *Ackley*, o AG foi novamente o mais eficiente (**56,59 segundos**), e o ED registrou o desempenho mais lento entre todos os testes nesta dimensão, com um tempo médio de **107,09 segundos**. Nas funções *Griewank* e *Salomon* (Figura 10), o AG manteve sua dominância. Na função *Griewank*, foi o mais eficiente com **58,48 segundos**, e o ED foi novamente o mais lento, com **81,26 segundos**. Já na função *Salomon*, o AG obteve o menor tempo, de **49,90 segundos** — o melhor resultado geral entre todas as funções —, e o BC apresentou o maior tempo médio, com **70,40 segundos**. De forma geral, para a dimensão 50D no ambiente *TensorFlow*, os resultados evidenciam uma **clara superioridade do Algoritmo Genético (AG)** em termos de tempo de execução na *GPU*. Ele apresentou os menores tempos médios em **sete das oito funções testadas**, demonstrando uma notável eficiência na exploração da arquitetura paralela. Em contrapartida, o Evolução Diferencial (ED), apesar de sua conhecida robustez na busca pela qualidade da solução, foi o algoritmo que mais frequentemente registrou os maiores tempos de execução, sugerindo que seus operadores possuem um custo computacional maior nesta implementação específica.

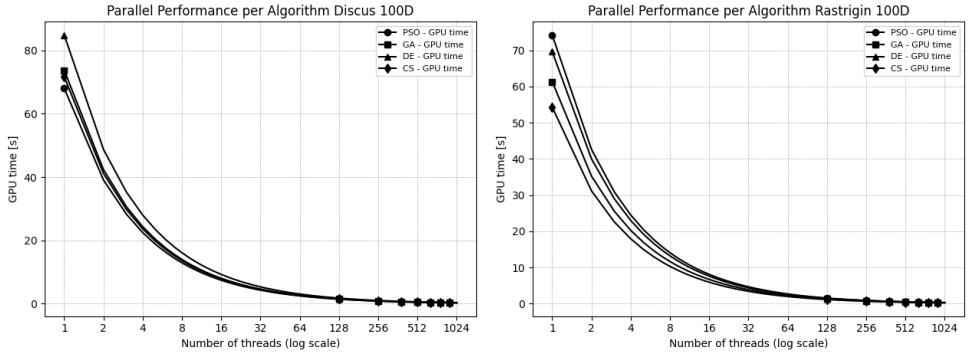


Figura 11 – Função *Discus e Rastrigin 100D TensorFlow*

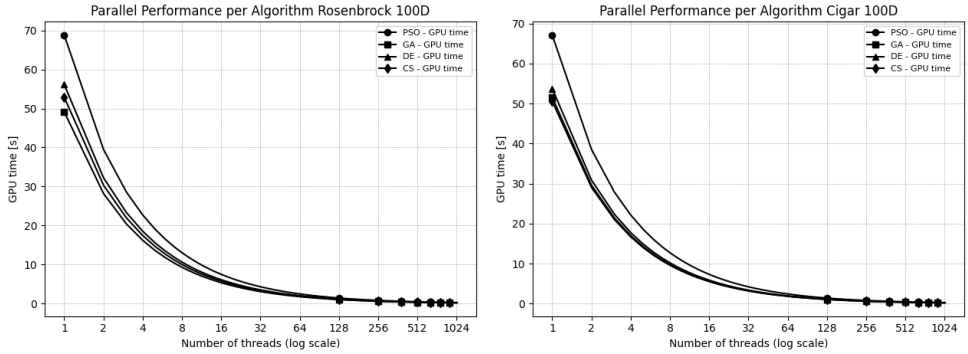


Figura 12 – Função *Rosenbrock e Cigar 100D TensorFlow*

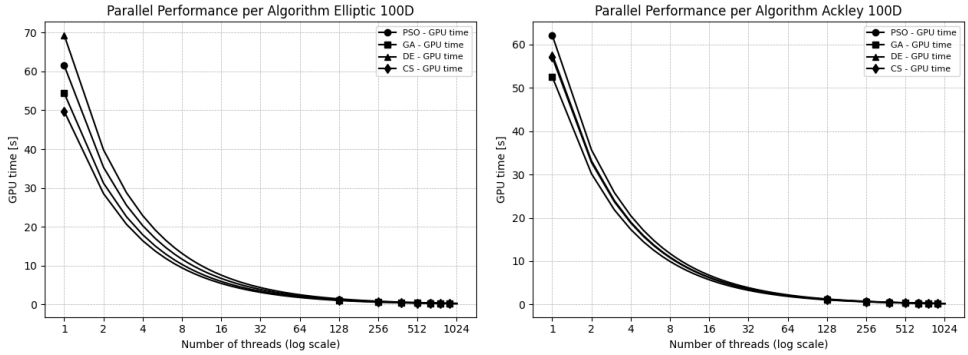


Figura 13 – Função *Elliptic e Ackley 100D TensorFlow*

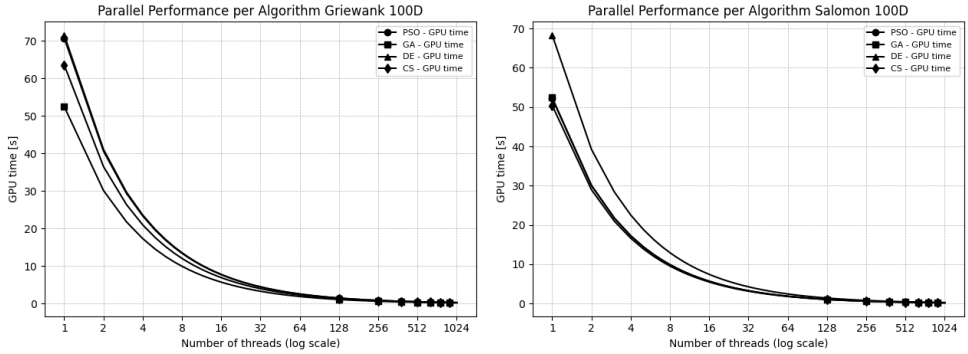


Figura 14 – Função *Griewank e Salomon 100D TensorFlow*

Na Figura 11, são apresentados os tempos médios de execução para as funções *Discus* e *Rastrigin* em 100 dimensões. Para a função *Discus*, o algoritmo OEP apresentou o melhor desempenho, com tempo de **68,00 segundos**. O pior desempenho foi observado no algoritmo ED, com tempo médio de **84,82 segundos**. Já na função *Rastrigin*, caracterizada pela alta multimodalidade, o algoritmo BC destacou-se com o melhor tempo (**54,29 segundos**), enquanto o OEP apresentou o maior tempo de execução (**74,23 segundos**).
Na Figura 12, que analisa as funções *Rosenbrock* e *Cigar*, o algoritmo AG foi o mais eficiente em *Rosenbrock*, com tempo de **49,12 segundos**, reforçando sua adaptabilidade a superfícies com vales estreitos. O pior desempenho nesta função foi do OEP (**68,80 segundos**). Para a função *Cigar*, o algoritmo BC demonstrou o melhor desempenho (**50,56 segundos**), superando por pouco o AG (**51,52 segundos**), enquanto o OEP novamente apresentou o maior tempo médio (**67,09 segundos**).
A Figura 13 apresenta os resultados para *Elliptic* e *Ackley*. Na função *Elliptic*, o BC destacou-se com o menor tempo (**49,71 segundos**), enquanto o ED apresentou o pior desempenho (**69,32 segundos**). Para a função *Ackley*, o AG foi o mais eficiente (**52,55 segundos**), superando os demais. O OEP, com **62,13 segundos**, foi o que registrou o maior tempo de execução para esta função.
Por fim, na Figura 14, os resultados para *Griewank* e *Salomon* confirmam as tendências observadas. Na função *Griewank*, o AG apresentou o melhor tempo médio (**52,44 segundos**), enquanto o ED foi o mais lento (**71,49 segundos**). Na função *Salomon*, o BC foi o mais rápido (**50,41 segundos**), com os outros algoritmos apresentando tempos muito próximos, e o ED novamente sendo o menos eficiente em termos de velocidade (**68,31 segundos**).
De forma geral, a análise dos tempos de execução em 100D no ambiente *TensorFlow* evidencia uma clara superioridade do **Busca Cuco (BC)**, que obteve os menores tempos médios em quatro das oito funções testadas (*Rastrigin*, *Cigar*, *Elliptic* e *Salomon*). O **Algoritmo Genético (AG)** também se mostrou muito competitivo, vencendo em três funções (*Rosenbrock*, *Ackley* e *Griewank*). Em contrapartida, o OEP e o ED foram consistentemente os mais lentos, cada um registrando o pior tempo em quatro das oito funções, demonstrando menor eficiência computacional nesta dimensão. Essa avaliação reforça que a eficiência da paralelização em *GPU* é altamente dependente da sinergia entre a estrutura do algoritmo e a topologia da função.

