

UNIVERSIDADE FEDERAL DO PAMPA

Angelo Geovanni Amaral Menezes

Análise de Desempenho de Algoritmos de Enxame Paralelizados em GPU

Alegrete
2019

Angelo Geovanni Amaral Menezes

Análise de Desempenho de Algoritmos de Enxame Paralelizados em GPU

Trabalho de Conclusão de Curso apresentado
ao Curso de Graduação em Ciência da Com-
putação da Universidade Federal do Pampa
como requisito parcial para a obtenção do tí-
tulo de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Claudio Schepke

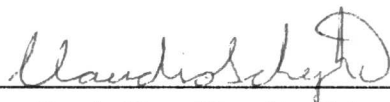
Alegrete
2019

Angelo Geovanni Amaral Menezes

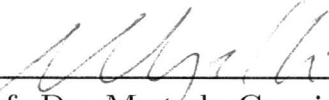
Análise de Desempenho de Algoritmos de Enxame Paralelizados em GPU

Trabalho de Conclusão de Curso apresentado
ao Curso de Graduação em Ciência da Com-
putação da Universidade Federal do Pampa
como requisito parcial para a obtenção do tí-
tulo de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 24 de junho de 2019.
Banca examinadora:



Prof. Dr. Claudio Schepke
Orientador
UNIPAMPA



Prof. Dr. Marcelo Caggiani Luizelli
UNIPAMPA



Prof. Dr. Marcelo Resende Thielo
UNIPAMPA

RESUMO

Algoritmos de enxames são um conjunto de heurísticas utilizados na busca de soluções ótimas em aplicações onde existem uma infinidade de possibilidades de respostas. Eles são extremamente versáteis e bastante utilizados em uma gama de diversos problemas, como por exemplo, no roteamento de veículos (BELL; MCMULLEN, 2004), predição de tendências do mercado financeiro (SHEN et al., 2011), problemas de fluxo de potência ótimo (MOHAMED et al., 2017), entre outros. Na busca por soluções ótimas, sendo que não se é necessário aos algoritmos compreenderem ou conhecerem completamente a função a ser avaliada, mantém-se a qualidade de resposta mesmo em casos de funções *black box*. Por natureza, estes algoritmos tem como característica uma relativa facilidade em sua paralelização, devido a uma certa independência dos componentes que formam o enxame, sendo que eles podem ser executados em um grande número de *threads*. Algumas aplicações que adotam algoritmos de enxame para a resolução de um problema exploram o paralelismo oferecido em CPU. Porém, a grande maioria não utiliza da computação em GPGPU para paralelização dos algoritmos. GPU oferecem um imenso número de *threads*, e alto desempenho em funções de cálculo básicas que são usadas na computação de *fitness* de um indivíduo ou de um enxame. Desta forma, este trabalho analisa e compara o desempenho de algoritmos de enxame implementados tanto em CPU quanto em GPU. Os resultados obtidos mostram que o desempenho em GPU é superior a CPU quando um número grande de elementos de enxame é utilizado.

Palavras-chave: Algoritmos de Enxame. Aplicações de Alto Desempenho. Paralelismo. GPU. GPGPU.

ABSTRACT

Swarm algorithms are a combined set of heuristics that are used in the search of optimal solutions in applications where there is a large amount of possibilities for answers. They are versatile and used in a diverse array of problems such as, vehicle routing problems (BELL; MCMULLEN, 2004), predicting trends in the financial market (SHEN et al., 2011), optimal power flow problems (MOHAMED et al., 2017), and many others. In the search for optimal solutions, since its not necessary for the algorithm to comprehend or know completely the problem that is being analyzed, keeping high quality answers even in black box scenarios. By nature these algorithms are relatively simple to paralelize, since the components that compose the swarm are semi-independent, making it possible to execute using a large number of threads. Some applications that use swarm algorithms for problem solving explore paralelism in CPU. But most aplications dont use GPGPU computing to paralelyze the algortihms. GPU offer a high number of threads, and fast computation of basic functions that are used in the evaluation of fitness of an individual or swarm. In this way, this work evaluate and compare both CPU and GPU implementations of swarm algorithms. The obtained results shows that the performance in GPU is better than CPU when a larger number of swarm elements are used.

Key-words: Swarm Algorithms. High Performance Applications. Parallelism. GPU. GPGPU.

LISTA DE FIGURAS

Figura 1 – Comparação de GFLOPS entre GPUs e CPUs. (OANCEA, 2014).	19
Figura 2 – Dança do Requebrado. (KARABOGA, 2005).	26
Figura 3 – Artificial Fish. (AZIZI, 2014).	30
Figura 4 – Exemplo do trajeto da mariposa voando, mantendo um ângulo fixo a fonte de luz e se aproximando em espiral. (MIRJALILI, 2015).	31
Figura 5 – Diagrama do modelo de paralelismo em CPU e modelo de paralelismo ingênuo em GPU. (TAN; DING, 2016).	33
Figura 6 – Diagrama do modelo de paralelismo multi fase em GPU. (TAN; DING, 2016).	34
Figura 7 – Diagrama do modelo de paralelismo em CPU e modelo de paralelismo somente em GPU. (TAN; DING, 2016).	35
Figura 8 – Função de Rastrigin.	43
Figura 9 – Tempos de execução para o algoritmo Particle Swarm Optimization (PSO) em CPU e GPU	45
Figura 10 – Fitness obtidos pelo algoritmo PSO em CPU e GPU	46
Figura 11 – Tempos de execução para o algoritmo Artificial Bee Colony (ABC) em CPU e GPU	47
Figura 12 – Fitness obtido utilizando o algoritmo ABC em CPU e GPU	47
Figura 13 – Tempos de execução para o algoritmo Moth-Flame Optimization (MFO) em CPU e GPU	48
Figura 14 – Fitness obtido utilizando o algoritmo MFO em CPU e GPU	49
Figura 15 – Tempos de execução para o algoritmo Artificial Fish Swarm Algorithm (AFSA) em CPU e GPU	49
Figura 16 – Fitness obtido utilizando o algoritmo AFSA em CPU e GPU	50

LISTA DE TABELAS

Tabela 1 – Trabalhos Relacionados.	38
Tabela 2 – Trabalhos Relacionados.	39
Tabela 3 – Ambiente de execução: CPU.	43
Tabela 4 – Ambiente de execução: GPU	43

LISTA DE ALGORITMOS

1	Particle Swarm Optimization (PSO)	25
2	Artificial Bee Colony (ABC)	27
3	Moth-Flame Optimization (MFO)	31
4	Paralelização ingênua	42

LISTA DE SIGLAS

ABC Artificial Bee Colony

ACO Ant Colony Optimization

ACS Ant Colony System

AF Artificial Fish

AFSA Artificial Fish Swarm Algorithm

BFO Bacterial Foraging Optimization

CPU Central Processing Unit

CSP Constraint Satisfaction Problems

FWA Fireworks Algorithm

GCM Conjugate Gradient Method

GPGPU General Purpose GPU

GPU Graphical Processing Unit

MFO Moth-Flame Optimization

PSO Particle Swarm Optimization

SM Streaming Multiprocessor

SP Stream processor

WSA Whale Swarm Algorithm

SUMÁRIO

1	INTRODUÇÃO	19
1.1	Objetivos do trabalho	20
1.2	Organização do texto	20
2	ALGORITMOS DE ENXAME	23
2.1	Particle Swarm Optimization	24
2.2	Artificial Bee Colony	25
2.3	Artificial Fish Swarm Algorithm	27
2.4	Moth-Flame Optimization	29
2.5	Conclusão do Capítulo	31
3	PARALELIZAÇÃO EM GPUS	33
3.1	Paralelismo Ingênuo	33
3.2	Paralelismo Multi Fase	33
3.3	Paralelismo somente em GPU	35
3.4	Paralelismo com múltiplos enxames	35
3.5	Conclusão do Capítulo	36
4	TRABALHOS RELACIONADOS	37
5	METODOLOGIA	41
5.1	Abordagem de Implementação	41
5.2	Bloco de código - Paralelismo ingênuo	41
5.3	Função de Rastrigin	41
5.4	Ambiente de testes	42
6	RESULTADOS	45
6.1	PSO	45
6.2	ABC	46
6.3	MFO	46
6.4	AFSA	48
6.5	Conclusão do Capítulo	50
7	CONSIDERAÇÕES FINAIS	51
	REFERÊNCIAS	53

1 INTRODUÇÃO

Algoritmos de Enxame são utilizados regularmente na solução de problemas diversos de otimização, especialmente aplicado aos problemas presentes nas áreas das engenharias, notando-se que cada vez mais, vem se aplicando algoritmos de enxame a problemas mais complexos, um exemplo de aplicação de algoritmos de enxame em problemas complexos é apresentado em (SHEN et al., 2011), onde temos a utilização do AFSA na predição de mercados financeiros.

Notando-se a importância de algoritmos eficientes para a solução destes problemas, durante a pesquisa realizada neste trabalho, foi possível perceber características inerentes dos algoritmos de enxame que possibilitam a sua paralelização em GPU. Sendo que, os algoritmos de enxame são compostos por diversos componentes semi independentes, desta forma, é possível uma implementação de forma paralela.

Em programas paralelizados em CPU, os mesmos podem não ter um melhor *speedup* a medida em que mais *threads* são adicionadas (TALLENT; MELLOR-CRUMMEY, 2009), devido a um *overhead* de comunicação, em programas paralelizados em GPU, procura-se utilizar o maior número de *threads* possível operando sobre os dados passados a GPU, pois o custo de transferência de informação entre CPU e GPU é extremamente alto.

Outro ponto a ser considerado é que, segundo (TAN; DING, 2016), quando algoritmos de enxame forem ser implementados em GPU, deve se priorizar a utilização de operações aritméticas de baixa precisão, devido ao seu desempenho superior se comparado com operações aritméticas de alta precisão em GPU, como demonstrado na Figura 1.

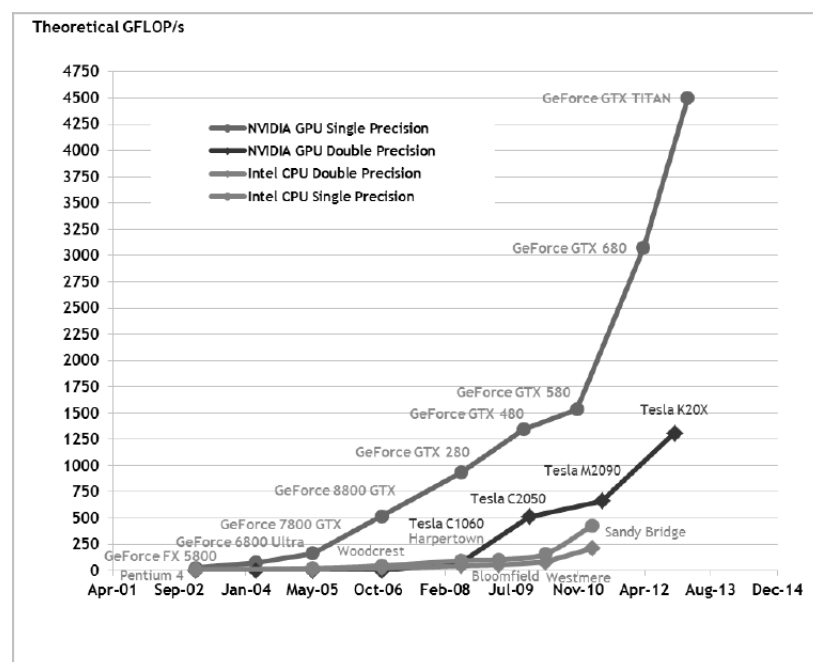


Figura 1 – Comparação de GFLOPS entre GPUs e CPUs. (OANCEA, 2014).

