UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ENGENHARIA QUÍMICA

ENG07053 - TRABALHO DE DIPLOMAÇÃO EM ENGENHARIA QUÍMICA

Paralelização do PSO em CUDA aplicada à estimação de parâmetros do modelo FOWM

Autor: Arthur Eckert Rüdiger

Orientador: Marcelo Farenzena

Porto Alegre, maio de 22

SUMÁRIO

Agradecimentos

Gostaria de começar agradecendo a todos os autores envolvidos nos trabalhos aqui citados, sem a dedicação de vocês este trabalho não seria possível.

Ao meu orientador, Marcelo Farenzena que se disponibilizou dia e noite a me guiar, ajudando a compor e refinar este trabalho acadêmico.

Aos mês pais e familiares que serviram de grande suporte principalmente nesta hora de necessidade.

E por fim mas não menos importante aos meus amigos, pois os melhores momentos desta vida são os que compartilhamos.

Resumo

O crescimento do volume produzido de petróleo em território brasileiro tem aumentado nos últimos anos, sendo que uma parcela significativa dessa produção provém de fontes marítimas. Devido à imensa complexidade de medir escoamentos multifásicos, modelos matemáticos simplificados foram propostos para controle e monitoramento de plataformas *Offshore*. Entretanto estes modelos necessitam que seus parâmetros sejam estimados com base em dados de operação ou proveniente de simulador fenomenológico rigoroso. Tal estimação tem se provado longa, ao ponto que trabalhos da literatura previamente propostos apresentam tempos de estimação variando de 12 a 24h. Neste trabalho uma estimação de parâmetros do modelo proposto posto por DIEHL et al. (2017) via Enxame de Partículas (PSO) é proposta de forma paralela. A implementação é realizada em CUDA C, utilizando o imenso processamento paralelo proveniente de Unidades de Processamento Gráfico (GPU). Os parâmetros do PSO são avaliados com base em uma métrica proposta neste trabalho que combina robustez e desempenho de modo a avaliar múltiplos tamanhos de enxame, topologias, escolhas de parâmetros como coeficentes de coerção e nostalgia. Os resultados obtidos foram comparados com uma versão sequencial proposta em C e com dois trabalhos da literatura sendo capaz de estimar os parâmetros em questão de minutos, representando uma redução no tempo de estimação de até 99,84%.

Palavras-chave: PSO, CUDA, FOWM, Estimação de Parâmetros

Lista de Figuras

Lista de Tabelas

LISTA DE SÍMBOLOS

α_{gr}	Fração de gás no *riser*
α_{lr}	Fração de líquido no *riser*
α_{gw}	Fração de gás no reservatório
α_{gt}	Fração de gás no *tubing*
Θ	Inclinação média do *riser*
ρ_l	Massa específica da fase líquida
ρ_{ai}	Massa específica do gás no anular
ρ_{mt}	Massa específica da mistura no *tubing*
ρ_{mres}	Massa específica da mistura no reservatório
ρ_{gt}	Massa específica do gás no *tubing*
χ	Coeficiente de constrição
ω_u	Fator de correção do volume
A_{ss}	Área da seção transversal da tubulação submersa
c_1	Coeficiente de nostalgia
c_2	Coeficiente de coerção
C_g	Coeficiente de vazão da válvula virtual
C_{out}	Coeficiente de vazão da válvula *choke*
D_a	Diâmetro do anular
D_{ss}	Diâmetro da tubulação submersa
D_t	Diâmetro do *tubing*
E	Fração de gás que entra diretamente no *riser*
E(x)	Média de x
F_{obj}	Função objetivo
g	Aceleração da gravidade
g_{best}	Melhor valor de F_{obj} lembrado pela enxame
h	Passo de integração
H_{vgl}	Distância vertical entre a árvore de natal e a válvula de *gas lift*

H_t	Distância vertical entre a árvore de natal e o reservatório
H_{pdg}	Distância vertical entre a árvore de natal e o sensor PDG
K_a	Coeficiente de vazão do anular para o *tubing*
K_r	Coeficiente de vazão no reservatório
K_w	Coeficiente de vazão da árvore de natal
l_{best}	Melhor valor de F_{obj} lembrado pela vizinhança
L_a	Comprimento do anular
L_{fl}	Comprimento da tubulação submersa
L_t	Comprimento do *tubing*
M	Massa molar do gás
m_{ga}	Massa de gás no anular
m_{gb}	Massa de gás na bolha
m_{gt}	Massa de gás no *tubing*
$m_{L,still}$	Massa mínima de gás no *riser*
m_{lt}	Massa de líquido no *tubing*
N	Número de partículas por enxame
N_{pts}	Número de pontos experimentais fornecidos
p	Posição da partícula
p_{best}	Melhor valor de F_{obj} lembrado pela partícula
P_{ai}	Pressão no anular
P_{bh}	Pressão na saída do reservatório
P_{eb}	Pressão interna da bolha
P_{pdg}	Pressão no ponto PDG
P_r	Pressão na saída do reservatório
P_{rb}	Pressão na base do *riser*
P_{rt}	Pressão no topo do *riser*
Ps	Pressão de saída da válvula *choke*
P_{tb}	Pressão na base do *tubing*

P_{tt}	Pressão no topo do *tubing*
r	Número de vizinhos próximos que se comunicam com a partícula
R	Constante universal dos gases
r_{xy}	Coeficiente de Pearson entre x e y
s_x	Desvio padrão de x
T	Temperatura
t	Tempo
t_{OLGA}	Tempo do simulador OLGA
t_{parada}	Tempo de parada do integrador
v	Velocidade da partícula
V_a	Volume do anular
V_{eb}	Volume da bolha
V_{gt}	Volume de gás no *tubing*
V_{lr}	Volume de líquido no *riser*
V_{ss}	Volume da tubulação submersa
V_t	Volume do *tubing*
W_{gc}	Vazão mássica de *gas lift*
W_g	Vazão mássica na válvula virtual
W_{iv}	Vazão mássica de gás do anular para o *tubing*
W_r	Vazão mássica do reservatório para o *bottom hole*
W_{gout}	Vazão mássica de gás na saída da *choke*
W_{lout}	Vazão mássica de líquido na saída da *choke*
W_{whg}	Vazão mássica de gás na árvore de natal
W_{whl}	Vazão mássica de líquido na árvore de natal
x	Variável medida OLGA
y	Variável medida inferida pelo FOWM
z	Abertura da válvula *choke*

Lista de Abreviaturas e Siglas

CUDA	*Compute Unified Device Architecture*
CPU	*Central Processing Unit*
FOWM	*Fast Offshore Wells Model*
GPU	*Graphics Processing Unit*
PDG	*Pressure Downhole Gauge*
PSO	*Particle Swarm Optimization*

1 Introdução

A extração de petróleo tem se tornado cada vez mais significativa no âmbito da economia brasileira. Dados da Agência Nacional do Petróleo (ANP) mostram que em 2018 produziram-se cerca de 2,679 milhões de barils de petróleo por dia, sendo 96% dessa produção proveniente de operações marítimas (*Offshore)* (ANP, 2018). Em 2020 esse valor aumentou para 3,026 milhões (+12,95%) (ANP, 2021) o que torna estudos relacionados a aumento na produção e segurança da operação cada vez mais atrativos.

A operação de plantas de extração de petróleo *Offshore* apresenta um grande desafio à indústria. Além da dificuldade de medição de escoamentos multifásicos devido à sua complexidade, muitos sensores operam com pouca ou nenhuma manutenção devido ao difícil acesso (DIEHL *et al.*, 2017). Assim, uma opção é recorrer à simulação da operação para o monitoramento da planta. Alguns modelos rigorosos baseados em Equações Diferenciais Parciais foram propostos na literatura (BALIÑO, 2014),(BENDLKSEN *et al.*, 1991),(SINÈGRE; PETIT; MÉNÉGATTI, 2006). Entretanto, a simulação de tais modelos é computacionalmente muito cara para ser utilizada com estratégias de controle em tempo real (DIEHL *et al.*, 2017).

Como alternativa, alguns modelos simplificados considerando apenas balanços de massa de seções específicas da planta foram propostos (EIKREM; AAMO; FOSS, 2008), (JAHANSHAHI; SKOGESTAD, 2011), (MEGLIO; KAASA; PETIT, 2009). Estes modelos semi-fenomenológicos (também conhecidos por caixa cinza) permitem utilizar estratégias de controle em tempo real mas necessitam que seus parâmetros sejam estimados com base em simulação rigorosa ou dados experimentais (DIEHL *et al.*, 2017).

Proposto por DIEHL et al. (2017) o *Fast Offshore Wells Model (FOWM)* é um modelo simplificado que permite o monitoramento e controle em tempo real de plataformas de extração de petróleo *Offshore* em águas profundas e ultraprofundas. A proposta do modelo é servir como gêmeo digital (*digital twin*) da planta, podendo estimar dados não medidos como pressões e vazões e prever golfadas severas. Entretanto, o modelo conta com 9 parâmetros que precisam ser estimados com base em dados provenientes da operação da planta ou de simulador rigoroso.

No trabalho original os autores mencionaram a dificuldade de estimação dos parâmetros, sendo propostas equações para valores estimados dos parâmetros. Desde então os trabalhos de APIO (2017), HÜFFNER (2017) e RODRIGUES (2018) propuseram técnicas para facilitar a estimação dos parâmetros do modelo. Apesar disso, ainda são necessárias diversas horas para realizar tal tarefa.

Algumas variáveis consideradas constantes para a estimação dos parâmetros, como temperatura, pressão do reservatório, pressão de saída, vazão de *gas lift* e abertura da válvula *choke*, podem sofrer variações ao longo do tempo. A reestimação periódica dos parâmetros pode também colaborar para que o modelo se mantenha sempre fiel aos dados experimentais, mesmo que estes sofram alterações devido a fatores não contabilizados.

O trabalho aqui apresentado possui como principal objetivo a introdução, ajuste e implementação de um algoritmo capaz de estimar os parâmetros do modelo FOWM em um curto período de tempo, visando viabilizar a reestimação periódica dos mesmos. Para isso se fez uso do Enxame de Partículas (PSO), um algoritmo estocástico de otimização originalmente proposto por Kennedy e Eberhart (1995). Este foi implementado em CUDA C, um modelo de programação heterogêno originalmente proposto pela empresa Nvidia em 2006 que permite utilizar a Unidade de Processamento Gráfico (GPU) para computação de propósito geral.

Os seguintes objetivos específicos foram traçados para alcançar este fim:

- Propor implementação de duas funções objetivo: mínimos quadrados e a função proposta por Dieh et al. (2017). Estas implementações levam em consideração limitações de memória do sistema;
- Avaliar a influência do passo de integração do integrador Runge-Kutta de 4ª ordem explícito na qualidade dos ajustes;
- Propor e utilizar um critério de escolha para ajustar os parâmetros do PSO de modo a obter boa robustez e eficiência;
- Avaliar a escalabilidade do algoritmo;

Este trabalho está estruturado da seguinte maneira: este trouxe uma breve introdução, motivação e objetivos. No Capítulo 2 serão introduzidos os conceitos necessários para a compreensão deste trabalho bem como a revisão da literatura acerca do tema, no Capítulo 3 será abordada a metodologia, o Capítulo 4 apresentará os resultados e discução e por fim o Capítulo 5 apresentará as conclusões e proporá trabalhos futuros.

2 Revisão Bibliográfica

Este capítulo tem como objetivo introduzir o leitor a alguns conceitos chaves para a compreensão deste trabalho.

2.1 Algoritmos de otimização baseados em população

Algoritmos de otimização baseados em população mantêm registro de múltiplas possíveis soluções para o problema a ser otimizado. Cada algoritmo tem seus próprios mecanismos e heurísticas. Entretanto, o fator em comum é a utilização de múltiplas soluções e alguma forma de comunicação que permite aos indivíduos compartilhar informação de modo a guiar a população para as melhores soluções encontradas. Alguns dos exemplos mais conhecidos são: *Ant Colony Optimization*, *Artificial Bee Colony* e *Particle Swarm Optimization*. (BABALOLA; OJOKOH; ODILI, 2020)

2.2 Enxame de partículas (PSO)

Inicialmente proposto por Kenedy e Eberhart em 1995, o enxame de partículas é um algoritmo de otimização que imita o comportamento de pássaros à procura de alimento. Cada partícula representa uma possível solução se movendo pelo hiperespaço. (KENNEDY; EBERHART, 1995)

Em sua forma base, o PSO apresenta as seguintes etapas: inicialização, avaliação da Função Objetivo, atualização do melhor da partícula, atualização do melhor global (ou da vizinhança), atualização da velocidade e atualização da posição. (KENNEDY; EBERHART, 1995)

Durante a etapa de inicialização as partículas recebem posições aleatórias dentro do espaço de busca. Fontes na literatura sugerem que as velocidades sejam inicializadas para pequenos valores aleatórios (ENGELBRECHT, 2012). Isto evita que as partículas deixem o espaço de busca com muita frequência nas iterações iniciais. Por fim, as variáveis que monitoram o menor valor da função objetivo já encontrado pela partícula (p_{best}) e pelo enxame (g_{best}) ou vizinhança (l_{best}) são inicializados a valores altos (geralmente infinito ou valor máximo permitido pela precisão). O Algoritmo 1 apresenta a etapa de inicialização em pseudocódigo.

Algoritimo 1: Inicialização PSO Global

para $(i \leftarrow 0; i < N°partículas; i \leftarrow i + 1)$:
  para $(j \leftarrow 0; j < N°dimensões; j \leftarrow j + 1)$:
    $Inicializa\ posição : p_{i,j} \leftarrow pmin_j + (pmax_j - pmin_j) * rand[0,1]_{i,j}$
    $Inicializa\ velocidade : v_{i,j} \leftarrow 0,01 * rand[-1,1]_{i,j} * \left(\frac{(pmax_j - pmin_j)}{2}\right)$
  fim para
  $pbest_i \leftarrow \infty$
fim para
$gbest \leftarrow \infty$

onde $p_{i,j}$ e $v_{i,j}$ são cada componente da posição e velocidade de cada partícula, rand[a,b] representa um número aleatório (utiliza-se distribuição uniforme) entre dois valores arbitrários a e b, $pmin_j$ e $pmax_j$ são respectivamente os limites inferior e superior do espaço de busca.

As etapas seguintes são realizadas até o critério de parada ser atingido (i.e. número máximo de iterações, tolerância de erro ou estagnação). A etapa de avaliação da função objetivo (F_{obj}) utiliza as coordenadas (posição (p)) das partículas para calcular a função objetivo. Este valor por sua vez é comparado ao menor valor encontrado até então (denominado p_{best}) pela respectiva partícula, substitui o mesmo caso este seja menor e suas coordenadas são salvas (Θ_{pbest}). O Algoritmo 2 apresenta a etapa descrita em pseudocódigo.

Algoritimo 2: Avaliação da Função Objetivo

para $(i \leftarrow 0; i < N°partículas; i \leftarrow i + 1)$:
  Computa: $Fobj(p_i)$
  se $(Fobj(p_i) < pbest_i)$:
    para $(j \leftarrow 0; j < N°dimensões; j \leftarrow j + 1)$:
      $\theta pbest_{i,j} \leftarrow p_{i,j}$
    fim para
    $pbest_i = Fobj(p_i)$
  fim se
fim para

Na etapa de atualização do melhor global (g_{best}) ou local (l_{best}), informação é trocada entre as partículas. A forma com que as partículas interagem entre si é geralmente representada por gráficos de nodos conectados por linhas, e é denominado topologia (ou sociometria). Algumas topologias mais comuns descritas na literatura estão representadas na Figura 2.1.

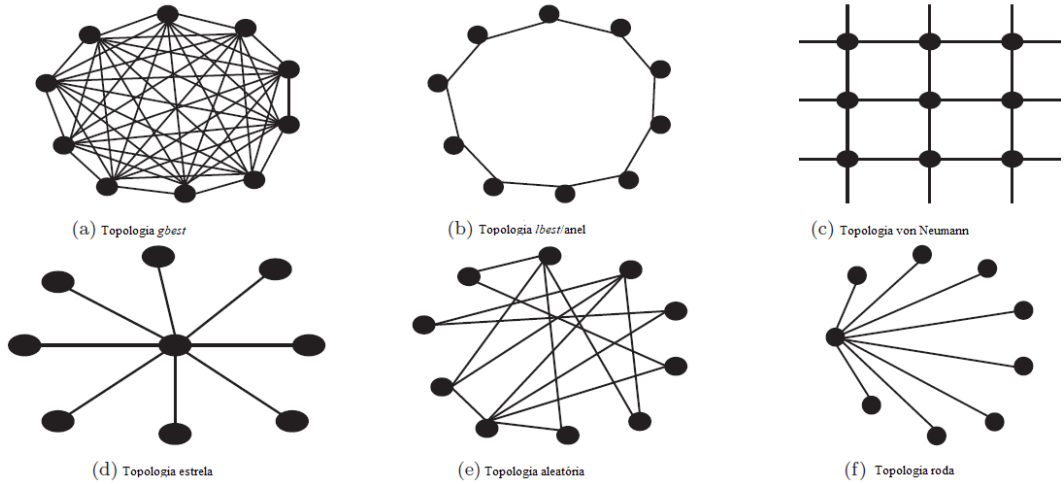

Figura 2.1 – Algumas das topologias mais comuns descritas na literatura (Figura adaptada de (HOUSSEIN *et al.*, 2021))

A escolha da topologia está diretamente relacionada à função objetivo a ser otimizada. Peng et al. (2022) demonstrou experimentalmente que em geral, topologias onde a informação é transmitida mais rapidamente (i.e mais conectadas) costumam apresentar melhores resultados para funções unimodais, enquanto topologias menos conectadas, devido à comunicação mais lenta, tendem a favorecer a saída das partículas de mínimos locais, sendo mais indicadas para funções multimodais.

Por fim ocorre a atualização da velocidade:

$$v_{(i,j)}^{t+1} = wv_{(i,j)}^{t} + c_1 r_{1(i,j)}^{t}\left(\theta p_{best(i,j)}^{t} - p_{(i,j)}^{t}\right) + c_2 r_{2(i,j)}^{t}\left(\theta g_{best(j)}^{t} - p_{(i,j)}^{t}\right) \quad (2.1)$$

e posição das partículas:

$$p_{(i,j)}^{t+1} = p_{(i,j)}^{t} + v_{(i,j)}^{t+1} \quad (2.2)$$

onde w é o coeficiente de inércia, c1 e c2 são os coeficientes de nostalgia e coerção respectivamente, os indices *i* e *j* representam a respectiva partícula e dimensão r_1 e r_2 são números aleatórios seguindo uma distribuição uniforme no intervalo (0,1] e Θ_{gbest} são as coordenadas da melhor partícula do enxame ou vizinhança.

Partículas eventualmente deixam o espaço de busca (i.e. $p_{(i,j)} > p_{maxj}$ ou $p_{(i,j)} < p_{minj}$), sendo que nesse caso a partícula é fixada no limite que excedeu e sua velocidade é reinicializada a uma fração da v_{max}, de modo a atraí-la para o centro do espaço de busca.

Clerc e Kennedy (2002) propuseram a utilização de um coeficiente de constrição (χ), de modo a alterar a equação da velocidade para:

$$\chi = \frac{2}{\left|2-C-\sqrt{C^2-4C}\right|} \tag{2.3}$$

$$v_{(i,j)}^{t+1} = \chi\left[v_{(i,j)}^{t} + c_1 r_{1(i,j)}^{t}\left(\theta p_{best(i,j)}^{t} - p_{(i,j)}^{t}\right) + c_2 r_{2(i,j)}^{t}\left(\theta g_{best(j)}^{t} - p_{(i,j)}^{t}\right)\right] \tag{2.4}$$

onde:

$$C = c_1 + c_2 \tag{2.5}$$

Tal modificação torna *w* um parâmetro dependente de c_1 e c_2. Isto permite uma análise mais direta de como a relação entre c_1 e c_2 altera o desempenho do algoritmo (sem ter que se preocupar com a perda de estabilidade). É importante mencionar que para algumas escolhas de parâmetros, as equações (2.1) e (2.4) são equivalentes.

O livro de Engelbrecht (2007) possui um capítulo dedicado ao PSO e suas variações. O leitor interessado pode encontrar mais informação sobre PSO e suas variações no trabalho de Houssein et al. (2021).

2.3 Arquitetura CPU/GPU

Unidades de Processamento Central *(Central Processing Units (CPUs))* possuem como principal função minimizar latência, ou seja, minimizar o tempo de execução de uma série de tarefas. Para fazer isso, cada núcleo da CPU possui uma grande quantidade de espaço ocupado por unidades de controle e memória. Isso permite que o chip troque rapidamente entre tarefas em execução, para evitar, por exemplo, esperas forçadas de acesso (NVIDIA, 2021).

Unidades de Processamento Gráfico *(Graphic Processing Units (GPUs))*, por outro lado, maximizam *througput*, ou seja, a capacidade de processamento (geralmente medido em FLOP/s (operações de ponto flutuante por segundo)). A arquitetura dos chips gráficos possui diversas unidades de processamento lógico aritmético (ALUs), com pouco espaço físico reservado para memória e controle (NVIDIA, 2021). Tal arquitetura permite a presença de muito mais núcleos para um mesmo espaço, o que tem se tornado atrativo para diversos ramos da computação científica. A Figura 2.2 ilustra ambas as arquiteturas.

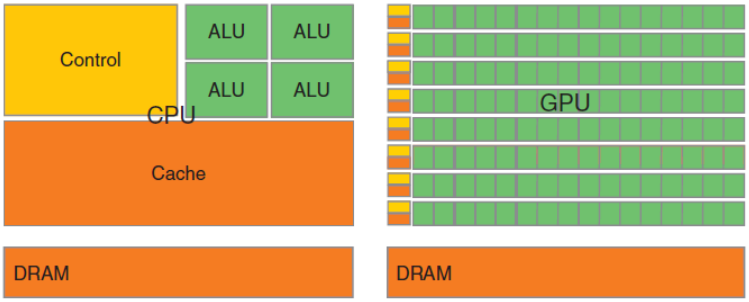

Figura 2.2 – Diferença entre arquiteturas da CPU e GPU (KIRK; HWU, 2013)

2.4 *Compute Unified Device Architecture* (CUDA)

Em novembro de 2006, o desenvolvedor de GPUs NVIDIA introduziu uma forma acessível e prática de utilizar GPUs para programação de propósito geral. Denominado CUDA (*Compute Unified Device Architecture*), a extensão da linguagem C permite utilizar o processamento de placas gráficas para resolução de problemas que se beneficiam de execução paralela. Isso permitiu que programadores pudessem utilizar o alto potencial de processamento paralelo para resolver problemas das mais diversas áreas (KIRK; HWU, 2013).

CUDA é um modelo de programação heterogênea que permite o uso conjunto da CPU e da GPU. Funções denominadas *kernels* são declaradas e chamadas pela CPU (denominada *Host*) e executadas pela GPU (ou GPUs) (denominada *Device*). A parcela executada pela GPU é compilada pelo NVCC, um compilador desenvolvido pela NVIDIA para tal propósito, enquanto a parcela executada pela CPU é executada pelo compilador tradicional. Esta abordagem permite que o programador opte por mover a parcela paralela do programa para a GPU enquanto a parcela sequencial é executada na CPU, otimizando o uso do *Hardware*.(KIRK; HWU, 2013)

A unidade base de execução é denominada *thread*. Cada *thread* corresponde a uma única execução do código escrito no *kernel*, e é responsável por executar uma parte do conjunto de dados de forma assíncrona na GPU. *Threads* são agrupados em blocos, que em conjunto formam uma grade (*grid*). Todos os *threads* em um mesmo bloco podem compartilhar dados de maneira eficiente por meio de memória de rápido acesso (*on chip*), além de poder usufruir de barreiras de sincronização que permitem que todos os *threads* no bloco esperem o término de uma determinada parcela do *kernel*. (KIRK; HWU, 2013)

O programador tem a liberdade de dimensionar a grade e os blocos, contanto que o número de *threads* por bloco (TPB) e o número de blocos por grade (BPG) não excedam o limite (1024 TPB no total e 65535 BPG para cada dimensão), além de poder optar por

utilizar blocos e grades de 1D, 2D ou 3D. A otimização do tamanho dos blocos é em geral uma tarefa complexa que leva em consideração aspectos do *Hardware*, o que foge do escopo deste trabalho. A Figura 2.3 apresenta a esquematização da hierarquia *thread*/bloco/grade, onde uma grade com 3x2 blocos de 4x3 *threads* cada foi ilustrada. (NVIDIA, 2021)

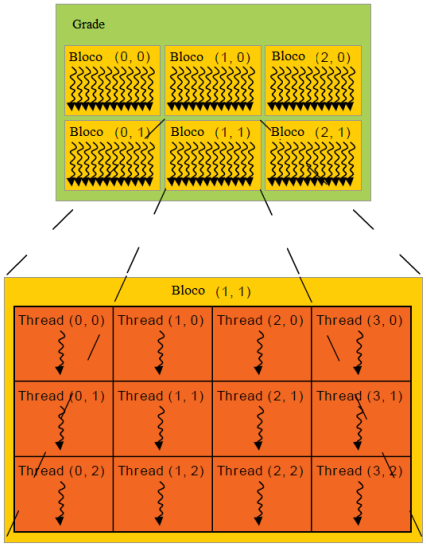

Figura 2.3 – Estrutura hierárquica de execução de dados (NVIDIA, 2021)

Simulação Fluido Dinâmica, Predição de Estrutura de Proteínas, Resolução de Equações Diferenciais Parciais (NVIDIA), Simulação de N corpos e Simulação de Corpos Rígidos (NVIDIA) são algumas das possíveis aplicações já demonstradas do CUDA. O leitor interessado pode acessar as implementações detalhadas dessas e mais aplicações no site oficial da NVIDIA, onde a coleção de 3 volumes *GPU Gems* se encontra disponível sem custo.