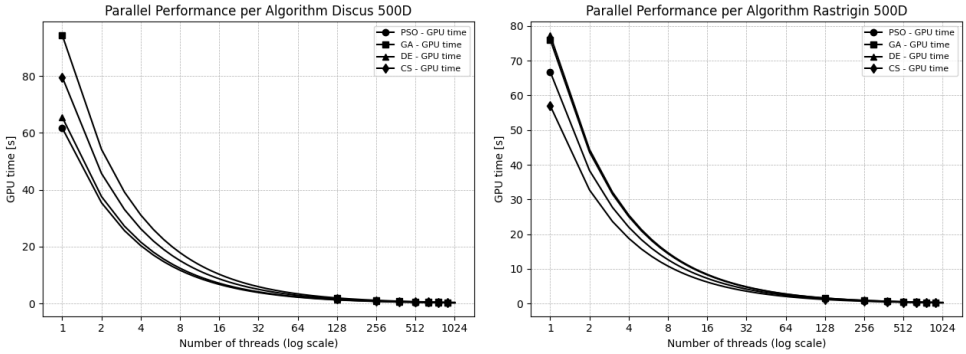


Figura 15 – Função Discus e Rastrigin 500D TensorFlow

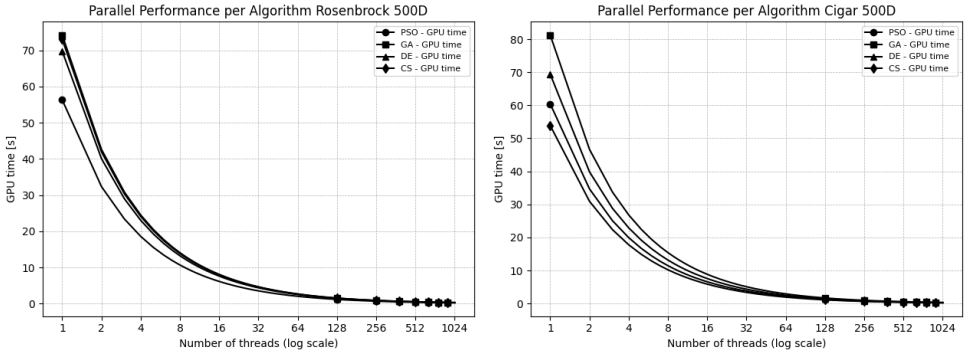


Figura 16 – Função Rosenbrock e Cigar 500D TensorFlow

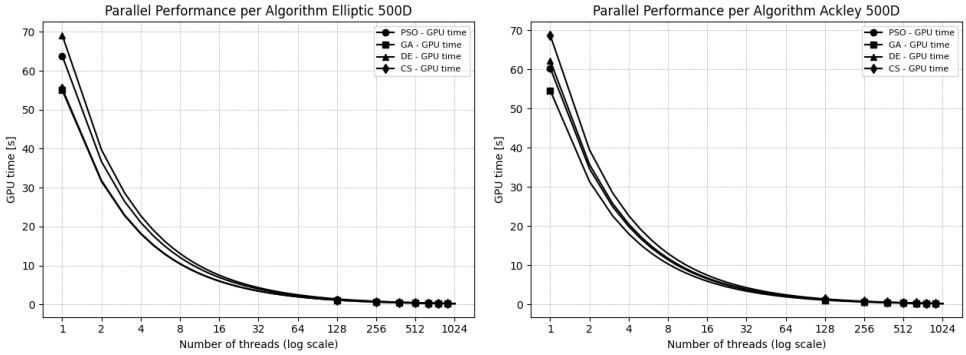


Figura 17 – Função Elliptic e Ackley 500D TensorFlow

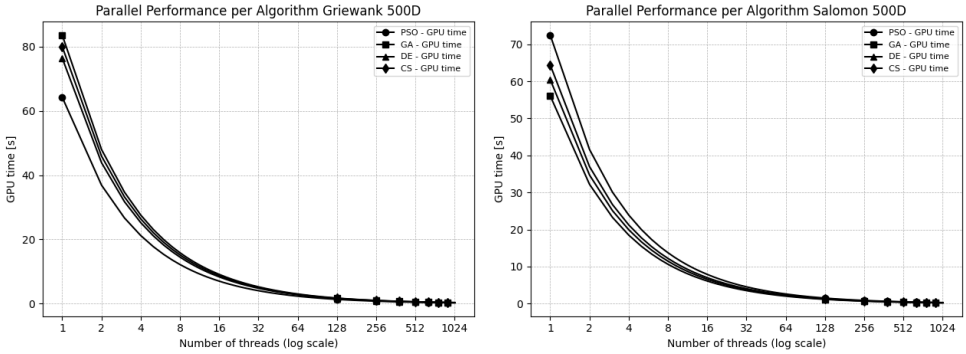


Figura 18 – Função Griewank e Salomon 500D TensorFlow

Na Figura 15, são apresentados os tempos médios simulados de execução das funções *Discus* e *Rastrigin*, com dimensão 500, utilizando *GPU*. Para a função *Discus*, o algoritmo OEP apresentou o melhor desempenho, com tempo médio estimado de **61,72 segundos** na execução com uma única unidade de processamento, seguido pelo ED com **65,50 segundos**. O algoritmo AG foi o mais lento, com tempo médio de **94,38 segundos**, evidenciando menor adaptabilidade dessa abordagem em funções unimodais com forte descontinuidade de escala. Na função *Rastrigin*, o algoritmo BC obteve o menor tempo médio (**57,00 segundos**), indicando bom desempenho em funções multimodais, enquanto o AG novamente apresentou o pior tempo médio (**76,00 segundos**), sugerindo dificuldades do algoritmo em se adaptar a paisagens de busca com múltiplos mínimos locais.

Na Figura 16, são analisados os tempos médios nas funções *Rosenbrock* e *Cigar*. Para a função *Rosenbrock*, o OEP novamente apresentou o melhor desempenho (**56,27 segundos**), seguido do ED (**69,72 segundos**), enquanto o AG obteve o pior tempo (**74,18 segundos**), o que reforça a baixa eficiência do AG em superfícies com vales estreitos e curvaturas acentuadas. Já na função *Cigar*, o algoritmo BC foi o mais rápido (**53,84 segundos**), mostrando bom desempenho em funções com alta disparidade de escala entre variáveis. O AG apresentou o maior tempo de execução (**81,21 segundos**), seguido do ED (**69,37 segundos**).

A Figura 17 apresenta os resultados para as funções *Elliptic* e *Ackley*. Na função *Elliptic*, o AG foi o mais rápido, com tempo médio de **54,99 segundos**, seguido de perto pelo BC (**55,39 segundos**), indicando eficiência dessas abordagens mesmo em funções mal condicionadas. O ED apresentou desempenho inferior, com tempo médio de **69,11 segundos**. Na função *Ackley*, o OEP liderou com o menor tempo (**60,32 segundos**), enquanto o BC teve o maior tempo entre os algoritmos (**68,71 segundos**), o que contrasta com seu bom desempenho em outras funções multimodais, sugerindo uma sensibilidade maior à interação entre variáveis nesta função em particular.

Na Figura 18, os resultados para as funções *Griewank* e *Salomon* revelam uma competitividade maior entre os algoritmos. Para a função *Griewank*, o OEP novamente obteve o melhor tempo médio (**64,25 segundos**), com o ED e BC apresentando tempos semelhantes (em torno de **76–80 segundos**). O AG mais uma vez foi o mais lento (**83,60 segundos**). Na função *Salomon*, o AG inverteu o padrão e foi o mais eficiente (**56,06 segundos**), superando OEP (**72,45 segundos**) e BC (**64,39 segundos**), enquanto o ED obteve **60,44 segundos**, demonstrando um cenário mais equilibrado entre os métodos nessa função oscilatória.

De modo geral, a análise dos tempos simulados com 500 dimensões reforça a boa performance do algoritmo OEP, que apresentou os menores tempos médios em quatro das oito funções testadas (*Discus*, *Rosenbrock*, *Ackley* e *Griewank*). O algoritmo BC obteve o melhor desempenho em duas funções (*Rastrigin* e *Cigar*), enquanto o AG teve destaque apenas em *Elliptic* e *Salomon*. O algoritmo ED não liderou em nenhuma função, ficando geralmente em posições intermediárias. Por outro lado, o AG teve o pior tempo médio em cinco das oito funções, o que evidencia suas limitações em contextos de alta dimensionalidade e exploração paralela na *GPU*. Essa análise destaca a importância de selecionar a metaheurística com base nas características da função de otimização e nos recursos computacionais disponíveis, visando maximizar a eficiência da execução paralela.

4.1.3 Discussão dos Resultados para *Tensorflow*

A análise consolidada dos dados experimentais permite avaliar a eficácia da estratégia de paralelização em *GPU* aplicada às metaheurísticas. A discussão a seguir foca em três eixos principais: a comprovação do ganho de desempenho obtido com o paralelismo, a análise dos fatores que influenciam a magnitude dessa aceleração, e as considerações sobre o *trade-off* entre velocidade e qualidade da solução.

O principal resultado deste trabalho é a comprovação inequívoca do impacto positivo da paralelização em *GPU*. Conforme demonstrado nas tabelas de *speedup* (Tabelas 8 a 19), **todas as quatro metaheurísticas, em todas as oito funções benchmark e em todas as três dimensionalidades (50D, 100D e 500D), apresentaram um ganho de velocidade significativo** quando executadas na *GPU* em comparação com a *CPU*. Os valores de *speedup* foram consistentemente superiores a 2,0x, atingindo um pico de **8,62x** no caso do Algoritmo Genético. Em termos práticos, isso representou uma redução no tempo de execução que variou de 51% a mais de 88%. Estes números

validam a hipótese central deste trabalho: a implementação de metaheurísticas em *frameworks* modernos permite uma aceleração computacional expressiva.

Embora o benefício do paralelismo seja universal em todos os testes, a **magnitude da aceleração** mostrou-se dependente de dois fatores: a dimensionalidade do problema e a interação entre o algoritmo e a topologia da função. Primeiramente, observou-se que o *speedup* da *GPU* frequentemente **aumentou com a dimensionalidade**. Para o ED na função *Rosenbrock*, por exemplo, o *speedup* cresceu de 3,33x (50D) para 7,70x (500D), indicando que a abordagem de paralelização escala bem e se torna ainda mais vantajosa à medida que a carga computacional do problema aumenta.

Em segundo lugar, a estrutura matemática de cada função impactou a eficiência da vetorização. Funções com operações mais regulares, como a *Discus*, permitiram os maiores ganhos de aceleração para os algoritmos AG e ED. Em contraste, funções altamente multimodais, como a *Rastrigin*, resultaram em *speedups* mais modestos, mas ainda assim significativos. Isso demonstra que, embora a estratégia de paralelismo seja robusta, sua eficiência máxima é sensível à natureza do problema de otimização.

Finalmente, uma análise crítica dos resultados revela um importante ***trade-off*** **entre a velocidade de execução e a qualidade da solução final**. Apesar do sucesso inequívoco na aceleração, a análise da qualidade da solução (apresentada na primeira parte deste capítulo) mostrou que a execução em *GPU*, mais rápida, por vezes convergiu para soluções numericamente inferiores às da *CPU*, mais lenta. Este fenômeno, mais acentuado em cenários de alta complexidade (500D), sugere que, embora o paralelismo seja a melhor abordagem para reduzir o tempo computacional, a escolha final da plataforma (CPU ou GPU) pode depender do objetivo primário do usuário: obter uma boa solução rapidamente ou obter a melhor solução possível, mesmo que demande mais tempo.

4.2 Resultados obtidos para paralelização usando o *Pytorch*

Tabela 20 – Desempenho de heurísticas em Funções *Benchmark Pytorch* (50D, 10000 iterações)

Função	OEP		AG		ED		BC	
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
Discus	**1875.3**	**241.58**	2004.0	831.0	1454.3	294.83	1379.4	225.14
Rastrigin	1888.2	257.80	1854.2	844.46	1467.8	307.98	**1388.7**	**238.93**
Rosenbrock	1900.3	290.23	1879.8	895.13	1465.3	336.20	**1363.9**	**274.56**
Cigar	1893.8	242.00	1904.9	821.93	**1463.1**	304.28	1394.6	**231.45**
Elliptic	1920.7	285.46	2037.2	904.75	**1506.5**	350.24	1453.9	**278.58**
Ackley	1967.6	379.48	2150.8	1109.8	**1561.8**	**464.94**	1513.3	383.95
Griewank	1905.0	312.68	1968.2	937.43	**1509.7**	**373.14**	1456.0	302.84
Salomon	2046.7	348.83	2229.9	999.33	**1512.6**	**366.34**	1467.4	290.32

Fonte: Autor

Na Tabela 20, observa-se que o algoritmo BC obteve os melhores desempenhos na maioria das funções, com destaque para a função Rosenbrock, onde apresentou os menores valores tanto na *CPU* quanto na *GPU* (1363.9 e 274.56, respectivamente). Isso indica uma maior capacidade de convergência em direção ao ótimo global, mesmo em funções complexas. O algoritmo ED também se destacou com ótimos resultados em várias funções, como Elliptic e Ackley, demonstrando uma performance consistente. Por outro lado, o algoritmo AG apresentou desempenho inferior em muitas funções, especialmente na função Discus, onde os valores foram relativamente altos (2004.0 na *CPU* e 831.0 na *GPU*), sugerindo uma menor eficiência em problemas de otimização mais diretos. Esses resultados reforçam a importância da escolha adequada do algoritmo conforme a natureza da função objetivo e a arquitetura computacional utilizada.