Porém, as implementações disponíveis, e que estão sendo utilizadas atualmente, são na maioria das vezes implementadas de forma não-paralelizada. Assim, nesse trabalho serão analisados os resultados obtidos pelos algoritmos implementados em CPU e em GPU

1.1 Objetivos do trabalho

O objetivo do trabalho é a análise de algoritmos de enxame implementados em GPU, comparados com os resultados obtidos com os mesmo implementados em CPU, simulando um problema complexo utilizando a Função de Rastrigin com um grande domínio de função.

Assim, verificar a viabilidade de uma biblioteca contendo implementações de algoritmos de enxame paralelizados em GPU.

Sendo que foram selecionados algoritmos de enxame que compartilham a semelhança de conterem cálculos de *fitness* a cada iteração. Dentre estes escolhidos, temos algoritmos clássicos e alguns mais recentes, visando cobrir uma diversidade grande de problemas onde um algoritmo possa ser eficiente na sua resolução. Também foi levado em consideração durante a escolha dos algoritmos o teorema No-free-lunch (WOLPERT; MACREARY, 1997), que diz que nenhum algoritmo é o melhor para a resolução de todos os problemas, implicando que mesmo que um algoritmo não é eficiente na solução de um certo problema, isso não significa que ele não é útil na solução de outras classes de problemas.

Objetivos específicos:

- Implementação - Desenvolver e analisar a implementação de algoritmos de enxame de forma paralelizada em GPU para algoritmos que ainda não foram paralelizados.
- Analisar o desempenho obtido pela implementação em GPU, e fazer comparações com a implementação em CPU.
- Verificar a viabilidade de implementação de uma biblioteca contendo algoritmos de enxame paralelizados em GPU.

1.2 Organização do texto

Este trabalho está organizado em 7 capítulos.

O capítulo 2 apresenta os algoritmos escolhidos, explicando o funcionamento dos mesmos.

Após, no capítulo 3 serão demonstradas as técnicas existentes na paralelização dos algoritmos, descrevendo pontos fortes e fracos das diferentes abordagens que se pode tomar na paralelização de algoritmos de enxame em Graphical Processing Units (GPUs).

No capítulo 4, será discutida a pesquisa extensiva que foi feita na área, estabelecendo o estado da arte, trabalhos importantes e trabalhos que estão diretamente relacionados com este trabalho.

Posteriormente, no capítulo 5 é apresentada a plataforma de programação que foi utilizada no desenvolvimento do trabalho, função utilizada nos testes e o ambiente computacional onde foram efetuados os mesmos.

No capítulo 6, são apresentados e discutidos os resultados de desempenho obtidos pelos algoritmos nas implementações testadas.

Finalmente, no capítulo 7 são feitas as considerações finais do trabalho, e pontos que podem ser abordados em trabalhos futuros.

2 ALGORITMOS DE ENXAME

Algoritmos de enxame são inspirados pelo comportamento coletivo, descentralizado e auto organizado demonstrados por certos animais ou padrões encontrados na natureza. Um algoritmo de enxame consiste em um grupo de indivíduos simples que se comunicam e trocam informações sobre o ambiente onde estão dispostos. Através deste comportamento individual simples é gerado um comportamento de enxame complexo, onde a força dos indivíduos é somada com os demais, beneficiando todos os membros do enxame. A partir desses comportamentos de acordo com (SERAPIÃO, 2009) podem ser sintetizados os seguintes princípios como inspiração na construção de algoritmos de enxame.

- Proximidade - Os indivíduos devem poder interagir entre si.
- Qualidade - Os indivíduos devem ser capazes de quantificar o seu comportamento.
- Diversidade - Permite ao sistema reagir a situações inesperadas.
- Estabilidade - Não são todas as variações de ambiente que devem gerar alterações no comportamento de um indivíduo.
- Adaptabilidade - Capacidade de se adequar a variações no ambiente.

Esses fatores levam a diversas qualidades necessárias a um algoritmo de busca por solução ótima, como por exemplo, a resiliência à fixação em soluções ótimas locais e poder operar de forma eficiente sobre problemas que não são conhecidos.

Algoritmos de enxame são úteis e eficientes em uma grande gama de problemas. Segundo o teorema No-free-lunch (WOLPERT; MACREADY, 1997), nenhum algoritmo é o melhor para a resolução de todos os problemas, implicando que mesmo que um algoritmo não é eficiente na solução de um certo problema, isso não significa que ele não é útil na solução de outras classes de problemas.

Os algoritmos de enxame são compostos por diversos componentes semi independentes. Desta forma, é relativamente simples implementá-los de forma paralela. Ao contrário de programas que podem não ter um melhor *speedup* a medida em que mais *threads* são adicionadas, devido a um *overhead* de comunicação. Em programas paralelizados em GPU, procura-se utilizar o maior número de *threads* possível operando sobre os dados passados a GPU, pois o custo de transferência de informação entre CPU e GPU é extremamente alto.

Sendo assim, deve-se buscar um aumento no desempenho através da execução de um grande número de *threads* simultâneas, sobre as informações passadas.

Segundo (TAN; DING, 2016), outro ponto a ser considerado é que, na maioria das aplicações, algoritmos de enxame não necessitam de operações aritméticas de alta precisão, podendo ser implementados utilizando métodos de cálculo mais rápidos porém

menos precisos, sem sacrificar a qualidade das respostas. Este fator contribui no *speedup* de implementações paralelizadas em GPU, devido ao fato de que GPUs possuem um alto desempenho nesse tipo de operação como já demonstrado na Figura 1.

Algoritmos de enxame destacam-se para a solução de diferentes tipos de problemas. Devido a este fato, foi levado em consideração a escolha de uma gama variada de algoritmos de enxames, incluindo algoritmos clássicos e algoritmos recentes. Nas próximas seções serão explicados, em detalhes, os algoritmos escolhidos para serem implementados na biblioteca.

2.1 Particle Swarm Optimization

PSO (EBERHART; KENNEDY, 1995) é um algoritmo baseado na abstração do comportamento de um bando de pássaros. Eles são representados por partículas que se comunicam entre si para a regulação da direção e velocidade individual a fim de convergir em uma solução ótima, similarmente ao comportamento apresentado na natureza por esses animais, onde há trocas de informações entre os indivíduos, a fim de se chegar ao local desejado ou encontrar alimento, geralmente utilizado na busca otimizações de funções.

O artigo original, (EBERHART; KENNEDY, 1995), é um trabalho muito importante para a área de inteligência artificial. Ele tem mais de 50.000 citações e serviu de base e inspiração para a criação de diversas variações de abordagem de PSO, como por exemplo, revisões do artigo do PSO pelos autores originais (SHI et al., 2001; SHI, 2004; POLI; KENNEDY; BLACKWELL, 2007; KENNEDY, 2011), PSO paralelizado para clusterização utilizando Map Reduce (ALJARAHI; LUDWIG, 2012) e estratégias de múltiplos enxames e mutações (CHEN et al., 2017).

No PSO, inicia-se as partículas que compõem o enxame com valores aleatórios para velocidade e posição inicial no espaço de resposta. O espaço de resposta representa o domínio de valores que podem ser inseridos para avaliação do problema, utilizando a função que define o problema.

A cada iteração do algoritmo é analisado o *fitness* de cada partícula, sendo o *fitness* o valor calculado da partícula de acordo com sua disposição no espaço de resposta, onde partículas mais próximas ao resultado esperado obtém um *fitness* melhor que as demais. Após o cálculo do *fitness* destas partículas são analisadas todas as partículas que compõem o enxame, a fim de encontrar a partícula com o melhor *fitness*. Com esses valores calculados, é atualizado a velocidade e posição da partícula, introduzindo valores aleatórios a esses parâmetros, para evitar que estas fiquem presas a soluções ótimas locais. No Algoritmo 1 tem-se o pseudocódigo do PSO, onde se tem como entrada o tamanho do enxame a ser utilizado.

O critério de parada do PSO é geralmente o número pré-definido de iterações, porém em problemas onde já se conhecem as respostas, ou em aplicação de treinamento de redes neurais, pode-se utilizar o melhor *fitness* como critério de parada.

Algoritmo 1: Particle Swarm Optimization (PSO)

```

Entrada Tamanho do enxame
Saída Posição com melhor fitness
partícula P [tamanho_enxame]
para cada partícula p em P faça
|  $P \leftarrow \text{Inicializa\_Partículas}()$ 
fim para cada
 $gBest \leftarrow P[0]$ 
para  $i \leftarrow 1$  até  $max\_it$  faça
| para cada  $i$  em Tamanho do enxame faça
| |  $P[i].fitness\_p \leftarrow fitness(p)$ 
| | se  $P[i].fitness\_p > P[i].pBest$  então
| | |  $P[i].pBest \leftarrow P[i].fitness\_p$ 
| | fim se
| | se  $P[i].pBest > gBest$  então
| | |  $gBest \leftarrow P[i].pBest$ 
| | fim se
| |  $v \leftarrow v + c1 * rand * (P[i].pBest - p) + c2 * rand(gBest - p)$ 
| |  $p \leftarrow p + v$ 
| fim para cada
fim para
retorna  $gBest$ 

```

Algumas implementações do PSO adicionam uma variável extra ao cálculo de velocidade e posição das partículas. Essa variável corresponde a inércia das partículas, o que limita ou não o quanto as partículas se deslocam no espaço de resposta, podendo assim ter-se um foco em buscas locais ou globais.

2.2 Artificial Bee Colony

Artificial Bee Colony (KARABOGA; BASTURK, 2007) é baseado no comportamento de uma colmeia de abelhas, sendo essa colmeia dividida em 3 tipos diferentes de abelhas:

- **Campeiras** - São abelhas enviadas para avaliar a qualidade das fontes de alimento ao redor da colmeia.
- **Seguidoras** - São enviadas para analisar possíveis fontes de alimentos e locais próximos fazendo buscas locais, as abelhas seguidoras também são divididas em duas sub-categorias.

Seguidoras empregadas - São as abelhas seguidoras que estão atualmente vinculadas a uma fonte de alimento, divulgando informações sobre esta fonte de alimento as demais abelhas.



Figura 2 – Dança do Requebrado. (KARABOGA, 2005).

Seguidoras desempregadas (abelhas olheiras) - São as abelhas seguidoras que estão procurando alguma fonte de comida, também são chamadas de abelhas olheiras.