Outros usos do CUDA associados à engenharia química incluem estudos sobre cinética (BORISOV; KUDRYAVTSEV; SHERSHNEV, 2019), (SARSEMBAYEV *et al.*, 2020), simulação molecular (GOLDSWORTHY, 2014), (OZHGIBESOV *et al.*, 2013), (KAZACHENKO *et al.*, 2015), (KULKARNI; UMALE, 2016) e separação de fluidos viscoelásticos (YANG; SU; GUO, 2012) no qual os autores obtiveram uma execução 190x mais rápida quando comparada à equivalente na CPU utilizando apenas um núcleo.

Devido a sua natureza trivialmente paralela, o PSO sinergiza muito bem com CUDA. De fato, algumas implementações já foram sugeridas na literatura (NEDJAH; DE MORAES CALAZAN; DE MACEDO MOURELLE, 2016), (HUSSAIN; HATTORI; FUJIMOTO, 2016). Em sua implementação paralela do PSO, (HUSSAIN; HATTORI; FUJIMOTO, 2016) obtiveram redução de até 46x no tempo de execução quando

comparado a versão sequencial. Implementações mais de 100x mais rápidas para aplicações trivialmente paralelas foram reportadas na literatura (KIRK; HWU, 2013).

(NEDJAH; DE MORAES CALAZAN; DE MACEDO MOURELLE, 2016) propuseram a implementação do PSO em CUDA com três formas diferentes de paralelismo:

-Estratégia orientada à partícula (*Particle-oriented Strategy*) (PoS):
neste caso, cada *thread* recebe uma partícula e executa todo o programa em série. Ocorre sincronização na etapa de comunicação

-Estratégia orientada à dimensão (*Dimension-oriented Strategy*) (DoS):
nesta estratégia, cada partícula é descrita por um conjunto de *threads* (cada *thread* recebendo apenas uma dimensão da partícula). Tal estratégia elimina a utilização de *loops*, tornando toda a execução de cada iteração paralela. Entretanto é limitada para casos onde a função objetivo pode ser computada independentemente para cada dimensão.

-Estratégia orientada a cooperação (*Cooperation-oriented Strategy*) (CoS):
neste caso o problema base é subdividido em problemas com menor número de dimensões, otimizado para cada um destes subproblemas. Assim como o DoS, essa abordagem é recomedada para problemas com um maior número de dimensões.

2.5 Golfadas

Escoamentos múltifásicos são caracterizados por sua complexidade e padrões variados que dependem de múltiplos fatores, entre eles características fisico-químicas dos fluidos envolvidos, temperatura, pressão, velocidades superficiais de escoamento e orientação da tubulação (FALCONE, 2009). A Figura 2.4 apresenta alguns padrões de escoamento para tubulações verticais (APIO, 2017).

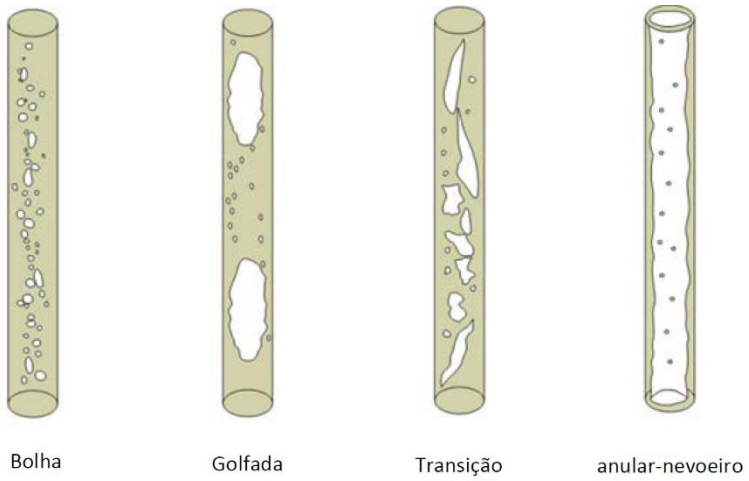

Figura 2.4 – Padrões de escoamento vertical (APIO, 2017).

Dentre os diversos padrões estudados na literatura, um dos importantes no âmbito de controle de operação são as Golfadas (HU, 2004). Golfadas são caracterizadas por um escoamento intermitente onde bolhas de gás do diâmetro da tubulação escoam de forma alternada com a fase líquida (FALCONE, 2009). O resultado se apresenta sob forma de cíclo limite estável, ou seja, as varíaveis não alcançam o estado estacionário e atingem um padrão ocilatório. Dependendo da amplitude das oscilações, essas variações podem causar danos a planta, podendo inclusive acarretar na parada da operação.

No sistema *gas lift*, segundo Hu (2004) as principais causas para golfadas, são:

2.5.1 Golfadas por Cabeceio Anular (Casing Heading)

Golfadas por Cabeceio Anular ocorrem quando a válvula de injeção de *gas lift* opera em região subsônica, nessa condição a taxa de injeção do gás depende da diferença de pressão entre o anular e o *tubing*. Duas condições são necessárias para a ocorrência deste fenômeno: O escoamento no *tubing* deve ser dominado pela gravidade e um grande volume de gás no anular (GEREVINI, 2017; HU, 2004). Assim, o cíclo é definido pelas seguintes etapas:

1. O gás escoa do anular para o *tubing*, reduzindo a pressão do *tubing* e acelerando a entrada de gás;
2. Com uma vazão insuficiente de injeção do gás, ocorre uma queda de pressão no anular;
3. Por consequência a pressão do casco se torna insuficiente para permitir o escoamento do gás para o *tubing*, acarretando no gradual aumento de pressão do casco;
4. Quando a pressão do casco se torna suficientemente alta o gás escoa para o *tubing* reiniciando o ciclo;

2.5.2 Golfadas por onda de Densidade (Density Wave)

Ocorre quando a válvula de injeção do sistema *gas lift* opera na região superssônica. Nessas condições o escoamento de gás para o *tubing* depende apenas da pressão do casco. Assim, pode se descrever o ciclo pelas etapas (APIO, 2017):

1. Inicialmente a pressão na saída do reservatório se encontra a valores muito baixos, resultando em um fluxo mínimo entre reservatório e *tubing*;
2. À medida que o gás é injetado, a pressão na saída do reservatório diminui gradualmente;
3. Quando a pressão do reservatório se torna maior que a pressão no fundo do poço o óleo passa a escoar, dominando a dinâmica de produção;

4. Eventualmente o escoamento do óleo cessa e o cíclo se repete;

2.5.3 Golfadas por Acidente (Terrain Riser)

Golfadas por Acidente são um fenômeno hidrodinâmico comum em plataformas *Offshore* que ocorrem pela obstrução do escoamento de gás no *riser* causado pela coluna de líquido. O gás se acumula na base do *riser*, aumentando gradualmente a pressão até que a mesma supere a pressão da coluna de líquido, de modo que todo o gás escoe até o topo do *riser*. Tal fenômeno gera comportamento oscilatório na planta, reduzindo a produção média e causando risco operacional aos equipamentos. A Figura 2.5 ilustra as fases que compõem o fenômeno de golfadas (DIEHL *et al.*, 2017).

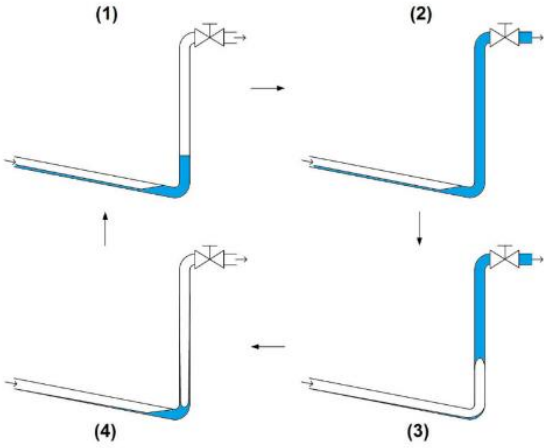

Figura 2.5 – Ilustração do comportamento cíclico de escoamento em um *riser* onde ocorrem golfadas (BILTOFT *et al.*, 2013).

O líquido acumula na base do *riser* (1). À medida que mais líquido e gás entram no sistema a presão aumenta gradualmente e o *riser* é preenchido com líquido (2). A pressão do gás preso pela coluna de líquido aumenta até que esta seja capaz de arrastar o líquido do *riser* para fora do sistema (3). Após o arraste o líquido se acumula na base do *riser* e o ciclo começa novamente (4) (BILTOFT *et al.*, 2013).

3 Materiais e Métodos

Neste capítulo será apresentado o caso de estudo (3.1), as funções objetivo propostas para análise do mesmo junto com os algoritmos usados para cálculo das mesmas (3.2), os experimentos realizados e *hardware* utilizado (3.3)

3.1 Caso de Estudo: Fast Offshore Wells Model (FOWM)

Proposto por Diehl et al. (2017), o *Fast Offshore Wells Model* (FOWM) incorpora modelos anteriormente propostos na literatura de modo a obter uma descrição completa da etapa de extração de petróleo em plataformas *Offshore* de águas profundas e ultraprofundas, cuja batimetria encontra-se ilustrada na Figura 3.1. O modelo se propõe a estimar varíaveis não medidas e prever a ocorrência de golfadas provenientes dos mecanismos descritos na seção 2.5. FOWM consiste de um sistema de seis equações diferenciais ordinárias (EDOs) e 9 parâmetros, considerando apenas os balanços de massa descritos pelas equações (3.1) a (3.6):

$$\frac{dm_{ga}}{dt} = W_{gc} - W_{iv} \tag{3.1}$$

$$\frac{dm_{gt}}{dt} = W_r \alpha_{gw} + W_{iv} - W_{whg} \tag{3.2}$$

$$\frac{dm_{lt}}{dt} = W_r\left(1 - \alpha_{gw}\right) - W_{whl} \tag{3.3}$$

$$\frac{dm_{gb}}{dt} = (1 - E)W_{whg} - W_g \tag{3.4}$$

$$\frac{dm_{gr}}{dt} = EW_{whg} + W_g - W_{gout} \tag{3.5}$$

$$\frac{dm_{lr}}{dt} = W_{whl} - W_{lout} \tag{3.6}$$

onde m_{ga} é a massa no anular, m_{gt} e m_{lt} são as massas de gás e líquido no *tubing* respectivamente m_{gb} é a massa de gás na bolha, m_{gr} e m_{lr} são as massas de gás e de líquido na linha de elevação (*riser*) respectivamente. *E* é a fração de gás que passa a bolha e α_{gw} é a fração mássica de gás nas condições de pressão e temperatura do reservatório. W_{gc} é a vazão de *gás lift* injetada no sistema, as demais vazões mássicas são descritas pelas equações (3.7) a (3.13):

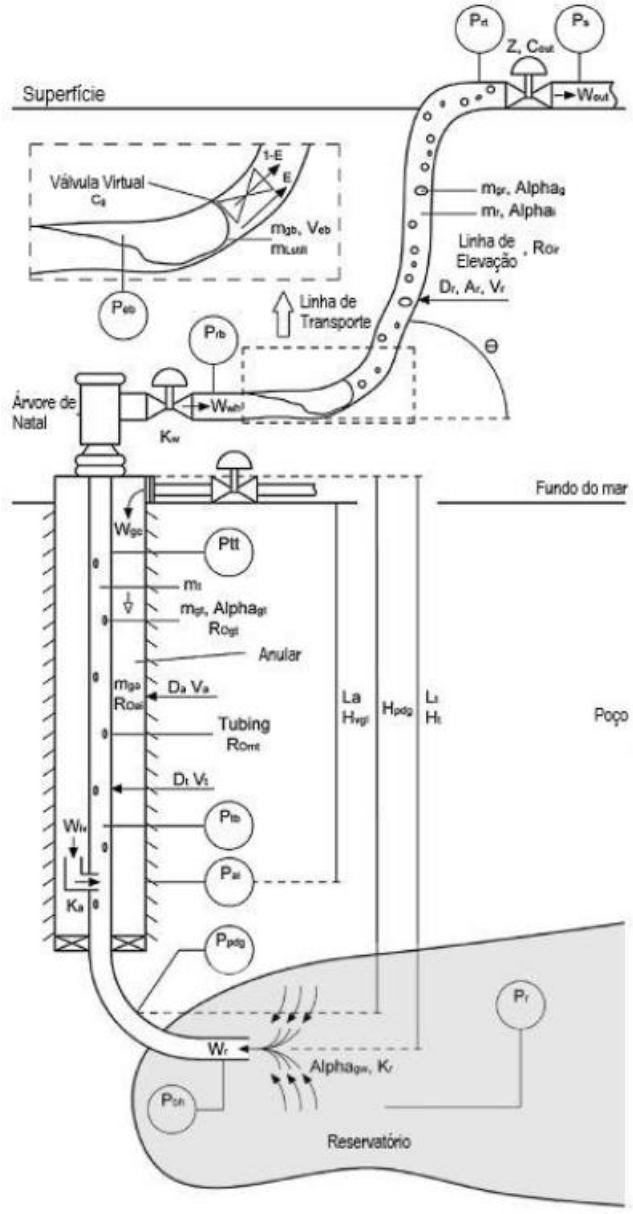

Figura 3.1 – Representação do Modelo FOWM (DIEHL *et al.*, 2017; KUCYK, 2021)

$$W_{iv} = K_a\sqrt{\rho_{ai}(P_{ai} - P_{tb})} \tag{3.7}$$

$$W_r = K_r[1 - 0{,}2(\frac{P_{bh}}{P_r}) - 0{,}8(\frac{P_{bh}}{P_r})^2] \tag{3.8}$$

$$W_{whg} = K_w\alpha_{gt}\sqrt{\rho_l(P_{tt} - P_{rb})} \tag{3.9}$$

$$W_{whl} = K_w(1 - \alpha_{gt})\sqrt{\rho_l(P_{tt} - P_{rb})} \tag{3.10}$$

$$W_g = C_g(P_{eb} - P_{rb}) \tag{3.11}$$

$$W_{gout} = \alpha_{gr}C_{out}z\sqrt{\rho_l(P_{rt} - P_s)} \tag{3.12}$$

$$W_{lout} = \alpha_{lr}C_{out}z\sqrt{\rho_l(P_{rt} - P_s)} \tag{3.13}$$

onde W_{iv} é a massa de gás que escoa do anular para o *tubing*, Wr é a vazão que sai do reservatório, W_{whg} e W_{whl} são respectivamente as vazões de gás e líquido na Árvore de Natal, W_g é a vazão da válvula virtual e W_{gout} e W_{lout} são as vazões de gás e líquido que passam pela válvula *choke.* A abertura da válvula *choke* é representada por z e ρ_l é a massa específica do líquido, considerada constante . K_a e K_w são os coeficientes de vazão entre o anular e o *tubing* e da Árvore de Natal. K_r é proporcional a produção do reservatório (DIEHL *et al.*, 2017).

A massa específica do gás no anular, ρ_{ai} é descrita pela equação (3.14). α_{gt} é a fração mássica de gás no *tubing*, enquanto α_{gr} e α_{lr} são as frações mássicas de gás e líquido na linha de elevação (descritos pelas equações 3.15-3.17). P_r e P_s são as pressões do reservatório e a jusante da *choke* e são consideradas constantes. As demais pressões estão descritas nas equações (3.18-3.25), são a pressão no anular (P_{ai}), *tubing* (P_{tb}), saída do reservatório (P_{bh}), posição PDG (P_{pdg}), topo do *tubing* (P_{tt}), à montante da bolha (P_{rb}), na bolha (P_{eb}) e no topo do *riser* (P_{rt}) respectivamente (DIEHL *et al.*, 2017).

$$\rho_{ai} = \frac{MP_{ai}}{RT} \tag{3.14}$$

$$\alpha_{gt} = \frac{m_{gt}}{m_{gt}+m_{lt}} \tag{3.15}$$

$$\alpha_{gr} = \frac{m_{gr}}{m_{gr}+m_{lr}} \tag{3.16}$$

$$\alpha_{lr} = 1 - \alpha_{gr} \tag{3.17}$$

$$P_{ai} = (\frac{RT}{V_a M} + \frac{gL_a}{V_a})m_{ga} \tag{3.18}$$

$$P_{tb} = P_{tt} + \rho_{mt} g H_{vgl} \tag{3.19}$$

$$P_{bh} = P_{pdg} + \rho_{mres} g (H_t - H_{pdg}) \tag{3.20}$$

$$P_{pdg} = P_{tb} + \rho_{mres} g (H_{pdg} - H_{vgl}) \tag{3.21}$$

$$P_{tt} = \frac{\rho_{gt} RT}{M} \tag{3.22}$$

$$P_{rb} = P_{rt} + \frac{(m_{lr}+m_{L,still}) g sen(\theta)}{A_{ss}} \tag{3.23}$$

$$P_{eb} = \frac{m_{gb} RT}{M V_{eb}} \tag{3.24}$$

$$P_{rt} = \frac{m_{gr} RT}{M(\omega_u V_{ss} - \frac{m_{lr}+m_{L,still}}{\rho_l})} \tag{3.25}$$

$$\rho_{mt} = \frac{m_{gt}+m_{lt}}{V_{gt}} \tag{3.26}$$

$$\rho_{gt} = \frac{m_{gt}}{V_{gt}} \tag{3.27}$$

$$V_{gt} = V_t - \frac{m_{lt}}{\rho_l} \tag{3.28}$$

$$A_{ss} = \frac{\pi D_{ss}^2}{4} \tag{3.29}$$

$$V_{ss} = \frac{\pi D_{ss}^2 L_r}{4} + \frac{\pi D_{ss}^2 L_{fl}}{4} \tag{3.30}$$

$$V_a = \frac{\pi D_a^2 L_a}{4} \tag{3.31}$$

$$V_t = \frac{\pi D_t^2 L_t}{4} \tag{3.32}$$

onde R é a constante universal dos gases, M a massa molar do gás, T a temperatura, g a aceleração gravitacional. V_a e L_a são o volume e comprimento do anular, ρ_{mt} e ρ_{gt} são as massas específicas da mistura e do gás no *tubing* respectivamente (calculadas por 3.26 e 3.27). ρ_{mres} é a massa específica da mistura no reservatório e é considerada constante. A distância vertical entre a Árvore de Natal e a válvula de *gás lift*, o transmissor do PDG e o a saída do reservatório são representados por H_{vgl}, H_{pdg} e H_t respectivamente. D_{ss}, D_t e D_a são os diâmetros da tubulação submersa, do *tubing* e do anular respectivamente (DIEHL et al., 2017). A_{ss} é a área transversal da tubulação submersa, Θ é a inclinação média do riser, $m_{L,still}$ é a massa mínima de líquido na tubulação submersa, V_{eb} é o volume da bolha. V_t e V_{gt} são o volume do *tubing* e o volume de gás no *tubing* respectivamente, L_t, L_r e L_{fl} são o comprimento do *tubing*, *riser* e tubulação no fundo do mar (*seabed*). V_{ss} e V_a são o volume da tubulação submersa e o volume do anular. Por fim ω_u é um parâmetro auxiliar do FOWM (DIEHL *et al.*, 2017).

No artigo original, os autores mencionaram a dificuldade de estimação dos parâmetros devido à presença de diversos mínimos locais, propondo inclusive uma função objetivo que penalize estes mínimos. O modelo consegue prever bem o comportamento oscilatório característico de golfadas severas, além de ter boa capacidade de extrapolação (DIEHL *et al.*, 2017).

Os parâmetros foram originalmente estimados utilizando o algoritmo Direct (FINKEL, 2003). Além disso, no artigo os autores propuseram equações para os valores iniciais. Desde então, os trabalhos de (HÜFFNER, 2017),(APIO, 2017) e (RODRIGUES, 2018) propuseram métodos para o ajuste de parâmetros do FOWM, em conjunto com técnicas específicas de ajustes de parâmetros desenvolvidas para modelos que contêm bifurcação de Hopf.

No trabalho atual, utilizou-se um espaço de busca baseado na ordem de grandeza parâmetros obtidos por Diehl *et al.* (2017) nos ajustes realizados para o Poço A, apresentados na Tabela 3.1, os Casos 1 à 3 representam o ajuste dos parâmetros do Poço A tomando como referência a P_{pdg}, P_{tt}, P_{rt} obtidos através de simulador rigoroso (Caso 1), apenas a P_{pdg} obtida através de simulador rigoroso (Caso 2) e utilizando dados de operação (Caso 3) (DIEHL *et al.*, 2017). O intervalo de busca estipulado para cada parâmetro se encontra na Tabela 3.2.

Tabela 3.1 – Parâmetros ajustados para o Poço A com três conjuntos de dados de referência (DIEHL *et al.*, 2017)

Parâmetro	Caso 1	Caso 2	Caso 3
$m_{L,still}$ (Kg)	4,96e2	1,96e3	6,22e1
C_g (m*s)	2,01e-4	2,05e-2	1,14e-3
C_{out} (m^2)	6,70e-3	1,97e-2	2,04e-3
V_{eb} (m^3)	1,15e-2	8,35e2	6,10e1
E (-)	4,04e-2	5,71e-1	1,54e-1
K_w (Kg/m)	1,34e-3	8,68e-4	6,88e-4
K_a (Kg/m)	1,82e-4	1,59e-4	2,29e-5
K_r (Kg/s)	2,58e2	1,31e2	1,27e2
ω_u (-)	1,00e0	7,65e0	2,78e0

Tabela 3.2 – Intervalo de busca dos parâmetros do FOWM

Parâmetro	Limite Inferior	Limite Superior
$m_{L,still}$ (Kg)	0	5000
C_g (m*s)	0	0,01
C_{out} (m^2)	0	0,01
V_{eb} (m^3)	0	500
E (-)	0	1
K_w (Kg/m)	0	0,01
K_a (Kg/m)	0	0,01
K_r (Kg/s)	0	1000
ω_u (-)	0	10

O PSO escolhido utiliza o coeficiente de constrição e as equações de movimento que englobam o mesmo (equações 2.3 a 2.5), mantendo a relação $C=c_1+c_2=4,1$. Consequentemente χ é mantido ao valor constante de 0,730. O uso desta versão permite avaliar a relação c_1/c_2 sem perda de estabilidade.

No trabalho atual, uma implementação paralela em CUDA do PSO aplicado à estimação de parâmetros do modelo FOWM é proposta. O modelo de paralelismo utilizado foi o orientado à partícula, ou seja, cada *thread* é responsável por uma única partícula. A escolha desta forma de paralelismo se dá devido a baixa dimensionalidade do problema e alta correlação dos parâmetros (RODRIGUES, 2018).

Os dados utilizados para realizar o ajuste dos parâmetros foram fornecidos pela Petrobras e correspondem às pressões P_{pdg}, P_{tt} e P_{rt}, provenientes da simulação de operação do Poço A (descrito por (DIEHL *et al.*, 2017)) em OLGA (BENDLKSEN *et al.*, 1991), um simulador industrial fenomenológico baseado em Equações Diferenciais Parciais (EDP), e são apresentados no Capítulo 4. A abertura da válvula *choke* e a vazão de *gas lift* foram mantidas constantes, nos valores de 22% e 80000 Sm³/dia respectivamente. Os valores dos parâmetros do poço A se encontram descritos na Tabela 3.3.

Tabela 3.3 – Especificações do Poço A (DIEHL *et al.*, 2017)

Parâmetro (unidade)	**Valor**
ρ_l (Kg/m³)	900
P_r (bar)	225
P_s (bar)	10
α_{gw}	0,0188
ρ_{mres} (Kg/m³)	892
M (kg/kmol)	18
T (K)	298
L_r (m)	1569
L_{ft} (m)	2928
L_t (m)	1639
L_a (m)	1118
H_t (m)	1279
H_{pdg} (m)	1117
H_{vgl} (m)	916
D_{ss} (m)	0,15
D_t (m)	0,15
D_a (m)	0,14

3.2 Função Objetivo

Neste trabalho duas funções objetivo foram utilizadas: Mínimos quadrados e a modificação dos Mínimos quadrados proposta por (DIEHL et al., 2017). Os mínimos quadrados foram aplicados a uma única pressão (P_{pdg}) e reflete casos onde poucas medidas se encontram disponíveis. A função proposta por Diehl et al. por outro lado foi utilizada para ajustar 3 pressões simultâneamente (P_{pdg}, P_{tt}, P_{rt}), e foi utilizada neste caso por penalizar soluções estáveis e ponderar bem o ajuste das pressões.

3.2.1 Mínimos quadrados

Na etapa inicial de ajuste dos parâmetros do PSO utilizou-se Mínimos quadrados aplicados à pressão P_{pdg}. Para tal, o sistema de EDOs é resolvido para cada partícula na GPU, via Runge Kutta de 4ª ordem explícito (KUTTA, 1901; RUNGE, 1895) (RK4) implementado pelo autor. Em cada iteração do RK4, o programa verifica se o tempo fornecido pelo conjunto de dados (t_{OLGA}) se encontra entre o tempo atual e o próximo ($t < t_{OLGA} < t+h$), realizando uma interpolação linear da pressão de modo a obter uma estimativa da mesma no tempo correspondente ao dos dados. Na sequência, o algoritmo computa o Erro Quadrado entre os valores de pressão predito pelo modelo caixa cinza e o rigoroso. Essa abordagem permite que o Erro Quadrado Médio seja computado enquanto a simulação do modelo ocorre. A escolha dessa abordagem oferece diversas vantagens quando comparada à alternativa (computar toda a simulação, salvar os dados e então computar o Erro Quadrado Médio):

- Permite utilizar memórias de leitura mais rápida *on chip* (registradores e memória compartilhada), o que não seria possível se todos os pontos da simulação fossem retidos;

- Para a alternativa, o algoritmo realizaria diversas leituras e escritas na memória global (*off chip*), o que além de ineficiente, acarreta em uma péssima escalabilidade (mesmo a memória global possui um limite, neste caso 4GB);

A representação da função objetivo em pseudocódigo se encontra no Algoritmo 3.