Tabela 21 – Desempenho de heurísticas em Funções *Benchmark Pytorch* (100D, 10000 iterações)

Função	OEP		AG		ED		BC	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Discus	1849.9	255.36	1883.9	802.15	1420.9	298.5	**1346.6**	**227.07**
Rastrigin	1913.0	281.79	2028.0	855.04	1481.0	311.16	**1370.2**	**241.38**
Rosenbrock	1931.1	314.36	1991.7	902.66	1503.5	343.43	**1401.1**	**271.23**
Cigar	1899.1	254.27	1934.6	816.75	1473.4	298.79	**1384.5**	**228.62**
Elliptic	1929.26	309.63	2083.1	900.33	1467.8	346.59	**1410.1**	**273.68**
Ackley	2054.2	409.96	2130.9	1114.7	1521.3	439.07	**1468.2**	**366.44**
Griewank	1984.78	323.27	2059.2	923.48	1490.4	367.65	**1429.2**	**298.63**
Salomon	2003.4	309.50	1959.2	903.87	1492.4	356.68	**1392.3**	**291.88**

Fonte: Autor

Na Tabela 21, o algoritmo BC foi o que mais se destacou, apresentando os melhores resultados em todas as oito funções, tanto na *CPU* quanto na *GPU*. Seu desempenho consistente indica uma excelente capacidade de adaptação a problemas de alta dimensionalidade, com valores significativamente inferiores aos demais algoritmos. O algoritmo ED também demonstrou resultados sólidos, com desempenho próximo ao do BC em várias funções, como *Elliptic* e *Salomon*. Por outro lado, o AG teve o pior desempenho na função *Discus*, com valores de 1883,9 na *CPU* e 802,15 na *GPU*. De forma geral, neste cenário de 100D, a implementação do AG mostrou-se a menos eficaz entre as quatro, consistentemente ficando mais distante do ótimo global.

Tabela 22 – Desempenho de heurísticas em Funções *Benchmark Pytorch* (500D, 10000 iterações)

Função	OEP		AG		ED		BC	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Discus	1857.3	248.01	1917.1	808.89	1485.2	299.70	**1401.8**	**234.15**
Rastrigin	1865.3	259.64	1914.07	849.34	1485.6	305.47	**1413.7**	**248.88**
Rosenbrock	1867.9	284.55	2251.2	971.62	1510.6	335.05	**1435.1**	**276.32**
Cigar	1835.6	244.19	1968.7	803.97	1489.6	293.00	**1425.5**	**231.90**
Elliptic	1877.6	289.90	2042.9	893.85	1532.4	338.58	**1435.0**	**278.30**
Ackley	1925.4	384.44	2000.3	1059.2	1614.1	469.50	**1449.1**	**367.83**
Griewank	1915.7	343.65	1942.6	947.87	1579.1	361.87	**1392.2**	**299.65**
Salomon	1903.65	301.88	1926.93	926.35	1560.1	351.11	**1369.0**	**275.61**

Fonte: Autor

Por fim, na Tabela 22, o algoritmo BC novamente apresentou os melhores resultados em todas as funções *benchmark*, superando as demais heurísticas tanto em desempenho na *CPU* quanto na *GPU*. Seu desempenho consistente reforça sua capacidade de explorar eficientemente o espaço de busca, mesmo em cenários de alta dimensionalidade. O algoritmo ED também demonstrou resultados competitivos, ficando próximo ao BC em várias funções, como Elliptic e Ackley. Em contrapartida, o algoritmo AG teve o pior desempenho em várias funções, especialmente na função Rosenbrock, com valores de 2251,2 na *CPU* e 971,62 na *GPU*, o que indica dificuldades em alcançar soluções próximas do ótimo global em ambientes com alta complexidade computacional.

Os resultados obtidos para as funções *benchmark* em 50D, 100D e 500D revelam padrões consistentes no desempenho dos algoritmos avaliados. O algoritmo ED (Evolução Diferencial) demonstrou excelente capacidade de aproximação ao ótimo global em diversas funções e dimensões, especialmente em 50D, onde liderou em praticamente todas as métricas. No entanto, à medida que a dimensionalidade aumentou, o algoritmo BC (Busca Cuco) passou a se destacar, obtendo os melhores resultados em praticamente todas as funções tanto em 100D quanto em 500D, evidenciando maior robustez e escalabilidade.

Por outro lado, o AG (Algoritmo Genético) apresentou desempenho inferior nas três dimensões, com dificuldades notáveis em funções como Discus e Rosenbrock, sugerindo menor adaptabilidade a problemas de alta

dimensionalidade. O OEP (Otimização por Enxame de Partículas) manteve desempenho intermediário ao longo dos testes, sem se destacar como o melhor em nenhuma dimensão ou função, mas também sem apresentar os piores resultados de forma consistente. Esses resultados reforçam a eficácia do ED em cenários de baixa a média dimensionalidade e a superioridade do BC em contextos mais complexos, enquanto AG e OEP enfrentam limitações em certos tipos de função e tamanhos de problema. É importante ressaltar que os parâmetros de controle de cada metaheurística (e.g., PC e PM para o AG; F e CR para o ED) foram mantidos fixos, baseados em valores padrão da literatura, para garantir uma base de comparação uniforme. Esta abordagem, no entanto, constitui uma limitação do estudo, pois não foi realizada uma etapa de **calibração de hiperparâmetros** para cada cenário. É plausível que o desempenho de algoritmos como o AG, que se mostrou consistentemente inferior, pudesse ser significativamente melhorado com um ajuste fino de suas taxas. Portanto, os resultados apresentados refletem o desempenho sob uma configuração padronizada, e uma futura linha de pesquisa seria investigar como o ranking de performance entre os algoritmos se alteraria após um processo rigoroso de otimização de seus respectivos parâmetros.

4.2.1 Resultados de *speedup* e eficiência em *Pytorch*

Os resultados serão apresentados referentes ao *speedup* e à eficiência dos algoritmos otimizados utilizando a plataforma *PyTorch*. Na analise de desempenho dos métodos em ambiente paralelo, foi avaliado como o tempo de execução se comporta à medida que o número de threads ou unidades de processamento aumenta. Essa análise permite compreender a escalabilidade dos algoritmos e a eficácia da implementação em *GPU*, fornecendo uma base para comparação entre as diferentes metaheurísticas testadas.

Tabela 23 – Resultados de Speedup e Eficiência OEP *Pytorch* (50D, 10000 iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	1875,3	241,58	**7,76**	0,30	+87,1%
Rastrigin	1888,2	257,8	**7,32**	0,29	+86,3%
Rosenbrock	1900,3	290,23	**6,55**	0,26	+84,7%
Cigar	1893,8	242,00	**7,83**	0,31	+87,2%
Elliptic	1920,7	285,46	**6,73**	0,26	+85,1%
Ackley	1967,6	379,48	**5,18**	0,20	+80,7%
Griewank	1905,0	312,68	**6,09**	0,24	+83,6%
Salomon	2046,7	348,83	**5,87**	0,23	+82,9%

Fonte: Autor

Na Tabela 23, o algoritmo OEP apresentou resultados altamente expressivos em termos de *speedup* e *eficiência* ao utilizar a GPU com suporte do framework **PyTorch**. A função *Cigar* obteve o melhor desempenho, com *speedup* de **7,83**, eficiência de **0,31** e ganho de **+87,2%**, seguida de *Discus* (**7,76**) e *Rastrigin* (**7,32**), todas com ganhos superiores a **+86%**. As funções *Rosenbrock* (**6,55**), *Elliptic* (**6,73**), *Griewank* (**6,09**) e *Salomon* (**5,87**) também mantiveram resultados robustos, com ganhos entre **+82,9%** e **+85,1%**. A função *Ackley* apresentou o menor *speedup*, com **5,18**, mas ainda assim com ganho expressivo de **+80,7%** e eficiência de **0,20**. De modo geral, o algoritmo OEP demonstrou excelente capacidade de paralelização na GPU, obtendo ganhos consideráveis em todas as funções *benchmark*, com *speedup* variando de **5,18** a **7,83** e eficiências entre **0,20** e **0,31**.

Tabela 24 – Resultados de Speedup e Eficiência AG *Pytorch* (50D, 10000 iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	2004,0	831,00	**2,41**	0,094	+58,5%
Rastrigin	1854,2	844,46	**2,20**	0,086	+54,5%
Rosenbrock	1879,8	895,13	**2,10**	0,082	+52,4%
Cigar	1904,9	821,93	**2,32**	0,091	+56,8%
Elliptic	2037,2	904,75	**2,25**	0,088	+55,6%
Ackley	2150,8	1109,80	**1,94**	0,076	+48,4%
Griewank	1968,2	937,43	**2,10**	0,082	+52,4%
Salomon	2229,9	999,33	**2,23**	0,087	+55,2%

Fonte: Autor

Na Tabela 24, os resultados de *Speedup* e *Eficiência* para o algoritmo AG (50D) indicam um desempenho razoável com ganho de desempenho moderado ao utilizar a GPU. A função *Discus* apresentou o melhor resultado, com *Speedup* de **2,41**, eficiência de **0,094** e ganho de **+58,5%**. Funções como *Cigar* (**2,32**), *Elliptic* (**2,25**) e *Salomon* (**2,23**) também se destacaram com ganhos acima de **+55%**. Já as funções *Ackley* (**1,94**) e *Rosenbrock* (**2,10**) apresentaram os menores *Speedups*, embora ainda com ganhos consideráveis, de **+48,4%** e **+52,4%**, respectivamente. De modo geral, o algoritmo AG demonstrou um bom aproveitamento da GPU, com *Speedup* variando de **1,94** a **2,41** e ganhos entre **+48,4%** e **+58,5%**, embora inferior aos demais algoritmos analisados.