- Escudeiras - Se não é possível a quantificação da qualidade das fontes de alimento após um certo número de tentativas pelas abelhas campeiras, estas se tornam abelhas escudeiras, e passam a conduzir buscas globais por fontes de alimento.

Cada abelha exerce sua função ao redor da colmeia, que representa o espaço da solução do problema. Elas comunicam-se com as demais abelhas através da dança do requebrado informando as demais sobre as diferentes fontes de alimento presentes no espaço ao redor da colmeia. Essas melhores fontes de alimento representam as melhores soluções do problema. A Figura 2 mostra os diferentes movimentos da dança das abelhas.

Inicialmente, o algoritmo recebe como entrada o número de fontes de alimento (SN), um limite máximo de tentativas de avaliação de uma fonte de alimento (limite), e um limite máximo de avaliações de *fitness* (MFE). Após a definição destes valores, é feita a inicialização da colmeia e é calculado um valor de *fitness* inicial, sendo este resultado inicial armazenado como a melhor solução. Após, dá-se início as repetições do algoritmo, onde a cada iteração do algoritmo são executadas as seguintes fases:

- Fase das abelhas seguidoras empregadas - Nesta fase, as abelhas fazem buscas pela vizinhança buscando as fontes de alimento mais nutritivas. Após uma fonte adequada ser encontrada, é calculado o *fitness*, e essa fonte é comparada com a informação de *fitness* inicial, mantendo a fonte de alimento com o melhor *fitness*. Após esta decisão, o *fitness* é comunicado as demais abelhas através da dança do requebrado.
- Fase das abelhas campeiras - Com a informação dos *fitness* das fontes de alimento, as abelhas campeiras escolhem de forma probabilística uma fonte de alimento.
- Fase das abelhas olheiras - Nesta fase, as abelhas seguidoras empregadas que ultrapassaram o número máximo de iterações de uma fonte de alimento se transformam em abelhas olheiras, e essa fonte de alimento previamente iterada é abandonada e as abelhas olheiras começam a procurar aleatoriamente por fontes de alimento no espaço de solução.

Algoritmo 2: Artificial Bee Colony (ABC)

Entrada Número de fontes de alimento, tentativas, avaliações de fitness.
Saída Posição com melhor *fitness*.

```

// Inicialização das fontes de alimento
fontes_alimento FS[num_fontes_alimento]
para  $i \leftarrow 1$  até  $num\_abelhas\_empregadas$  faça
     $FS\_posição \leftarrow posição\_aleatória()$ 
     $FS\_fitness \leftarrow fitness(FS\_posição)$ 
     $FS\_tentativas \leftarrow 0$ 
fim para
 $melhor \leftarrow melhor\_fonte(FS)$ 
enquanto  $iteração\_atual < max\_it$  faça
    // Abelhas empregadas procuram novas fontes de alimento, e é feita
    // a análise do fitness destas fontes
    Fase_abelhas_empregadas ()
    // Abelhas olheiras analisam as melhores fontes de alimento de
    // acordo com o cálculo de probabilidade
    Fase_abelhas_olheiras ()
    // Abelhas escudeiras verificam uma fonte de alimento aleatória
    Fase_abelhas_escudeiras ()
    se  $melhor\_fonte(FS) > melhor$  então
         $melhor \leftarrow melhor\_fonte(FS)$ 
    fim se
     $iteração\_atual \leftarrow iteração\_atual + 1$ 
fim enqto
retorna  $melhor$ 

```

Ao final de cada iteração, os melhores resultados são salvos, e comparados com o melhores resultados obtidos anteriormente, mantendo o melhor valor ao longo das iterações. O critério de parada é um número de iterações pré-definido, No algoritmo 2 tem-se o pseudocódigo do algoritmo.

As aplicações do ABC são tradicionalmente em problemas de otimização, especialmente na área de engenharia em problemas de distribuição de energia, com em (BAI; EKE; LEE, 2017).

2.3 Artificial Fish Swarm Algorithm

Artificial Fish Swarm Algorithm (LI, 2003), é baseado no comportamento demonstrado por certos peixes na busca por comida, na evasão de predadores e na migração para melhores locais. O comportamento de enxame é mais seguro e eficiente para os peixes, sendo que através do comportamento individual de cada peixe e comunicação com os demais, estes conseguem encontrar locais na água onde se tem mais comida, ou comida de melhor qualidade. Assim similarmente ao PSO e ABC, pode-se abstrair certos conceitos e implementá-los em um algoritmo de enxame, utilizando a água como local de resposta

e os locais com comida como pontos máximos.

Algumas diferenças do AFSA se comparado ao PSO é a utilização de alguns aspectos similares a algoritmos genéticos como o uso de gerações de indivíduos. Outras qualidades herdadas dos algoritmos genéticos incluem a independência do gradiente de informação da função analisada e a capacidade de resolver problemas multi dimensionais complexos sem a necessidade de ajuste de parâmetros. Porém, AFSA não utiliza funções de cruzamento e mutação de algoritmos genéticos.

O AFSA é utilizado na resolução de problemas de otimização, *data mining*, processamento de imagens, treinamento de redes neurais e em processamento de sinais.

No algoritmo AFSA clássico abordado em (NESHAT et al., 2014), tem-se cinco comportamentos básicos apresentados pelos Artificial Fish (AF).

- AF_Prey - O comportamento mais básico dos AF, representado na figura 3, e utilizado na busca por alimento, depende da posição do AF e se alguma fonte de alimento está em seu campo de visão, assim decidindo a direção que o AF vai tomar. Esse comportamento é composto pelas seguintes funções.

$$Xj = Xi + Visual.rand() \quad (2.1)$$

Nessa função Xj corresponde a posição futura do AF, que é o resultado da posição atual do AF Xi somada com o resultado da função $Visual.rand()$, sendo que esta função retorna um valor resultante da escolha de um valor aleatório do campo de visão do AF. Esse campo de visão corresponde a concentração de alimentos Y , que é a representação da função do problema a ser resolvido.

Se o próximo estado do campo de visão for melhor que o estado atual ($Yi < Yj$), utiliza-se a seguinte função para dar um passo nessa direção.

$$Xi^{(t+1)} \leftarrow Xi^{(t)} + \frac{Xj - Xi^{(t)}}{\|Xj - Xi^{(t)}\|} \cdot Step.rand() \quad (2.2)$$

Se a condição ($Yi < Yj$) for falsa, é escolhido aleatoriamente um novo estado Xj . Esse novo estado é analisado com a condição anterior. Se o número máximo de tentativas (*try_numbers*) for ultrapassado sem sucesso, uma nova posição para o AF deve ser calculada aleatoriamente utilizando a Função 2.3.

$$Xj^{(t+1)} \leftarrow Xi^{(t)} + Visual.rand() \quad (2.3)$$

- AF_Swarm - Como definido anteriormente, os AF procuram deslocar-se utilizando informações de outros AF e das fontes de comida ao seu redor, mantendo uma

formação de enxame. Esse comportamento de enxame se dá através da seguinte maneira, considerando o X_i o estado atual do AF, X_c a posição central e nf o número de companheiros no setor atual ($d_{ij} < \text{Visual}$), n sendo o número total de peixes. Se $(Y_c > Y_i \wedge \frac{nf}{n} < \delta)$ for verdadeiro, isso significa que o centro dos companheiros tem mais alimento (melhor *fitness*) e não tem uma grande concentração de AF, então o AF dá um passo para o centro dos companheiros.

$$X_i^{(t+1)} \leftarrow X_i^{(t)} + \frac{X_c - X_i^{(t)}}{\|X_c - X_i^{(t)}\|} \cdot \text{Step.rand()} \quad (2.4)$$

Caso isso não for verdadeiro, executa-se o padrão AF_Prey .

- **AF_Follow** - Esse comportamento define as ações tomadas pelos AF. Quando um ou mais AF encontram uma fonte de alimento, sendo que após essa descoberta os AF que compõem o enxame se deslocam rapidamente até essa fonte de alimento. Sendo o X_i o estado atual do AF, e este desloca-se até a posição X_j onde se tem um companheiro, no setor atual ($d_{ij} < \text{Visual}$), que contém a melhor fonte de alimento Y_j . Se $(Y_j > Y_i \wedge \frac{nf}{n} < \delta)$ for verdadeiro, isso significa que o estado X_j possui uma melhor concentração de comida (melhor *fitness*) e não tem uma grande concentração de AF, então o AF se desloca até a posição X_j .

$$X_i^{(t+1)} \leftarrow X_i^{(t)} + \frac{X_j - X_i^{(t)}}{\|X_j - X_i^{(t)}\|} \cdot \text{Step.rand()} \quad (2.5)$$

Caso isso não for verdadeiro, executa-se o padrão AF_Prey .

- **AF_Move** - Os AF nadam de forma aleatória no espaço de resposta, na procura por alimentos ou companheiros. Sendo esse comportamento a função básica do comportamento AF_Prey , onde o AF se desloca para um local aleatório no seu campo de visão.

2.4 Moth-Flame Optimization

Mariposas conseguem se direcionar de forma excelente à noite utilizando um método chamado de orientação transversa de navegação. Esse método de orientação utiliza a luz vindo da lua, sendo que como a lua está distante das mariposas, estas se aproveitam desse fato e voam mantendo um ângulo fixo a luz da lua. Assim elas conseguem percorrer enormes distâncias em linha reta. Porém se confundem com luzes vindas de outras fontes, podendo ficar presas ao redor de fontes de luz artificiais, voando em espiral ao redor delas. Esse comportamento das mariposas é chamado de espiral da morte.

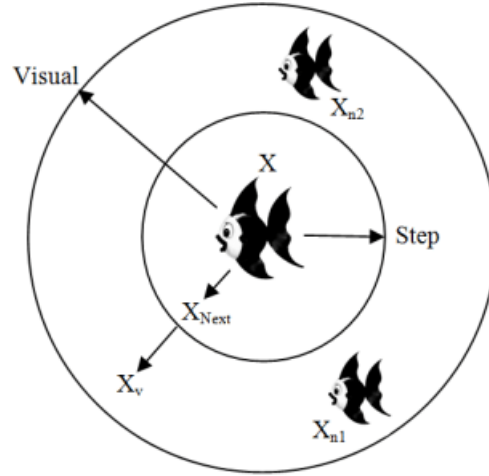


Figura 3 – Artificial Fish. (AZIZI, 2014).

Com isso em mente, Moth-Flame Optimization (MIRJALILI, 2015) utiliza o conceito de localização espacial como meio de orientação de uma geração de mariposas que percorrem o espaço de solução procurando manter um ângulo constante a fonte de luz próxima. Esse espaço de solução representa o problema que deve ser resolvido, e as fontes de luz representam as soluções. Assim, movem-se em espiral até encontrar uma solução ótima, conforme pode ser visto na Figura 4.

Porém muitas vezes as mariposas ficam presas em espirais da morte, onde elas ficam circulando a fonte de luz em um ângulo fixo sem conseguir escapar. Para evitar esse comportamento, o algoritmo utiliza uma tabela para salvar o melhor progresso das mariposas. Assim, mesmo que uma mariposa fique presa em uma espiral da morte ao redor de uma solução ótima local, o progresso será salvo e as outras mariposas poderão encontrar o máximo global.