Algoritimo 3: Computando o Erro Quadrado

$Fobj \leftarrow 0$
$valor \leftarrow 0$
enquanto $(t <= t_{parada})$:
  se $(valor < N_{pts}$ **e** $t_{OLGA}[valor] <= t + h)$:
    $Computa : Ppdg\ no\ instante\ t$
    $Computa : RK4$
    $Computa : Ppdg\ no\ instante\ t + h$
    $Interpola : Ppdg\ entre\ t\ e\ t + h$
    $Fobj \leftarrow Fobj + (Ppdg - Ppdg_{OLGA}[valor])^2$
    $valor \leftarrow valor + 1$
  fim se
  $Computa : RK4$
fim enquanto
$Fobj \leftarrow Fobj \div N_{pts}$

onde t_{OLGA} e $Ppdg_{OLGA}$ são vetores que contém os valores de tempo e Ppdg respectivamente gerados pelo simulador OLGA, aos quais o modelo se ajusta, N_{pts} é o número total de pontos fornecidos e valor é uma variável auxiliar usada para acessar cada valor do conjunto de dados. Fobj é a função objetivo a ser calculada (Erro Quadrado) e RK4 é responsável por integrar o conjunto de EDOs para um único passo h.

A escolha do passo de integração adequado para ser utilizado no RK4 é de extrema importância. O passo de integração deve ser apenas pequeno o suficiente para que ocorra

convergência. De modo a obter o passo ideal de integração os parâmetros foram estimados com um passo pequeno (h=0,5 s). O FOWM foi então simulado variando o passo de integração para um bom conjunto de parâmetros obtidos. O passo foi então aumentado e os parâmetros foram re-ajustados de modo a verificar se o passo maior é adequado. O procedimento foi repetido até que não fosse mais possível obter resultados satisfatórios. Para esta etapa, utilizou-se os passos 0,5 s, 1 s, 5 s e 10 s.

O tempo de parada (t_{parada}) da simulação é outro parâmetro importante. No trabalho atual, todo o conjunto de dados disponível para a condição de operação supramencionada foi utilizado. Isto corresponde a um total de 1440 pontos experimentais resultando num intervalo de simulação de 86340 s. Se houver necessidade de reduzir ainda mais o tempo de execução do algoritmo aqui apresentado, o autor recomenda que um estudo avaliando o tempo mínimo de simulação para estimação dos parâmetros seja realizado.

3.2.2 Função Objetivo Proposta por Diehl et al. (2017)

O PSO ajustado foi também utilizado para estimar os parâmetros utilizando 3 pressões: P_{pdg}, P_{tt} e P_{rt}. Nesta etapa, a função objetivo proposta em (DIEHL *et al.*, 2017). foi utilizada.

$$F_{obj} = \sum_{j=1}^{M} \frac{1}{\omega_j} \left[\frac{1}{N} \sum_{i=1}^{N} \omega_i \left(y_{j,i} - x_{j,i} \right)^2 \right] \tag{3.33}$$

$$\omega_i = \left(\frac{x_{i,j} - E(x_j)}{s_{j,x}} \right)^2 \tag{3.34}$$

$$\omega_j = \left(\frac{r_{xy} - 1}{2} \right) \tag{3.35}$$

onde o índice j representa cada pressão (i.e. P_{pdg}, P_{tt} e P_{rt}), o índice i representa cada ponto, x são as pressões obtidas do simulador OLGA e y são as pressões estimadas pelo FOWM, E(x) e s_x são a média e o desvio padrão de cada pressão e r_{xy} é o coeficiente de Pearson.

Esta função demanda que algumas variáveis a mais sejam computadas, como a média, desvio padrão e coeficiente de Pearson. Considerando que nenhum valor da simulação é salvo na memória, é necessária uma maneira de computar tais valores *on the fly*. Computar a média é trivial, basta acumular a soma dos valores de cada pressão e posteriormente dividir pelo número total de pontos utilizado. O desvio padrão, entretanto não pode ser computado de maneira tradicional, uma vez que seria necessário conhecer a média *a priori*. Para computar o desvio padrão utilizou-se a equação 3.36.

$$s_x^2 = \frac{Npts\left(E(x^2) - E^2(x) \right)}{Npts - 1} \tag{3.36}$$

onde s_x é o desvio padrão, N_{pts} o número de pontos, $E(x^2)$ é a média dos quadrados e $E^2(x)$ o quadrado da média. Assim, basta acumular a soma do quadrado de cada pressão durante a etapa de simulação (de modo a obter $E(x^2)$) e posteriormente utilizar a média ao quadrado ($E^2(x)$) para calcular a variância. O coeficiente de Pearson pode ser calculado a partir da média e variância e da soma do produto ($x_{i,j}$,$y_{i,j}$) acumulado. O coeficiente de Pearson é definido como:

$$r_{xy} = \frac{\sum(x_i y_i) - Npts E(x) E(y)}{(Npts-1) s_x s_y} \tag{3.37}$$

onde r_{xy} é o coeficiente de Pearson e mede correlação entre os dados (1 indica correlação perfeita, -1 indica correlação negativa perfeita e 0 indica nenhuma correlação). A média e desvio padrão dos dados do OLGA (E(x) e s_x) são pré-computados na CPU, uma vez que são constantes.

O Algoritmo 4 computa a função objetivo proposta por Diehl et al..

Algoritimo 4: Fobj Diehl

$Fobj \leftarrow 0$
$valor \leftarrow 0$
para $(j \leftarrow 0; j < NVM; j \leftarrow j+1)$:
  $E(y)[j] \leftarrow 0$
  $s_y[j] \leftarrow 0$
  $r_{xy}[j] \leftarrow 0$
  $\omega_i[j] \leftarrow 0$
  $MSE[j] \leftarrow 0$
fim para
enquanto $(t <= t_{parada})$:
  se $(valor < N_{pts}$ e $t_{OLGA}[valor] <= t+h)$:
  para $(j \leftarrow 0; j < NVM; j \leftarrow j+1)$:
    $Computa : y[j]$ *no instante* t
  fim para
    $Computa : RK4$
  para $(j \leftarrow 0; j < NVM; j \leftarrow j+1)$:
    $Computa : y[j]$ *no instante* t
    $Interpola : y[j]$ *entre* t *e* $t+h$
  fim para
  para $(j \leftarrow 0; j < NVM; j \leftarrow j+1)$:
    $E(y)[j] \leftarrow E(y)[j] + y[j]$
    $s_y[j] \leftarrow s_y[j] + y[j]^2$
    $r_{xy}[j] \leftarrow r_{xy}[j] + y[j] * x[j][valor]$
    $\omega_i[j] \leftarrow \left(\frac{x[j][valor] - E(x)[j]}{s_x[j]}\right)^2$
    $MSE[j] \leftarrow \omega_i[j] * (y[j] - x[j][valor])^2$
  fim para
    $valor \leftarrow valor + 1$
  fim se
  $Computa : RK4$
fim enquanto
para $(j \leftarrow 0; j < NVM; j \leftarrow j+1)$:
  $E(y)[j] \leftarrow E(y)[j] \div N_{pts}$
  $s_y[j] \leftarrow s_y[j] \div N_{pts}$
  $s_y[j] \leftarrow \sqrt{\frac{N_{pts}(s_y[j] - E(y)^2[j])}{N_{pts}-1}}$
  $r_{xy}[j] \leftarrow \frac{(r_{xy}[j] - N_{pts}(E(y)[j] * E(x)[j]))}{(N_{pts}-1) s_x s_y}$
  $r_{xy}[j] \leftarrow \left(\frac{r_{xy}[j]+1}{2}\right)$
  $Fobj \leftarrow Fobj + \frac{MSE[j]}{r_{xy}[j] * N_{pts}}$
fim para

onde NVM é o número de variáveis medidas e MSE é utilizado para acumular o quadrado da soma da diferença entre as pressões fornecidas pelo simulador OLGA (denotados por x) e as pressões estimadas pelo FOWM (denotados por y), multiplicado pelos pesos ω_i calculados pela equação 3.34. O algoritmo completo escrito em CUDA se encontra no Apêndice A.

3.3 Experimentos Realizados

De modo a avaliar robustez e desempenho das configurações aqui estudadas, a seguinte métrica é proposta:

$$\mathrm{CE} = (\mathrm{FC} * \mathrm{Iterconv} + 2 * (1 - \mathrm{FC}) * \mathrm{MaxIter})/100 \qquad (3.38)$$

onde CE é o critério de escolha, FC a fração de simulações que atingiu o valor crítico, Iter_{conv} é o número de iterações necessário para que a média aritmética das simulações que alcançaram o mínimo global atinja o valor crítico e Max_{Iter} é o número máximo de iterações (neste estudo 10000). A escolha de tal métrica penaliza simulações que não atingem o valor crítico, ao ponto que se torna mais favorável ao algoritmo tal valor de forma tardia com mais frequência do que prematuramente com menos frequência.

Durante todos os experimentos, a seguinte condição inicial foi utilizada:

Tabela 3.4 – Condição inicial utilizada

$\mathbf{m_{ga}}$	2805.14932589 kg
$\mathbf{m_{gt}}$	2125.01254427 kg
$\mathbf{m_{lt}}$	9103.27649409 kg
$\mathbf{m_{gb}}$	7857.40023097 kg
$\mathbf{m_{gr}}$	1316.40535031 kg
$\mathbf{m_{lr}}$	36810.25978214 kg

Além disso, esperou-se 500 s para comparar ao primeiro ponto dos dados do OLGA (i.e $t_{OLGA}[0] = 500$ s), para garantir que a simulação do FOWM se encontre no ciclo limite estável. Dados sobre as condições iniciais do problema são de dificil obtenção, uma vez que as massas não podem ser diretamente medidas (HÜFFNER, 2017). As condições iniciais para este trabalho foram variadas de modo a obter valores que pertencessem ao ciclo limite da condição de operação aqui estudada.

O processo de otimização dos parâmetros do PSO passou por 3 etapas:

-Na primeira etapa, cinco tamanhos de enxame (N=64, N=128, N=256, N=512, N=1024) foram testados com 6 topologias (Global, Anel (r=2), Círculo (r=N/16, r=N/8 e

r=N/4) e Von Neumann (VN)). A topologia "Círculo" é a generalização do Anel, onde cada partícula se comunica com os "r" vizinhos mais próximos. Neste estudo optou-se por utilizar r como função de N para manter constante o número de iterações que esta topologia leva para que a informação se propague para todas as partículas.

-Na segunda etapa as 5 melhores configurações obtidas na etapa anterior (5 menores valores de CE) foram testadas com 5 configurações de parâmetros diferentes: c1=c2/3, c1=c2/2, c1=c2, c1=2*c2 e c1=3*c2.

-Por fim, de modo a guiar quanto à escalabilidade do algoritmo, a melhor configuração obtida na etapa anterior foi testada com 3 configurações diferentes. Nessa etapa, o número de partículas foi variado de 2048 a 16384. Três configurações diferentes foram testadas para avaliar a velocidade de convergência. As configurações foram:

-Enxames Individuais (EI) – O PSO foi executado N_{total}/N_{enxame} vezes em paralelo, onde N_{total} corresponde ao número total de partículas (i.e 2048 à 16384 partículas) e N_{enxame} corresponde ao tamanho de enxame utilizado, obtido das etapas anteriores, sem comunicação nenhuma entre os enxames;

-Migração (M) – O PSO foi executado com N_{total}/N_{enxame} vezes em paralelo, a cada 100 iterações ocorre migração entre os enxames. Durante a migração a pior partícula de um dado enxame (maior p_{best}) é substituída pela melhor partícula entre todos os enxames (menor p_{best});

-Enxame Único (EU) – O PSO foi executado 1 única vez com N_{total} partículas;

Nesta etapa, tanto o tempo quanto o número de iterações até convergência foram medidos. O intuito é encontrar a melhor configuração para o *Hardware* utilizado e servir como guia para quanto a escalabilidade do algorítimo para GPUs maiores.

Por fim, o tempo de execução para estimação dos parâmetros foi medido em ambos os casos (apenas com a P_{pdg} e com as 3 pressões). Esta etapa serve para demonstrar a aplicabilidade do algorítimo na rotina operacional da planta.

As simulações foram compiladas no Microssoft Visual Studio 2019, em modo *Release*, em um notebook Lenovo ideapad Gaming 3i, com processador intel i7 de 10ª geração (*clock* base de 2.6 GHz) e placa de vídeo NVIDIA Geforce GTX 1650. A versão utilizada do CUDA foi a 11.4 e utilizou-se o compilador NVCC (NVIDIA, 2021).

Todas as simulações foram replicadas 50 vezes de modo a obter um bom estimador da fração de convergência. O critério de convergência utilizado neste estudo foi $F_{obj} < 4.10^{10}$ Pa^2 para o caso da Ppdg e $F_{obj} < 1{,}4.10^{12}$ Pa^2 para o caso envolvendo as 3 pressões. Estes valores foram obtidos observando os ajustes por meio de gráficos e não seguem nenhum critério rigoroso. Em simulações com dados reais (em contraste com funções *benchmark*

cujos mínimos são conhecidos) não existe um conhecimento *a priori* do valor da função objetivo no mínimo global, o que torna difícil de utilizar a mesma como critério de parada. O autor sugere que estudos futuros sejam realizados visando obter um critério de parada mais rigoroso para o algoritmo aqui apresentado.

4 Resultados

Neste Capítulo serão apresentados os resultados obtidos nas etapas de ajuste do PSO. Seção 4.1 apresenta a escolha do passo de integração (h). As seções 4.2, 4.3 e 4.4 realizam as etapas de ajuste do PSO conforme descrito no Capítulo 3. A seção 4.5 compara o tempo de execução da melhor configuração obtida na seção 4.4 em versão paralela e sequencial. Por fim, a seção 4.6 aplica os ajustes realizados neste capítulo à função objetivo proposta por Diehl, descrita na seção 3.2.2.

4.1 Escolha do passo de integração

Neste trabalho utilizou-se um passo de integração (h) de 5 s, o que resulta numa simulação levemente ruidosa, mas captura o formato do modelo suficientemente bem para ser usado no PSO. Como mencionado no Capítulo 3, os parâmetros foram inicialmente estimados para o menor passo avaliado (h=0,5 s), aumentou-se então o passo para 1 s, avaliando graficamente se o ajuste dos parâmetros para este passo apresenta bons resultados quando utilizado com passos menores. Este processo foi repetido para h =5 s, onde a simulação apresentou um pouco de ruído, apesar disso, quando utilizados em conjunto com os passos de integração menores (h=0,5 s e h=1 s) os parâmetros ajustados obtiveram boa adesão aos dados do OLGA, como ilustrado na Figura 4.1. O mesmo não pode ser dito para h=10 s.

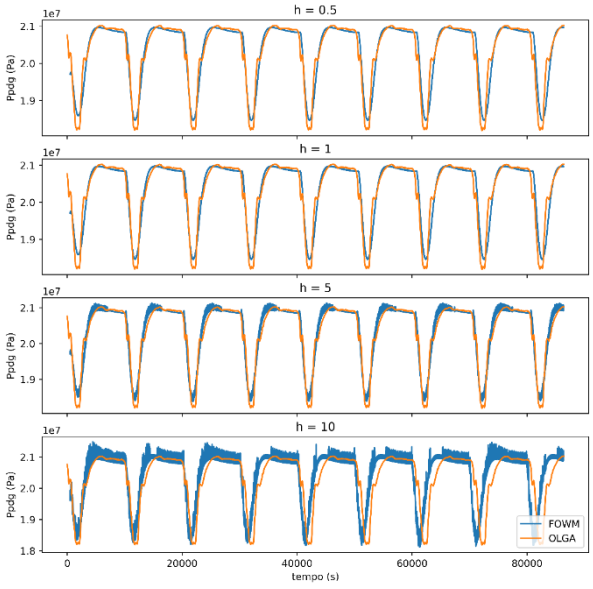

Figura 4.1 – Passo de integração para um mesmo conjunto de parâmetros.

4.2 Número de Partículas e Topologia

Para o limite de iterações imposto neste trabalho (10000), configurações com 64 e 128 partículas tiveram convergência praticamente nula. A fração de convergência teve crescimento proporcional ao número de partículas, alcançando um valor máximo de 0,86. Apesar de lógica, esta tendência vai de encontro com as implementações sugeridas para o PSO sequencial, que utilizam tamanhos de enxame relativamente pequenos (o PSO referência (*Standard PSO)* propõe 50 partículas). É importante ressaltar que esta comparação só é válida para o valor escolhido de iterações. Como nenhuma análise quanto à estagnação do enxame foi realizada, convergência para um tempo mais longo de simulação não pode ser descartada.

As Figuras 4.2, 4.3 e 4.4 apresentam a evolução da função objetivo média e da fração de convergência para N=1024, N=512 e N=256, respectivamente.

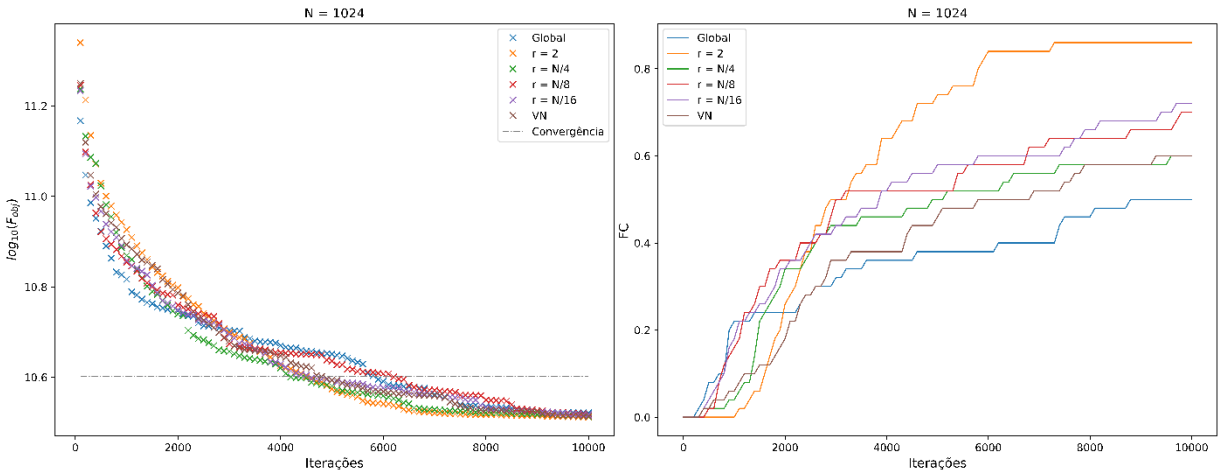

Figura 4.2 – Evolução da função objetivo para simulações convergentes (esquerda) e evolução da fração de simulações que convergiu (direita) para N =1024.

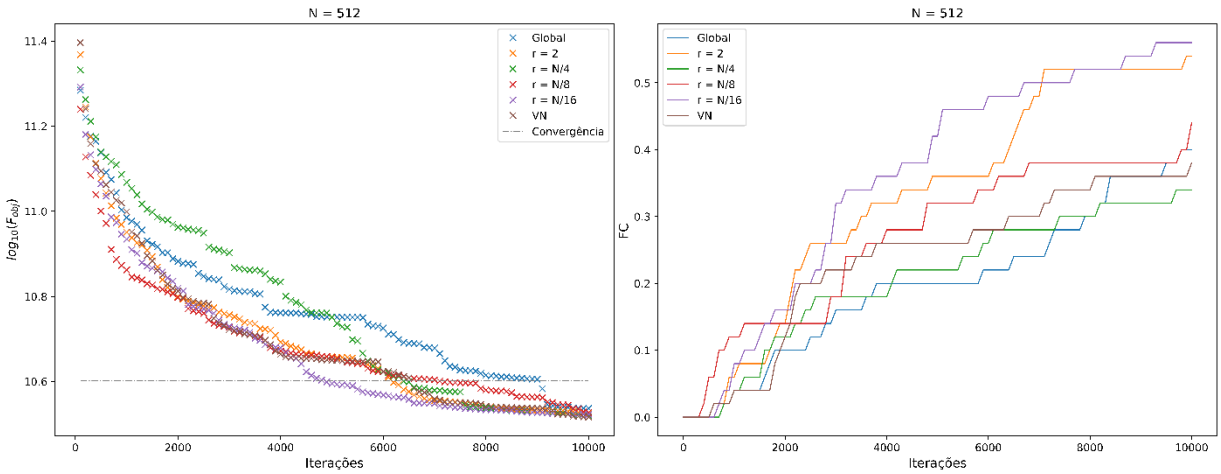

Figura 4.3 – Evolução da função objetivo para simulações convergentes (esquerda) e evolução da fração de simulações que convergiu (direita) para N =512.

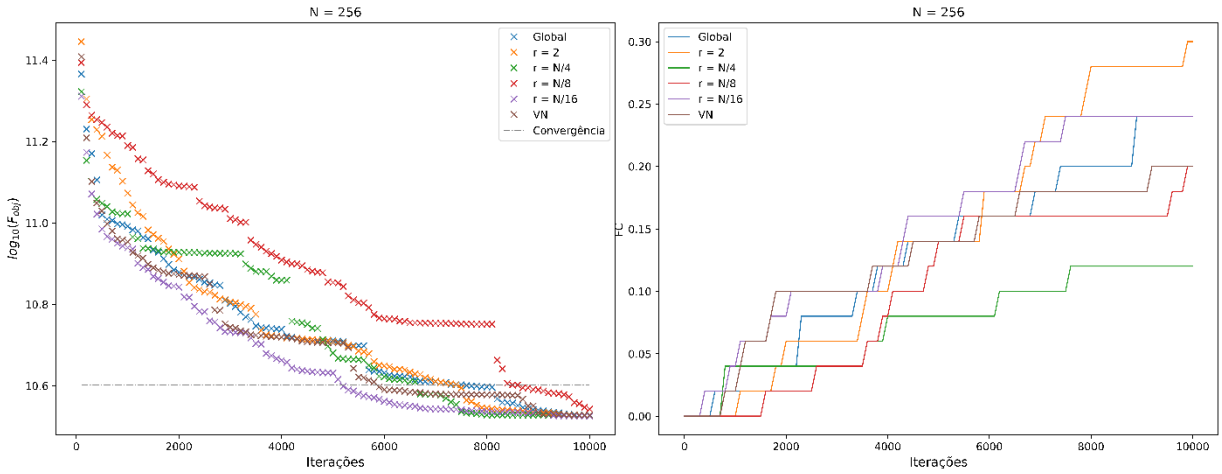

Figura 4.4 – Evolução da função objetivo para simulações convergentes (esquerda) e evolução da fração de simulações que convergiu (direita) para N =256.

De modo a facilitar a visualização do leitor, o critério de escolha calculado para todas as configurações desta etapa se encontra na Figura 4.5 (com as 5 melhores configurações marcadas em vermelho).

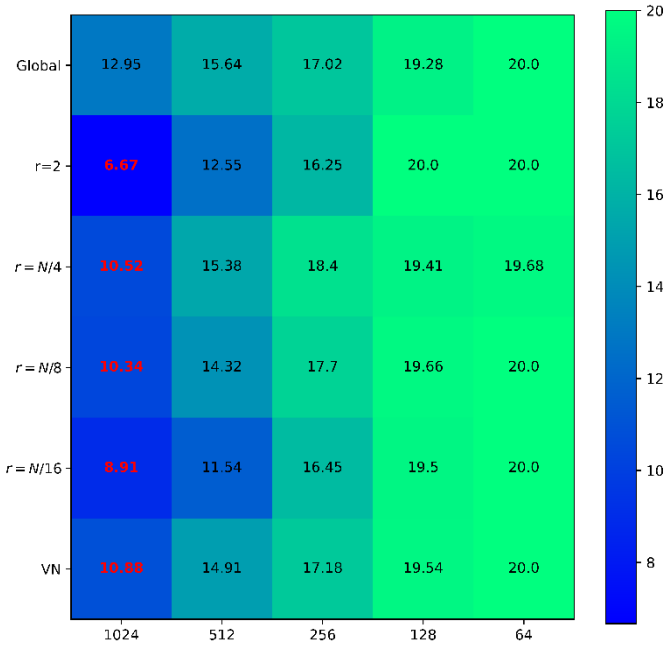

Figura 4.5 – Critério de escolha para todas as configurações testadas na primeira etapa.

As cinco melhores configurações utilizaram 1024 partículas, com as topologias r=2, r=N/16, r=N/8, r=N/4 e VN, deixando de fora apenas a topologia global, o que corrobora com a afirmação de Peng et al. (2022) mencionada no Capítulo 2 sobre a eficiência de topologias menos conectadas para minimização de funções multimodais.

4.3 Coeficiente de Nostalgia e Coerção

Nesta etapa foi dada sequência ao ajuste dos parâmetros do PSO. Para todas as simulações nesta etapa os valores de c_1 e c_2 foram variados. Assim como na etapa anterior, tanto a média da Fobj quanto a fração de convergência foram monitoradas (Figuras 4.6 a 4.10). Cada Figura representa a variação dos coeficientes para as cinco melhores configurações obtidas na seção 4.2.