Tabela 25 – Resultados de *Speedup* e Eficiência ED *Pytorch* (50D, 10000 Iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	1454,3	294,83	**4,93**	0,1926	+79,73%
Rastrigin	1467,8	307,98	**4,77**	0,1863	+79,00%
Rosenbrock	1465,3	336,20	**4,36**	0,1703	+77,06%
Cigar	1463,1	304,28	**4,81**	0,1879	+79,21%
Elliptic	1506,5	350,24	**4,30**	0,1680	+76,75%
Ackley	1561,8	464,94	**3,36**	0,1312	+70,23%
Griewank	1509,7	373,14	**4,05**	0,1582	+75,29%
Salomon	1512,6	366,34	**4,13**	0,1613	+75,78%

Fonte: Autor

Na Tabela 25, o algoritmo ED apresentou um desempenho sólido e consistente para 50D. A função *Discus* destacou-se com o maior *Speedup* (**4,93**), seguida por *Cigar* (**4,81**) e *Rastrigin* (**4,77**), todas com eficiências próximas de **0,19** e ganhos acima de **+79%**. As funções *Salomon* (**4,13**), *Griewank* (**4,05**) e *Rosenbrock* (**4,36**) também apresentaram bons resultados, com eficiências entre **0,1582** e **0,1703**. A função *Ackley* teve o menor desempenho, com *Speedup* de **3,36**, eficiência de **0,1312** e ganho de **+70,23%**. No geral, o algoritmo ED demonstrou boa escalabilidade com a GPU, mantendo *Speedups* acima de **3,3** e ganhos consistentes em todas as funções analisadas.

Tabela 26 – Resultados de Speedup e Eficiência BC *Pytorch* (50D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	1379,40	225,14	**6,13**	0,2395	+83,68%
Rastrigin	1388,70	238,93	**5,81**	0,2277	+82,80%
Rosenbrock	1363,90	274,56	**4,97**	0,1941	+79,87%
Cigar	1394,60	231,45	**6,03**	0,2355	+83,40%
Elliptic	1453,90	278,58	**5,22**	0,2039	+80,84%
Ackley	1513,30	383,95	**3,94**	0,1539	+74,63%
Griewank	1456,00	302,84	**4,81**	0,1879	+79,21%
Salomon	1467,40	290,32	**5,05**	0,1973	+80,22%

Fonte: Autor

Na Tabela 26, observa-se que o algoritmo BC apresentou um desempenho sólido para 50D. A função *Discus* obteve o maior *Speedup*, com **6,13**, seguida por *Cigar* (6,03) e *Rastrigin* (5,81), todas com eficiências superiores a **0,22** e ganhos acima de **+82%**. As funções *Rosenbrock*, *Griewank* e *Salomon* apresentaram *Speedups* entre **4,81** e **5,05**, com ganhos que variaram entre **+79%** e **+80%**. A função *Elliptic* obteve *Speedup* de **5,22**, com ganho de **+80,84%** e eficiência de **0,2039**. O menor desempenho foi registrado na função *Ackley*, com *Speedup* de **3,94**, eficiência de **0,1539** e ganho de **+74,63%**. No geral, o algoritmo BC mostrou-se eficiente na maioria das funções, com bom aproveitamento da GPU.

Tabela 27 – Resultados de Speedup e Eficiência OEP *Pytorch* (100D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	1849,90	255,36	**7,24**	0,2830	+86,20%
Rastrigin	1913,00	281,79	**6,79**	0,2652	+85,27%
Rosenbrock	1931,10	314,36	**6,14**	0,2400	+83,72%
Cigar	1899,10	254,27	**7,47**	0,2918	+86,61%
Elliptic	1929,26	309,63	**6,23**	0,2434	+83,95%
Ackley	2054,20	409,96	**5,01**	0,1957	+80,04%
Griewank	1984,78	323,27	**6,14**	0,2398	+83,71%
Salomon	2003,40	309,50	**6,47**	0,2529	+84,55%

Fonte: Autor

Na Tabela 27, o algoritmo OEP apresentou o melhor desempenho entre os algoritmos analisados para o cenário de 100D. A função *Cigar* obteve o maior *Speedup*, com **7,47**, seguida das funções *Discus* (7,24) e *Rastrigin* (6,79), com eficiências entre **0,2652** e **0,2918**, e ganhos superiores a **+85%**. Funções como *Rosenbrock*, *Elliptic*, *Griewank* e *Salomon* apresentaram *Speedups* entre **6,14** e **6,47**, demonstrando desempenho bastante consistente, com ganhos acima de **+83%**. A função *Ackley* teve o menor resultado, com *Speedup* de **5,01**, eficiência de **0,1957** e ganho de **+80,04%**. Esses resultados indicam que o OEP foi o algoritmo mais eficaz em aproveitar a paralelização na GPU neste conjunto de testes.

Tabela 28 – Resultados de Speedup e Eficiência AG *Pytorch* (100D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	1883,90	802,15	**2,35**	0,0917	+57,42%
Rastrigin	2028,00	855,04	**2,37**	0,0926	+57,84%
Rosenbrock	1991,70	902,66	**2,21**	0,0862	+54,68%
Cigar	1934,60	816,75	**2,37**	0,0925	+57,78%
Elliptic	2083,10	900,33	**2,31**	0,0904	+56,78%
Ackley	2130,90	1114,70	**1,91**	0,0747	+47,69%
Griewank	2059,20	923,48	**2,23**	0,0871	+55,15%
Salomon	1959,20	903,87	**2,17**	0,0847	+53,87%

Fonte: Autor

Na Tabela 28, o algoritmo AG apresentou desempenho mais modesto em comparação aos demais algoritmos analisados. As funções *Rastrigin* e *Cigar* obtiveram os melhores *Speedups*, ambos com **2,37**, seguidas de *Elliptic* (2,31) e *Discus* (2,35), com ganhos superiores a **+56%** e eficiências em torno de **0,09**. A função *Ackley* apresentou o menor *Speedup* (1,91), com ganho de **+47,69%** e eficiência de **0,0747**, evidenciando maior dificuldade de paralelização. De forma geral, os ganhos foram consistentes, variando entre **+47%** e **+58%**, o que demonstra um ganho significativo, embora inferior ao observado com algoritmos como BC e ED.

Tabela 29 – Resultados de Speedup e Eficiência ED *Pytorch* (100D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	1420,9	298,50	**4,76**	0,1859	+78,99%
Rastrigin	1481,0	311,16	**4,76**	0,1859	+78,99%
Rosenbrock	1503,5	343,43	**4,38**	0,1710	+77,16%
Cigar	1473,4	298,79	**4,93**	0,1926	+79,72%
Elliptic	1467,8	346,59	**4,23**	0,1654	+76,39%
Ackley	1521,3	439,07	**3,46**	0,1353	+71,14%
Griewank	1490,4	367,65	**4,05**	0,1584	+75,33%
Salomon	1492,4	356,68	**4,18**	0,1634	+76,10%

Fonte: Autor

Na Tabela 29, o algoritmo ED apresentou desempenho sólido em termos de *Speedup* e eficiência. As funções *Cigar* (4,93), *Discus* (4,76) e *Rastrigin* (4,76) destacaram-se com ganhos superiores a **+78%** e eficiências acima de **0,18**. Em seguida, aparecem as funções *Rosenbrock* (4,38), *Salomon* (4,18), *Griewank* (4,05) e *Elliptic* (4,23), com ganhos entre **+75%** e **+77%**. A função *Ackley* apresentou o menor desempenho relativo (3,46), mas ainda assim obteve um ganho expressivo de **+71%** e eficiência de **0,1353**. Os resultados demonstram que o ED possui boa escalabilidade na GPU para 100 dimensões, com destaque para as funções de paisagem mais simples como *Cigar* e *Discus*.

Tabela 30 – Resultados de Speedup e Eficiência BC *Pytorch* (100D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	1346,6	227,07	**5,93**	0,2317	+83,14%
Rastrigin	1370,2	241,38	**5,68**	0,2217	+82,38%
Rosenbrock	1401,1	271,23	**5,17**	0,2018	+80,64%
Cigar	1384,5	228,62	**6,06**	0,2366	+83,49%
Elliptic	1410,1	273,68	**5,15**	0,2013	+80,59%
Ackley	1468,2	366,44	**4,01**	0,1565	+75,04%
Griewank	1429,2	298,63	**4,79**	0,1869	+79,11%
Salomon	1392,3	291,88	**4,77**	0,1863	+79,04%

Fonte: Autor

A Tabela 30 apresenta os resultados de *Speedup* e eficiência para o algoritmo BC em funções com 100 dimensões e 10.000 iterações. Os melhores desempenhos foram observados nas funções *Cigar* (6,06) e *Discus* (5,93), com ganhos superiores a **+83%** e eficiências acima de **0,23**. Em seguida, destacam-se as funções *Rastrigin* (5,68), *Rosenbrock* (5,17) e *Elliptic* (5,15), com ganhos variando entre **+80%** e **+82%**. As funções *Griewank* (4,79) e *Salomon* (4,77) mantiveram bom desempenho, com ganhos em torno de **+79%**. A função com menor desempenho foi *Ackley* (4,01), ainda assim apresentando um ganho considerável de **+75%**. Esses resultados mostram que o BC obteve um desempenho consistente em 100D, com *Speedup* e eficiência elevados na maioria das funções.

Tabela 31 – Resultados de Speedup e Eficiência OEP *Pytorch* (500D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	1857,30	248,01	**7,49**	0,2926	+86,65%
Rastrigin	1865,30	259,64	**7,18**	0,2804	+86,08%
Rosenbrock	1867,90	284,55	**6,56**	0,2563	+84,77%
Cigar	1835,60	244,19	**7,52**	0,2938	+86,69%
Elliptic	1877,60	289,90	**6,48**	0,2531	+84,56%
Ackley	1925,40	384,44	**5,01**	0,1957	+80,03%
Griewank	1915,70	343,65	**5,57**	0,2176	+82,06%
Salomon	1903,65	301,88	**6,31**	0,2465	+84,15%

Fonte: Autor

A Tabela 31 apresenta os resultados de *Speedup* e eficiência para o algoritmo OEP em funções com 500 dimensões e 10.000 iterações. Os melhores desempenhos foram observados nas funções *Cigar* (7,52) e *Discus* (7,49), com ganhos superiores a **+86,65%** e eficiências acima de **0,2926%**. A função *Rastrigin* (7,18) também apresentou excelente resultado, com ganho de **+86,08%**. As funções *Rosenbrock* (6,56), *Elliptic* (6,48) e *Salomon* (6,31) mantiveram ganhos elevados, acima de **+84%**. Já as funções *Griewank* (5,57) e *Ackley* (5,01) apresentaram ganhos ligeiramente inferiores, mas ainda expressivos, com **+82,06%** e **+80,03%**, respectivamente. Esses resultados destacam o OEP como o algoritmo com maior aproveitamento de GPU entre os testados.