No artigo onde foi definido o algoritmo MFO (MIRJALILI, 2015), foi utilizada a seguinte espiral a ser usada pelas mariposas.

$$S(M_i, F_j) = D_i \times e^{bt} \times \cos(2\pi t) + F_j \quad (2.6)$$

Onde o ponto de origem da espiral é a posição da mariposa, o ponto final da espiral é a fonte de luz e a espiral não pode sair do espaço de solução. Na função da espiral utilizada, D_i é a distância entre a mariposa i e a fonte de luz j , b é uma constante que define a forma dessa espiral e t é um número aleatório no intervalo $[-1,1]$.

Também ao longo da execução do algoritmo, o número de fontes de luzes vai se reduzindo, sendo escolhidas as melhores fontes de luzes e as piores vão sendo descartadas. Abaixo está representada a função que corresponde a esse comportamento.

$$N^o \text{ de fontes de luz} = \text{round}\left(N - l \times \frac{N - 1}{T}\right) \quad (2.7)$$

Algoritmo 3: Moth-Flame Optimization (MFO)

Entrada Tamanho do enxame, número de iterações.
Saída Posição com melhor *fitness*.

```

// Inicialização
mariposas M[tamanho_enxame]
para  $i \leftarrow 1$  até tamanho_enxame faça
    |  $M\_posicao[i] \leftarrow posicao\_aleatoria()$ 
    |  $M\_Fitness[i] \leftarrow fitness(M\_posicao[i])$ 
fim para
iteracao_atual  $\leftarrow 0$ 
enquanto iteracao_atual < max_it faça
    | calcula_num_fontes ()
    | // De acordo com a equação 2.7
    | atualiza_constante_convergencia ()
    | atualiza_proximidade_fonte ()
    | para  $i \leftarrow 1$  até tamanho_enxame faça
    | |  $M\_Fitness[i] \leftarrow fitness(M\_posicao[i])$ 
    | fim para
    | atualiza_vetor_mariposas ()
    | atualiza_fontes ()
fim enqto
retorna posicao_melhor_fitness (M)

```

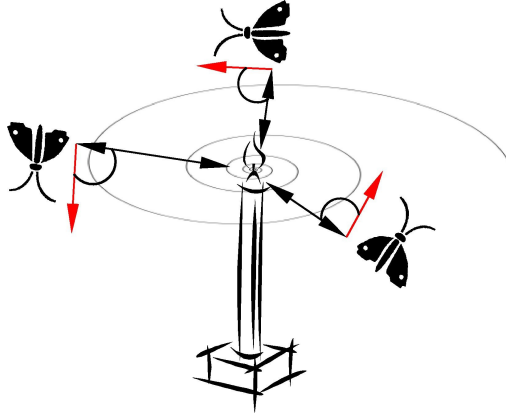


Figura 4 – Exemplo do trajeto da mariposa voando, mantendo um ângulo fixo a fonte de luz e se aproximando em espiral. (MIRJALILI, 2015).

Onde l é o número atual de iterações, N é o número máximo de fontes de luz e T corresponde ao número máximo de iterações. No algoritmo 3, tem-se o pseudocódigo representando o algoritmo.

2.5 Conclusão do Capítulo

Neste capítulo foram apresentados quatro algoritmos baseados em enxame. Estes algoritmos foram escolhidos devido a sua similaridade. Sendo que em todos, há a necessidade de se calcular o *fitness* a cada iteração do algoritmo.

3 PARALELIZAÇÃO EM GPUS

Segundo (TAN; DING, 2016), pode-se caracterizar as abordagens de paralelização de algoritmos de enxame em GPU em quatro modelos de paralelismo distintos: ingênuo, multi fase, somente em GPU, múltiplos enxames. A seguir serão abordados estes modelos.

3.1 Paralelismo Ingênuo

Um princípio básico de computação paralela é identificar o gargalo de um algoritmo e otimizá-lo. Em algoritmos de enxame, a avaliação do *fitness* apesar de ser uma função simples, na maioria dos algoritmos precisa ser calculada para cada elemento que compõem o enxame (n) a cada iteração do algoritmo (m), isso gera uma complexidade $O(n * m)$, sendo essa complexidade multiplicada também ao custo do cálculo de *fitness*.

As outras funções que compõem um algoritmo de enxame, não são tão facilmente paralelizáveis. Assim pode-se implementar uma solução paralela heterogênea, conforme detalhado na Figura 5. Desta forma, aproveita-se do desempenho de CPUs em comunicação e geração de números aleatórios, e do grande número de *threads* e desempenho em funções de baixa precisão de GPUs, e a possibilidade de cálculo de *fitness* de diversos elementos do enxame ao mesmo tempo.

3.2 Paralelismo Multi Fase

O modelo de paralelismo ingênuo possui uma grande facilidade de implementação e possibilidade de modularização do bloco do código de cálculo de *fitness*, assim podendo ser facilmente adotado em outros algoritmos. Uma grande desvantagem se dá ao fato da necessidade de transferência de informação sobre o enxame, da memória principal à

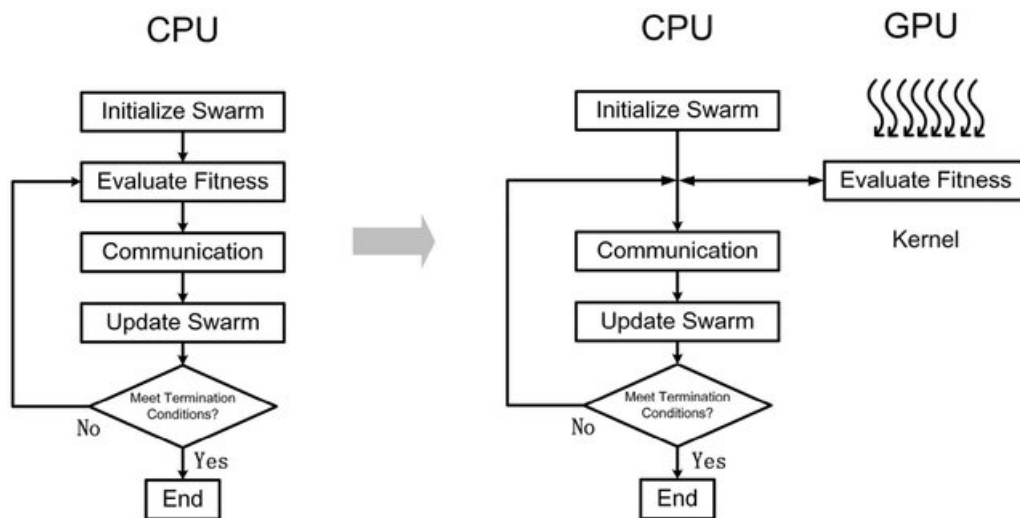


Figura 5 – Diagrama do modelo de paralelismo em CPU e modelo de paralelismo ingênuo em GPU. (TAN; DING, 2016).

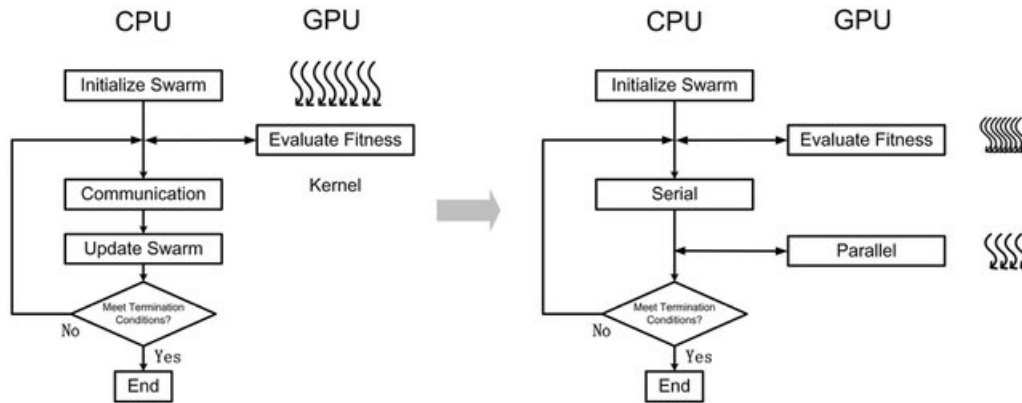


Figura 6 – Diagrama do modelo de paralelismo multi fase em GPU. (TAN; DING, 2016).

memória da GPU. Após a paralelização, da parte explicitamente paralela do algoritmo, o gargalo do desempenho será transferido a outras partes do algoritmo, como por exemplo, a atualização de velocidade e posição de partículas no PSO.

Com isso em mente, pode-se também buscar extrair desempenho utilizando paralelismo em GPU dessas outras partes do algoritmo, conforme pode ser visto na Figura 6. Neste caso, leva-se em conta na implementação de que essas funções do algoritmo são executadas em fases, não necessariamente em paralelo com funções de análise de *fitness*. Assim pode-se categorizar essas funções nos seguintes itens.

- Operações de vetor - Pode-se formular diversas operações que compõem algoritmos de enxames como operações de vetor, como o exemplo dado do PSO dado anteriormente, operações de atualização de velocidade e posição, ou gerações de faísca no algoritmo de fogos de artifício (TAN; ZHU, 2010). Essas operações de vetor podem ser implementadas de maneira altamente granular, utilizando uma *thread* para cada elemento do vetor.
- Redução - Funções de cálculo de máximo e mínimo utilizado em algoritmos como o Fireworks Algorithm (FWA), cálculo de distância em MFO, entre outras, em arquiteturas de GPU mais antigas podem ser resolvidas utilizando operadores atômicos e memória compartilhada. Já em arquiteturas mais recentes, é resolvido utilizando deslocamento de dados entre os registradores.
- Classificação - Utilizado em alguns algoritmos para a seleção de um certo indivíduo do enxame. É interessante também ser paralelizado em GPU devido ao fato de que as informações sobre os elementos do enxame já estão na memória da GPU.
- Construção de caminho - Um dos pontos focais de pesquisa na área de paralelização de algoritmos de enxame, principalmente na roleta de seleção da construção de caminho no Ant Colony Optimization (ACO).

- Atualização de feromônios - Outro ponto que vem sendo bastante pesquisado, sendo composta por duas tarefas principais, depósito e atualização de feromônio. A atualização de feromônio pode ser trivialmente paralelizado. Já o depósito de feromônios torna-se difícil de ser paralelizado, devido ao fato de que múltiplas formigas podem depositar feromônio ao mesmo tempo no mesmo local da matriz de feromônios.

Um ponto negativo desta abordagem é que ela não pode ser utilizada na implementação de todos os algoritmos de enxame como a implementação ingênua.

3.3 Paralelismo somente em GPU

Comparado com o desempenho computacional das GPUs, a comunicação entre GPU e CPU é extremamente lenta, o que pode afetar negativamente o resultado final obtido pela implementação heterogênea de algoritmos de enxame. Assim pode-se pensar em uma implementação completamente em GPU, obtendo um melhor desempenho por não ter um grande *overhead* de comunicação entre CPU e GPU, conforme mostrado na Figura 7.