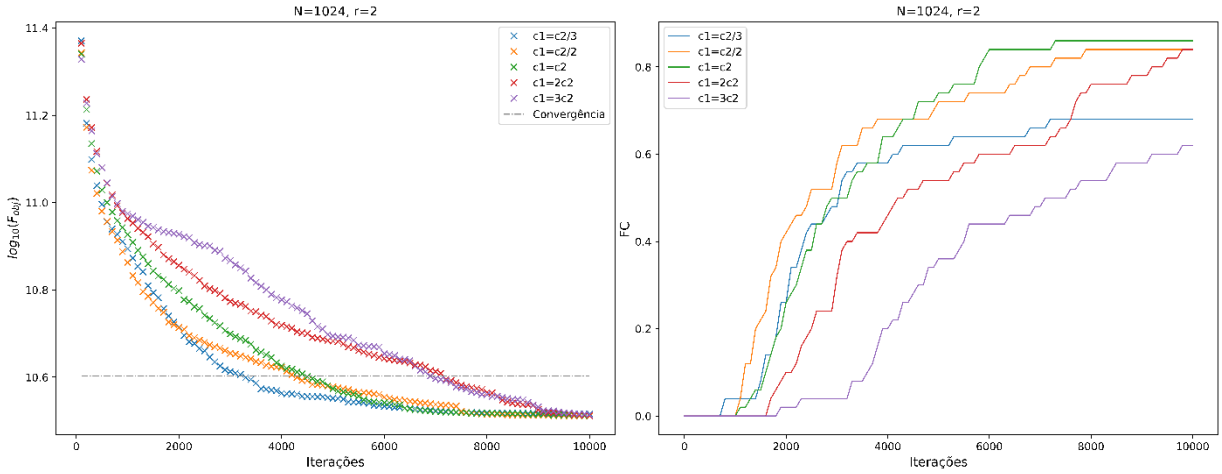

Figura 4.6 – Evolução da função objetivo para simulações convergentes (esquerda) e evolução da fração de simulações que convergiu (direita) para N =1024, topologia anel (r=2).

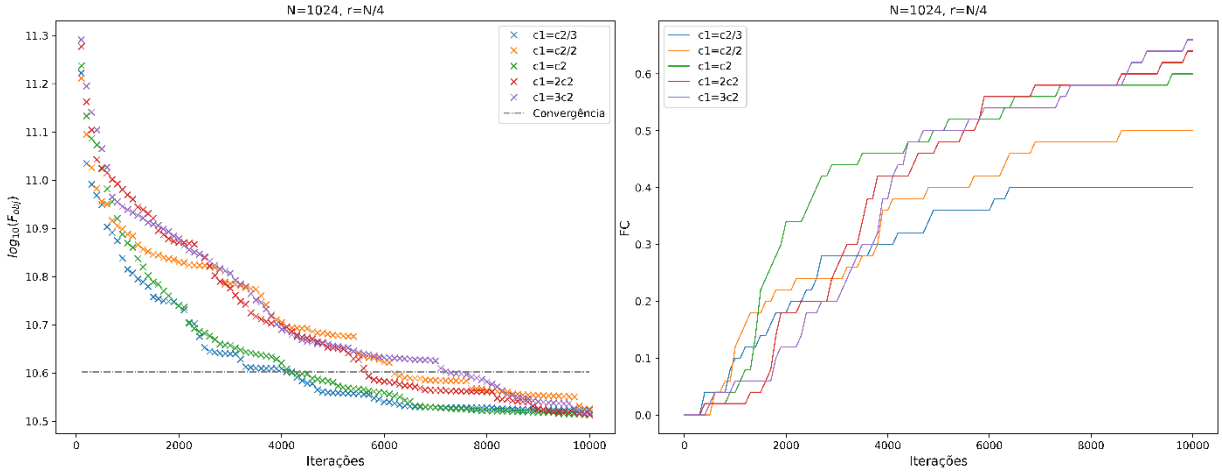

Figura 4.7 – Evolução da função objetivo para simulações convergentes (esquerda) e evolução da fração de simulações que convergiu (direita) para N =1024, topologia círculo (r=N/4).

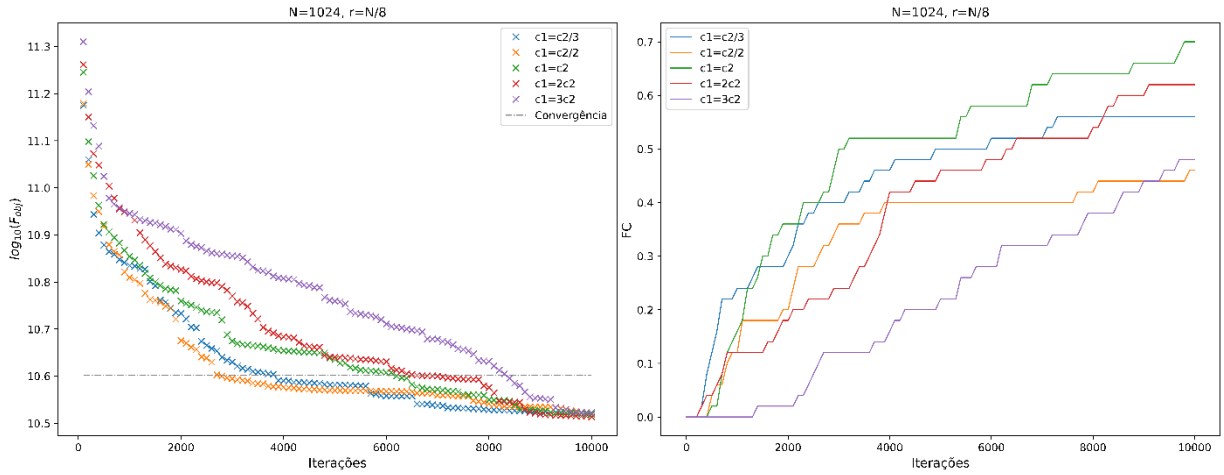

Figura 4.8 – Evolução da função objetivo para simulações convergentes (esquerda) e evolução da fração de simulações que convergiu (direita) para N =1024, topologia círculo (r=N/8).

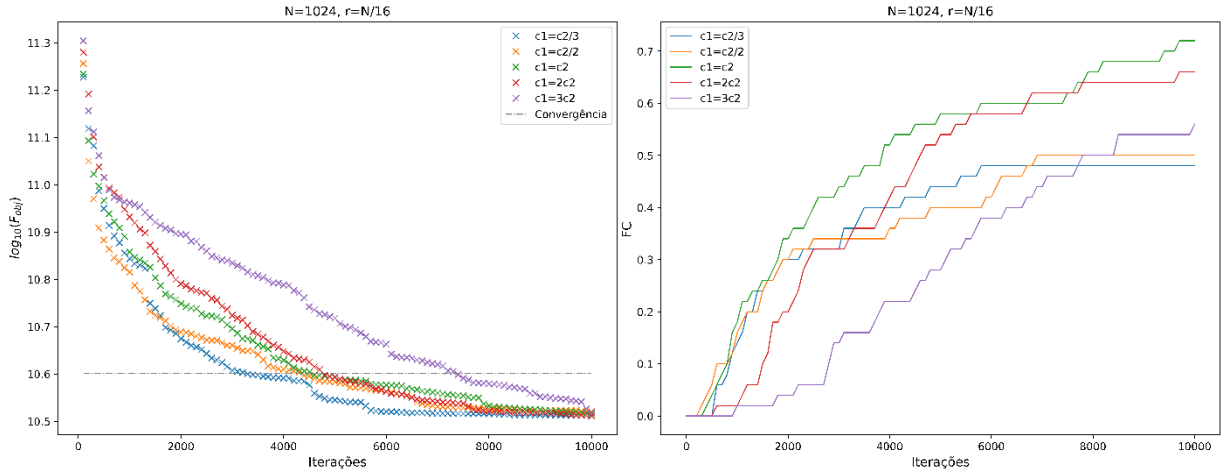

Figura 4.9 – Evolução da função objetivo para simulações convergentes (esquerda) e evolução da fração de simulações que convergiu (direita) para N =1024, topologia círculo (r=N/16).

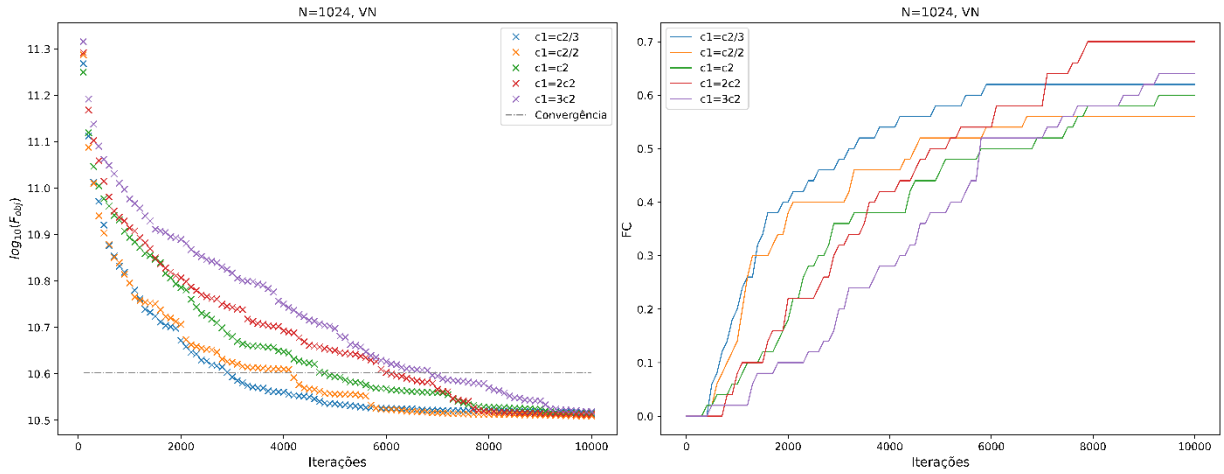

Figura 4.10 – Evolução da função objetivo para simulações convergentes (esquerda) e evolução da fração de simulações que convergiu (direita) para N =1024, topologia Von Neumann (VN).

Assim como na etapa anterior o critério de escolha das simulações se encontra na Figura 4.11 (com a melhor configuração marcada em vermelho).

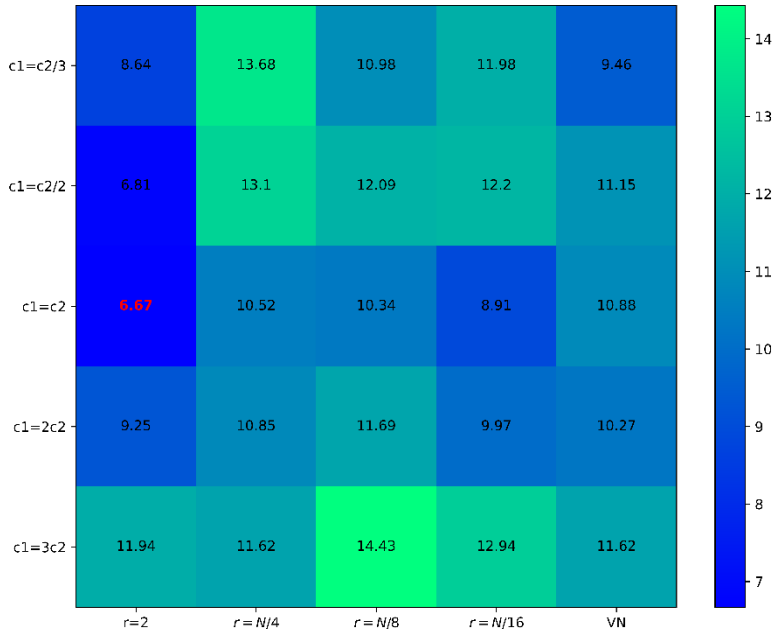

Figura 4.11 – Critério de escolha para todas as configurações testadas na segunda etapa.

Os melhores resultados foram obtidos para c_1=c_2, r=2. Em outras palavras, variar a relação c_1/c_2 não trouxe benefícios à robustez e desempenho do algoritmo.

4.4 Escalabilidade

Por fim, para a melhor configuração obtida, a escalabilidade foi testada. Nesta etapa as simulações foram realizadas até o critério de convergência ser atingido (em contraste com as etapas anteriores que simularam 10000 iterações). A razão para isso se dá ao longo tempo de execução que seria necessário para simular todas as iterações nas versões escalonadas.

As FCs para cada configuração se encontram na Tabela 4.1, o número médio de iterações para convergência se encontra na Figura 4.12, e o tempo de execução para cada configuração se encontra na Figura 4.13.

Tabela 4.1 – Valores de FC para versões escalonadas com 3 configurações: Enxames Individuais (EI), Migração (M) e Enxame Único (EU).

Estrutura/N_{total}	**2048**	**4096**	**8192**	**16384**
EI	1,00	1,00	1,00	1,00
M	0,96	1,00	1,00	1,00
EU	1,00	1,00	1,00	1,00

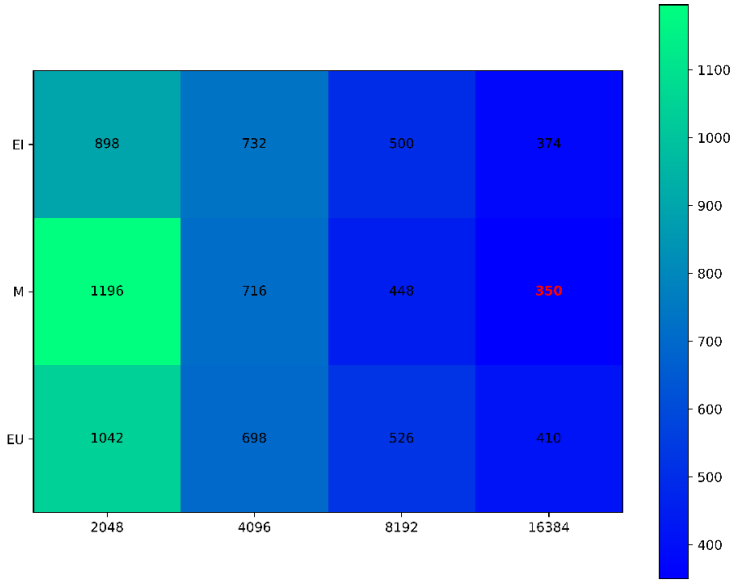

Figura 4.12 – Número médio de iterações para convergência para as versões escalonadas.

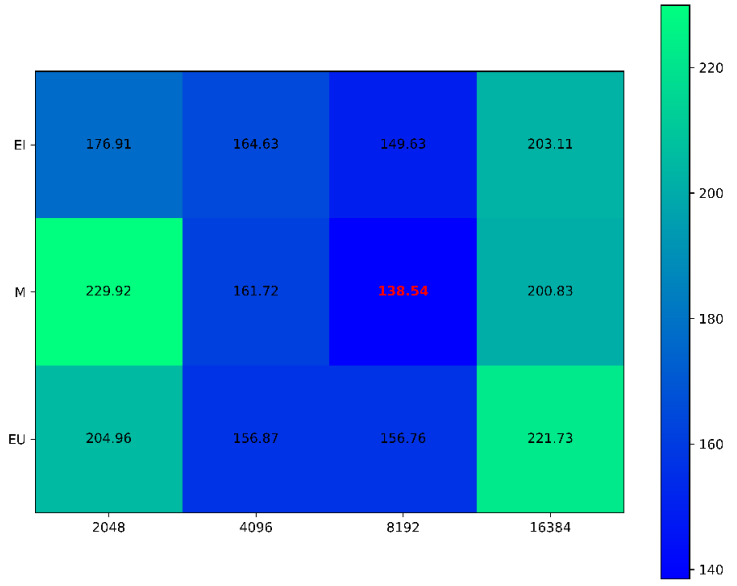

Figura 4.13 – Tempo médio em segundos para convergência para as versões escalonadas

Para a GPU utilizada neste trabalho, a configuração que obteve menor tempo médio foi a N=8192 com Migração, alcançando um tempo médio de 138,53 s. O número médio de iterações serve como guia para quanto a utilização de GPUs maiores. A Figura 4.12 mostra

que o número médio de iterações necessário para convergência continua a diminuir com o aumento do número de partículas.

4.5 Comparação com outras implementações

Nesta etapa é realizado um comparativo entre o tempo médio por iteração para o algoritmo implementado em CUDA quando comparado a uma implementação similar em C. A configuração utilizada foi a obtida na seção 4.4 (N=8192 com Migração, Topologia de Anel c_1=c_2). O tempo médio por iteração foi medido ao longo de 100 iterações, a implementação sequencial utilizou apenas 1 núcleo da CPU e seu código pode ser encontrado no Apêndice B.

A implementação sequencial levou em média 44,09 s por iteração, enquanto a paralela executa cada iteração em 0,31 s (142,58x mais rápido). Para esta configuração o algoritmo sequencial levaria 19752,32 s (5,49 h) para atingir o valor crítico.

Os trabalhos de Apio (2017) e Rodrigues (2018) propõem técnicas para estimação dos mesmos parâmetros. Rodrigues (2018) utilizou um PSO com 50 partículas e 2000 iterações, com um tempo médio de 870 ms para cada avaliação da função objetivo, resultando num tempo total de aproximadamente 24h. Apio (2017) por outro lado reportou um tempo total de 12h, utilizando o método *single shooting*. Assim, a configuração ótima obtida neste trabalho utilizada para ajustar os parâmetros tomando como referência apenas a P_{pdg} obteve uma redução de 99,84% e 99,68% no tempo necessário para estimação dos parâmetros, quando comparado aos tempos reportados por Rodrigues (2018) e Apio (2017) respectivamente. Tal discrepâcia provavelmente ocorre por duas razões:

- Este trabalho utiliza *hardware* (GPU) com capacidadde de processamento muito maior do que CPU numa faixa de preço equivalente.

- Este trabalho optou por deliberadamente utilizar um passo de integração maior, além de realizar todas as operações com precisão simples o que resulta num tempo de simulação muito menor. No trabalho de Rodrigues (2018) foram utilizados integradores com passo variável e controle de erro, como o Radau5ODE, além de precisão dupla (padrão do Python).

4.6 Análise para 3 pressões

Os resultados do ajuste simultâneo para P_{pdg}, P_{tt} e P_{rt} se encontram na Figura 4.14. Para a melhor configuração obtida no quesito tempo de execução, estimar os parâmetros

utilizando a função proposta por Diehl et al. levou em média 476,48 s (em média 1598 iterações), enquanto utilizando apenas a P_{pdg} o algoritmo levou 138,53 s (448 iterações), o que ilusra a maior complexidade de ajustar o modelo a três variáveis medidas simultaneamente. Ainda assim, o algorítimo não falhou em alcançar o valor crítico nenhuma vez e obteve um tempo de execução baixo o suficiente para o propósito dete trabalho. A função objetivo proposta por Diehl et al. fez um ótimo trabalho ponderando diferentes pressões, de modo que apesar de ser em média uma ordem de magnitude menor, o ajuste de P_{rt} recebeu tanto peso quanto as demais pressões.

A estimação utilizando apenas os dados da P_{pdg} se encontra na Figura 4.15 e resultou num bom ajuste para a P_{pdg} e P_{tt}, mas os resultados obtidos para o ajuste da P_{rt} apresentaram menor amplitude nos picos (entretanto esta colocação é meramente qualitativa, uma vez que não foram realizados testes estatísticos neste trabalho), o que vai ao encontro dos resultados obtidos por Diehl et al. (2017). Segundo eles, a estimação de parâmetros do FOWM usando apenas a P_{pdg} não garante o ajuste global do mesmo. Ainda assim, é útil poder utilizar o modelo dependendo apenas da P_{pdg} para obtenção dos parâmetros, uma vez que medidas de todas as pressões nem sempre estão disponíveis (DIEHL *et al.*, 2017). Os parâmetros obtidos para ambos os casos estão na Tabela 4.2. Os parâmetros foram também estimados para múltiplas aberturas da válvula *choke* e seus valores se encontram no Apêncice C.

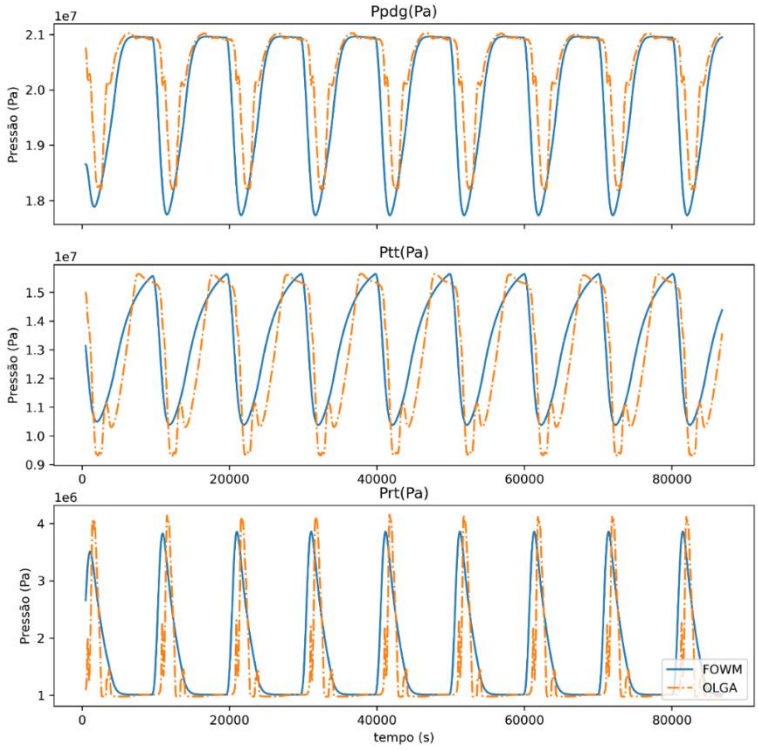

Figura 4.14 – Ajuste simultâneo para 3 pressões (P_{pdg}, P_{tt} e P_{rt}).

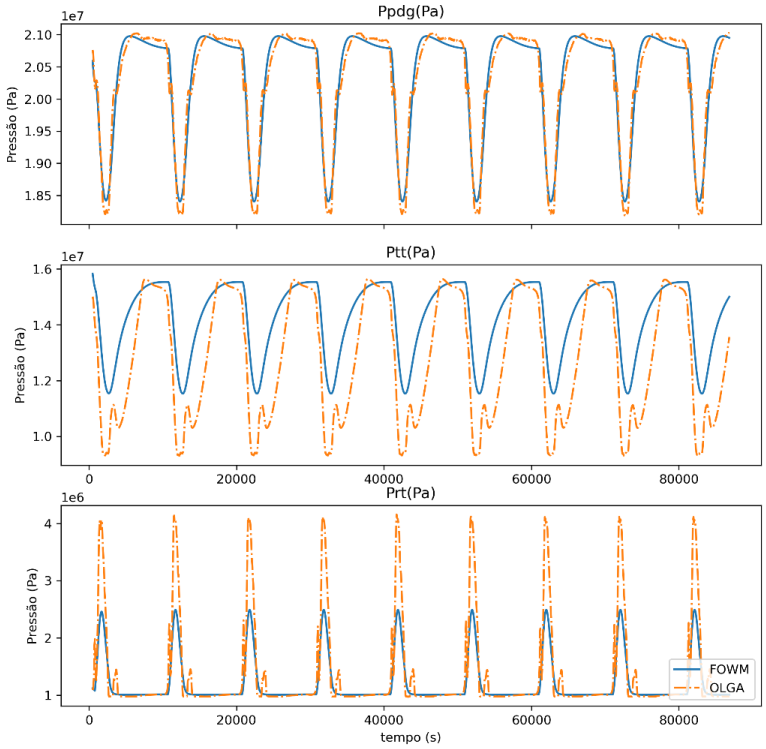

Figura 4.15 – Ajuste usando apenas a P_{pdg}.

Tabela 4.2 – Parâmetros obtidos para o ajuste do FOWM utilizando uma e múltiplas pressões

Parâmetro (Unid)	**Apenas P_{pdg}**	**P_{pdg}, P_{tt}, P_{rt}**
$m_{L,still}$ (Kg)	4724.7231445312	54.8192481995
C_g ($m*s$)	0.0000552986	0.0000016152
C_{out} (m^2)	0.0048283422	0.0036079551
V_{eb} (m^3)	80.6512756348	87.3540649414
E (-)	0.1437970549	0.0336449295
K_w (Kg/m)	0.0004694668	0.0008459875
K_a (Kg/m)	0.0017206073	0.0014850982
K_r (Kg/s)	140.4575347900	147.1688995361
ω_u (-)	2.3304784298	1.1671241522
F_{obj}	**35554226176.000000**	**1390402207744.000000**

5 Conclusões e Trabalhos Futuros

Este trabalho teve como meta ajustar o PSO paralelo implementado em CUDA ao ajuste de parâmetros do modelo FOWM proposto por (DIEHL *et al.*, 2017). Para isto, três etapas de otimização foram realizadas:

A primeira avaliou o efeito do tamanho de enxame e topologia na eficiencia e robustez do algoritmo. A segunda avaliou a relação dos coeficientes de nostalgia (c_1) e coerção (c_2) e por fim estratégias para aplicação do algoritmo em *hardware* mais potente (GPU maior) foram avaliadas, de modo a escalar o algoritmo até 16384 partículas. A melhor configuração obtida para o *Hardware* utilizado foram enxames de 8 enxames de 1024 partículas cada com Topologia r=2, c_1=c_2 com migração entre enxames. O tempo médio necessário para estimação dos parâmetros utilizando Mínimos quadrados aplicados à Ppdg foi de 138,53 s.

Os resultados obtidos foram comparados com uma versão sequencial escrita em C e com dois trabalhos da literatura que propõem a estimação de parâmetros do FOWM: (APIO, 2017) e (RODRIGUES, 2018). A implementação proposta neste trabalho foi capaz de estimar os parâmetros substâncialmente mais rápido do que os trabalhos anteriores (12h (APIO, 2017) e 24h (RODRIGUES, 2018)).