Tabela 32 – Resultados de Speedup e Eficiência AG *Pytorch* (500D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	1917,10	808,89	**2,37**	0,0926	+57,82%
Rastrigin	1914,07	849,34	**2,25**	0,0879	+55,65%
Rosenbrock	2251,20	971,62	**2,32**	0,0906	+56,84%
Cigar	1968,70	803,97	**2,45**	0,0957	+59,18%
Elliptic	2042,90	893,85	**2,29**	0,0894	+56,27%
Ackley	2000,30	1059,20	**1,89**	0,0740	+47,08%
Griewank	1942,60	947,87	**2,05**	0,0801	+51,21%
Salomon	1926,93	926,35	**2,08**	0,0813	+51,90%

Fonte: Autor

A Tabela 32 apresenta os resultados de *Speedup* e eficiência para o algoritmo AG em funções com 500 dimensões e 10.000 iterações. A função *Cigar* obteve o maior *Speedup* (2,45), com um ganho de **+59,18%** e eficiência de **0,0957%**. As funções *Discus* (2,37), *Rosenbrock* (2,32) e *Elliptic* (2,29) também apresentaram desempenhos relevantes, com ganhos entre **+56,27%** e **+57,82%**. Já as funções *Rastrigin* (2,25), *Griewank* (2,05) e *Salomon* (2,08) tiveram ganhos na faixa de **+51,21%** a **+55,65%**. A função *Ackley* apresentou o menor desempenho relativo, com *Speedup* de 1,89 e ganho de **+47,08%**. Os resultados indicam uma melhoria considerável no desempenho com o uso de GPU, embora inferior aos observados com os algoritmos ED e BC.

Tabela 33 – Resultados de Speedup e Eficiência ED *Pytorch* (500D, 10000 Iterações)

Função	Tempo (s)		Speedup	Eficiência	Ganho (%)
	CPU	GPU			
Discus	1485,2	299,7	**4,96**	0,1938	+79,82%
Rastrigin	1485,6	305,47	**4,86**	0,1898	+79,44%
Rosenbrock	1510,6	335,05	**4,51**	0,1762	+77,82%
Cigar	1489,6	293	**5,08**	0,1984	+80,34%
Elliptic	1532,4	338,58	**4,53**	0,1769	+77,91%
Ackley	1614,1	469,5	**3,44**	0,1344	+70,92%
Griewank	1579,1	361,87	**4,36**	0,1703	+77,09%
Salomon	1560,1	351,11	**4,44**	0,1734	+77,49%

Fonte: Autor

A Tabela 33 apresenta os resultados de *Speedup* e eficiência para o algoritmo ED em funções com 500 dimensões e 10.000 iterações. A função *Cigar* obteve o maior *Speedup* (5,08), com um ganho de **+80,34%**, seguida por *Discus* (4,96) e *Rastrigin* (4,86), com ganhos de **+79,82%** e **+79,44%**, respectivamente. Outras funções, como *Rosenbrock* (4,51), *Elliptic* (4,53), *Griewank* (4,36) e *Salomon* (4,44), apresentaram ganhos entre **+77,09%** e **+77,91%**. A função *Ackley* teve o menor desempenho relativo, com um *Speedup* de 3,44 e ganho de **+70,92%**. De forma geral, o algoritmo ED demonstrou ganhos significativos com a utilização da GPU, com eficiências variando entre **0,13%** e **0,20%**.

Tabela 34 – Resultados de Speedup e Eficiência BC *Pytorch* (500D, 10000 Iterações)

Função	**Tempo (s)**		**Speedup**	**Eficiência**	**Ganho (%)**
	CPU	**GPU**			
Discus	1401,8	234,15	**5,99**	0,2340	+83,30%
Rastrigin	1413,7	248,88	**5,68**	0,2219	+82,40%
Rosenbrock	1435,1	276,32	**5,19**	0,2027	+80,74%
Cigar	1425,5	231,9	**6,15**	0,2402	+83,73%
Elliptic	1435,0	278,3	**5,16**	0,2016	+80,60%
Ackley	1449,1	367,83	**3,94**	0,1539	+74,63%
Griewank	1392,2	299,65	**4,65**	0,1816	+78,48%
Salomon	1369,0	275,61	**4,97**	0,1941	+79,87%

Fonte: Autor

A Tabela 34 apresenta os resultados de *Speedup* e eficiência para o algoritmo BC, aplicado a funções com 500 dimensões e 10.000 iterações. A função *Cigar* obteve o maior *Speedup* (6,15), com um ganho expressivo de **+83,73%**. Outras funções como *Discus* (5,99), *Rastrigin* (5,68) e *Rosenbrock* (5,19) também apresentaram ganhos significativos, variando entre **+80,74%** e **+83,30%**. A função *Ackley* teve o menor desempenho relativo (3,94), com ganho de **+74,63%**. No geral, todas as funções demonstraram vantagens claras com a execução paralela em GPU, com eficiências variando entre **0,15%** e **0,24%**, considerando o uso de uma GPU com 2560 núcleos CUDA.

4.2.2 Resultados Gráficos Tempo *GPU* Por Algoritimos *PyTorch*

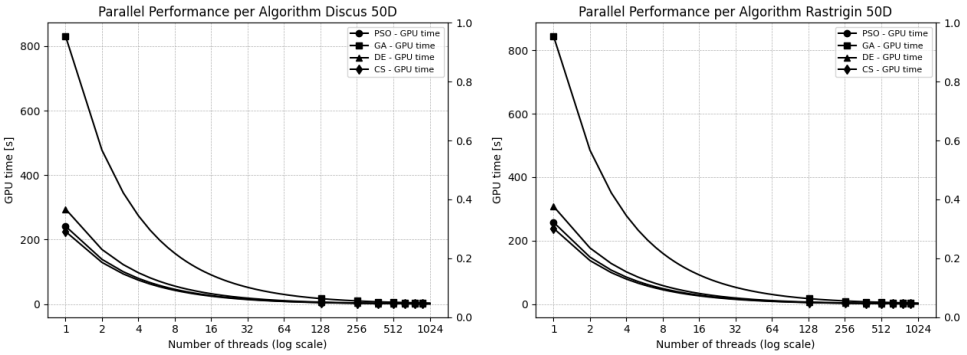


Figura 19 – Função *Discus e Rastrigin 50D PyTorch*

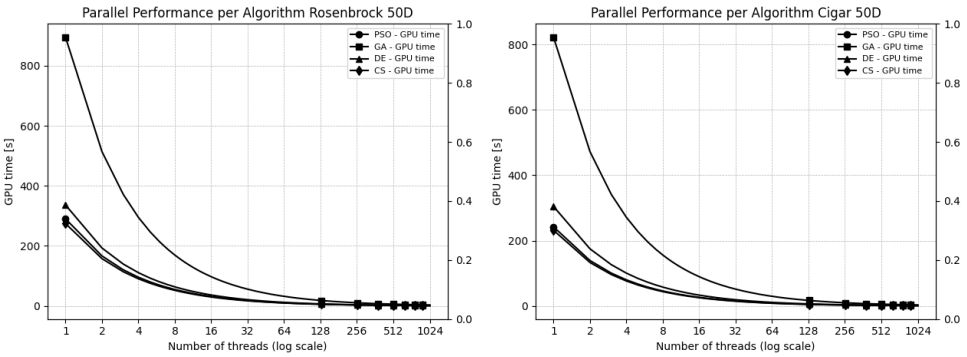


Figura 20 – Função *Rosenbrock e Cigar 50D PyTorch*

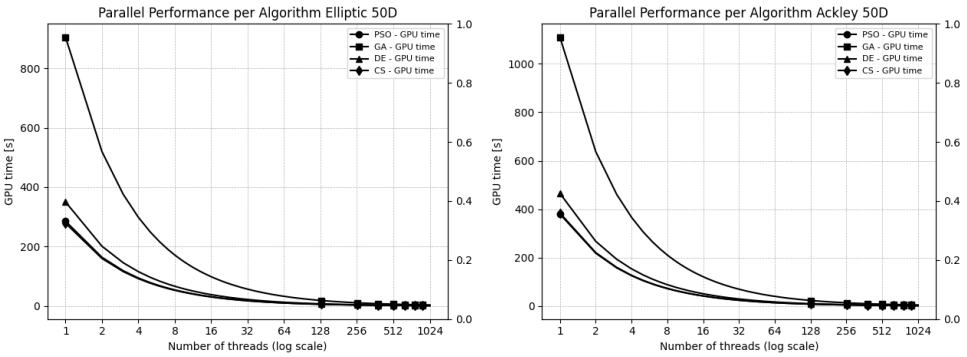


Figura 21 – Função *Elliptic e Ackley 50D PyTorch*

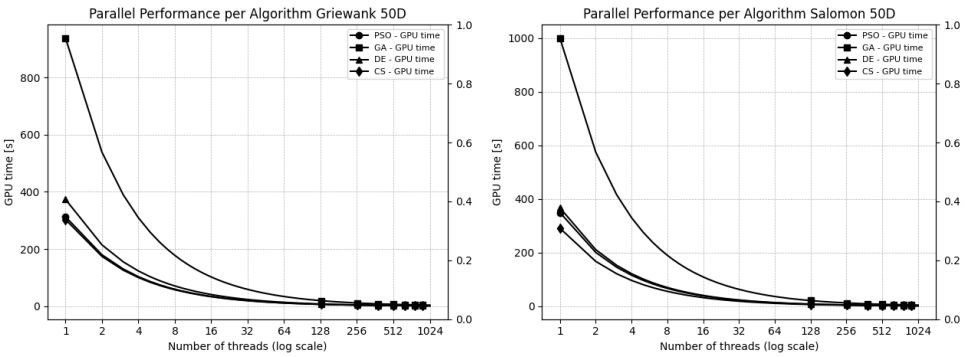


Figura 22 – Função *Griewank e Salomon 50D PyTorch*

Com base nos tempos de execução na *GPU* apresentados na Figura 19, a função *Discus* apresentou o menor tempo de *GPU*, com 225,14 ms, utilizando o algoritmo BC. Por outro lado, o algoritmo AG registrou o maior tempo para a mesma função, com 831 ms. A discrepância de desempenho entre esses dois algoritmos pode ser atribuída às diferenças na forma como exploram o paralelismo computacional. De forma semelhante, para a função *Rastrigin*, o algoritmo BC obteve o melhor tempo de *GPU*, com 238,93 ms, enquanto o AG novamente apresentou o pior resultado, com 844,46 ms. Esses dados evidenciam a eficiência do BC na utilização dos recursos da *GPU*, ao passo que o AG se mostra menos otimizado para computação paralela, elemento essencial para alcançar tempos de execução mais rápidos em ambientes com aceleração por *GPU*. A estrutura mais simples e direta do BC pode favorecer sua implementação eficiente em arquiteturas massivamente paralelas. Já o AG, por depender de operações mais custosas e sequenciais, tende a apresentar gargalos quando submetido a execuções em *GPU*.

Na Figura 20, observa-se que a função *Cigar* obteve o menor tempo de execução na *GPU*, com 231,45 ms, ao ser otimizada pelo algoritmo BC. Em contrapartida, o algoritmo AG apresentou o pior tempo para a função *Rosenbrock*, com 895,13 ms. A diferença de desempenho registrada reforça a eficiência do algoritmo BC na utilização efetiva da potência computacional da *GPU*, tornando-o a escolha preferida para funções com espaços de alta dimensionalidade. Esses resultados reforçam a tendência de que o BC maximiza a utilização da *GPU* de forma eficiente, enquanto o desempenho do AG permanece inferior entre os algoritmos testados, sugerindo que o AG requer aprimoramentos adicionais para obter melhor desempenho em ambientes com aceleração por *GPU*. A adaptabilidade do BC a diferentes tipos de funções pode ser um diferencial importante em contextos práticos de otimização. Isso o torna mais versátil e eficiente para aplicações que exigem resposta rápida e uso intensivo de *GPU*.