3.4 Paralelismo com múltiplos enxames

Algoritmos de enxame sofrem com a rápida deterioração da capacidade de resolução de problemas a medida em que o espaço de busca e dimensões do problema crescem. Para resolver isso, na resolução de problemas maiores, pode-se dividi-lo em subproblemas menores e aplicar os algoritmos nesses subproblemas, fazendo a comunicação entre esses enxames.

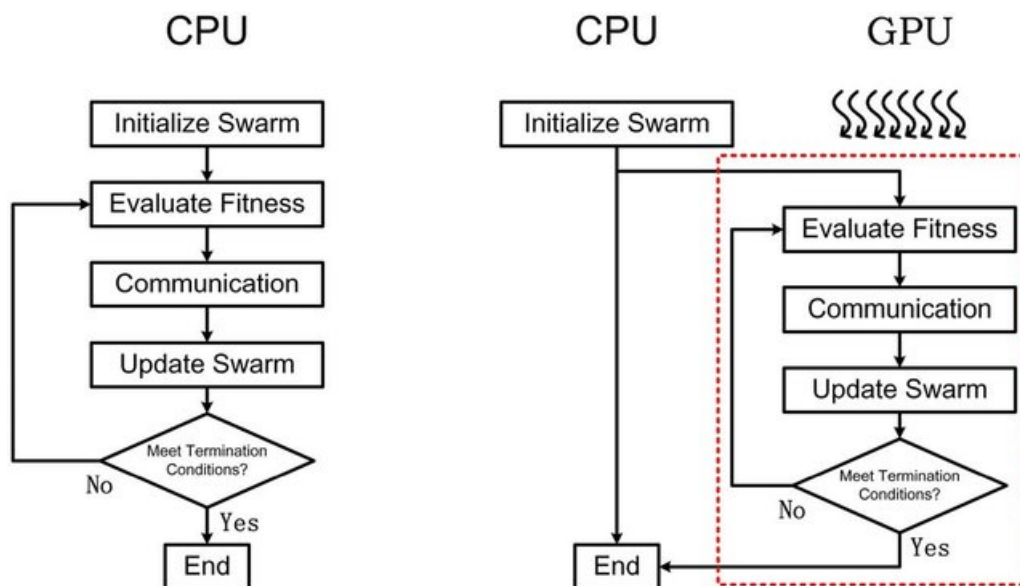


Figura 7 – Diagrama do modelo de paralelismo em CPU e modelo de paralelismo somente em GPU. (TAN; DING, 2016).

Na abordagem de múltiplos enxames tem-se dois modelos:

- Múltiplos enxames autônomos - São executadas múltiplas cópias independentes do algoritmo em paralelo, sendo que cada instância pode convergir a uma solução ótima diferente.
- Múltiplos enxames colaborativos - Analisam partes diferentes do problema e trocam informação entre si, a fim de convergir na melhor solução.

3.5 Conclusão do Capítulo

O foco deste trabalho é analisar o desempenho de implementações ingênuas de algoritmos de enxame. Também deverá ser feito um *profile* da execução de implementações sequenciais dos algoritmos, a fim de coletar resultados de tempo execução e compará-los aos resultados obtidos na abordagem paralela. Este tipo de comparativo é chamado de *speedup*.

4 TRABALHOS RELACIONADOS

O foco desta pesquisa, está na busca de algoritmos ou técnicas de paralelização de algoritmos do tipo enxame para GPU. Por ser um assunto relativamente recente, primeiramente, para a realização deste trabalho foi feita uma extensa pesquisa na literatura, a fim de agrupar o que há de melhor e mais recente sobre o assunto. Foram identificados possíveis focos de pesquisa e elaboração de trabalhos futuros. Neste capítulo estão dispostos alguns dos trabalhos mais importantes na área. A pesquisa foi realizada utilizando o *Google Scholar* e *Google Search*. Nestes, foram inseridos os seguintes termos de busca (Swarm Intelligence, Swarm algorithms, PSO, Parallel PSO, Boids, Parallel Boids, Parallel Swarm Algorithms, GPU Parallel Swarm Algorithms).

Após a análise inicial dos artigos selecionados a partir do resultado dessa pesquisa, foram feitas novas pesquisas sobre os pontos mais interessantes levantados pelos artigos selecionados, como por exemplo, técnicas de paralelismo e novos algoritmos. Alguns dos assuntos abordados pelos artigos previamente selecionados foram abandonados. Sendo que no final da etapa de pesquisa, foram analisados mais de 60 trabalhos distintos da área. Alguns citados diretamente neste trabalho, e os demais trabalhos relacionados estão dispostos nas Tabela 1 e Tabela 2.

Os trabalhos relacionados apresentados na Tabela 1 e Tabela 2 resgataram da literatura os algoritmos de enxame paralelizados. Após a pesquisa ficou aparente a escassez de implementações em GPU. Outro ponto levantado é um domínio no uso do PSO na elaboração dos trabalhos e nos trabalhos que utilizam do paralelismo em GPU. Na maioria das vezes é utilizada um modelo de paralelismo mais focado em GPU, o que dificulta a compreensão dos algoritmos por quem não possui conhecimento na área de programação em GPGPU e dificulta o uso destas implementações em outros algoritmos.

Tabela 1 – Trabalhos Relacionados.

Autores	Trabalho	Descrição
(TAN; DING, 2016)	A Survey on GPU-Based Implementation of Swarm Intelligence Algorithms	<i>Survey</i> compreensivo sobre estratégias de paralelização de algoritmos de enxame utilizando GPU, diferenças de desempenho entre CPU e GPU, ferramentas disponíveis e também aborda métricas para mensurar performance.
(MUSSI; DAOLIO; CAGNONI, 2011)	Evaluation of parallel particle swarm optimization algorithms within the CUDATM architecture	Implementação do PSO em GPU utilizando CUDA, com um modelo de paralelismo rodando mais código em GPU, sendo feito também uma avaliação dos resultados obtidos. Aborda em detalhes o funcionamento das <i>threads</i> em CUDA.
(DALI; BOUA-MAMA, 2015)	GPU-PSO : Parallel Particle Swarm Optimization approaches on Graphical Processing Unit for Constraint Reasoning: Case of Max-CSPs	Neste trabalho é utilizado uma implementação do PSO em GPU para a resolução de Constraint Satisfaction Problems (CSP), sendo CSPs problemas onde temos diversos elementos cujos estados devem satisfazer uma série de restrições, como por exemplo, alocação de recursos. Inicialmente no trabalho são abordados questões como o PSO em si, e arquitetura de GPUs e elementos do CUDA. Após é definida a estrutura da implementação do PSO a ser utilizada, sendo que neste trabalho também é utilizada um modelo de paralelismo com foco em GPU, e finalmente são analisados os resultados obtidos e comparados com implementações clássicas do PSO.
(OUYANG et al., 2015)	Parallel hybrid PSO with CUDA for ID heat conduction equation	É utilizada uma implementação modificada do PSO, usando em conjunto com Conjugate Gradient Method (GCM) para o cálculo de equações de condutividade de calor, sendo que estas equações são computacionalmente complexas. O cálculo dessas equações foi computado utilizando o algoritmo híbrido definido em GPU, utilizando uma abordagem de paralelismo similar a utilizada nos outros trabalhos. Os resultados foram comparados entre implementações paralelas do PSO, GCM e o algoritmo híbrido.
(SKINDEROWICZ, 2016)	The GPU-based parallel Ant Colony System	Aborda a implementação do algoritmo Ant Colony System (ACS) em GPU utilizando também um modelo de paralelismo focado em GPU, abordando questões como geração de números aleatórios, comunicação de elementos do enxame através do depósito de feromônios e analisando os resultados obtidos.

Tabela 2 – Trabalhos Relacionados.

(CHANG et al., 2005)	A parallel particle swarm optimization algorithm with communication strategies	Analisa o impacto da comunicação feita entre as partículas do enxame, buscando encontrar um equilíbrio entre qualidade de resposta e desempenho do algoritmo a partir da regulação da comunicação entre partículas em um algoritmo paralelizado tradicionalmente em CPU.
(ZHOU; TAN, 2009)	GPU-based Parallel Particle Swarm Optimization	Implementação do PSO em GPU utilizando como base a definição original do algoritmo dada por (EBERHART; KENNEDY, 1995), abordando algumas técnicas para melhorar o desempenho do PSO em GPU, como por exemplo, a criação de números aleatórios em CPU e armazenamento em memória da GPU, devido a dificuldade de criação de números aleatórios em GPU pela falta de aritmética de alta precisão em GPU.

5 METODOLOGIA

Este capítulo apresenta os aspectos metodológicos a serem seguidos para o desenvolvimento do trabalho.

5.1 Abordagem de Implementação

O trabalho foi desenvolvido utilizando a linguagem Python e a biblioteca PyCUDA (Klöckner et al., 2012) devido a sua facilidade de utilização e diversas implementações de algoritmos de enxame já existentes, como algumas bibliotecas de algoritmos de enxame (SwarmPackagePy, EvoloPy (FARIS et al., 2016), Hive), sendo este um dos objetivos do trabalho: implementar uma abordagem de paralelismo em GPU que possa ser facilmente utilizada em algoritmos já existentes. Foram utilizadas nesse trabalho implementações já existentes dos algoritmos abordados no Capítulo 2, levando em consideração a possibilidade do bloco de código responsável pelo cálculo paralelo de *fitness* poder ser utilizada em mais algoritmos no futuro.

Os códigos resultantes estão disponíveis na plataforma *GitHub* em (MENEZES, 2019), assim podendo ser utilizado e melhorado pela comunidade.

5.2 Bloco de código - Paralelismo ingênuo

O bloco de código responsável pelo paralelismo desenvolvido para o trabalho, foi desenvolvido em Python e C, e fica encarregado de receber os dados da CPU, alocar a memória na GPU e transferir estes dados a memória alocada, utilizando funções do PyCUDA (Klöckner et al., 2012). O código recebe dos algoritmos o número de elementos que compõem o enxame, e dois vetores de posição dos elementos, para o eixo X e Y. Com estes dados são alocados dois vetores na memória da GPU e os dados das posições são passado a eles. também é criado na memória da GPU um vetor que armazena o resultado do cálculo do *fitness* sobre estas posições.

Com os dados presentes na memória da GPU, é calculado o *fitness* em *threads* na GPU, onde cada *thread* fica encarregada de calcular o valor de *fitness* de um elemento do enxame. Essa atribuição de *threads* é feita utilizando o cálculo do identificador da *thread*, sendo utilizado um *block* CUDA unidimensional, de tamanho igual ao número de elementos do enxame, representado no pseudocódigo 4.