No trabalho apresentado por Rodrigues (2018), os parâmetros do FOWM foram estimados com apenas 50 partículas em 2000 iterações. No trabalho atual, versões do algorítimo com menor enxame (64 e 128 partículas) foram incapazes de encontrar soluções. Isto se dá pela complexidade da função objetivo que apresenta diversos mínimos locais. No seu trabalho, Rodrigues (2018) apresentou técnicas para simplificar a função objetivo (i.e. tornar mais convexa), o que explica o menor esforço computacional empregado. Vale ressaltar que apesar de ter apresentado tempo de execução menor, o método aqui proposto realizou muito mais avaliações da função objetivo do que o proposto por Rodrigues (2018).

Por fim a capacidade do algoritmo de ajustar os parâmetros do FOWM foi demonstrada para o ajuste simultâneo de 3 pressões (P_{pdg}, P_{tt} e P_{rt}) utilizando a função objetivo proposta por Diehl et al.. Apesar de levar mais tempo, o algorítimo foi capaz de estimar os parâmetros em menos de 10 minutos, além de não falhar em alcançar o mínimo global nenhuma vez em 50 simulações.

O autor sugere que em trabalhos futuros as técnicas apresentadas em Ricardo e Apio para estimação de parâmetros de sistemas com bifurcação de Hopf sejam aplicadas em conjunto com o algoritmo aqui proposto. Além disso, explorar a implementação de

integradores de passo variável no ambiente CUDA poderia reduzir ainda mais o tempo necessário para estimação dos parâmetros.

Como mencionado no Capítulo 3, o critério de parada não foi rigorosamente definido, assim, atualmente não é possível saber se houve convergência dos parâmetros sem observações gráficas. Desta forma, propor um critério de parada rigorosamente definido é extremamente desejável.

Referências

ANDRIES P. ENGELBRECHT. **Computational Inteligence An Introduction**. [*S. l.: s. n.*], 2007.

APIO, Andressa. **ESTIMAÇÃO DE PARÂMETROS EM MODELOS COM CICLO LIMITE**. 2017. [*s. l.*], 2017.

BABALOLA, Asegunloluwa Eunice; OJOKOH, Bolanle Adefowoke; ODILI, Julius Beneoluchi. A Review of Population-Based Optimization Algorithms. *In*: , 2020. **2020 International Conference in Mathematics, Computer Engineering and Computer Science, ICMCECS 2020**. [*S. l.*]: Institute of Electrical and Electronics Engineers Inc., 2020.

BALIÑO, J. L. Modeling and simulation of severe slugging in air-water systems including inertial effects. **Journal of Computational Science**, [*s. l.*], v. 5, n. 3, p. 482–495, 2014.

BENDLKSEN, Kjell H *et al.* The Dynamic Two-Fluid Model OLGA: Theory and Application. **SPE Production Engineering**, [*s. l.*], v. 6, n. 02, p. 171–180, 1991. Disponível em: https://doi.org/10.2118/19451-PA.

BILTOFT, Jakob *et al.* Recreating riser slugging flow based on an economic lab-sized setup ?. *In*: , 2013. **IFAC Proceedings Volumes (IFAC-PapersOnline)**. [*S. l.*]: IFAC Secretariat, 2013. p. 47–52.

BORISOV, S. P.; KUDRYAVTSEV, A. N.; SHERSHNEV, A. A. Influence of detailed mechanisms of chemical kinetics on propagation and stability of detonation wave in H2/O2 mixture. *In*: , 2019. **Journal of Physics: Conference Series**. [*S. l.*]: Institute of Physics Publishing, 2019.

CLERC, Maurice; KENNEDY, James. **The Particle Swarm-Explosion, Stability, and Convergence in a Multidimensional Complex SpaceIEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION**. [*S. l.: s. n.*], 2002.

DÉCIO, Diretor-Geral *et al.* **AGÊNCIA NACIONAL DO PETRÓLEO, GÁS NATURAL E BIOCOMBUSTÍVEIS Superintendente-adjunto de Segurança Operacional e Meio Ambiente**. [*S. l.: s. n.*], 2018.

DIEHL, Fabio C. *et al.* Fast Offshore Wells Model (FOWM): A practical dynamic model for multiphase oil production systems in deepwater and ultra-deepwater scenarios. **Computers and Chemical Engineering**, [*s. l.*], v. 99, p. 304–313, 2017.

EIKREM, Gisle Otto; AAMO, Ole Morten; FOSS, Bjarne A. **On Instability in Gas Lift Wells and Schemes for Stabilization by Automatic Control**. [*S. l.: s. n.*], 2008.

ENGELBRECHT, Andries. Particle swarm optimization: Velocity initialization. *In*: , 2012. **2012 IEEE Congress on Evolutionary Computation, CEC 2012**. [*S. l.: s. n.*], 2012.

FALCONE, Gioia. Chapter 1 Multiphase Flow Fundamentals. *In*: FALCONE, Gioia; HEWITT, G F; ALIMONTI, Claudio (org.). **Developments in Petroleum Science**. [*S. l.*]: Elsevier, 2009. v. 54, p. 1–18. *E-book*. Disponível em: https://www.sciencedirect.com/science/article/pii/S0376736109054016.

FINKEL, Daniel E. **DIRECT Optimization Algorithm User Guide**. [*S. l.: s. n.*], 2003. Disponível em: http://www4.ncsu.edu/. .

GEREVINI, Giovani Gonçalves. **ATENUAÇÃO DE GOLFADAS EM SISTEMAS DE ELEVAÇÃO DE PETRÓLEO EM AMBIENTE OFFSHORE**. 2017. [*s. l.*], 2017.
GOLDSWORTHY, M. J. A GPU-CUDA based direct simulation Monte Carlo algorithm for real gas flows. **Computers and Fluids**, [*s. l.*], v. 94, p. 58–68, 2014.
HOUSSEIN, Essam H. *et al.* Major Advances in Particle Swarm Optimization: Theory, Analysis, and Application. **Swarm and Evolutionary Computation**, [*s. l.*], v. 63, 2021.
HU, Bin. **Characterizing gas-lift instabilities**. [*S. l.: s. n.*], 2004.
HÜFFNER, Leonardo Nardi. **FUNÇÃO OBJETIVO PARA ESTIMAÇÃO DE PARÂMETROS DE MODELOS COM CICLO LIMITE**. 2017. [*s. l.*], 2017.
HUSSAIN, Md Maruf; HATTORI, Hiroshi; FUJIMOTO, Noriyuki. A CUDA Implementation of the Standard Particle Swarm Optimization. [*s. l.*], 2016.
JAHANSHAHI, Esmaeil; SKOGESTAD, Sigurd. Simplified dynamical models for control of severe slugging in multiphase risers. *In*: , 2011. **IFAC Proceedings Volumes (IFAC-PapersOnline)**. [*S. l.*]: IFAC Secretariat, 2011. p. 1634–1639.
KAZACHENKO, Sergey *et al.* Algorithms for GPU-based molecular dynamics simulations of complex fluids: Applications to water, mixtures, and liquid crystals. **Journal of Computational Chemistry**, [*s. l.*], v. 36, n. 24, p. 1787–1804, 2015.
KENNEDY, James; EBERHART, Russell. Particle Swarm Optimization. *In*: , 1995. **Anais [...]**. [*S. l.: s. n.*], 1995. p. 1942–1948.
KIRK, David B; HWU, Wen-mei W. **Programming Massively Parallel Processors: A Hands-on Approach**. [*S. l.: s. n.*], 2013.
KUCYK, Daniel. **Reinforcement Learning aplicado para otimização da produção de poços de elevação de petróleo**. 2021. [*s. l.*], 2021.
KULKARNI, Manjiri K; UMALE, J S. **Prediction of Chemical Bond Formation using efficient CUDA based HPC Framework**. [*S. l.: s. n.*], 2016.
KUTTA, Wihelm. Beitrag_zur_näherungsweisen_Integration. [*s. l.*], 1901.
MEGLIO, Florent di; KAASA, Glenn-Ole; PETIT, Nicolas. A first principle model for multiphase slugging flow in vertical risers. *In*: , 2009. **Anais [...]**. [*S. l.: s. n.*], 2009. p. 8244–8251.
MESSIAS BOLSONARO, Jair *et al.* **MINISTÉRIO DE MINAS E ENERGIA AGÊNCIA NACIONAL DO PETRÓLEO, GÁS NATURAL E BIOCOMBUSTÍVEIS PRESIDENTE DA REPÚBLICA DIRETOR-GERAL**. [*S. l.: s. n.*], 2021. Disponível em: https://www.gov.br/anp/pt-br/centrais-de-conteudo/publicacoes. .
NEDJAH, Nadia; DE MORAES CALAZAN, Rogério; DE MACEDO MOURELLE, Luiza. Particle, dimension and cooperation-oriented PSO parallelization strategies for efficient high-dimension problem optimizations on graphics processing units. **Computer Journal**, [*s. l.*], v. 59, n. 6, p. 810–835, 2016.
NVIDIA. **CUDA C++ Programming Guide Design Guide**. [*S. l.: s. n.*], 2021.
NVIDIA. **GPU Gems 2**. [*S. l.*], [*s. d.*]. Disponível em: https://developer.nvidia.com/gpugems/gpugems2/copyright. Acesso at: 10 Apr. 2022 a.
NVIDIA. **GPU Gems 3**. [*S. l.*], [*s. d.*]. Disponível em: https://developer.nvidia.com/gpugems/gpugems3/contributors. Acesso at: 10 Apr. 2022 b.

OZHGIBESOV, M. S. *et al.* Studies on argon collisions with smooth and rough tungsten surfaces. **Journal of Molecular Graphics and Modelling**, [*s. l.*], v. 45, p. 45–49, 2013.

PENG, Jian *et al.* Impact of population topology on particle swarm optimization and its variants: An information propagation perspective. **Swarm and Evolutionary Computation**, [*s. l.*], v. 69, 2022.

RODRIGUES, Ricardo França. **NOVA METODOLOGIA PARA ESTIMAÇÃO DE PARÂMETROS DE MODELOS COM BIFURCAÇÃO HOPF**. 2018. [*s. l.*], 2018.

RUNGE, C. **Ueber die numerische Aufl6sung von Differentialgleichungen**. [*S. l.: s. n.*], 1895.

SARSEMBAYEV, M. *et al.* Using the Cuda Technology to Speed up Computations in Problems of Chemical Kinetics. **Cybernetics and Systems Analysis**, [*s. l.*], v. 56, n. 4, p. 675–682, 2020.

SINÈGRE, Laure; PETIT, Nicolas; MÉNÉGATTI, Philippe. **Predicting instabilities in gas-lifted wells simulation**. [*S. l.: s. n.*], 2006.

YANG, Keda; SU, Jiaye; GUO, Hongxia. GPU accelerated numerical simulations of viscoelastic phase separation model. **Journal of Computational Chemistry**, [*s. l.*], v. 33, n. 18, p. 1564–1571, 2012. Disponível em: https://doi.org/10.1002/jcc.22990.

Apêndice A: Implementação do PSO aplicado ao ajuste de parâmetros do FOWM em CUDA C

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include<cuda.h>
#include <curand.h>
#include <time.h>
#include<string.h>


#define PI (3.14159265358979323846f)

#define Wgc (float(100000.0*(101325.0*18.0)/(293.0*8314.0*24.0*3600.0))) //vazao de gas lift convertida de m3/dia para kg/s
#define NVM (4) //"Numero de variaveis Medidas" (Pressões, abertura de válvula choke)
#define samplesize (5999) //tamanho da amostra (i.e. No de pontos)
#define STPB (128) //tamanho do bloco
#define N (1024) //numero de particulas por enxame
#define L (8) //numero de enxames

//memoria constante
__constant__ float c_Wc1c2[3];//coef de inercia, nostalgia e coercao
__constant__ float c_lim[18];//limites superior e inferior de cada dimensao
__constant__ float c_y_med[NVM];//media dos dados para cada VM
__constant__ float c_y_stdev[NVM];//desvio dos dados para cada VM

__constant__ float c_param_planta_ag[12];//parametros da planta

//definindo funções do modelo
__device__ float d_max(float x, float y) {
   if (x > y) {
      return x;
   }
   else {
      return y;
   }
}

__device__ float Vgt(float* m, int i, float* param_modelo) {

   float Vgt = c_param_planta_ag[7] - m[6 * i + 2] / c_param_planta_ag[8];
   return Vgt;
}

__device__ float rhogt(float* m, int i, float* param_modelo) {
```

```
    float rhogt = m[6 * i + 1] / Vgt(m, i, param_modelo);
    return rhogt;
}

__device__ float Pai(float* m, int i, float* param_modelo) {

    float Pai = c_param_planta_ag[0] * m[6 * i + 0];
    return Pai;
}

__device__ float alphagt(float* m, int i, float* param_modelo) {

    float alphagt = m[6 * i + 1] / (m[6 * i + 1] + m[6 * i + 2]);
    return alphagt;
}

__device__ float alphagr(float* m, int i, float* param_modelo) {

    float alphagr = m[6 * i + 4] / (m[6 * i + 4] + m[6 * i + 5]);
    return alphagr;
}

__device__ float Peb(float* m, int i, float* param_modelo) {

    float Peb = m[6 * i + 3] * c_param_planta_ag[1] / param_modelo[9 * i + 3];
    return Peb;
}

__device__ float rhomt(float* m, int i, float* param_modelo) {

    float rhomt = (m[6 * i + 1] + m[6 * i + 2]) / c_param_planta_ag[7];
    return rhomt;
}

__device__ float alphalr(float* m, int i, float* param_modelo) {

    float alphalr = 1 - alphagr(m, i, param_modelo);
    return alphalr;
}

__device__ float Ptt(float* m, int i, float* param_modelo) {

    float Ptt = rhogt(m, i, param_modelo) * c_param_planta_ag[1];
    return Ptt;
}

__device__ float rhoai(float* m, int i, float* param_modelo) {

    float rhoai = Pai(m, i, param_modelo) / c_param_planta_ag[1];
    return rhoai;
}

__device__ float Prt(float* m, int i, float* param_modelo) {
```

```
        float Prt = m[6 * i + 4] * c_param_planta_ag[1] / (param_modelo[9 * i + 8] *
c_param_planta_ag[9] - (m[6 * i + 5] + param_modelo[9 * i + 0]) /
c_param_planta_ag[8]);
        return Prt;
    }

    __device__ float Prb(float* m, int i, float* param_modelo) {

        float Prb = Prt(m, i, param_modelo) + (m[6 * i + 5] + param_modelo[9 * i + 0]) *
c_param_planta_ag[5];
        return Prb;
    }

    __device__ float Ptb(float* m, int i, float* param_modelo) {

        float Ptb = Ptt(m, i, param_modelo) + rhomt(m, i, param_modelo) *
c_param_planta_ag[2];
        return Ptb;
    }

    __device__ float Ppdg(float* m, int i, float* param_modelo) {

        float Ppdg = Ptb(m, i, param_modelo) + c_param_planta_ag[4];
        return Ppdg;
    }

    __device__ float Pbh(float* m, int i, float* param_modelo) {

        float Pbh = Ppdg(m, i, param_modelo) + c_param_planta_ag[3];
        return Pbh;
    }

    //Vazões
    __device__ float Wiv(float* m, int i, float* param_modelo) {

        float Wiv = param_modelo[9 * i + 6] * sqrtf(d_max(0.0f, rhoai(m, i, param_modelo)
* (Pai(m, i, param_modelo) - Ptb(m, i, param_modelo))));
        return Wiv;
    }

    __device__ float Wr(float* m, int i, float* param_modelo) {

        float Wr = d_max(0.0f, param_modelo[9 * i + 7] * (1.0f - (0.2f * Pbh(m, i,
param_modelo) / c_param_planta_ag[6]) - ((0.8f * Pbh(m, i, param_modelo) /
c_param_planta_ag[6]) * (Pbh(m, i, param_modelo) / c_param_planta_ag[6]))));
        return Wr;
    }

    __device__ float Wwhg(float* m, int i, float* param_modelo) {

        float Wwhg = param_modelo[9 * i + 5] * sqrtf(d_max(0.0f, c_param_planta_ag[8] *
(Ptt(m, i, param_modelo) - Prb(m, i, param_modelo)))) * alphagt(m, i, param_modelo);
```

```
    return Wwhg;
}

__device__ float Wwhl(float* m, int i, float* param_modelo) {

    float Wwhl = param_modelo[9 * i + 5] * sqrtf(d_max(0.0f, c_param_planta_ag[8] *
(Ptt(m, i, param_modelo) - Prb(m, i, param_modelo))) * (1 - alphagt(m, i,
param_modelo)));
    return Wwhl;
}

__device__ float Wg(float* m, int i, float* param_modelo) {

    float Wg = param_modelo[9 * i + 1] * d_max(0.0f, (Peb(m, i, param_modelo) -
Prb(m, i, param_modelo)));
    return Wg;
}

__device__ float Wgout(float* m, int i, float* param_modelo,float z) {

    float Wgout = alphagr(m, i, param_modelo) * param_modelo[9 * i + 2] * z *
sqrtf(d_max(0.0f, c_param_planta_ag[8] * (Prt(m, i, param_modelo) -
c_param_planta_ag[10])));
    return Wgout;
}

__device__ float Wlout(float* m, int i, float* param_modelo,float z) {

    float Wlout = alphalr(m, i, param_modelo) * param_modelo[9 * i + 2] * z *
sqrtf(d_max(0.0f, c_param_planta_ag[8] * (Prt(m, i, param_modelo) -
c_param_planta_ag[10])));
    return Wlout;
}

//EDOs

__device__ void Dm(float* out, float* m, float* param_modelo, int i, float z) {

    float Pai = c_param_planta_ag[0] * m[0];
    float Vgt = c_param_planta_ag[7] - m[2] / c_param_planta_ag[8];
    float alphagt = m[1] / (m[1] + m[2]);
    float alphagr = m[4] / (m[4] + m[5]);
    float Peb = m[3] * c_param_planta_ag[1] / param_modelo[9 * i + 3];
    float rhomt = (m[1] + m[2]) / c_param_planta_ag[7];
    float Prt = m[4] * c_param_planta_ag[1] / (param_modelo[9 * i + 8] *
c_param_planta_ag[9] - (m[5] + param_modelo[9 * i + 0]) / c_param_planta_ag[8]);

    float rhoai = Pai / c_param_planta_ag[1];
    float Prb = Prt + (m[5] + param_modelo[9 * i + 0]) * c_param_planta_ag[5];
    float rhogt = m[1] / Vgt;
    float alphalr = 1.0f - alphagr;
    float Wg = param_modelo[9 * i + 1] * ((0.0f > (Peb - Prb)) ? 0.0f : (Peb - Prb));
```

```
        float Wgout = alphagr * param_modelo[9 * i + 2] * z * sqrtf((0.0f > (c_param_planta_ag[8] * (Prt - c_param_planta_ag[10]))) ? 0.0f : (c_param_planta_ag[8] * (Prt - c_param_planta_ag[10])));

        float Ptt = rhogt * c_param_planta_ag[1];
        float Wlout = alphalr * param_modelo[9 * i + 2] * z * sqrtf((0.0f > (c_param_planta_ag[8] * (Prt - c_param_planta_ag[10]))) ? 0.0f : (c_param_planta_ag[8] * (Prt - c_param_planta_ag[10])));

        float Ptb = Ptt + rhomt * c_param_planta_ag[2];
        float Wwhg = param_modelo[9 * i + 5] * sqrtf((0.0f > c_param_planta_ag[8] * (Ptt - Prb)) ? 0.0f : c_param_planta_ag[8] * (Ptt - Prb)) * alphagt;
        float Wwhl = param_modelo[9 * i + 5] * sqrtf((0.0f > c_param_planta_ag[8] * (Ptt - Prb)) ? 0.0f : c_param_planta_ag[8] * (Ptt - Prb)) * (1.0f - alphagt);

        float Ppdg = Ptb + c_param_planta_ag[4];

        float Pbh = Ppdg + c_param_planta_ag[3];
        float Wiv = param_modelo[9 * i + 6] * sqrtf((0.0f > rhoai * (Pai - Ptb)) ? 0.0f : rhoai * (Pai - Ptb));

        float Wr = param_modelo[9 * i + 7] * ((0.0f > (1.0f - (0.2f * Pbh / c_param_planta_ag[6]) - ((0.8f * Pbh / c_param_planta_ag[6]) * ( Pbh / c_param_planta_ag[6])))) ? 0.0f : (1.0f - (0.2f * Pbh / c_param_planta_ag[6]) - ((0.8f * Pbh / c_param_planta_ag[6]) * (Pbh / c_param_planta_ag[6]))));

        out[0] = Wgc - Wiv;
        out[1] = Wr * c_param_planta_ag[11] + Wiv - Wwhg;
        out[2] = Wr * (1.0f - c_param_planta_ag[11]) - Wwhl;
        out[3] = (1.0f - param_modelo[9 * i + 4]) * Wwhg - Wg;
        out[4] = param_modelo[9 * i + 4] * Wwhg + Wg - Wgout;
        out[5] = Wwhl - Wlout;
    }

    __device__ void RK4(void(*Dm)(float*, float*, float*, int, float), float* G, int i, float* d_pos, float h, float z) {

        float temp[6];

        float K0[6];
        float K1[6];
        float K2[6];
        float K3[6];

        for (int k = 0; k < 6; k++) {
           temp[k] = G[6 * i + k];
        }

        Dm(K0, temp, d_pos, i,z);
        for (int k = 0; k < 6; k++) {
           temp[k] = G[6 * i + k] + h * K0[k] / 2.0f;
        }
```

```
        Dm(K1, temp, d_pos, i,z);

        for (int k = 0; k < 6; k++) {
           temp[k] = G[6 * i + k] + h * K1[k] / 2.0f;
        }

        Dm(K2, temp, d_pos, i,z);

        for (int k = 0; k < 6; k++) {
           temp[k] = G[6 * i + k] + h * K2[k];
        }

        Dm(K3, temp, d_pos, i,z);

        for (int k = 0; k < 6; k++) {
           G[6 * i + k] = G[6 * i + k] + h * (K0[k] + 2.0f * K1[k] + 2.0f * K2[k] + K3[k]) /
6.0f;
        }
    }

    //Fobj
    //Minimos Quadrados
    __global__ void Fobj(float* pos, float passo, float* cost, float* d_Dados, float t_stop)
    {

        int i = threadIdx.x;
        int ind = threadIdx.x + blockIdx.x * blockDim.x;
        float t = 0.0f;
        float h = passo;

        float s_cost=0.0f;

        float m[6];

        m[0] = 2805.14932589;
        m[1] = 2125.01254427;
        m[2] = 9103.27649409;
        m[3] = 7857.40023097;
        m[4] = 1316.40535031;
        m[5] = 36810.25978214;

        __shared__ float G[STPB * 6];
        __shared__ float s_pos[STPB * 9];

        for (int k = 0; k < 9; k++) {
           s_pos[9 * i + k] = pos[9 * ind + k];
        }

        __syncthreads();


        for (int k = 0; k < 6; k++) {
           G[i * 6 + k] = m[k];
```

```
        }

        int value = 0;

        float z = d_Dados[(NVM + 1) * value + 4] / 100.0f;

        while (t <= t_stop) {


           if (value < samplesize && d_Dados[(NVM+1) * value + 0] <= t + h) {

              float D0 = d_Dados[(NVM + 1) * value + 0];
              float D1 = d_Dados[(NVM + 1) * value + 1];
              float z = d_Dados[(NVM + 1) * value + 4] / 100.0f;

              float P1 = Ppdg(G, i, s_pos) * (t + h - D0);

              t = t + h;

              RK4(Dm, G, i, s_pos, h,z);

              //essa etapa é responsável pela interpolação linear
              P1 = (P1 + Ppdg(G, i, s_pos) * (D0 - t + h)) / h;

              //calculo dos valores necessários para a função objetivo
              float MSE1 = ((P1 - D1));

              s_cost += (MSE1 * MSE1);

              value = value + 1;
           }

           t = t + h;

           RK4(Dm, G, i, s_pos, h,z);

        }


        cost[ind] = s_cost/samplesize; //saída da função objetivo

    }

    //Fobj proposta por Diehl
    __global__ void Fobj_Diehl(float* pos, float passo, float* cost, float* d_Dados, float
t_stop)
    {

        int i = threadIdx.x;
        int ind = threadIdx.x + blockIdx.x * blockDim.x;
        float t = 0.0f;
        float h = passo;
        float MSE;
```

```
float rxy[NVM-1];
float x_med[NVM-1];
float x_stdev[NVM-1];
float wi;
float s_cost[NVM-1];

for (int k = 0; k < NVM-1; k++) {
   rxy[k]=0.0f;
   x_med[k] = 0.0f;
   x_stdev[k] = 0.0f;
   s_cost[k] = 0.0f;
}

float m[6];

m[0] = 7629.49953301;
m[1] = 1506.46645264;
m[2] = 20249.26259091;
m[3] = 2135.35823438;
m[4] = 1130.58624768;
m[5] = 15196.84541979;

__shared__ float G[STPB * 6];
__shared__ float s_pos[STPB * 9];

for (int k = 0; k < 9; k++) {
   s_pos[9 * i + k] = pos[9 * ind + k];
}

__syncthreads();


for (int k = 0; k < 6; k++) {
   G[i * 6 + k] = m[k];
}

int value = 0;