A análise dos tempos de execução na *GPU* para as funções *Elliptic* e *Ackley* (Figura 21) reforça a superioridade computacional do Busca Cuco (BC) nesta plataforma. O BC registrou o menor tempo para a função *Elliptic* (**278,58 ms**), enquanto o Algoritmo Genético (AG) foi o mais lento na função *Ackley*, com um tempo de **1109,8 ms**. A performance inferior do AG é especialmente notável em **paisagens de otimização** como a da função *Ackley*, que é caracterizada por uma bacia de atração ampla com múltiplos mínimos locais. Este tipo de topologia complexa parece expor os gargalos na implementação paralela do AG, que não consegue explorar o espaço de busca com a mesma eficiência. Em contraste, a abordagem de busca mais direta do BC se mostra mais robusta e escalável, mantendo sua alta performance mesmo em funções desafiadoras.

Por fim, na Figura 22, o melhor desempenho foi observado para a função *Salomon*, utilizando o algoritmo BC, que obteve tempo de execução de 290,32 ms. Em contrapartida, o pior tempo de *GPU* também foi registrado para a função *Salomon*, mas otimizada com o algoritmo AG, que atingiu 999,33 ms. Esses resultados destacam, mais uma vez, a eficiência do algoritmo BC na exploração das capacidades da *GPU* em comparação com as demais abordagens. O desempenho lento do AG sugere que ele pode não estar plenamente otimizado para execução em *GPU*, enquanto o BC demonstra clara vantagem na minimização do tempo de processamento e na maximização da eficiência computacional em diferentes funções de *benchmark*. A robustez do BC diante de funções com características distintas reforça sua aplicabilidade em diferentes domínios de engenharia e ciência de dados. Assim, seu uso torna-se estratégico quando se busca alto desempenho em tarefas de otimização com restrições de tempo.

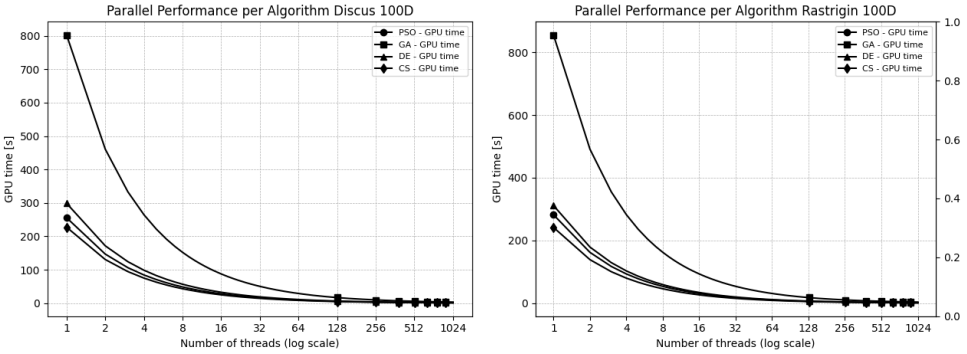


Figura 23 – Função *Discus e Rastrigin 100D PyTorch*

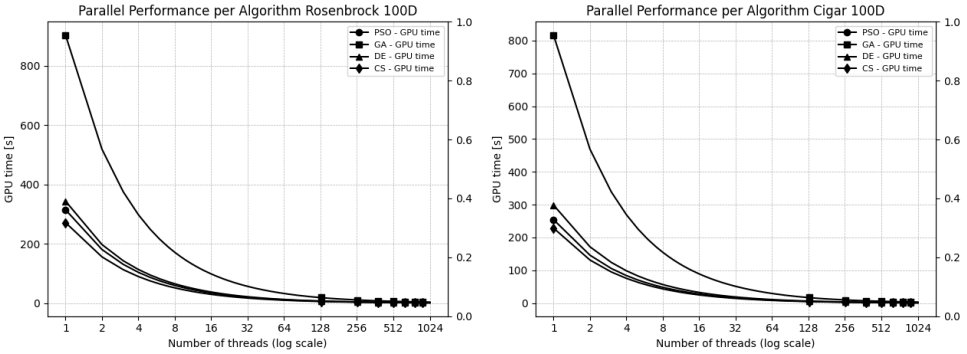


Figura 24 – Função *Rosenbrock e Cigar 100D PyTorch*

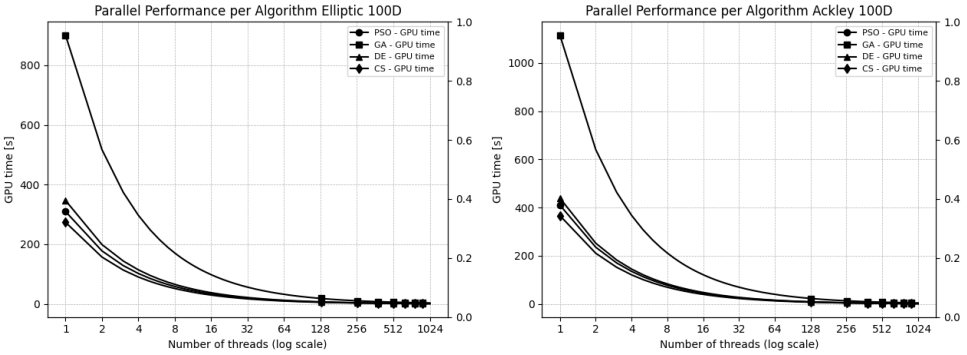


Figura 25 – Função *Elliptic e Ackley 100D PyTorch*

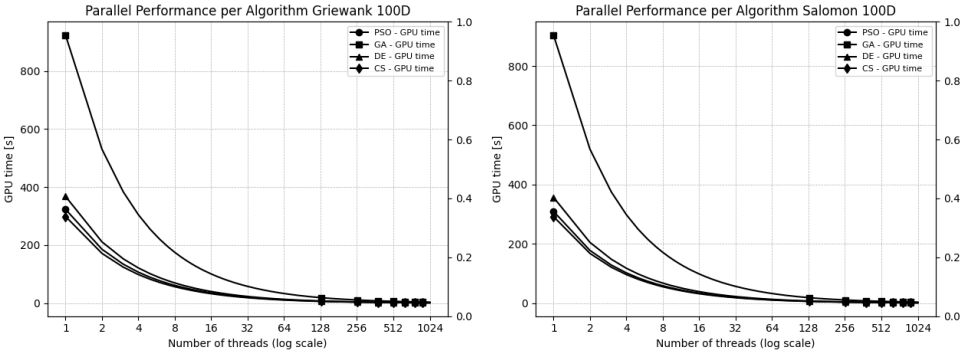


Figura 26 – Função *Griewank e Salomon 100D PyTorch*

Com base nos tempos de execução na *GPU* apresentados na Figura 23, o melhor desempenho foi observado para a função *Discus* utilizando o algoritmo BC, que obteve o menor tempo de execução, com 227,07 ms. Por outro lado, o pior tempo de execução na *GPU* foi registrado para a função *Rastrigin* otimizada com o algoritmo AG, atingindo 855,04 ms. Esses resultados reforçam a eficiência superior do algoritmo BC na utilização dos recursos da *GPU*, apresentando consistentemente tempos de processamento mais rápidos em diferentes funções de *benchmark*, em comparação com os demais algoritmos. A disparidade de desempenho sugere que o algoritmo BC é mais eficiente na exploração da paralelização, sendo capaz de lidar com cálculos mais complexos de forma eficaz, enquanto a execução do AG permanece mais lenta, mesmo com otimização para *GPU*. Os resultados obtidos indicam que a estrutura do BC se adapta melhor ao modelo de execução paralela, aproveitando melhor os núcleos de processamento disponíveis. Além disso, a simplicidade relativa dos mecanismos de atualização do BC pode favorecer sua escalabilidade em cenários computacionalmente intensivos.

Em relação aos tempos de execução na *GPU* mostrados na Figura 24, o melhor desempenho foi registrado para a função *Cigar* com o algoritmo BC, que alcançou o menor tempo, com 228,62 ms. Por outro lado, o maior tempo de execução foi observado para a função *Rosenbrock* otimizada com o algoritmo AG, atingindo 902,66 ms. Esses resultados destacam a capacidade superior do algoritmo BC em explorar de forma eficiente o paralelismo da *GPU*, mantendo menores tempos de execução em comparação com outras abordagens de otimização. Esse padrão confirma que o BC é mais adequado para aceleração com *GPU*, especialmente ao lidar com problemas de otimização que exigem altos recursos computacionais, enquanto o AG apresenta desempenho inferior. A diferença de desempenho pode também estar relacionada ao maior custo computacional das operações genéticas, como *crossover* e *mutação*, que tendem a ser menos paralelizáveis. Por sua vez, o BC adota uma abordagem mais direta de busca, o que contribui para um fluxo de execução mais fluido e eficiente na *GPU*.

Considerando os tempos de execução na *GPU* ilustrados na Figura 25, o melhor desempenho foi obtido com a função *Elliptic* otimizada com o algoritmo BC, resultando em um tempo de 273,68 ms. O tempo mais lento foi observado para a função *Ackley* otimizada com o algoritmo AG, que levou 1114,7 ms. Essa diferença de desempenho evidencia, mais uma vez, a eficiência superior do algoritmo BC na utilização dos recursos da *GPU* em diferentes funções de *benchmark*. A execução mais lenta do algoritmo AG ressalta suas limitações em explorar plenamente o poder computacional paralelo da *GPU*, reforçando a vantagem do BC nesse tipo de tarefa de otimização. Essa constatação sugere que a estrutura interna do AG pode gerar gargalos que comprometem sua eficiência em arquiteturas paralelas. Além disso, o desempenho consistente do BC indica uma melhor compatibilidade com a execução massivamente paralela típica das *GPUs* modernas.

Por fim, os tempos de execução na *GPU* apresentados na Figura 26, a função *Salomon* utilizando o algoritmo BC apresentou o melhor desempenho, com tempo de execução de 291,88 ms. Em contrapartida, o pior tempo foi registrado para a função *Griewank* otimizada com o algoritmo AG, alcançando 923,48 ms. Isso confirma, mais uma vez, a eficiência do algoritmo BC na minimização do tempo de computação em diferentes problemas de otimização quando se utilizam recursos de *GPU*. O desempenho inferior do AG neste caso evidencia sua utilização subótima da *GPU*, reforçando a conclusão de que o BC supera o AG em termos de eficiência temporal na *GPU* em diversas funções de otimização. Esses achados são relevantes para aplicações que exigem alta performance computacional em tempo reduzido, como simulações em larga escala. Dessa forma, o BC se consolida como uma alternativa robusta e veloz para otimização paralela com uso intensivo de *GPU*.