5.3 Função de Rastrigin

Para avaliação dos algoritmos, foi utilizada a função de Rastrigin representada em 5.1, com duas dimensões e espaço de solução $[-10000, 10000]$, esse grande espaço de solução foi escolhido a fim de simular um problema de otimização de grande escala. Sendo

Algoritmo 4: Paralelização ingênua

Entrada Tamanho do enxame, Vetor de posições X, vetor de posições Y
Saída Vetor de fitness

```

// Alocação de memória na GPU
posições_X_GPU = cuda.mem_alloc(posicoes_X.nbytes)
posições_Y_GPU = cuda.mem_alloc(posicoes_Y.nbytes)
resultado_fitness_GPU = cuda.mem_alloc(posicoes_X.nbytes)
// Alocação de memória na GPU
kernel = SourceModule(
#include <math.h>
__global__ void fitness(float * posições_X_GPU, float * posições_Y_GPU,
float * resultado_fitness_GPU)
{
int i ← blockDim.x * blockIdx.x + threadIdx.x;
// Exemplo de cálculo de fitness de uma esfera em GPU
resultado_fitness_GPU[i]
← pow(posições_X_GPU[i], 2) + pow(posições_Y_GPU[i], 2);
)
// Chamada de função
fitness = kernel.get_function("fitness")
fitness (posições_X_GPU, posições_Y_GPU, resultado_fitness_GPU)
block = (tamanho_enxame, 1, 1))
// Obtenção dos resultados
vetor_fitness [tamanho_enxame]
cuda.memcpy_dtoh(vetor_fitness, resultado_fitness_GPU)
retorna vetor_fitness

```

escolhida essa função pelo fato de conter diversos mínimos locais, como pode ser observado na Figura 8.

$$f(x_1 \cdots x_n) = 10n + \sum_{i=1}^n (x_i^2 - 10\cos(2\pi x_i))$$

$$-10000 \leq x_i \leq 10000$$

$$\text{mínimo em } f(0, \cdots, 0) = 0 \quad (5.1)$$

5.4 Ambiente de testes

O ambiente computacional que foi utilizado neste trabalho para os testes de execução é composto por uma máquina com dois *Intel Xeon E5-2650*, 128GB de memória RAM e uma GPU *Nvidia Quadro 5000M*, sendo estes componentes apresentados com mais detalhes nas Tabelas 3 e 4.

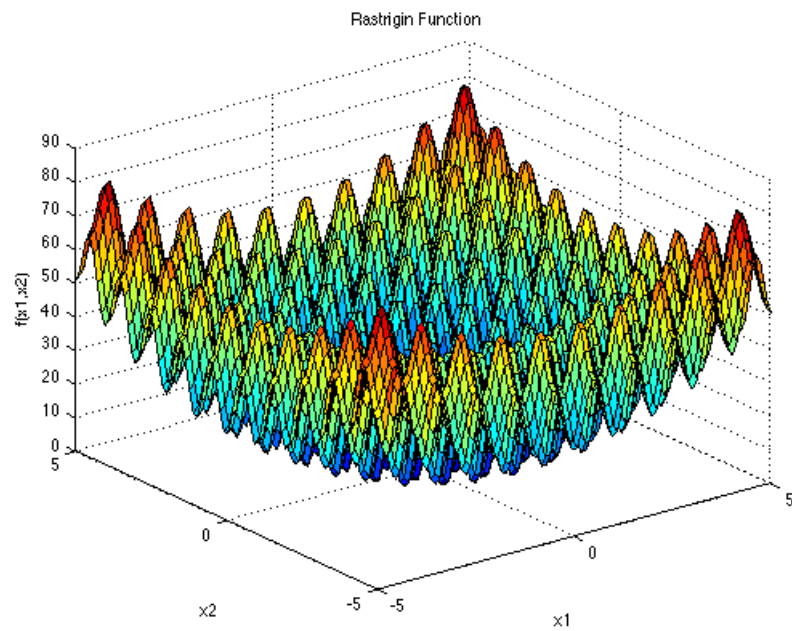


Figura 8 – Função de Rastrigin.

Tabela 3 – Ambiente de execução: CPU.

Características	Xeon E5-2650 ($\times 2$)
Frequência	2.00 GHz
Núcleos	8 ($\times 2$)
<i>Threads</i>	16 ($\times 2$)
<i>Cache</i> L1	32 KB
<i>Cache</i> L2	256 KB
<i>Cache</i> L3	20 MB
Memória RAM	128 GB

Tabela 4 – Ambiente de execução: GPU

Características	Quadro 5000M
Frequência	1.15 GHz
SM	11
SP	352
<i>Warp</i>	32
<i>Threads</i> por SM	1536
<i>Cache</i> L1	48 KB
<i>Cache</i> L2	640 KB
Memória Global	2.5 GB
Largura de banda de memória	211 GBps

6 RESULTADOS

Para a obtenção dos resultados foram executados os 4 algoritmos sobre a função de Rastrigin (Equação 5.1), com 100 iterações para cada um dos algoritmos, sendo este teste repetido 10 vezes, resgatando a média do tempo de execuções e o melhor *fitness* obtido. Para a função Rastrigin busca-se o mínimo global da função. Sendo assim, os *fitness* mais próximos de 0 são os melhores.

6.1 PSO

Na Figura 9 temos os tempos de execução obtidos pelo algoritmo em CPU e GPU. Já na Figura 10 temos os *fitness* obtidos em CPU e GPU. Podemos observar que o tempo de execução cresceu linearmente a medida que foram inseridos mais elementos no enxame. Proporcionalmente também foi obtido um melhor *fitness* devido a uma melhor cobertura do espaço de solução.

O tempo de execução em GPU está representado na Figura 9. Podemos perceber que para um número menor de partículas temos um desempenho inferior em relação a implementação com CPU. A perda de desempenho dá-se pelo fato de que há uma necessidade de transferência de dados entre CPU e GPU. Após a transferência dos dados, pode-se calcular vários *fitness* dos elementos em paralelo. Utilizando um número menor de elementos do enxame, o tempo de transferência de dados para a GPU é superior do que apenas calcular os *fitness* dos elementos em CPU. Porém, a partir de 5120 elementos do enxame, para este algoritmo, podemos ver que o tempo de execução em GPU já é levemente superior se comparado com CPU. A medida em que aumenta-se o número de elementos do enxame, a diferença no tempo de execução entre as duas implementações

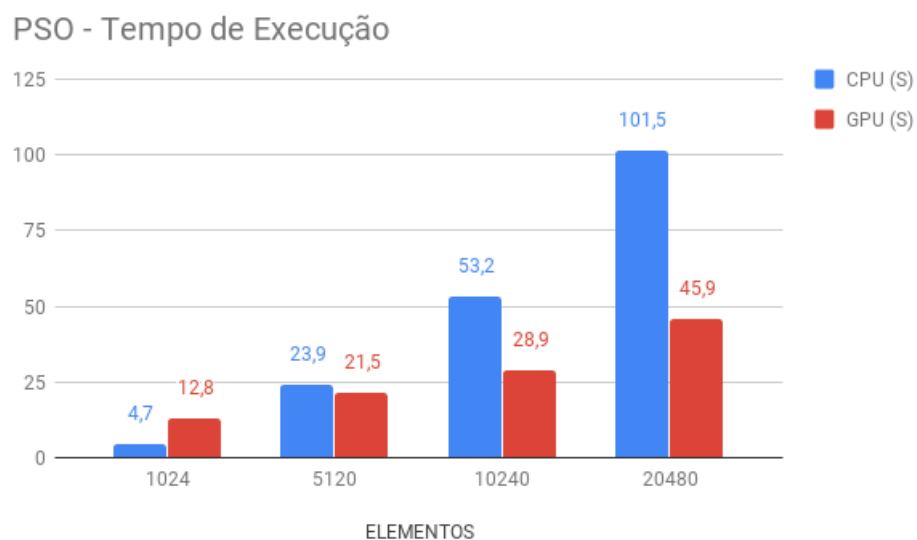


Figura 9 – Tempos de execução para o algoritmo PSO em CPU e GPU

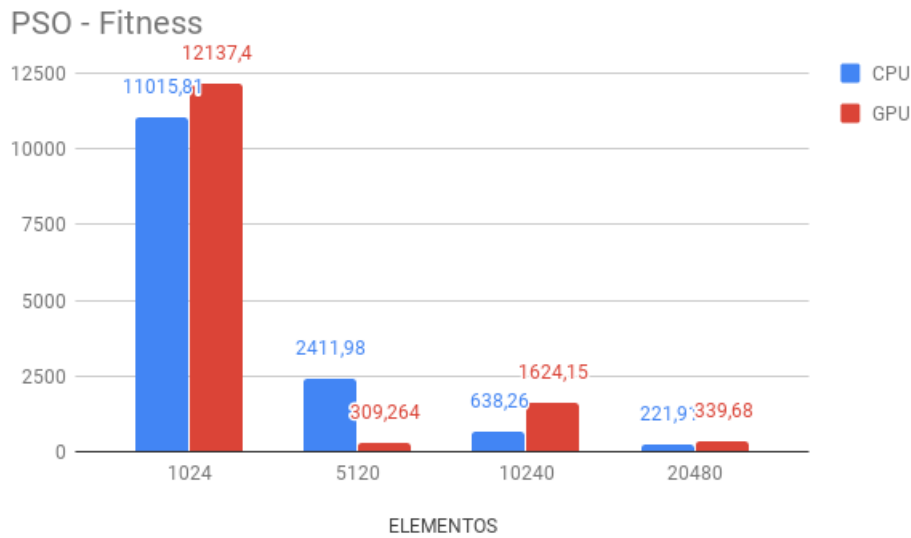


Figura 10 – Fitness obtidos pelo algoritmo PSO em CPU e GPU

aumenta, sendo a implementação em GPU mais rápida. Foi obtido um *speed-up* máximo de 55% para o enxame com 20480 elementos.

Já na Figura 10 podemos observar que o *fitness* obtido pelo enxame com 5120 elementos é melhor do os resultados obtidos pelos enxames com um número maior de elementos. Isso ocorre devido ao fato de que uma partícula do enxame pode ser criada próxima ao ponto ótimo da função. Quando aumentamos o número de partículas do enxame também aumentamos a chance disso acontecer.

6.2 ABC

O algoritmo ABC, com um número alto de elementos de enxame utilizados no trabalho, demonstrou-se que escala de maneira péssima, visto que a cada fase do algoritmo, as abelhas executam diversas ações, como a avaliação e criação de novas fontes de alimento, transformação de tipos de abelhas. Assim como cada abelha tem diversas tarefas atribuídas, quando se aumenta o número de elementos, acarreta-se um aumento drástico do tempo de execução. Porém, conseguiu-se um valor bom de *fitness*, escapando das soluções ótimas locais, como podemos ver nas Figuras 12 e 12. Com a implementação paralela em GPU, conseguiu-se diminuir um pouco o tempo de execução do algoritmo, como podemos ver na Figura 11, com a paralelização da inicialização das abelhas.