//float z = 0.22f;
float z = d_Dados[(NVM + 1) * value + 4]/100.0f;

while (t <= t_stop) {


   if (value < samplesize && d_Dados[(NVM + 1) * value + 0] <= t + h) {

      float D0 = d_Dados[(NVM + 1) * value + 0];
      float D1 = d_Dados[(NVM + 1) * value + 1];
      float D2 = d_Dados[(NVM + 1) * value + 2];
      float D3 = d_Dados[(NVM + 1) * value + 3];
      float z = d_Dados[(NVM + 1) * value + 4] / 100.0f;
```

```
            float P1 = Ppdg(G, i, s_pos) * (t + h - D0);
            float P2 = Ptt(G, i, s_pos) * (t + h - D0);
            float P3 = Prt(G, i, s_pos) * (t + h - D0);

            t = t + h;

            RK4(Dm, G, i, s_pos, h,z);

            //essa etapa é responsável pela interpolação linear

            P1 = (P1 + Ppdg(G, i, s_pos) * (D0 - t + h)) / h;
            P2 = (P2 + Ptt(G, i, s_pos) * (D0 - t + h)) / h;
            P3 = (P3 + Prt(G, i, s_pos) * (D0 - t + h)) / h;

            MSE = (P1 - D1);
            x_med[0] += P1;
            x_stdev[0] += (P1 * P1);
            rxy[0] += (P1 * D1);
            wi = ((D1 - c_y_med[0]) / c_y_stdev[0]) * ((D1 - c_y_med[0]) / c_y_stdev[0]);
            s_cost[0] += wi * (MSE * MSE);
            //s_cost[0] += wi * (MSE1 * MSE1);
            //calculo dos valores necessários para a função objetivo
            //float MSE1 = ((P1 - D1)) / NVM;//(P1 - D1) / NVM;
            MSE = (P2 - D2);
            x_med[1] += P2;
            x_stdev[1] += (P2 * P2);
            rxy[1] += (P2 * D2);
            wi = ((D2 - c_y_med[1]) / c_y_stdev[1]) * ((D2 - c_y_med[1]) / c_y_stdev[1]);
            s_cost[1] += wi * (MSE * MSE);

            MSE = (P3 - D3);
            x_med[2] += P3;
            x_stdev[2] += (P3 * P3);
            rxy[2] += (P3 * D3);
            wi = ((D3 - c_y_med[2]) / c_y_stdev[2]) * ((D3 - c_y_med[2]) / c_y_stdev[2]);
            s_cost[2] += wi * (MSE * MSE);

            value = value + 1;
         }

         t = t + h;

         RK4(Dm, G, i, s_pos, h,z);

      }

      //printf("%f\n",value);
      for (int k = 0; k < NVM-1; k++) {
         x_med[k] = x_med[k] / (samplesize);//média
         x_stdev[k] = sqrtf((x_stdev[k] - x_med[k] * x_med[k] * samplesize) / (samplesize
- 1));//desvio padrão
         rxy[k] = (rxy[k] - samplesize * x_med[k] * c_y_med[k]) / ((samplesize - 1) *
x_stdev[k] * c_y_stdev[k]);//coef Pearson
```

```
            //evitar que ocorra eventuais "crashes" por divizão por zero
            rxy[k] = d_max(0.001f, (rxy[k] + 1.0f) / 2.0f);
            s_cost[k] = (1 / rxy[k]) * s_cost[k] / samplesize;
            if (k > 0) {
               s_cost[0] += s_cost[k];
            }
         }

         cost[ind] = (s_cost[0]);//saída da função objetivo

      }

      //PSO

      //inicializa as partículas
      __global__ void particle_initialize(float* pos, float* vel, float* R1, float* R2, float*
pbest, float* lbest, int* l)
      {
         //definindo indices
         int ind = threadIdx.x + blockIdx.x * blockDim.x;


         //importante verificar se os dados se encontram dentro dos limites estabelecidos
         for (int k = 0; k < 9; k++) {

            //inicializando posição e velocidade
            pos[k + ind * 9] = c_lim[2 * k + 0] + R1[k + ind * 9] * (c_lim[2 * k + 1] - c_lim[2
* k + 0]);
            vel[k + ind * 9] = 0.01f*((c_lim[2 * k + 1] - c_lim[2 * k + 0]) / 2.0f) * (2.0f *
R2[k + ind * 9] - 1.0f);
         }

         pbest[ind] = INFINITY;
         lbest[ind] = INFINITY;
         l[ind] = ind;

      }

      //atualiza a posição e velocidade para o PSO Global
      __global__ void base_PSO_particle_PV(float* pos, float* vel, float* R1, float* R2,
float* theta_pbest, int* g, int n) {

         //definindo indices
         int ind = threadIdx.x + blockIdx.x * blockDim.x;


         for (int k = 0; k < 9; k++) {
            //atualizando velocidade e posição
            vel[k + ind * 9] = c_Wc1c2[0] * vel[k + ind * 9] + c_Wc1c2[1] * R1[k + ind * 9]
* (theta_pbest[k + ind * 9] - pos[k + ind * 9]) + c_Wc1c2[2] * R2[k + ind * 9] *
(theta_pbest[g[(blockIdx.x * blockDim.x) / N] * 9 + k] - pos[k + ind * 9]);

            pos[k + ind * 9] = pos[k + ind * 9] + vel[k + ind * 9];
```

```
            //verificando se as particulas sairam do espaço de busca
            if (pos[k + ind * 9] > c_lim[2 * k + 1] || pos[k + ind * 9] < c_lim[2 * k + 0]) {

               //caso em  que a partícula é maior que o limite superior
               if (pos[k + ind * 9] > c_lim[2 * k + 1]) {
                  pos[k + ind * 9] = c_lim[2 * k + 1];
                  vel[k + ind * 9] = -0.75f * (c_lim[2 * k + 1] - c_lim[2 * k + 0]) / 2.0f;
               }

               //caso em  que a partícula é menor que o limite inferior
               else {
                  pos[k + ind * 9] = c_lim[2 * k + 0];
                  vel[k + ind * 9] = 0.75f * (c_lim[2 * k + 1] - c_lim[2 * k + 0]) / 2.0f;
               }

            }
         }

      }

      //atualiza a posição e velocidade para outras topologias
      __global__ void local_PSO_particle_PV(float* pos, float* vel, float* R1, float* R2,
float* theta_pbest, int* l, int n) {

         //definindo indices
         int ind = threadIdx.x + blockIdx.x * blockDim.x;


         for (int k = 0; k < 9; k++) {
            //atualizando velocidade e posição
            vel[k + ind * 9] = c_Wc1c2[0] * vel[k + ind * 9] + c_Wc1c2[1] * R1[k + ind * 9]
* (theta_pbest[k + ind * 9] - pos[k + ind * 9]) + c_Wc1c2[2] * R2[k + ind * 9] *
(theta_pbest[l[ind] * 9 + k] - pos[k + ind * 9]);

            pos[k + ind * 9] = pos[k + ind * 9] + vel[k + ind * 9];

            //verificando se as particulas sairam do espaço de busca
            if (pos[k + ind * 9] > c_lim[2 * k + 1] || pos[k + ind * 9] < c_lim[2 * k + 0]) {

               //caso em  que a partícula é maior que o limite superior
               if (pos[k + ind * 9] > c_lim[2 * k + 1]) {
                  pos[k + ind * 9] = c_lim[2 * k + 1];
                  vel[k + ind * 9] = -0.01f * (c_lim[2 * k + 1] - c_lim[2 * k + 0]) / 2.0f;
               }

               //caso em  que a partícula é menor que o limite inferior
               else {
                  pos[k + ind * 9] = c_lim[2 * k + 0];
                  vel[k + ind * 9] = 0.01f * (c_lim[2 * k + 1] - c_lim[2 * k + 0]) / 2.0f;
               }

            }
```

```
        }

    }

    //compara Fobj e pbest
    __global__ void particle_pbest(float* cost, float* pbest, float* lbest, int* l, float* theta_pbest, float* pos, int n) {

        //definindo indices
        int ind = threadIdx.x + blockIdx.x * blockDim.x;

        //comparando custo a pbest
        if (cost[ind] < pbest[ind]) {

            for (int k = 0; k < 9; k++) {
                //atualizando os theta_pbest
                theta_pbest[k + ind * 9] = pos[k + ind * 9];
            }

            //atualizando pbest
            pbest[ind] = cost[ind];

        }
        if (lbest[ind] > pbest[ind]) {
            lbest[ind] = pbest[ind];
            l[ind] = ind;
        }
    }

    //Topologias

    //Circulo+Anel
    __global__ void circulo_PSO_find_lbest(float* pbest, float* lbest, int* l, int n, int r) {

        int i = threadIdx.x;
        int ind = threadIdx.x + blockIdx.x * blockDim.x;

        __shared__ float s_lbest[N];
        __shared__ float s_n_lbest[N];
        __shared__ int s_n_l[N];
        __shared__ int s_l[N];

        s_lbest[i] = lbest[ind];
        s_n_lbest[i] = lbest[ind];
        s_l[i] = l[ind];
        s_n_l[i] = l[ind];

        __syncthreads();

        for (int k = 1; k <= r / 2; k++) {
            if (s_lbest[i] > s_n_lbest[(N + i + k) % N]) {
                s_lbest[i] = s_n_lbest[(N + i + k) % N];
                s_l[i] = s_n_l[(N + i + k) % N];
```

```
            }
            if (s_lbest[i] > s_n_lbest[(N + i - k) % N]) {
                s_lbest[i] = s_n_lbest[(N + i - k) % N];
                s_l[i] = s_n_l[(N + i - k) % N];
            }
        }

        if (lbest[ind] > s_lbest[i]) {
            lbest[ind] = s_lbest[i];
            l[ind] = s_l[i];

        }
        if (s_l[i]<0 || s_l[i]>L * N - 1) {
            printf("%f\n", lbest[ind]);
            printf("%d\n", ind);
        }

    }

    //Von Neumann
    __global__ void von_neumann(float* pbest, float* lbest, int* l, int n) {

        int i = threadIdx.x;
        int j = threadIdx.y;


        __shared__ float s_lbest[N];
        __shared__ float s_n_lbest[N];
        __shared__ int s_n_l[N];
        __shared__ int s_l[N];

        s_lbest[i + blockDim.x * j] = lbest[i + blockDim.x * j + blockIdx.x*blockDim.x*blockDim.y];
        s_n_lbest[i + blockDim.x * j] = lbest[i + blockDim.x * j + blockIdx.x * blockDim.x * blockDim.y];
        s_l[i + blockDim.x * j] = l[i + blockDim.x * j + blockIdx.x * blockDim.x * blockDim.y];
        s_n_l[i + blockDim.x * j] = l[i + blockDim.x * j + blockIdx.x * blockDim.x * blockDim.y];

        __syncthreads();

        if (s_lbest[i + blockDim.x * j] > s_n_lbest[(blockDim.x + i + 1) % blockDim.x + blockDim.x * j]) {
            s_lbest[i + blockDim.x * j] = s_n_lbest[(blockDim.x + i + 1) % blockDim.x + blockDim.x * j];
            s_l[i + blockDim.x * j] = s_n_l[(blockDim.x + i + 1) % blockDim.x + blockDim.x * j];
        }
        if (s_lbest[i + blockDim.x * j] > s_n_lbest[(blockDim.x + i - 1) % blockDim.x + blockDim.x * j]) {
            s_lbest[i + blockDim.x * j] = s_n_lbest[(blockDim.x + i - 1) % blockDim.x + blockDim.x * j];
```

```
            s_l[i + blockDim.x * j] = s_n_l[(blockDim.x + i - 1) % blockDim.x + blockDim.x
* j];
        }
        if (s_lbest[i + blockDim.x * j] > s_n_lbest[i + blockDim.x * ((blockDim.y + j + 1) %
blockDim.y)]) {
            s_lbest[i + blockDim.x * j] = s_n_lbest[i + blockDim.x * ((blockDim.y + j + 1) %
blockDim.y)];
            s_l[i + blockDim.x * j] = s_n_l[i + blockDim.x * ((blockDim.y + j + 1) %
blockDim.y)];
        }
        if (s_lbest[i + blockDim.x * j] > s_n_lbest[i + blockDim.x * ((blockDim.y + j - 1) %
blockDim.y)]) {
            s_lbest[i + blockDim.x * j] = s_n_lbest[i + blockDim.x * ((blockDim.y + j - 1) %
blockDim.y)];
            s_l[i + blockDim.x * j] = s_n_l[i + blockDim.x * ((blockDim.y + j - 1) %
blockDim.y)];
        }

        if (lbest[i + blockDim.x * j + blockIdx.x * blockDim.x * blockDim.y] > s_lbest[i +
blockDim.x * j]) {
            lbest[i + blockDim.x * j + blockIdx.x * blockDim.x * blockDim.y] = s_lbest[i +
blockDim.x * j];
            l[i + blockDim.x * j + blockIdx.x * blockDim.x * blockDim.y] = s_l[i +
blockDim.x * j];
        }

    }

    //Global
    __global__ void base_PSO_find_gbest(float* pbest, float* gbest, int* g, int n) {

        //definindo indices
        int i = threadIdx.x;
        int ind = threadIdx.x + blockDim.x * blockIdx.x;

        //inicializando memória compartilhada
        __shared__ int s_gind[N];
        __shared__ float s_gbest[N];

        //passando os dados da memória global para compartilhada
        s_gbest[i] = pbest[ind];
        s_gind[i] = ind;

        __syncthreads();

        //algoritmo de reduce para obter o gbest dentre os pbest
        for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
            if (i < stride) {
                if (s_gbest[i] > s_gbest[i + stride]) {
                    s_gbest[i] = s_gbest[i + stride];
                    s_gind[i] = s_gind[i + stride];
                }
```

```
            }
            __syncthreads();
        }


        //passando o "gbest do bloco" da memória local para a global.Cada bloco lida com 1024 pbests, assim
        //é necessário lançar um kernel subsequente para obter gbest dentre os "gbest do bloco"
        if (i == 0) {
            if (gbest[blockIdx.x] > s_gbest[0]) {
                gbest[blockIdx.x] = s_gbest[0];
                g[blockIdx.x] = s_gind[0];
            }
        }
    }

    //Utilizado para casos em que o número de partículas por enxame é maior que 1024
    __global__ void circulo_PSO_find_lbest2(float* lbest, int* l, float* lbest_temp, int* l_temp) {

        int i = threadIdx.x;
        int ind = threadIdx.x + blockIdx.x * blockDim.x;

        __shared__ float s_lbest[STPB];
        __shared__ float s_lbest_r[STPB];
        __shared__ float s_lbest_l[STPB];


        __shared__ int s_l[STPB];
        __shared__ int s_l_r[STPB];
        __shared__ int s_l_l[STPB];

        s_lbest[i] = lbest_temp[ind];
        s_l[i] = l_temp[ind];

        s_lbest_r[i] = lbest_temp[(N * L + ind + 1) % (N * L)];
        s_l_r[i] = l_temp[(N * L + ind + 1) % (N * L)];

        s_lbest_l[i] = lbest_temp[(N * L + ind - 1) % (N * L)];
        s_l_l[i] = l_temp[(N * L + ind - 1) % (N * L)];

        if (s_lbest[i] > s_lbest_r[i]) {
            s_lbest[i] = s_lbest_r[i];
            s_l[i] = s_l_r[i];
        }

        if (s_lbest[i] > s_lbest_l[i]) {
            s_lbest[i] = s_lbest_l[i];
            s_l[i] = s_l_l[i];
        }
```

```
    if (lbest[ind] > s_lbest[i]) {
        lbest[ind] = s_lbest[i];
        l[ind] = s_l[i];
    }
}

//Migracao
//Troca o pior pbest do enxame pelo melhor pbest dentre todos os enxames
__global__ void migracao_global(float* theta_pbest,float* pbest, float* gbest, int* g) {

    //definindo indices
    int i = threadIdx.x;
    int ind = threadIdx.x + blockDim.x * blockIdx.x;

    //inicializando memória compartilhada
    __shared__ int s_pind[N];
    __shared__ float s_pbest[N];

    //passando os dados da memória global para compartilhada
    s_pbest[i] = pbest[ind];
    s_pind[i] = ind;

    int gmin;

    if (i == 0) {
        float gtest = gbest[0];
        gmin = g[0];

        for (int k = 1; k < gridDim.x; k++) {
            if (gtest > gbest[k]) {
                gtest = gbest[k];
                gmin = g[k];
            }
        }
    }

    __syncthreads();

    //algoritmo de reduce para obter o gbest dentre os pbest
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (i < stride) {
            if (s_pbest[i] < s_pbest[i + stride]) {
                s_pbest[i] = s_pbest[i + stride];
                s_pind[i] = s_pind[i + stride];
            }

        }
        __syncthreads();
    }
```

```
        //passando o "gbest do bloco" da memória local para a global.Cada bloco lida com
1024 pbests, assim
        //é necessário lançar um kernel subsequente para obter gbest dentre os "gbest do
bloco"
        if (i == 0) {
            pbest[s_pind[0]] = pbest[gmin];
            for (int k=0;k<9;k++){
               theta_pbest[9*s_pind[0] + k] = theta_pbest[9*gmin+k];
            }
          }
    }




    //funções na cpu
    //calcula média e desvio padrão dos dados fornecidos (experimental ou simulador
rigoroso)
    __host__ void y_medstdev(float* Dados, int ind_stop, float* y_med, float* y_stdev) {

        int ind = 0;

        for (int k = 0; k < NVM; k++) {
           y_med[k] = 0.0f;
           y_stdev[k] = 0.0f;
        }

        while (ind<ind_stop) {

           for (int k = 0; k < NVM; k++) {

              y_med[k] += Dados[(NVM + 1) * ind + k + 1];
              y_stdev[k] += Dados[(NVM + 1) * ind + k + 1] * Dados[(NVM + 1) * ind + k
+ 1];

           }
           ind = ind + 1;
    }
        //ind = ind - 1;
        for (int k = 0; k < NVM; k++) {
           y_stdev[k] = sqrtf((y_stdev[k] - y_med[k] * y_med[k] / ind) / (ind - 1));
           y_med[k] = y_med[k] / ind;
        }
    }

    //função para salvar dados para .csv (fonte: http://codingstreet.com/create-csv-file-in-
c/)
    __host__ void create_marks_csv(char* filename,  const int m, const int n, float
dados[]) {

        printf("\n Creating %s file", filename);
```

```
    FILE* f;

    int i, j;

    //filename = strcat(filename, ".csv");

    f = fopen(filename, "w+");


    for (i = 0; i < m; i++) {
       for (j = 0; j < n; j++) {

          fprintf(f, "%f,", dados[i * n + j]);
       }
       fprintf(f, "\n");
    }

    fclose(f);

    printf("\n %sfile created\n", filename);

}


int main()
{
    //reinicia a GPU por precaucao
    cudaDeviceReset();

    //declaracao das constantes da planta
    float param_planta_ag[12];

    float  rhol = 900.0f; //kg / m3
    float  Pr = 225e5f; //Pa
    float  Ps = 10e5f; //Pa
    float  alphagw = 0.0188f; //adm
    float  rhomres = 892.0f; //kg / m3
    float  M = 18.0f; //kg / kmol
    float  T = 298.0f; //K
    float  Lr = 1569.0f; //m
    float  Lft = 2928.0f; //m
    float  Lt = 1639.0f; //m
    float  La = 1118.0f; //m
    float  Ht = 1279.0f; //m
    float  Hpdg = 1117.0f; //m
    float  Hvgl = 916.0f; //m
    float  Dss = 0.15f; //m
    float  Dt = 0.15f; //m
    float  Da = 0.14f; //m
    float  g = 9.81f;//m/s2
    float  R = 8314.0f;//J/mol*K

    float Ass = (PI * Dss * Dss) / 4.0f;
```

```
float Vss = (Ass * (Lr + Lft));
float Va = PI * Da * Da * La / 4.0f;
float Vt = PI * Dt * Dt * Lt / 4.0f;
float theta = PI / 4.0f;

//Essa etapa simplesmente precomputa alguns valores, como RT/M por exemplo
param_planta_ag[0] = R * T / (M * Va) + g * La / Va;
param_planta_ag[1] = R * T / M;
param_planta_ag[2] = g * Hvgl;
param_planta_ag[3] = rhomres * g * (Ht - Hpdg);
param_planta_ag[4] = rhomres * g * (Hpdg - Hvgl);
param_planta_ag[5] = g * sinf(theta) / Ass;
param_planta_ag[6] = Pr;
param_planta_ag[7] = Vt;
param_planta_ag[8] = rhol;
param_planta_ag[9] = Vss;
param_planta_ag[10] = Ps;
param_planta_ag[11] = alphagw;

//copia as constantes da planta para a memoria constante
cudaMemcpyToSymbol(c_param_planta_ag, param_planta_ag, sizeof(param_planta_ag));

//leitura dos dados experimentais de arquivo do tipo .csv (Fonte: https://stackoverflow.com/questions/46384395/how-to-move-values-from-a-csv-file-to-a-float-array#46384495
//Importante: assume-se que o arquivo fornecido será do tipo .csv com NVM+1 colunas, sendo:
//1a coluna contem o tempo
//2a coluna até a coluna NVM contem as pressões na ordem (Ppdg,Ptt,Prt)
//A última coluna contem a abertura da valvula

FILE* fp;
fp = fopen("OLGA3.csv", "r");
float *Dados;
Dados = (float*)malloc(samplesize * (NVM+1) * sizeof(float));

int Dados_ind = 0;

if (fp == NULL) {
   printf("failed to open file\n");
   return 1;
}

while (fscanf(fp, "%f", &Dados[Dados_ind++]) == 1) {
   fscanf(fp, ",");
}
fclose(fp);

int ind = 0;
float D0 = Dados[0]-2000;

while (ind <= samplesize - 1) {
```

```
            //deslocando o tempo para que t0 = 0
            Dados[(NVM + 1) * ind] = (Dados[(NVM + 1) * ind] - D0) > 0 ? Dados[(NVM + 1) * ind] - D0 : 0;

            ind = ind + 1;
        }
        //

        //transfere os dados para o device
        float* d_Dados;
        cudaMalloc((void**)&d_Dados, samplesize * (NVM + 1) * sizeof(float));
        cudaMemcpy(d_Dados, Dados, samplesize * (NVM + 1) * sizeof(float), cudaMemcpyHostToDevice);

        //definindo limites de busca:
        float lim[18];

        //lim inf
        for (int i = 0; i < 9; i++) {
            lim[i * 2 + 0] = 0.0f;
        }

        //lim sup
        lim[0 * 2 + 1] = 5e3f;
        lim[1 * 2 + 1] = 1e-2f;
        lim[2 * 2 + 1] = 1e-2f;
        lim[3 * 2 + 1] = 5e2f;
        lim[4 * 2 + 1] = 1e0f;
        lim[5 * 2 + 1] = 1e-2f;
        lim[6 * 2 + 1] = 1e-2f;
        lim[7 * 2 + 1] = 1e3f;
        lim[8 * 2 + 1] = 1e1f;

        //imprime os limites para melhor controle
        for (int i = 0; i < 9; i++) {
            printf("lim_inf param[%d] = %f",i,lim[i * 2 + 0]);
            printf("    lim_sup param[%d] = %f\n",i,lim[i * 2 + 1]);
        }

        //transferindo limites para memoria constante
        cudaMemcpyToSymbol(c_lim, lim, sizeof(lim));

        //definindo coeficientes de inercia, coercao e nostalgia (utilizada a versão do CF-PSO (coef de constrição))
        float c1 = 1.0f * 4.1f / 2.0f;
        float c2 = 4.1f - c1;
        float X = 2.0f / fabs(2.0f - (c1 + c2) - sqrtf((c1 + c2) * (c1 + c2) - 4 * (c1 + c2)));
        float Wc1c2[3] = { X ,X * c1 ,X * c2 };

        //imprimindo coeficientes para controle
        printf("X = %f\n", X);
        printf("c1 = %f\n", X * c1);
```

```
printf("c2 = %f\n", X * c2);
cudaMemcpyToSymbol(c_Wc1c2, Wc1c2, sizeof(Wc1c2));

//Controle da simulacao
//char* nome = "PSODiehl\.csv"; //nome do arquivo de saída
const int n = L*N; //número total de particulas
int a = n / STPB; //dimensionamento do bloco
const int n_iter = 10000; //numero máximo de iterações
const int n_run = 1; //número de vezes que o algoritmo vai rodar
int r = 2; //parametro da topologia "circulo"
float h = 5.0f; //passo de integração
float t_stop = Dados[(samplesize-1) * (NVM+1)]; //ultimo valor de tempo dos dados
printf("t_stop = %f\n", t_stop);

//usado para topologia Von Neumann (agrupa em uma grade)
int base = int(log2(N)) / 2;
int altura = int(log2(N)) - base;
base = 1 << base;
altura = 1 << altura;

//dimensionamento blocos/grade
dim3 TPB_VN(base, altura, 1);
dim3 NB_VN(n/N, 1, 1);

//declarando variaveis host
float* theta_pbest;
float* gbest;
int* gind;
float* y_med;
float* y_stdev;

//alocando memoria para o host
theta_pbest = (float*)malloc(9 * n * sizeof(float));
gbest = (float*)malloc(n / N * sizeof(float));
gind = (int*)malloc(n / N * sizeof(int));
y_med = (float*)malloc(NVM * sizeof(float));
y_stdev = (float*)malloc(NVM * sizeof(float));

//computa media e desvio dos dados (usado em Fobj_Diehl)
y_medstdev(Dados,samplesize,y_med,y_stdev);

//declarando variaveis device
float* d_cost;
float* d_pos;
float* d_vel;
float* d_theta_pbest;
float* d_pbest;
float* d_gbest;
int* d_gind;
float* d_lbest;
int* d_l;

//valores usados para o caso em que o tamanho do enxame excede 1024 partículas
```

```
float* d_lbest_temp;
int* d_l_temp;

float* d_R1;
float* d_R2;