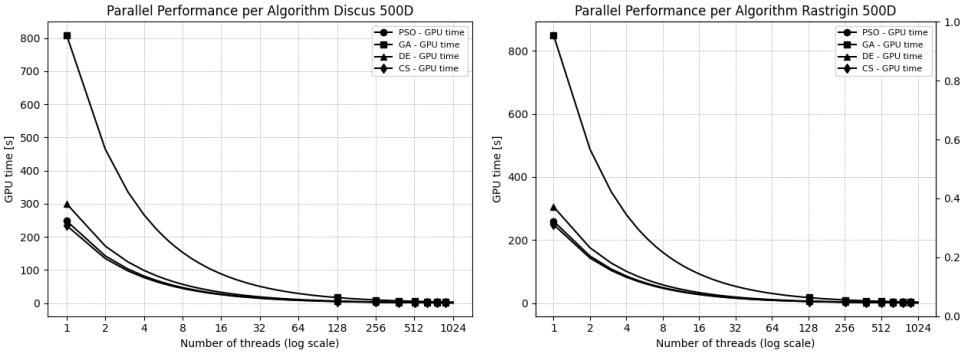


Figura 27 – Função *Discus e Rastrigin 500D PyTorch*

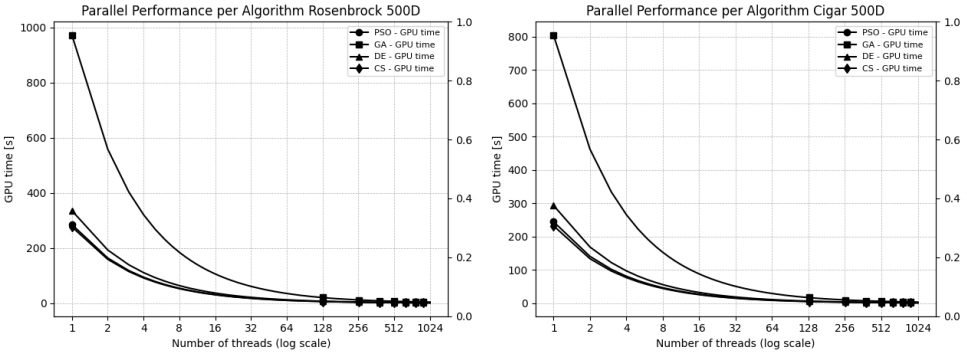


Figura 28 – Função *Rosenbrock e Cigar 500D PyTorch*

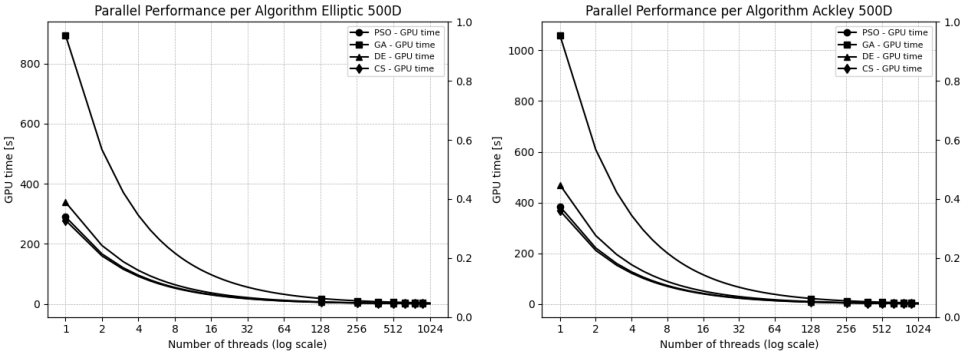


Figura 29 – Função *Elliptic e Ackley 500D PyTorch*

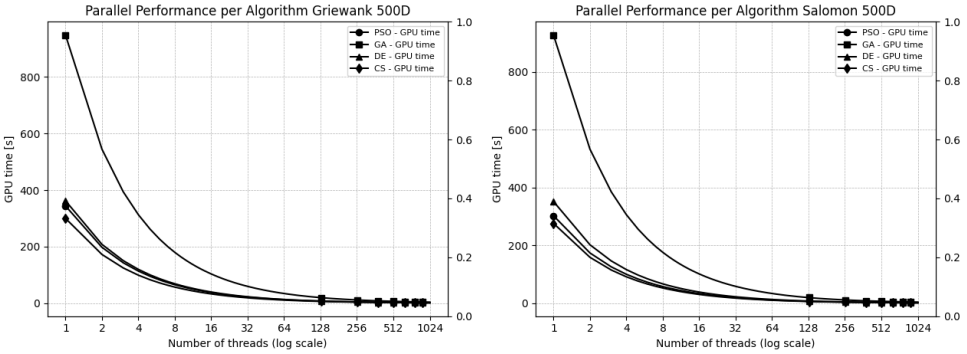


Figura 30 – Função *Griewank e Salomon 500D PyTorch*

Em relação aos tempos de execução na *GPU* apresentados na Figura 27, o melhor desempenho foi observado para a função *Discus* otimizada com o algoritmo BC, que obteve o menor tempo na *GPU*, com 234,15 ms. O pior tempo foi registrado para a função *Rastrigin* com o algoritmo AG, atingindo 849,34 ms. Esses resultados destacam a eficiência do algoritmo BC na redução dos tempos computacionais e reforçam sua adequação para tarefas de otimização em alta dimensão em plataformas com *GPU*.

Na Figura 28, o resultado mais eficiente foi obtido para a função *Cigar*, também utilizando o algoritmo BC, que alcançou o menor tempo de execução na *GPU*, com 231,9 ms. Por outro lado, a função *Rosenbrock*, com o algoritmo AG, apresentou o maior tempo de *GPU*, atingindo 971,62 ms. Esses dados reforçam a vantagem do algoritmo BC na redução do tempo computacional, especialmente em funções desafiadoras como a *Cigar*, quando avaliadas em alta dimensionalidade.

Considerando o tempo de *GPU*, o algoritmo BC obteve o melhor desempenho, com os menores tempos de execução para as funções *Elliptic* (278,3 ms) e *Ackley* (367,83 ms). Isso indica que o BC está altamente otimizado para execução em *GPU*, aproveitando de forma eficiente os recursos de processamento paralelo. O algoritmo OEP, embora não tenha alcançado os menores tempos, apresentou desempenho competitivo, com 289,9 ms para *Elliptic* e 384,44 ms para *Ackley*. Em contraste, o algoritmo AG apresentou os maiores tempos de execução na *GPU*, com 893,85 ms para *Elliptic* e 1059,2 ms para *Ackley*, evidenciando sua menor eficiência na utilização do potencial total da aceleração por *GPU*.

Ainda em termos de tempo de *GPU*, o algoritmo BC demonstrou o melhor desempenho também para as funções *Griewank* e *Salomon*, com tempos de 299,65 ms e 275,61 ms, respectivamente. Esses resultados evidenciam a eficiência do BC na utilização dos recursos da *GPU* para tarefas de otimização em alta dimensão. O algoritmo OEP apresentou desempenho competitivo, com 343,65 ms para *Griewank* e 301,88 ms para *Salomon*; embora ligeiramente mais lento que o BC, ainda demonstra forte eficiência computacional. Por outro lado, o algoritmo AG novamente apresentou os maiores tempos, com 947,87 ms para *Griewank* e 926,35 ms para *Salomon*, indicando que sua paralelização na *GPU* é menos eficaz, resultando em desempenho inferior em comparação com os algoritmos OEP e BC.

4.2.3 Discussão dos Resultados para *PyTorch*

A análise dos experimentos realizados com a biblioteca *PyTorch* revela um cenário de desempenho distinto daquele observado com o *TensorFlow*, destacando a profunda influência do *framework* no comportamento das metaheurísticas. A discussão a seguir aborda a performance geral dos algoritmos, o impacto da dimensionalidade e, crucialmente, uma comparação direta entre os resultados obtidos nas duas plataformas.

A principal conclusão da análise de qualidade da solução no *PyTorch* é a **notável e consistente superioridade do algoritmo Busca Cuco (BC)**. Enquanto em 50D a disputa foi mais equilibrada, nos cenários de alta dimensionalidade (100D e 500D), o BC obteve os melhores resultados (menores valores da função objetivo) em **todas as oito funções benchmark**, superando as demais metaheurísticas de forma conclusiva. O Evolução Diferencial (ED), que se destacou no TensorFlow, apresentou um desempenho competitivo em 50D, mas não conseguiu acompanhar a robustez do BC à medida que a complexidade do problema aumentava. Em contrapartida, o Algoritmo Genético (AG) demonstrou ser o menos eficaz, consistentemente apresentando os piores valores de aptidão, especialmente em 500D.

Do ponto de vista da aceleração, o ambiente *PyTorch* proporcionou um *speedup* significativo para todos os algoritmos, com valores que chegaram a superar **7,8x**. Os algoritmos OEP e BC foram os mais rápidos em termos de tempo de execução na *GPU*, enquanto o AG foi, novamente, o mais lento.

A observação mais significativa, no entanto, é a aparente **ausência do trade-off entre velocidade e qualidade para o Busca Cuco**. Diferentemente do cenário no TensorFlow, onde o algoritmo mais preciso (ED) era um dos mais lentos, no PyTorch, o BC conseguiu unir os dois mundos: foi consistentemente um dos mais rápidos na execução em *GPU* e, ao mesmo tempo, o que encontrou as soluções de maior qualidade em alta dimensionalidade.

Isso sugere que a implementação do BC se alinha de forma excepcionalmente eficiente com a maneira como o *PyTorch* gerencia as operações vetorizadas e a execução em *CUDA*.

4.3 Análise Comparativa: TensorFlow e PyTorch

Para consolidar as descobertas deste trabalho, esta seção finaliza com uma comparação direta entre os resultados de desempenho obtidos nos frameworks *TensorFlow* e *PyTorch*, seguida por uma discussão sobre as implicações e a sinergia observada entre os algoritmos e as plataformas.

A Tabela 35 sintetiza os melhores resultados para o cenário de maior complexidade (500D), permitindo uma avaliação direta do impacto da escolha do *framework*.

Tabela 35 – Comparativo de Melhor Desempenho (Qualidade e Speedup) entre Frameworks (500D)

Função	**Qualidade da Solução (Menor Valor)**		**Melhor Aceleração (Speedup)**	
	TensorFlow	**PyTorch**	**TensorFlow**	**PyTorch**
Discus	BC (10666)	**BC (234,15)**	AG (7,64x)	**OEP (7,49x)**
Rastrigin	ED (3204.5)	**BC (248,88)**	ED (4,45x)	**OEP (7,18x)**
Rosenbrock	ED (1555.3)	**BC (276,32)**	**ED (7,70x)**	OEP (6,56x)
Cigar	ED (4461.8)	**BC (231,90)**	AG (6,40x)	**OEP (7,52x)**
Elliptic	ED (12.608)	**BC (278,30)**	AG (3,44x)	**OEP (6,48x)**
Ackley	ED (1.4713)	**BC (367,83)**	AG (3,59x)	**OEP (5,01x)**
Griewank	ED (0.0074)	**BC (299,65)**	AG (6,31x)	**OEP (5,57x)**
Salomon	ED (1.9080)	**BC (275,61)**	AG (3,29x)	**OEP (6,31x)**

Fonte: Autor. Os melhores resultados entre os dois frameworks estão em negrito.