6.3 MFO

Na Figura 13, tem-se o tempo de execução do MFO em CPU. Este algoritmo obteve o tempo de execução mais baixo entre os algoritmos testados, superando até o PSO. Ao mesmo tempo, obteve os melhores resultados de *fitness* constantemente, em ambas

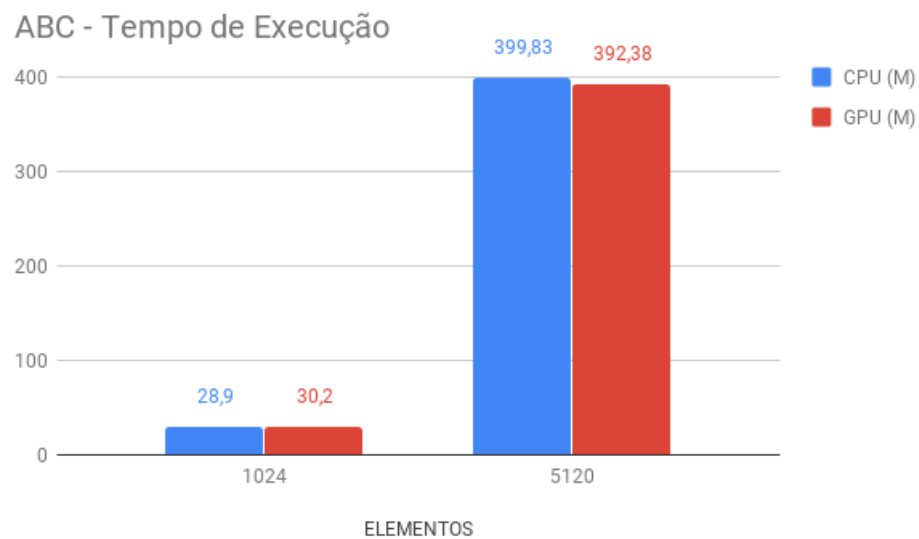


Figura 11 – Tempos de execução para o algoritmo ABC em CPU e GPU

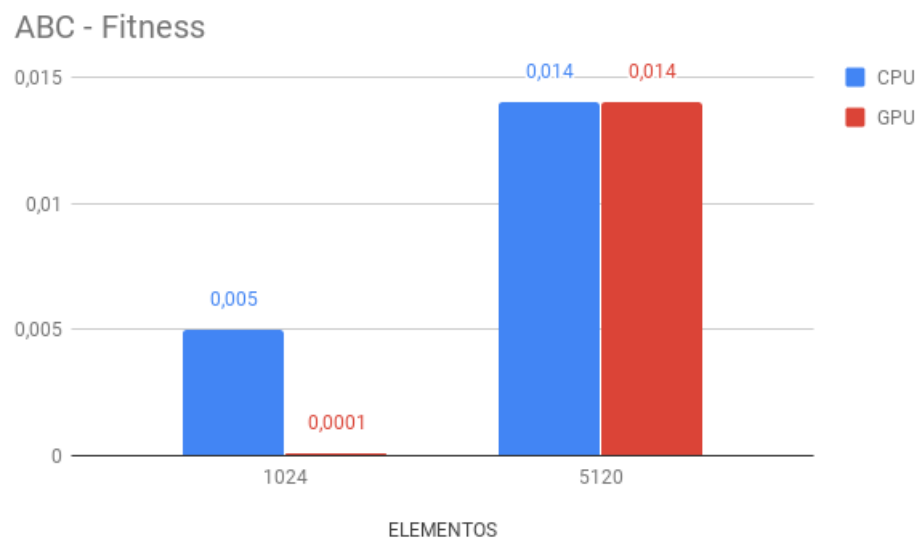


Figura 12 – Fitness obtido utilizando o algoritmo ABC em CPU e GPU

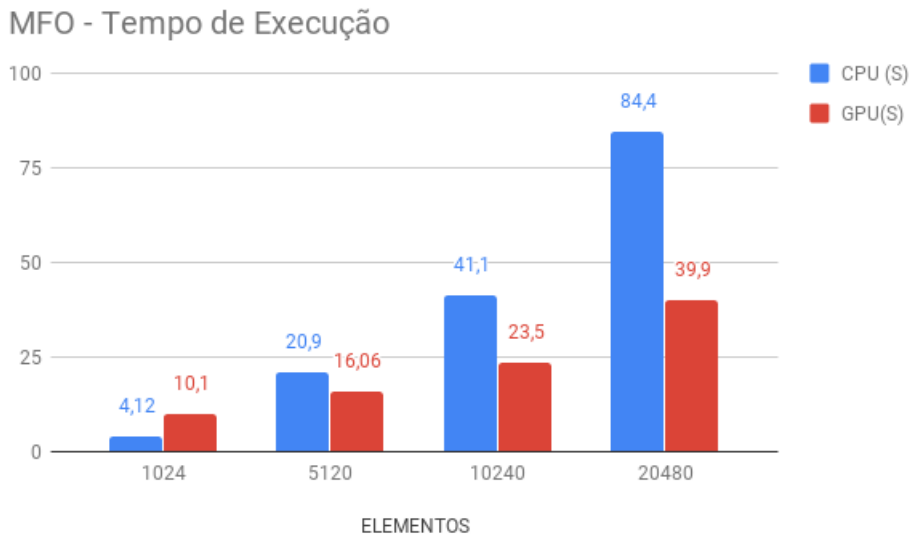


Figura 13 – Tempos de execução para o algoritmo MFO em CPU e GPU

implementações, como podemos ver na Figura 13. Já os outros algoritmos demonstraram certos comportamentos anômalos na questão de *fitness*, como por exemplo, a obtenção de um *fitness* melhor utilizando um número menor de partículas. Isso se dá ao fato dos algoritmos em si não conseguirem se livrar das soluções ótimas locais e depender de uma certa sorte na criação das partículas, esperando que algum elemento seja criado próximo ao ótimo global. Já no MFO, observa-se bons *fitness* em todos os testes conduzidos. Isso se dá pela maneira como as mariposas se aproximam das possíveis soluções, movendo-se em espirais. Assim elas não ficam presas às soluções ótimas locais. Pode-se observar na Figura 13 que o tempo de execução aumenta de forma linear. Utilizando da implementação paralela, consegue-se diminuir o tempo de inicialização das partículas como pode-se verificar na figura 13, onde obteve-se um *speed-up* substancial, sendo que a implementação em GPU tem um tempo de execução menor que a metade do tempo original e mantendo a qualidade das respostas.

6.4 AFSA

Os resultados de tempo de execução obtidos utilizando o AFSA, representado na Figura 15, nos mostram que, apesar de o tempo de execução não aumentar de forma totalmente exponencial a medida que o número de AFs aumentam, este algoritmo se torna inviável em aplicações que requerem um número maior de elementos de enxame, sendo apenas um pouco mais rápido que o ABC. Também não há uma boa qualidade de resposta, como pode ser visto nos *fitness* expressados na Figura 16, devido ao fato de que mesmo com o comportamento de *Leap* dos AFs, estes raramente chegam a utilizá-lo, e os AFs tendem a permanecer em formação de enxame. Mesmo quando o comportamento de

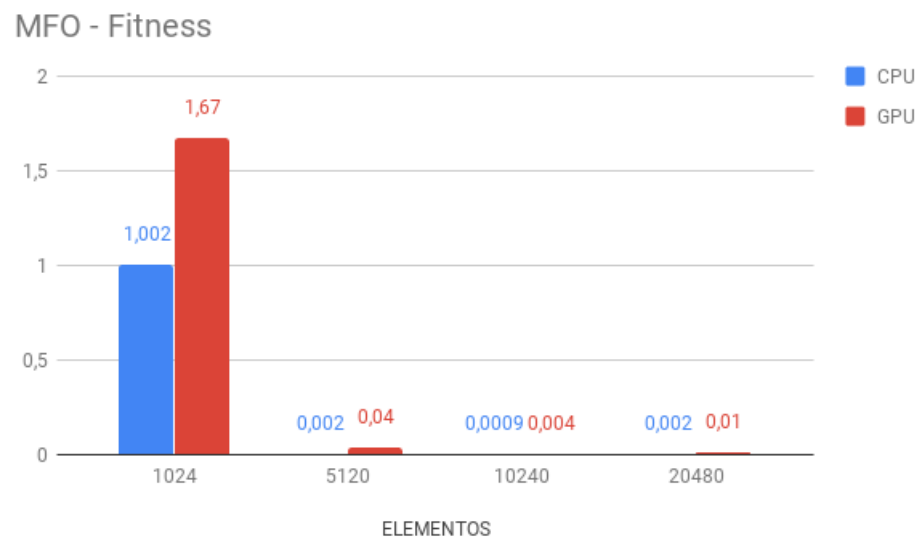


Figura 14 – Fitness obtido utilizando o algoritmo MFO em CPU e GPU

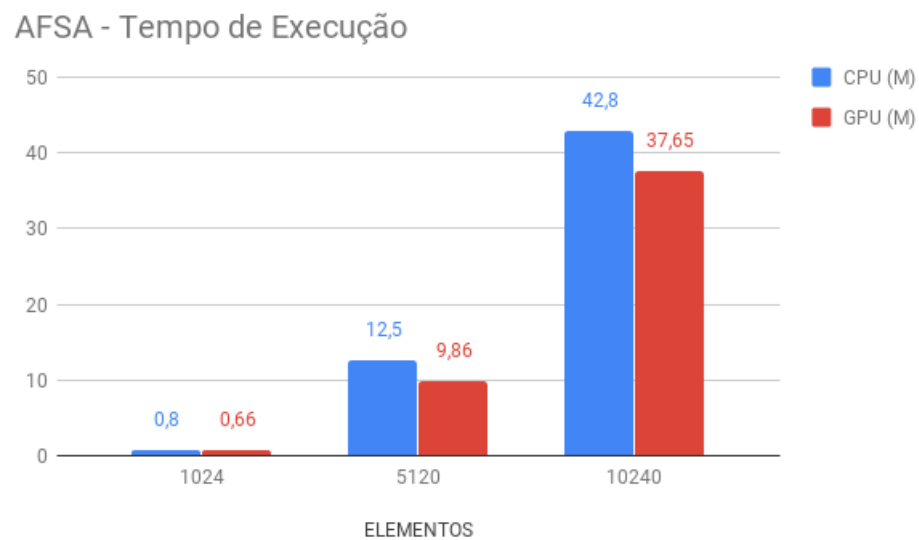


Figura 15 – Tempos de execução para o algoritmo AFSA em CPU e GPU

salto é ativado, ainda fica-se dependente da aleatoriedade de saltar para um local melhor que o anterior. Também nota-se a dependência de uma boa posição inicial de algum elemento do enxame para a obtenção de um bom resultado. Esse comportamento pode ser claramente observado na Figura 16. Utilizando a implementação paralela, é possível diminuir um pouco o tempo de execução, como pode-se observar na Figura 15.