//declarando vetor de armazenamento (para gerar o arquivo .csv com a saida dos dados)
float* out_gbest;
out_gbest = (float*)malloc((n / N) * (n_iter/100 +1)*sizeof(float));
float* out_time_iter;
out_time_iter = (float*)malloc(n_run*2 * sizeof(float));

//Início da simulação
for (int run=0;run<n_run;run++){

   printf("\nExecucao de numero %d:\n", run + 1);

//gerador de numeros "aleatorios"
curandGenerator_t gen;
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
curandSetPseudoRandomGeneratorSeed(gen, time(NULL));


//medindo tempo
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

//alocando memoria para o device
cudaMalloc((void**)&d_cost, n * sizeof(float));
cudaMalloc((void**)&d_pos, 9 * n * sizeof(float));
cudaMalloc((void**)&d_vel, 9 * n * sizeof(float));
cudaMalloc((void**)&d_theta_pbest, 9 * n * sizeof(float));
cudaMalloc((void**)&d_pbest, n * sizeof(float));
cudaMalloc((void**)&d_gbest, (n / N) * sizeof(float));
cudaMalloc((void**)&d_gind, (n / N) * sizeof(int));
cudaMalloc((void**)&d_lbest, n * sizeof(float));
cudaMalloc((void**)&d_l, n * sizeof(int));

cudaMalloc((void**)&d_lbest_temp, n * sizeof(float));
cudaMalloc((void**)&d_l_temp, n * sizeof(int));

cudaMalloc((void**)&d_R1, 9 * n * sizeof(float));
cudaMalloc((void**)&d_R2, 9 * n * sizeof(float));

//copiando media e desvio dos dados para a memoria constante
cudaMemcpyToSymbol(c_y_med, y_med, NVM * sizeof(float));
cudaMemcpyToSymbol(c_y_stdev, y_stdev, NVM * sizeof(float));

//imprimindo media e desvio para controle
for (int k = 0; k < NVM-1; k++) {
   printf("med[%d] = %f\n", k + 1, y_med[k]);
```

```
            printf("stdev[%d] = %f\n", k + 1, y_stdev[k]);
        }

        //inicializa gbest
        for (int k = 0; k < n / N; k++) {
            gbest[k] = INFINITY;
        }

        //copia gbest para device
        cudaMemcpy(d_gbest, gbest, (n / N) * sizeof(float), cudaMemcpyHostToDevice);

        //inicia a contagem do tempo
        cudaEventRecord(start, 0);

        //gera os numeros aleatorios usados na inicializacao
        curandGenerateUniform(gen, d_R1, n * 9);
        curandGenerateUniform(gen, d_R2, n * 9);

        //inicializa as posicoes,velocidades e pbests/lbests
        particle_initialize << <a, n / a >> > (d_pos, d_vel, d_R1, d_R2, d_pbest, d_lbest,
d_l);

        //variavel que verifica se o criterio de convergencia foi atingido
        int check = 0;

        for (int iter = 0; iter <= n_iter; iter++) {

            //computa funcao objetivo (descomentar apenas 1)
            Fobj << <a, n / a >> > (d_pos, h, d_cost, d_Dados, t_stop); //MSE
            //Fobj_Diehl << <a, n / a >> > (d_pos, h, d_cost, d_Dados, t_stop);

            //verifica se cost < pbest
            particle_pbest << <a, n / a >> > (d_cost, d_pbest, d_lbest, d_l, d_theta_pbest,
d_pos, n);

            //topologia (descomentar apenas 1)
            //base_PSO_find_gbest<<<n/N,N>>>(d_pbest, d_gbest, d_gind, n);
            circulo_PSO_find_lbest << <n / N, N >> > (d_pbest, d_lbest, d_l, n, r);
            //von_neumann<<<NB_VN,TPB_VN>>>(d_pbest,d_lbest,d_l,n);

            //cudaMemcpy(d_lbest_temp, d_lbest, n * sizeof(float),
cudaMemcpyDeviceToDevice);
            //cudaMemcpy(d_l_temp, d_l, n * sizeof(int), cudaMemcpyDeviceToDevice);
            //circulo_PSO_find_lbest2 << <a, n / a >> > (d_lbest, d_l, d_lbest_temp,
d_l_temp);

            //cudaMemcpy(d_lbest, d_lbest_temp, n * sizeof(float),
cudaMemcpyDeviceToDevice);
            //cudaMemcpy(d_l, d_l_temp, n * sizeof(int), cudaMemcpyDeviceToDevice);

            //gera os numeros aleatorios usados no calculo da velocidade
            curandGenerateUniform(gen, d_R1, n * 9);
            curandGenerateUniform(gen, d_R2, n * 9);
```

```
        //calcula velocidade e posicao (descomentar apenas 1)
        //base_PSO_particle_PV << <a, n / a >> > (d_pos, d_vel, d_R1, d_R2,
d_theta_pbest, d_gind, n);
        local_PSO_particle_PV << <a, n / a >> > (d_pos, d_vel, d_R1, d_R2,
d_theta_pbest, d_l, n);

        if (iter % 100 == 0) {

            //copia o gbest para a CPU para verificar convergencia
            base_PSO_find_gbest << <n / N, N >> > (d_pbest, d_gbest, d_gind, n);
            cudaMemcpy(gbest, d_gbest, (n / N) * sizeof(float),
cudaMemcpyDeviceToHost);

            printf("========================\n");

            for (int k = 0; k < n / N; k++) {
                /*
                if (gbest[0] > gbest[k]) {
                    gbest[0] = gbest[k];
                    gind[0] = gind[k];
                }
                */
                printf("gbest da simulacao %d na iteracao %d: %f\n", k + 1, iter, gbest[k]);
                //salva os valores de gbest para serem escritos posteriormente em um arquivo
.csv
                //out_gbest[(iter/100)*(n / N)+k]= gbest[k];

                //verifica se o criterio de convergencia foi atingido
                if (gbest[k] < 4.0e10) {//4.0e10 p/ Fobj e 1.2e12 p/ Fobj_Pearson
                    check = 1;

                }
            }
            //printf("gbest da simulacao na iteracao %d: %f\n", iter, gbest[0]);
            printf("========================\n");

            //migracao
            migracao_global<<<n / N, N >>>(d_theta_pbest, d_pbest, d_gbest, d_gind);

            //se o criterio foi atingido encerra o loop
            if (check==1){
                out_time_iter[2 * run] = float(iter);
                break;
            }
        }
        if (iter == n_iter) {
            out_time_iter[2 * run] = float(iter);
        }

    }
```

```
        //salva os dados obtidos num arquivo .csv
        //create_marks_csv(nome,n_iter / 100 + 1, n / N, out_gbest);


        //copia os parametros e gbest de volta para a CPU
        cudaMemcpy(theta_pbest, d_theta_pbest, 9 * n * sizeof(float),
cudaMemcpyDeviceToHost);
        base_PSO_find_gbest << <n / N, N >> > (d_pbest, d_gbest, d_gind, n);
        cudaMemcpy(gbest, d_gbest, (n / N) * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(gind, d_gind, (n / N) * sizeof(int), cudaMemcpyDeviceToHost);
        for (int k = 0; k < n / N; k++) {
           if (gbest[0] > gbest[k]) {
              gbest[0] = gbest[k];
              gind[0] = gind[k];
           }
        }

        //para a contagem do tempo
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        float ElapsedTime;
        cudaEventElapsedTime(&ElapsedTime, start, stop);
        float tempo = ElapsedTime;

        out_time_iter[2 * run + 1] = tempo;


        //imprime o tempo de simulacao
        printf("Tempo de Simulacao: %f ms\n", tempo);


    //imprime resultados (parametros e gbest)
    //for (int i = 0; i < n / N; i++) {
        for (int i = 0; i < 1; i++) {
        printf("========================\n");
        printf("Simulacao %d:\n", i + 1);
        for (int k = 0; k < 9; k++) {
           printf("param[%d]= %.10f\n", k, theta_pbest[gind[0] * 9 + k]);
        }
        printf("gbest: %f\n", gbest[0]);
        printf("========================\n");
    }

    }

        //create_marks_csv(nome, n_run, 2, out_time_iter);


        //reinicia a GPU
        cudaDeviceReset();