A análise numérica da Tabela 35 revela duas tendências claras. Primeiramente, do ponto de vista da **qualidade da solução**, a implementação em *PyTorch* com o algoritmo Busca Cuco (BC) foi **drasticamente superior**, obtendo os menores valores em todas as oito funções. Em segundo lugar, ao analisar a **aceleração computacional (*speedup*)**, o cenário foi mais competitivo: o *TensorFlow* se destacou em acelerar os algoritmos AG e ED, enquanto o *PyTorch* demonstrou maior capacidade em acelerar o OEP.

Essa divergência de resultados aponta para a conclusão mais importante deste trabalho: não há uma única "melhor combinação"de algoritmo e ferramenta, mas sim uma forte **sinergia** e **interdependência** entre eles.

- No **TensorFlow**, a análise revelou um claro *trade-off*: o **ED** foi o algoritmo de maior precisão, mas não o mais rápido, exigindo uma escolha do usuário entre velocidade e qualidade.
- No **PyTorch**, esse *trade-off* foi menos evidente, pois o **BC** se consolidou como a opção dominante em ambos os critérios para problemas de alta dimensionalidade.

Essa diferença pode ser atribuída à maneira distinta como cada *framework* traduz as operações das metaheurísticas em **instruções paralelas para a GPU**. A natureza dinâmica do *PyTorch* pode ter beneficiado a implementação do BC, enquanto as otimizações de grafo do *TensorFlow* podem ter favorecido as operações vetoriais do ED. Portanto, a conclusão final é que a eficácia da paralelização em *GPU* depende da complexa interação entre o algoritmo, a topologia do problema e a arquitetura do *framework*, sendo a escolha conjunta desses elementos um fator determinante para alcançar tanto a máxima velocidade quanto a melhor qualidade de solução.

5 CONCLUSÃO

Este trabalho teve como objetivo central investigar de que forma a implementação de paralelismo com metaheurísticas em *frameworks* como *TensorFlow* e *PyTorch* influencia o desempenho computacional e a qualidade das soluções obtidas em problemas de otimização de alta dimensionalidade. Respondendo diretamente à **pergunta de pesquisa**, a análise dos resultados revelou que a paralelização teve um impacto **inequivocamente positivo no desempenho computacional**, mas uma influência **complexa e dependente do contexto na qualidade da solução**.

Com base nesses achados, a **hipótese** inicial foi **parcialmente confirmada**. A premissa de que o paralelismo reduziria significativamente o tempo de execução foi amplamente validada, com acelerações (*speedup*) que atingiram até **8,62x** e uma redução de tempo superior a 88%. Contudo, a premissa de que a qualidade da solução seria sempre "melhorada ou mantida"não se sustentou universalmente. Em vez disso, os resultados demonstraram um sutil, porém crucial, *trade-off* entre velocidade e precisão, que se manifestou de maneiras distintas em cada *framework* e se tornou a principal contribuição desta análise.

A principal contribuição deste trabalho foi a identificação de um claro *trade-off* no ambiente TensorFlow: o algoritmo de maior precisão (ED) não foi o mais rápido, enquanto os mais velozes (OEP, AG) não foram os mais precisos. Em contraste, no ambiente PyTorch, este *trade-off* foi suplantado pela sinergia entre o algoritmo Busca Cuco (BC) e o *framework*, que se consolidou como a abordagem dominante tanto em velocidade de execução quanto em qualidade da solução em cenários de alta dimensionalidade. Isso demonstra que a escolha do framework não é neutra, mas sim um fator que interage com a estrutura de cada metaheurística, influenciando o resultado final da otimização.

Por fim, como sugestão para **trabalhos futuros**, ressalta-se a importância da continuidade desta análise de desempenho. O campo da computação de alto desempenho está em constante evolução, e seria de grande valor reavaliar as metaheurísticas aqui estudadas em novas gerações de *GPUs* e versões atualizadas do *PyTorch* e *TensorFlow*, para verificar se os *trade-offs* e sinergias observados entre algoritmos e *frameworks* se mantêm. Adicionalmente, a expansão da análise para ambientes com múltiplos dispositivos (*multi-GPU*) ou múltiplos nós computacionais (*multi-node*), explorando estratégias de paralelismo distribuído como o modelo de ilhas, constitui um próximo passo natural. Tal investigação permitiria compreender a escalabilidade real desses algoritmos em larga escala e os novos gargalos de comunicação que surgem, contribuindo para a aplicação de metaheurísticas em problemas de otimização ainda mais complexos e exigentes.

REFERÊNCIAS

ABADI, E. A. M. *TensorFlow: Large-scale machine learning on heterogeneous distributed systems*. [S.l.], ,2016. Disponível em: <https://www.tensorflow.org/>. 13, 16, 30

ALBA, E.; TOMASSINI, M. *Parallelism and evolutionary algorithms*. [S.l.], 2002. v. 6, 443 - 462 p. Disponível em: <https://www.researchgate.net/publication/3418716_Parallelism_and_evolutionary_algorithms/>. 16, 19, 20

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. [S.l.]: ACM, 1967. p. 483–485. 17

ATTUX, R. *et al. Algoritmos Evolutivos – Visão Geral*. [S.l.], 2021. Disponível em: <https://www.dca.fee.unicamp.br/~lboccato/topico_3_algoritmos_evolutivos_visao_geral.pdf>. 18, 19, 21

BARNEY, B. *Introduction to Parallel Computing*. [S.l.], 2021. <https://hpc-tutorials.llnl.gov/>. 18

CHEUNG *et al. A Nonhomogeneous Cuckoo Search Algorithm Based on Quantum Mechanism for Real Parameter Optimization*. [S.l.], 2016. v. 47, 1-12 p. 24

CHIEN, S. W. D. *et al. TensorFlow Doing HPC*. [S.l.], 2019. 509-518 p. 15

DORNELES, L. D. L. *Comparando diferentes implementações paralelas de algoritmos genéticos em GPUs com CUDA*. [S.l.], 2021. Disponível em: <https://www.lume.ufrgs.br/bitstream/handle/10183/224285/001128636.pdf?sequence=1/>. Acesso em: Acesso em: 25 ago. 2021. 15

EBERHART R.; KENNEDY, J. *A new optimizer using particle swarm theory. In: Micro Machine and Human Science*. [S.l.], 1995. 39–43 p. Disponível em: <https://www.lume.ufrgs.br/bitstream/handle/10183/224285/001128636.pdf?sequence=1/>. 22

GOLDSCHMIDT. *Complemento IV-Introdução aos Algoritmos Genéticos*. [S.l.], 2015. Disponível em: <https://eic.cefet-rj.br/dm2ed/complementos/dm2ed-ComplementoIV.pdf/>. Acesso em: Acesso em: 19 ago. 2021. 20, 22

HENNESSY, D. A. P. J. L. *Computer Architecture: A Quantitative Approach*. 6th. ed. [S.l.], 2017. Disponível em: <https://www.elsevier.com/books/computer-architecture/patterson/978-0-12-804522-5>. 26

KUMAR. *A Systematic Review on High-Performance Computing Techniques for Metaheuristics*. [S.l.], 2023. Disponível em: <https://dl.acm.org/doi/10.1145/3550484>. 13

LI, S. *et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training*. [S.l.], 2020. abs/2006.15704. 15

LIANG *et al. Problem Definitions and Evaluation Criteria for the CEC 2014 Special Session and Competition on Single Objective Real-Parameter Numerical Optimization*. [S.l.], 2013. Disponível em: <https://www.ntu.edu.sg/cas/epr/cec2014/publications/CEC2014-benchmark-technical-report.pdf>. 32

MARINTECH, F. *Mooring Line and Anchor: Mooring Line Type*. [S.l.], 1995. Disponível em: <http://repositorio.poli.ufrj.br/monografias/monopoli10022180.pdf/>. Acesso em: Acesso em: 15 ago. 2021. 23

MASSAGO, S. *Introdução ao Algoritmo Genético, UFSCAR*. [S.l.], 2013. Disponível em: <https://www.dm.ufscar.br/profs/sadao/download/?file=article/algoritmos-geneticos.pdf/>. Acesso em: Acesso em: 16 ago. 2021. 20

NELLI, F. *Parallel And High Performance Programming With Python*. [S.l.], 2023. v. 1, 1-339 p. 26, 27

NUMPY. *NumPy v1.21 Manual*. [S.l.], 2021. Disponível em: <https://numpy.org/doc/stable/>. Acesso em: Acesso em: 19 ago. 2021. 16

PACHECO; AL, E. *Algoritmos Genéticos: Princípios e Aplicações*. [S.l.], 2021. Disponível em: <http://www.inf.ufsc.br/~mauro.roisenberg/ine5377/Cursos-ICA/CE-intro_apost.pdf/>. Acesso em: Acesso em: 19 ago. 2021. 20

PYTORCH. *PyTorch Documentation*. [S.l.], 2020. Disponível em: <https://pytorch.org/>. 15, 16

ROCHA; SARAMAGO, S. F. P. N. C. *Estudo de algumas Estratégias da Evolução Diferencial*. [S.l.], 2011. Disponível em: <https://files.cercomp.ufg.br/weby/up/498/o/Faimison2010.pdf/>. Acesso em: Acesso em: 19 ago. 2021. 23, 24

ROVEDA; UGO. *O que é Python, para que serve e por que aprender?* [S.l.], 2021. Disponível em: <https://files.cercomp.ufg.br/weby/up/498/o/Faimison2010.pdf/>. Acesso em: Acesso em: 19 ago. 2021. 16

SMITH. *Studying the Impact of TensorFlow and PyTorch Bindings on Machine Learning Software Quality*. [S.l.], 2023. Disponível em: <https://www.researchgate.net/publication/382080360_Studying_the_Impact_of_TensorFlow_and_PyTorch_Bindings_on_Machine_Learning_Software_Quality>. 13

TANABE, R.; FUKUNAGA, A. *Success-history based parameter adaptation for differential evolution*. [S.l.], 2013. 71–78 p. 24

TAVARES; NEDJAH. *Utilização de Otimização por Enxame de Partículas e Algoritmos Genéticos em Rastreamento de Padrões*. [S.l.], 2015. Disponível em: <http://abricom.org.br/wp-content/uploads/2016/01/cbic2015_submission_49.pdf/>. Acesso em: Acesso em: 19 ago. 2021. 22

YANG; SUASH, X.-S. . *Cuckoo Search via Levy Flights*. [S.l.], 2010. Disponível em: <https://www.researchgate.net/publication/45904981_Cuckoo_Search_via_Levy_Flights/>. Acesso em: Acesso em: 16 ago. 2021. 25