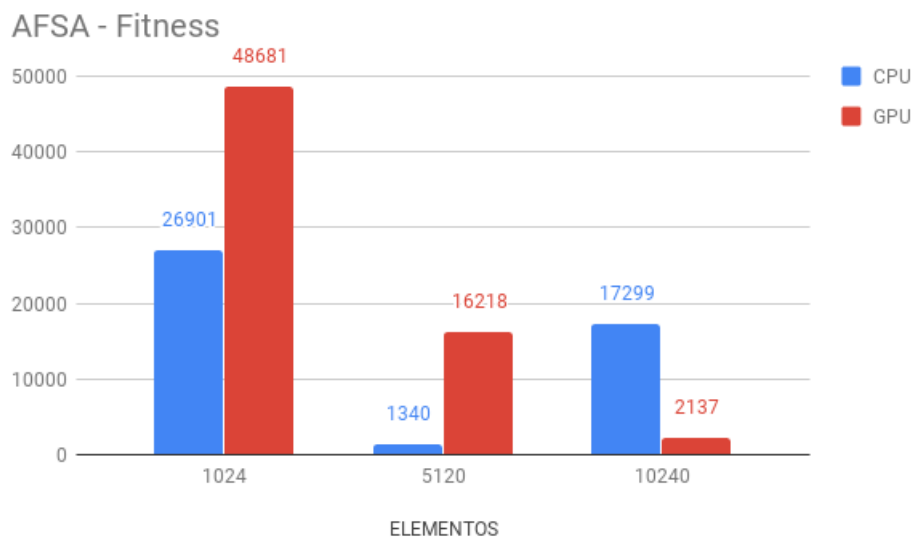


Figura 16 – Fitness obtido utilizando o algoritmo AFSA em CPU e GPU

6.5 Conclusão do Capítulo

Através dos resultados obtidos nesta seção, pode-se verificar certos pontos, como por exemplo, a dificuldade dos algoritmos que se movimentam em linhas retas, no espaço de solução, em encontrar um bom *fitness*. Estes ficam mais dependentes de um número maior de elementos do enxame.

Também podemos verificar que, apesar do PSO ser o algoritmo mais utilizado e influente na área de algoritmos de enxame, não conseguiu-se um bom resultado utilizando o mesmo. Surpreendentemente, o MFO, que é um algoritmo mais recente e menos conhecido, obteve um bom *fitness* e um bom tempo de execução.

7 CONSIDERAÇÕES FINAIS

Com a implementação ingênua de paralelismo em GPU implementada neste trabalho, é possível a fácil integração deste bloco de códigos a algoritmos de enxame ou outros algoritmos, que utilizam-se de diversos elementos dispostos no espaço de solução do problema com um *fitness* específico. Nas implementações, obteve-se um bom *speed-up* na maioria dos algoritmos, com alguns algoritmos alcançando um ganho de desempenho considerável, levando em consideração que é uma abordagem híbrida, que pode ser facilmente acoplada a diversos algoritmos. A implementação em GPU levou metade do tempo se comparado com a implementação em CPU nos algoritmos MFO e PSO.

A implementação desenvolvida durante o trabalho pode ser aplicada a problemas que requerem um grande número de elementos de enxame, como os problemas abordados em (DALI; BOUAMAMA, 2015) e (OUYANG et al., 2015), acoplando a algoritmos compatíveis já existentes.

Na literatura há alguns trabalhos recentes importantes como em (TAN; DING, 2016). Ainda tem-se diversos pontos a serem explorados que não foram abordados neste trabalho, como por exemplo:

- Impacto no desempenho causado por variáveis dos algoritmos (coeficientes de convergência no PSO, chances de salto no AFSA e taxa de abelhas empregadas no ABC).
- Diferentes tipos de implementação de paralelismo em GPU, principalmente o paralelismo somente em GPU, já que neste trabalho verificamos que o tempo de transferência entre CPU e GPU é alto e acaba prejudicando o desempenho para enxames que não utilizam um número alto de elementos.
- Avaliação da implementação paralela ingênua em GPU desenvolvida no trabalho para outros algoritmos, podendo ser adaptada para o uso em algoritmos como, por exemplo, o algoritmo genético, ou novos algoritmos de enxame Bacterial Foraging Optimization (BFO), FWA e Whale Swarm Algorithm (WSA).

Ainda existem pontos a serem pesquisados na área de algoritmos de enxame, sendo as implementações em GPU algo pouco explorado, e algo promissor, se implementados de maneira a minimizar a transferência de dados para a GPU.

Também fica disponível para a utilização e melhoramento os códigos desenvolvidos durante o trabalho em (MENEZES, 2019).

REFERÊNCIAS

- ALJARAH, I.; LUDWIG, S. A. Parallel particle swarm optimization clustering algorithm based on mapreduce methodology. In: IEEE. **Nature and biologically inspired computing (NaBIC), 2012 fourth world congress on.** [S.l.], 2012. p. 104–111. Citado na página 24.
- AZIZI, R. Empirical study of artificial fish swarm algorithm. **arXiv preprint arXiv:1405.4138**, 2014. Citado 2 vezes nas páginas 9 e 30.
- BAI, W.; EKE, I.; LEE, K. Y. An improved artificial bee colony optimization algorithm based on orthogonal learning for optimal power flow problem. **Control Engineering Practice**, Elsevier, v. 61, p. 163–172, 2017. Citado na página 27.
- BELL, J. E.; MCMULLEN, P. R. Ant colony optimization techniques for the vehicle routing problem. **Advanced engineering informatics**, Elsevier, v. 18, n. 1, p. 41–48, 2004. Citado 2 vezes nas páginas 5 e 7.
- CHANG, J.-F. et al. A parallel particle swarm optimization algorithm with communication strategies. 2005. Citado na página 39.
- CHEN, Y. et al. Particle swarm optimizer with two differential mutation. **Applied Soft Computing**, Elsevier, v. 61, p. 314–330, 2017. Citado na página 24.
- DALI, N.; BOUAMAMA, S. Gpu-pso: parallel particle swarm optimization approaches on graphical processing unit for constraint reasoning: case of max-csps. **Procedia Computer Science**, Elsevier, v. 60, p. 1070–1080, 2015. Citado 2 vezes nas páginas 38 e 51.
- EBERHART, R.; KENNEDY, J. A new optimizer using particle swarm theory. In: IEEE. **Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on.** [S.l.], 1995. p. 39–43. Citado 2 vezes nas páginas 24 e 39.
- FARIS, H. et al. Evolopy: An open-source nature-inspired optimization framework in python. In: **IJCCI (ECTA).** [S.l.: s.n.], 2016. p. 171–177. Citado na página 41.
- KARABOGA, D. **An idea based on honey bee swarm for numerical optimization.** [S.l.], 2005. Citado 2 vezes nas páginas 9 e 26.
- KARABOGA, D.; BASTURK, B. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. **Journal of global optimization**, Springer, v. 39, n. 3, p. 459–471, 2007. Citado na página 25.
- KENNEDY, J. Particle swarm optimization. In: **Encyclopedia of machine learning.** [S.l.]: Springer, 2011. p. 760–766. Citado na página 24.
- Klöckner, A. et al. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. **Parallel Computing**, v. 38, n. 3, p. 157–174, 2012. ISSN 0167-8191. Citado na página 41.
- LI, X. A new intelligent optimization-artificial fish swarm algorithm. **Doctor thesis, Zhejiang University of Zhejiang, China**, 2003. Citado na página 27.

- MENEZES, A. **GPUSwarm**. [S.l.]: GitHub, 2019. <<https://github.com/AngeloGiovanniMenezes/GPUSwarm>>. Citado 2 vezes nas páginas 41 e 51.
- MIRJALILI, S. Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm. **Knowledge-Based Systems**, v. 89, p. 228 – 249, 2015. ISSN 0950-7051. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0950705115002580>>. Citado 3 vezes nas páginas 9, 30 e 31.
- MOHAMED, A.-A. A. et al. Optimal power flow using moth swarm algorithm. **Electric Power Systems Research**, Elsevier, v. 142, p. 190–206, 2017. Citado 2 vezes nas páginas 5 e 7.
- MUSSI, L.; DAOLIO, F.; CAGNONI, S. Evaluation of parallel particle swarm optimization algorithms within the cudaTM architecture. **Information Sciences**, Elsevier, v. 181, n. 20, p. 4642–4657, 2011. Citado na página 38.
- NESHAT, M. et al. Artificial fish swarm algorithm: a survey of the state-of-the-art, hybridization, combinatorial and indicative applications. **Artificial intelligence review**, Springer, v. 42, n. 4, p. 965–997, 2014. Citado na página 28.
- OANCEA, B. **Parallel computing in economics - an overview of the software frameworks**. 2014. Citado 2 vezes nas páginas 9 e 19.
- OUYANG, A. et al. Parallel hybrid pso with cuda for ld heat conduction equation. **Computers & Fluids**, Elsevier, v. 110, p. 198–210, 2015. Citado 2 vezes nas páginas 38 e 51.
- POLI, R.; KENNEDY, J.; BLACKWELL, T. Particle swarm optimization. **Swarm intelligence**, Springer, v. 1, n. 1, p. 33–57, 2007. Citado na página 24.
- SERAPIÃO, A. B. d. S. Fundamentos de otimização por inteligência de enxames: uma visão geral. **Sba: Controle & Automação Sociedade Brasileira de Automatica**, SciELO Brasil, v. 20, n. 3, p. 271–304, 2009. Citado na página 23.
- SHEN, W. et al. Forecasting stock indices using radial basis function neural networks optimized by artificial fish swarm algorithm. **Knowledge-Based Systems**, Elsevier, v. 24, n. 3, p. 378–385, 2011. Citado 3 vezes nas páginas 5, 7 e 19.
- SHI, Y. Particle swarm optimization. **IEEE connections**, v. 2, n. 1, p. 8–13, 2004. Citado na página 24.
- SHI, Y. et al. Particle swarm optimization: developments, applications and resources. In: IEEE. **evolutionary computation, 2001. Proceedings of the 2001 Congress on**. [S.l.], 2001. v. 1, p. 81–86. Citado na página 24.
- SKINDEROWICZ, R. The gpu-based parallel ant colony system. **Journal of Parallel and Distributed Computing**, Elsevier, v. 98, p. 48–60, 2016. Citado na página 38.
- TALLENT, N. R.; MELLOR-CRUMMEY, J. M. Effective performance measurement and analysis of multithreaded applications. In: ACM. **ACM Sigplan Notices**. [S.l.], 2009. v. 44, n. 4, p. 229–240. Citado na página 19.

TAN, Y.; DING, K. A survey on gpu-based implementation of swarm intelligence algorithms. **IEEE transactions on cybernetics**, IEEE, v. 46, n. 9, p. 2028–2041, 2016. Citado 8 vezes nas páginas 9, 19, 23, 33, 34, 35, 38 e 51.

TAN, Y.; ZHU, Y. Fireworks algorithm for optimization. In: SPRINGER. **International Conference in Swarm Intelligence**. [S.l.], 2010. p. 355–364. Citado na página 34.

WOLPERT, D. H.; MACREADY, W. G. No free lunch theorems for optimization. **IEEE transactions on evolutionary computation**, IEEE, v. 1, n. 1, p. 67–82, 1997. Citado 2 vezes nas páginas 20 e 23.

ZHOU, Y.; TAN, Y. Gpu-based parallel particle swarm optimization. In: IEEE. **Evolutionary Computation, 2009. CEC'09. IEEE Congress on**. [S.l.], 2009. p. 1493–1500. Citado na página 39.