        return 0;
    }
```

Apêndice B: Implementação do PSO aplicado ao ajuste de parâmetros do FOWM em C (Sequencial)

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include<cuda.h>
#include <curand.h>
#include <time.h>
#include<string.h>


#define PI (3.14159265358979323846f)

#define z (0.22f) //abertura da choke
#define  Wgc  (float(80000.0*(101325.0*18.0)/(293.0*8314.0*24.0*3600.0)))  //vazao de gas lift convertida de m3/dia para kg/s
#define NVM (3) //"Numero de variaveis Medidas"
#define samplesize (1440) //tamanho da amostra (i.e. No de pontos)
#define STPB (128) //tamanho do bloco
#define N (1024) //umero de particulas

//memoria constante
float c_Wc1c2[3] = { 072,1.4,1.4 };//coef de inercia, nostalgia e coercao
float c_lim[18];//limites superior e inferior de cada dimensao
float c_y_med[NVM];//media dos dados para cada VM
float c_y_stdev[NVM];//desvio dos dados para cada VM

float c_param_planta_ag[12];//parametros da planta

//definindo funções do modelo
__host__ float d_max(float x, float y) {
   if (x > y) {
      return x;
   }
   else {
      return y;
   }
}

__host__ float Vgt(float* m, int i, float* param_modelo) {

   float Vgt = c_param_planta_ag[7] - m[6 * i + 2] / c_param_planta_ag[8];
   return Vgt;
}
```

```
__host__ float rhogt(float* m, int i, float* param_modelo) {

   float rhogt = m[6 * i + 1] / Vgt(m, i, param_modelo);
   return rhogt;
}

__host__ float Pai(float* m, int i, float* param_modelo) {

   float Pai = c_param_planta_ag[0] * m[6 * i + 0];
   return Pai;
}

__host__ float alphagt(float* m, int i, float* param_modelo) {

   float alphagt = m[6 * i + 1] / (m[6 * i + 1] + m[6 * i + 2]);
   return alphagt;
}

__host__ float alphagr(float* m, int i, float* param_modelo) {

   float alphagr = m[6 * i + 4] / (m[6 * i + 4] + m[6 * i + 5]);
   return alphagr;
}

__host__ float Peb(float* m, int i, float* param_modelo) {

   float Peb = m[6 * i + 3] * c_param_planta_ag[1] / param_modelo[9 * i + 3];
   return Peb;
}

__host__ float rhomt(float* m, int i, float* param_modelo) {

   float rhomt = (m[6 * i + 1] + m[6 * i + 2]) / c_param_planta_ag[7];
   return rhomt;
}

__host__ float alphalr(float* m, int i, float* param_modelo) {

   float alphalr = 1 - alphagr(m, i, param_modelo);
   return alphalr;
}

__host__ float Ptt(float* m, int i, float* param_modelo) {

   float Ptt = rhogt(m, i, param_modelo) * c_param_planta_ag[1];
   return Ptt;
}

__host__ float rhoai(float* m, int i, float* param_modelo) {
```

```
    float rhoai = Pai(m, i, param_modelo) / c_param_planta_ag[1];
    return rhoai;
}

__host__ float Prt(float* m, int i, float* param_modelo) {

    float Prt = m[6 * i + 4] * c_param_planta_ag[1] / (param_modelo[9 * i + 8] *
c_param_planta_ag[9] - (m[6 * i + 5] + param_modelo[9 * i + 0]) /
c_param_planta_ag[8]);
    return Prt;
}

__host__ float Prb(float* m, int i, float* param_modelo) {

    float Prb = Prt(m, i, param_modelo) + (m[6 * i + 5] + param_modelo[9 * i + 0]) *
c_param_planta_ag[5];
    return Prb;
}

__host__ float Ptb(float* m, int i, float* param_modelo) {

    float Ptb = Ptt(m, i, param_modelo) + rhomt(m, i, param_modelo) *
c_param_planta_ag[2];
    return Ptb;
}

__host__ float Ppdg(float* m, int i, float* param_modelo) {

    float Ppdg = Ptb(m, i, param_modelo) + c_param_planta_ag[4];
    return Ppdg;
}

__host__ float Pbh(float* m, int i, float* param_modelo) {

    float Pbh = Ppdg(m, i, param_modelo) + c_param_planta_ag[3];
    return Pbh;
}

//Vazões
__host__ float Wiv(float* m, int i, float* param_modelo) {

    float Wiv = param_modelo[9 * i + 6] * sqrtf(d_max(0.0f, rhoai(m, i, param_modelo)
* (Pai(m, i, param_modelo) - Ptb(m, i, param_modelo))));
    return Wiv;
}

__host__ float Wr(float* m, int i, float* param_modelo) {
```

```
    float Wr = d_max(0.0f, param_modelo[9 * i + 7] * (1.0f - (0.2f * Pbh(m, i, param_modelo) / c_param_planta_ag[6]) - ((0.8f * Pbh(m, i, param_modelo) / c_param_planta_ag[6]) * (Pbh(m, i, param_modelo) / c_param_planta_ag[6]))));
    return Wr;
}

__host__ float Wwhg(float* m, int i, float* param_modelo) {

    float Wwhg = param_modelo[9 * i + 5] * sqrtf(d_max(0.0f, c_param_planta_ag[8] * (Ptt(m, i, param_modelo) - Prb(m, i, param_modelo)))) * alphagt(m, i, param_modelo);
    return Wwhg;
}

__host__ float Wwhl(float* m, int i, float* param_modelo) {

    float Wwhl = param_modelo[9 * i + 5] * sqrtf(d_max(0.0f, c_param_planta_ag[8] * (Ptt(m, i, param_modelo) - Prb(m, i, param_modelo))) * (1 - alphagt(m, i, param_modelo)));
    return Wwhl;
}

__host__ float Wg(float* m, int i, float* param_modelo) {

    float Wg = param_modelo[9 * i + 1] * d_max(0.0f, (Peb(m, i, param_modelo) - Prb(m, i, param_modelo)));
    return Wg;
}

__host__ float Wgout(float* m, int i, float* param_modelo) {

    float Wgout = alphagr(m, i, param_modelo) * param_modelo[9 * i + 2] * z * sqrtf(d_max(0.0f, c_param_planta_ag[8] * (Prt(m, i, param_modelo) - c_param_planta_ag[10])));
    return Wgout;
}

__host__ float Wlout(float* m, int i, float* param_modelo) {

    float Wlout = alphalr(m, i, param_modelo) * param_modelo[9 * i + 2] * z * sqrtf(d_max(0.0f, c_param_planta_ag[8] * (Prt(m, i, param_modelo) - c_param_planta_ag[10])));
    return Wlout;
}

//EDOs

__host__ void Dm(float* out, float* m, float* param_modelo, int i) {

    float Pai = c_param_planta_ag[0] * m[0];
    float Vgt = c_param_planta_ag[7] - m[2] / c_param_planta_ag[8];
```

```
        float alphagt = m[1] / (m[1] + m[2]);
        float alphagr = m[4] / (m[4] + m[5]);
        float Peb = m[3] * c_param_planta_ag[1] / param_modelo[9 * i + 3];
        float rhomt = (m[1] + m[2]) / c_param_planta_ag[7];
        float Prt = m[4] * c_param_planta_ag[1] / (param_modelo[9 * i + 8] *
c_param_planta_ag[9] - (m[5] + param_modelo[9 * i + 0]) / c_param_planta_ag[8]);

        float rhoai = Pai / c_param_planta_ag[1];
        float Prb = Prt + (m[5] + param_modelo[9 * i + 0]) * c_param_planta_ag[5];
        float rhogt = m[1] / Vgt;
        float alphalr = 1.0f - alphagr;
        float Wg = param_modelo[9 * i + 1] * ((0.0f > (Peb - Prb)) ? 0.0f : (Peb - Prb));
        float Wgout = alphagr * param_modelo[9 * i + 2] * z * sqrtf((0.0f >
(c_param_planta_ag[8] * (Prt - c_param_planta_ag[10]))) ? 0.0f : (c_param_planta_ag[8]
* (Prt - c_param_planta_ag[10])));

        float Ptt = rhogt * c_param_planta_ag[1];
        float Wlout = alphalr * param_modelo[9 * i + 2] * z * sqrtf((0.0f >
(c_param_planta_ag[8] * (Prt - c_param_planta_ag[10]))) ? 0.0f : (c_param_planta_ag[8]
* (Prt - c_param_planta_ag[10])));

        float Ptb = Ptt + rhomt * c_param_planta_ag[2];
        float Wwhg = param_modelo[9 * i + 5] * sqrtf((0.0f > c_param_planta_ag[8] * (Ptt -
Prb)) ? 0.0f : c_param_planta_ag[8] * (Ptt - Prb)) * alphagt;
        float Wwhl = param_modelo[9 * i + 5] * sqrtf((0.0f > c_param_planta_ag[8] * (Ptt -
Prb)) ? 0.0f : c_param_planta_ag[8] * (Ptt - Prb)) * (1.0f - alphagt);

        float Ppdg = Ptb + c_param_planta_ag[4];

        float Pbh = Ppdg + c_param_planta_ag[3];
        float Wiv = param_modelo[9 * i + 6] * sqrtf((0.0f > rhoai * (Pai - Ptb)) ? 0.0f : rhoai *
(Pai - Ptb));

        float Wr = param_modelo[9 * i + 7] * ((0.0f > (1.0f - (0.2f * Pbh /
c_param_planta_ag[6]) - ((0.8f * Pbh / c_param_planta_ag[6]) * (Pbh /
c_param_planta_ag[6])))) ? 0.0f : (1.0f - (0.2f * Pbh / c_param_planta_ag[6]) - ((0.8f * Pbh
/ c_param_planta_ag[6]) * (Pbh / c_param_planta_ag[6]))));

        out[0] = Wgc - Wiv;
        out[1] = Wr * c_param_planta_ag[11] + Wiv - Wwhg;
        out[2] = Wr * (1.0f - c_param_planta_ag[11]) - Wwhl;
        out[3] = (1.0f - param_modelo[9 * i + 4]) * Wwhg - Wg;
        out[4] = param_modelo[9 * i + 4] * Wwhg + Wg - Wgout;
        out[5] = Wwhl - Wlout;
    }

    __host__ void RK4(void(*Dm)(float*, float*, float*, int), float* G, int i, float* d_pos,
float h) {

        float temp[6];
```

```
        float K0[6];
        float K1[6];
        float K2[6];
        float K3[6];


        for (int k = 0; k < 6; k++) {
           temp[k] = G[6 * i + k];
        }

        Dm(K0, temp, d_pos, i);
        for (int k = 0; k < 6; k++) {
           temp[k] = G[6 * i + k] + h * K0[k] / 2.0f;
        }

        Dm(K1, temp, d_pos, i);

        for (int k = 0; k < 6; k++) {
           temp[k] = G[6 * i + k] + h * K1[k] / 2.0f;
        }

        Dm(K2, temp, d_pos, i);

        for (int k = 0; k < 6; k++) {
           temp[k] = G[6 * i + k] + h * K2[k];
        }

        Dm(K3, temp, d_pos, i);

        for (int k = 0; k < 6; k++) {
           G[6 * i + k] = G[6 * i + k] + h * (K0[k] + 2.0f * K1[k] + 2.0f * K2[k] + K3[k]) / 6.0f;
        }
    }

    //Fobj

    __host__ void Fobj(float* pos, float passo, float* cost, float* Dados, float t_stop,int
ind)
    {

        int i = 0;
        float t = 0.0f;
        float h = passo;

        float s_cost = 0.0f;

        float G[6];

        G[0] = 2805.14932589;
```

```
G[1] = 2125.01254427;
G[2] = 9103.27649409;
G[3] = 7857.40023097;
G[4] = 1316.40535031;
G[5] = 36810.25978214;


float s_pos[9];

for (int k = 0; k < 9; k++) {
   s_pos[k] = pos[9 * ind + k];
   //printf("%f\n", s_pos[k]);
}

int value = 0;

while (t <= t_stop) {


   if (value < samplesize && Dados[(NVM + 1) * value + 0] <= t + h) {

      float D0 = Dados[(NVM + 1) * value + 0];
      float D1 = Dados[(NVM + 1) * value + 1];
      //float D2 = Dados[(NVM + 1) * value + 2];
      //float D3 = Dados[(NVM + 1) * value + 3];

      float P1 = Ppdg(G, i, s_pos) * (t + h - D0);
      //float P2 = Ptt(G, i, s_pos) * (t + h - D0);
      //float P3 = Prt(G, i, s_pos) * (t + h - D0);

      t = t + h;

      RK4(Dm, G, i, s_pos, h);

      //essa etapa é responsável pela interpolação linear

      P1 = (P1 + Ppdg(G, i, s_pos) * (D0 - t + h)) / h;
      //P2 = (P2 + Ptt(G, i, s_pos) * (D0 - t + h)) / h;
      //P3 = (P3 + Prt(G, i, s_pos) * (D0 - t + h)) / h;


      //calculo dos valores necessários para a função objetivo
      float MSE1 = ((P1 - D1));// / NVM;//(P1 - D1) / NVM;
      //float MSE2 = ((P2 - D2)) / NVM;
      //float MSE3 = ((P3 - D3)*2.0f) / NVM;
      s_cost += (MSE1 * MSE1);// +(MSE2 * MSE2) + (MSE3 * MSE3);


      value = value + 1;
   }
```

```
            t = t + h;

            RK4(Dm, G, i, s_pos, h);

        }

        //printf("%d\n",value);
        cost[ind] = s_cost / value; // float(value);// / ((value - 1) / 100);
        //printf("%f\n", cost[ind]);
                            //printf("%d",(value-1)/20);

    }

    __host__ void Fobj_Diehl(float* pos, float passo, float* cost, float* Dados, float t_stop, int ind)
    {

        int i = 0;
        float t = 0.0f;
        float h = passo;
        float MSE;

        float rxy[NVM];
        float x_med[NVM];
        float x_stdev[NVM];
        float wi;
        float s_cost[NVM];

        for (int k = 0; k < NVM; k++) {
            rxy[k] = 0.0f;
            x_med[k] = 0.0f;
            x_stdev[k] = 0.0f;
            s_cost[k] = 0.0f;
        }

        //float s_cost = 0.0f;

        float m[6];

        m[0] = 2805.14932589;
        m[1] = 2125.01254427;
        m[2] = 9103.27649409;
        m[3] = 7857.40023097;
        m[4] = 1316.40535031;
        m[5] = 36810.25978214;

        float G[6];
        float s_pos[9];
```

```
for (int k = 0; k < 9; k++) {
   s_pos[k] = pos[9 * ind + k];
}



for (int k = 0; k < 6; k++) {
   G[k] = m[k];
}

int value = 0;

while (t <= t_stop) {


   if (value < samplesize && Dados[(NVM + 1) * value + 0] <= t + h) {

      float D0 = Dados[(NVM + 1) * value + 0];
      float D1 = Dados[(NVM + 1) * value + 1];
      float D2 = Dados[(NVM + 1) * value + 2];
      float D3 = Dados[(NVM + 1) * value + 3];

      float P1 = Ppdg(G, i, s_pos) * (t + h - D0);
      float P2 = Ptt(G, i, s_pos) * (t + h - D0);
      float P3 = Prt(G, i, s_pos) * (t + h - D0);

      t = t + h;

      RK4(Dm, G, i, s_pos, h);

      //essa etapa é responsável pela interpolação linear

      P1 = (P1 + Ppdg(G, i, s_pos) * (D0 - t + h)) / h;
      P2 = (P2 + Ptt(G, i, s_pos) * (D0 - t + h)) / h;
      P3 = (P3 + Prt(G, i, s_pos) * (D0 - t + h)) / h;

      MSE = (P1 - D1);
      x_med[0] += P1;
      x_stdev[0] += (P1 * P1);
      rxy[0] += (P1 * D1);
      wi = ((D1 - c_y_med[0]) / c_y_stdev[0]) * ((D1 - c_y_med[0]) / c_y_stdev[0]);
      s_cost[0] += wi * (MSE * MSE);
      //s_cost[0] += wi * (MSE1 * MSE1);
      //calculo dos valores necessários para a função objetivo
      //float MSE1 = ((P1 - D1)) / NVM;//(P1 - D1) / NVM;
      MSE = (P2 - D2);
      x_med[1] += P2;
      x_stdev[1] += (P2 * P2);
      rxy[1] += (P2 * D2);
```

```
            wi = ((D2 - c_y_med[1]) / c_y_stdev[1]) * ((D2 - c_y_med[1]) / c_y_stdev[1]);
            s_cost[1] += wi * (MSE * MSE);

            MSE = (P3 - D3);
            x_med[2] += P3;
            x_stdev[2] += (P3 * P3);
            rxy[2] += (P3 * D3);
            wi = ((D3 - c_y_med[2]) / c_y_stdev[2]) * ((D3 - c_y_med[2]) / c_y_stdev[2]);
            s_cost[2] += wi * (MSE * MSE);

            value = value + 1;
          }

          t = t + h;

          RK4(Dm, G, i, s_pos, h);

        }

        //printf("%f\n",value);
        for (int k = 0; k < NVM; k++) {
          x_med[k] = x_med[k] / (value);
          x_stdev[k] = sqrtf((x_stdev[k] - x_med[k] * x_med[k] * value) / (value - 1));
          rxy[k] = (rxy[k] - value * x_med[k] * c_y_med[k]) / ((value - 1) * x_stdev[k] *
c_y_stdev[k]);
          //evitar que ocorra eventuais "crashes" por divizão por zero
          rxy[k] = d_max(0.001f, (rxy[k] + 1.0f) / 2.0f);
          s_cost[k] = (1 / rxy[k]) * s_cost[k] / value;
          if (k > 0) {
            s_cost[0] += s_cost[k];
          }
        }
        //printf("%f  %d\n", s_cost[0],ind);

        cost[ind] = (s_cost[0]);

        //printf("%f  %d\n", s_cost[i],ind);

        //cost[ind] = s_cost[0]+ s_cost[1]+ s_cost[2]; // float(value);// / ((value - 1) / 100);
        //printf("%d",(value-1)/20);

    }

    //PSO

    __host__ void particle_initialize(float* pos, float* vel, float* R1, float* R2, float*
pbest, float* lbest, int n, int ind)
    {
      //definindo indices
```

```
        //importante verificar se os dados se encontram dentro dos limites estabelecidos
        for (int k = 0; k < 9; k++) {

            //inicializando posição e velocidade
            pos[k + ind * 9] = c_lim[2 * k + 0] + R1[k + ind * 9] * (c_lim[2 * k + 1] - c_lim[2 * k
+ 0]);
            vel[k + ind * 9] = 0.01f * ((c_lim[2 * k + 1] - c_lim[2 * k + 0]) / 2.0f) * (2.0f * R2[k +
ind * 9] - 1.0f);
        }

        pbest[ind] = INFINITY;
        lbest[ind] = INFINITY;

    }

    __host__ void base_PSO_particle_PV(float* pos, float* vel, float* R1, float* R2, float*
theta_pbest, int* g, int n, int ind) {

        //definindo indices


        for (int k = 0; k < 9; k++) {
            //atualizando velocidade e posição
            vel[k + ind * 9] = c_Wc1c2[0] * vel[k + ind * 9] + c_Wc1c2[1] * R1[k + ind * 9] *
(theta_pbest[k + ind * 9] - pos[k + ind * 9]) + c_Wc1c2[2] * R2[k + ind * 9] *
(theta_pbest[g[0] * 9 + k] - pos[k + ind * 9]);

            pos[k + ind * 9] = pos[k + ind * 9] + vel[k + ind * 9];

            //verificando se as particulas sairam do espaço de busca
            if (pos[k + ind * 9] > c_lim[2 * k + 1] || pos[k + ind * 9] < c_lim[2 * k + 0]) {

                //caso em  que a partícula é maior que o limite superior
                if (pos[k + ind * 9] > c_lim[2 * k + 1]) {
                    pos[k + ind * 9] = c_lim[2 * k + 1];
                    vel[k + ind * 9] = -0.75f * (c_lim[2 * k + 1] - c_lim[2 * k + 0]) / 2.0f;
                }

                //caso em  que a partícula é menor que o limite inferior
                else {
                    pos[k + ind * 9] = c_lim[2 * k + 0];
                    vel[k + ind * 9] = 0.75f * (c_lim[2 * k + 1] - c_lim[2 * k + 0]) / 2.0f;
                }

            }
        }

    }
```

```
__host__ void local_PSO_particle_PV(float* pos, float* vel, float* R1, float* R2, float* theta_pbest, int* l, int n, int ind) {

    //definindo indices



    for (int k = 0; k < 9; k++) {
      //atualizando velocidade e posição
      vel[k + ind * 9] = c_Wc1c2[0] * vel[k + ind * 9] + c_Wc1c2[1] * R1[k + ind * 9] * (theta_pbest[k + ind * 9] - pos[k + ind * 9]) + c_Wc1c2[2] * R2[k + ind * 9] * (theta_pbest[l[ind] * 9 + k] - pos[k + ind * 9]);

      pos[k + ind * 9] = pos[k + ind * 9] + vel[k + ind * 9];

      //verificando se as particulas sairam do espaço de busca
      if (pos[k + ind * 9] > c_lim[2 * k + 1] || pos[k + ind * 9] < c_lim[2 * k + 0]) {

        //caso em  que a partícula é maior que o limite superior
        if (pos[k + ind * 9] > c_lim[2 * k + 1]) {
          pos[k + ind * 9] = c_lim[2 * k + 1];
          vel[k + ind * 9] = -0.01f * (c_lim[2 * k + 1] - c_lim[2 * k + 0]) / 2.0f;
        }

        //caso em  que a partícula é menor que o limite inferior
        else {
          pos[k + ind * 9] = c_lim[2 * k + 0];
          vel[k + ind * 9] = 0.01f * (c_lim[2 * k + 1] - c_lim[2 * k + 0]) / 2.0f;
        }

      }
    }

  }

  __host__ void particle_pbest(float* cost, float* pbest, float* lbest, int* l, float* theta_pbest, float* pos, int n, int ind) {

    //definindo indices

    //comparando custo a pbest
    if (cost[ind] < pbest[ind]) {

      for (int k = 0; k < 9; k++) {
        //atualizando os theta_pbest
        theta_pbest[k + ind * 9] = pos[k + ind * 9];
      }

      //atualizando pbest
```

```
            pbest[ind] = cost[ind];

        }
        if (lbest[ind] > pbest[ind]) {
            lbest[ind] = pbest[ind];
            l[ind] = ind;
        }
        //printf("%f\n", pbest[ind]);
    }


    //Topologias
    __host__ void cpy_lbest(float* lbest, int* l, float* n_lbest, int* n_l,int n) {
        for (int k = 0; k < n; k++) {
            n_lbest[k] = lbest[k];
            n_l[k] = l[k];
        }
    }


    __host__ void circulo_PSO_find_lbest(float* pbest, float* lbest, int* l, float* n_lbest,
int* n_l, int n, int r, int ind) {


        for (int k = 1; k <= r / 2; k++) {
            if (lbest[ind] > n_lbest[(N + ind + k) % N]) {
                lbest[ind] = n_lbest[(N + ind + k) % N];
                l[ind] = n_l[(N + ind + k) % N];
            }
            if (lbest[ind] > n_lbest[(N + ind - k) % N]) {
                lbest[ind] = n_lbest[(N + ind - k) % N];
                l[ind] = n_l[(N + ind - k) % N];
            }
        }

        if (lbest[ind] > n_lbest[ind]) {
            lbest[ind] = n_lbest[ind];
            l[ind] = n_l[ind];
        }

    }

    __host__ void base_PSO_find_gbest(float* pbest, float* gbest, int* g, int n) {


        //inicializando memória compartilhada
        int s_gind;
        float s_gbest;

        //passando os dados da memória global para compartilhada
```

```
        for(int j=0;j<n/N;j++){
           s_gbest = pbest[N*j];
           s_gind = N*j;



        //algoritimo de reduce para obter o gbest dentre os pbest
           for (int k = 0; k < N; k++) {
                 if (s_gbest > pbest[N*j+k]) {
                    s_gbest = pbest[N*j+k];
                    s_gind = N*j+k;
                 }

              }

           if (gbest[j] > s_gbest) {
              gbest[j] = s_gbest;
              g[j] = s_gind;
           }
        }
     }

     //Migracao
     __host__ void migracao_global(float* theta_pbest, float* pbest, float* gbest, int* g,
int n) {

        //inicializando memória compartilhada
        float s_pbest;
        int s_pind;


        //passando os dados da memória global para compartilhada

        int gmin;

        float gtest = gbest[0];
        gmin = g[0];

        for (int k = 0; k < n / N; k++) {
           if (gtest > gbest[k]) {
              gtest = gbest[k];
              gmin = g[k];
           }
        }


        //algoritimo de reduce para obter o gbest dentre os pbest
        for (int k = 0; k < n / N; k++) {
           s_pind = k * N;
           s_pbest = pbest[k * N];
```

```
            for (int ind = 0; ind < N; ind++) {
                if (s_pbest > pbest[k * N + ind]) {
                    s_pbest = pbest[k * N + ind];
                    s_pind = k * N + ind;
                }
            }
            pbest[s_pind] = pbest[gmin];
            for (int k = 0; k < 9; k++) {
                theta_pbest[9 * s_pind + k] = theta_pbest[9 * gmin + k];
            }
        }
    }



    //funções na cpu
    __host__ void y_medstdev(float* Dados, int ind_stop, float* y_med, float* y_stdev) {

        int ind = 0;

        for (int k = 0; k < NVM; k++) {
            y_med[k] = 0.0f;
            y_stdev[k] = 0.0f;
        }

        while (ind < ind_stop) {

            for (int k = 0; k < NVM; k++) {

                y_med[k] += Dados[(NVM + 1) * ind + k + 1];
                y_stdev[k] += Dados[(NVM + 1) * ind + k + 1] * Dados[(NVM + 1) * ind + k + 1];

            }
            ind = ind + 1;
        }
        //ind = ind - 1;
        for (int k = 0; k < NVM; k++) {
            y_stdev[k] = sqrtf((y_stdev[k] - y_med[k] * y_med[k] / ind) / (ind - 1));
            y_med[k] = y_med[k] / ind;
        }
    }

    //função para salstdev dados para csv (fonte: http://codingstreet.com/create-csv-file-
in-c/)
    __host__ void create_marks_csv(char* filename, const int m, const int n, float
dados[]) {

        printf("\n Creating %s file", filename);

        FILE* f;
```

```
    int i, j;

    //filename = strcat(filename, ".csv");

    f = fopen(filename, "w+");


    for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++) {

        fprintf(f, "%f,", dados[i * n + j]);
      }
      fprintf(f, "\n");
    }

    fclose(f);

    printf("\n %sfile created\n", filename);

}

__host__ void radom(float* R,int n) {

    //srand(time(NULL));

    for (int k = 0; k < 9*n; k++) {
      R[k] = (double) rand() / (RAND_MAX+1);
      //printf("% f", R[k]);
    }
}

int main()
{

    //declaracao das constantes da planta
    float param_planta_ag[12];

    float  rhol = 900.0f; //kg / m3
    float  Pr = 225e5f; //Pa
    float  Ps = 10e5f; //Pa
    float  alphagw = 0.0188f; //adm
    float  rhomres = 892.0f; //kg / m3
    float  M = 18.0f; //kg / kmol
    float  T = 298.0f; //K
    float  Lr = 1569.0f; //m
    float  Lft = 2928.0f; //m
    float  Lt = 1639.0f; //m
    float  La = 1118.0f; //m
    float  Ht = 1279.0f; //m
```

```
float  Hpdg = 1117.0f; //m
float  Hvgl = 916.0f; //m
float  Dss = 0.15f; //m
float  Dt = 0.15f; //m
float  Da = 0.14f; //m
float  g = 9.81f;//m/s2
float  R = 8314.0f;//J/mol*K

float Ass = (PI * Dss * Dss) / 4.0f;
float Vss = (Ass * (Lr + Lft));
float Va = PI * Da * Da * La / 4.0f;
float Vt = PI * Dt * Dt * Lt / 4.0f;
float theta = PI / 4.0f;

param_planta_ag[0] = R * T / (M * Va) + g * La / Va;
param_planta_ag[1] = R * T / M;
param_planta_ag[2] = g * Hvgl;
param_planta_ag[3] = rhomres * g * (Ht - Hpdg);
param_planta_ag[4] = rhomres * g * (Hpdg - Hvgl);
param_planta_ag[5] = g * sinf(theta) / Ass;
param_planta_ag[6] = Pr;
param_planta_ag[7] = Vt;
param_planta_ag[8] = rhol;
param_planta_ag[9] = Vss;
param_planta_ag[10] = Ps;
param_planta_ag[11] = alphagw;

//copia as constantes da planta para a memoria constante
for (int i = 0; i < 12; i++) {
   c_param_planta_ag[i] = param_planta_ag[i];
}

//leitura dos dados experimentais (Fonte:
FILE* fp;
fp = fopen("dados_OLGA2.csv", "r");
float* Dados;
Dados = (float*)malloc(samplesize * (NVM + 1) * sizeof(float));

int Dados_ind = 0;

if (fp == NULL) {
   printf("failed to open file\n");
   return 1;
}

while (fscanf(fp, "%f", &Dados[Dados_ind++]) == 1) {
   fscanf(fp, ",");
}
fclose(fp);
```

```
        int ind = 0;
        float D0 = Dados[0];

        while (ind <= samplesize - 1) {

           //deslocando o tempo para que t0 = 0
           Dados[(NVM + 1) * ind] = (Dados[(NVM + 1) * ind] - D0) > 0 ? Dados[(NVM + 1) *
ind] - D0 : 0;

           ind = ind + 1;
        }

        //definindo limites de busca:
        float lim[18];

        //lim inf
        for (int i = 0; i < 9; i++) {
           lim[i * 2 + 0] = 0.0f;
        }

        //lim sup
        lim[0 * 2 + 1] = 5e3;
        lim[1 * 2 + 1] = 1e-2;
        lim[2 * 2 + 1] = 1e-2;
        lim[3 * 2 + 1] = 5e2;
        lim[4 * 2 + 1] = 1e0;
        lim[5 * 2 + 1] = 1e-2;
        lim[6 * 2 + 1] = 1e-2;
        lim[7 * 2 + 1] = 1e3;
        lim[8 * 2 + 1] = 1e1;

        //imprime os limites para melhor controle
        for (int i = 0; i < 9; i++) {
           printf("lim_inf param[%d] = %f", i, lim[i * 2 + 0]);
           c_lim[2 * i + 0] = lim[i * 2 + 0];
           printf("    lim_sup param[%d] = %f\n", i, lim[i * 2 + 1]);
           c_lim[2 * i + 1] = lim[i * 2 + 1];
        }

        //transferindo limites para memoria constante
        //cudaMemcpyToSymbol(c_lim, lim, sizeof(lim));

        //definindo coeficientes de inercia, coercao e nostalgia (utilizada a versão do CF-
PSO (coef de constrição))
        float c1 = 1.0f * 4.1f / 2.0f;
        float c2 = 4.1f - c1;
        float X = 2.0f / fabs(2.0f - (c1 + c2) - sqrtf((c1 + c2) * (c1 + c2) - 4 * (c1 + c2)));
        float Wc1c2[3] = { X ,X * c1 ,X * c2 };

        //imprimindo coeficientes para controle
```

```
printf("X = %f\n", X);
printf("c1 = %f\n", X * c1);
printf("c2 = %f\n", X * c2);

for (int k = 0; k < 3; k++) {
   c_Wc1c2[k] = Wc1c2[k];
}

//Controle da simulacao
//char* nome = "PSOn1024rNs4c1c2s2\.csv";
const int n = 8*1024;
const int n_iter = 100;
int r = 2;
float h = 5.0f;
float t_stop = Dados[(samplesize - 1) * (NVM + 1)]; //ultimo valor de tempo dos dados
printf("t_stop = %f\n", t_stop);

//declarando variaveis host
float* theta_pbest;
float* gbest;
float* pbest;
float* cost;
float* pos;
float* vel;
float* lbest;
int* l;
float* n_lbest;
int* n_l;

int* gind;
float* y_med;
float* y_stdev;

float* R1;
float* R2;


//alocando memoria para o host
theta_pbest = (float*)malloc(9 * n * sizeof(float));
pos = (float*)malloc(9 * n * sizeof(float));
vel = (float*)malloc(9 * n * sizeof(float));
pbest = (float*)malloc(n * sizeof(float));
cost = (float*)malloc(n * sizeof(float));
lbest = (float*)malloc(n * sizeof(float));
n_lbest = (float*)malloc(n * sizeof(float));
gbest = (float*)malloc(n / N * sizeof(float));
gind = (int*)malloc(n / N * sizeof(int));
l = (int*)malloc(n * sizeof(int));
n_l = (int*)malloc(n * sizeof(int));
```

```
y_med = (float*)malloc(NVM * sizeof(float));
y_stdev = (float*)malloc(NVM * sizeof(float));

R1 = (float*)malloc(9 * n * sizeof(float));
R2 = (float*)malloc(9 * n * sizeof(float));

//computa media e desvio dos dados (usado em Fobj_Diehl)
y_medstdev(Dados, samplesize, y_med, y_stdev);


//declarando vetor de armazenamento (para gerar o arquivo .csv com a saida dos dados)
//float* out_gbest;
//out_gbest = (float*)malloc((n / N) * (n_iter / 100 + 1) * sizeof(float));


//gerador de numeros "aleatorios"
/*
curandGenerator_t gen;
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
curandSetPseudoRandomGeneratorSeed(gen, time(NULL));
*/


//medindo tempo



//copiando media e desvio dos dados para a memoria constante
for (int k = 0; k < NVM; k++) {
   c_y_med[k]= y_med[k];
   c_y_stdev[k] = y_stdev[k];
}


//imprimindo media e desvio para controle
for (int k = 0; k < NVM; k++) {
   printf("med[%d] = %f\n", k + 1, y_med[k]);
   printf("stdev[%d] = %f\n", k + 1, y_stdev[k]);
}

//inicializa gbest
for (int k = 0; k < n / N; k++) {
   gbest[k] = INFINITY;
}

//copia gbest para device


//gera os numeros aleatorios usados na inicializacao
```

```
        radom(R1, n);
        radom(R2, n);

        //inicializa as posicoes,velocidades e pbests/lbests
        for (int ind = 0; ind < n; ind++)
        particle_initialize(pos, vel, R1, R2, pbest, lbest, n, ind);

        //variavel que verifica se o criterio de convergencia foi atingido
        int check = 0;

        clock_t start = clock();

        for (int iter = 0; iter <= n_iter; iter++) {

           //computa funcao objetivo
           //Fobj(d_pos, h, d_cost, Dados, t_stop,ind); //MSE
           for (int ind=0;ind<n;ind++)
           Fobj( pos, h,  cost, Dados, t_stop,ind);

           //verifica se cost<pbest
           for (int ind = 0; ind < n; ind++)
           particle_pbest( cost,  pbest,  lbest,  l,  theta_pbest,  pos, n, ind);

           //topologia
           //base_PSO_fin gbest<<<n/N,N>>>( pbest,  gbest,  gind, n);
           cpy_lbest(lbest, l, n_lbest, n_l, n);
           for (int ind = 0; ind < n; ind++)
           circulo_PSO_find_lbest( pbest,  lbest,  l, n_lbest, n_l, n, r, ind);
           //von_neumann<<<NB_VN,TPB_VN>>>( pbest, lbest, l,n);

           //gera os numeros aleatorios usados no calculo da velocidade
           radom(R1,n);
           radom(R2, n);

           //calcula velocidade e posicao
           //base_PSO_particle_PV << <a, n / a >> > ( pos,  vel,  R1,  R2,  theta_pbest,  gind,
n);
           for (int ind = 0; ind < n; ind++)
           local_PSO_particle_PV( pos,  vel,  R1,  R2,  theta_pbest,  l, n, ind);

           if (iter % 100 == 0) {

              //copia o gbest para a CPU para verificar convergencia
              base_PSO_find_gbest(pbest,  gbest,  gind, n);
              migracao_global(theta_pbest, pbest, gbest, gind, n);

              printf("========================\n");

              for (int k = 0; k < n / N; k++) {
                 printf("gbest da simulacao %d na iteracao %d: %f\n", k + 1, iter, gbest[k]);
```

```
            //salva os valores de gbest para serem escritos posteriormente em um
arquivo .csv
            //out_gbest[(iter/100)*(n / N)+k]= gbest[k];

            //verifica se o criterio de convergencia foi atingido
            if (gbest[k] < 4e10) {
               check = 1;

            }
         }

         printf("========================\n");

         //se o criterio foi atingido encerra o loop
         if (check == 1) {
            break;
         }
      }

   }

   //salva os dados obtidos num arquivo .csv
   //create_marks_csv(nome,n_iter / 100 + 1, n / N, out_gbest);

   //copia os parametros e gbest de volta para a CPU
   base_PSO_find_gbest( pbest,  gbest,  gind, n);

   //para a contagem do tempo
   clock_t end = clock();
   double elapsed_time = (end - start) / (double)CLOCKS_PER_SEC;

   //imprime o tempo de simulacao
   printf("Tempo de Simulacao: %f s\n", float(elapsed_time));

   //imprime resultados (parametros e gbest)
   for (int i = 0; i < n / N; i++) {
      printf("========================\n");
      printf("Simulacao %d:\n", i + 1);
      for (int k = 0; k < 9; k++) {
         printf("param[%d]= %.10f\n", k, theta_pbest[gind[i] * 9 + k]);
      }
      printf("gbest: %f\n", gbest[i]);
      printf("========================\n");
   }


   return 0;
}
```

Apêndice C: Estimação de parâmetros para múltiplas aberturas da válvula *choke*

O objetivo deste apêndice é contemplar os parâmetros estimados para múltiplas aberturas da válvula *choke*. Os parâmetros foram estimados utilizando as pressões P_{pdg}, P_{tt} e P_{rt} por meio da função objetivo proposta por Diehl para uma vazão de *gas lift* fixa de 80000 m^3/d com as aberturas de 2%, 5%, 8%, 12%, 18%, 22%, 30%, 50%, 75% e 100%. O algoritmo realizou 10000 iterações com as configurações ótimas obtidas da seção 4.4 (8 enxames de 1024 partículas cada com migração, c1=c2 e topologia r=2).

Os parâmetros obtidos neste apêndice utilizaram como condição inicial da simulação os estados: m_{ga} = 7629.49953301 kg, m_{gt} = 1506.46645264 kg, m_{lt} = 20249.26259091 kg, m_{gb} = 2135.35823438 kg, m_{gr} = 1130.58624768 kg, m_{lr} = 15196.84541979kg.

Um tempo de 2000 segundos foi esperado antes da coleta do primeiro ponto (i.e. $t_{OLGA}[0]$ = 2000 s), o tempo mais longo de simuação neste caso se dá por utilizar uma condição inicial genérica igual para todas as aberturas de válvula. Se comparado aos parâmetros da seção 4.6, é possivel observar que a condição inicial tem grande efeito no ajuste dos mesmos.

Os ajustes do FOWM são ilustrados nas Figuras C.1 à C.10, enquanto os parâmetros e o respectivo valor da função objetivo se encontram nas Tabelas C.1 à C.10.

Tabela C.1 – Parâmetros para abertura de 2%

Parâmetro (Unid)	**Valor**
$m_{L,still}$ (Kg)	811.1069335938
C_g (m_*s)	0.0000228054
C_{out} (m^2)	0.0098920893
V_{eb} (m^3)	118.9060897827
E (-)	0.1053475663
K_w (Kg/m)	0.0027717161
K_a (Kg/m)	0.0091771986
K_r (Kg/s)	339.7560119629
ω_u (-)	0.5852149725
F_{obj}	**2946267611136.000000**

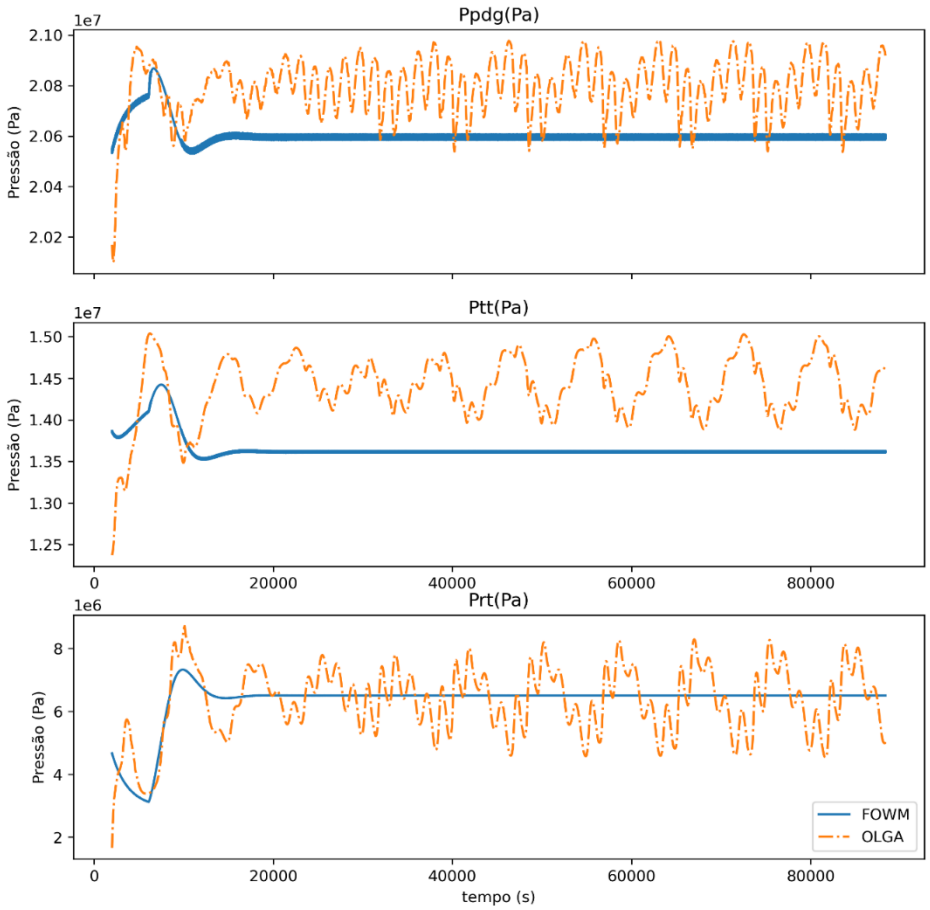

Figura C.1 – Ajuste dos parâmetros do FOWM utilizando P_{pdg}, P_{tt} e P_{rt} para abertura da *choke* de 2%

Tabela C.2 – Parâmetros para abertura de 5%

Parâmetro (Unid)	**Valor**
$m_{L,still}$ (Kg)	0.0000000000
C_g (m*s)	0.0003985868
C_{out} (m^2)	0.0081827706
V_{eb} (m^3)	116.9555130005
E (-)	0.0000000000
K_w (Kg/m)	0.0004906736
K_a (Kg/m)	0.0019843704
K_r (Kg/s)	231.6345367432
ω_u (-)	0.6550435424
F_{obj}	**1656323440640.000000**

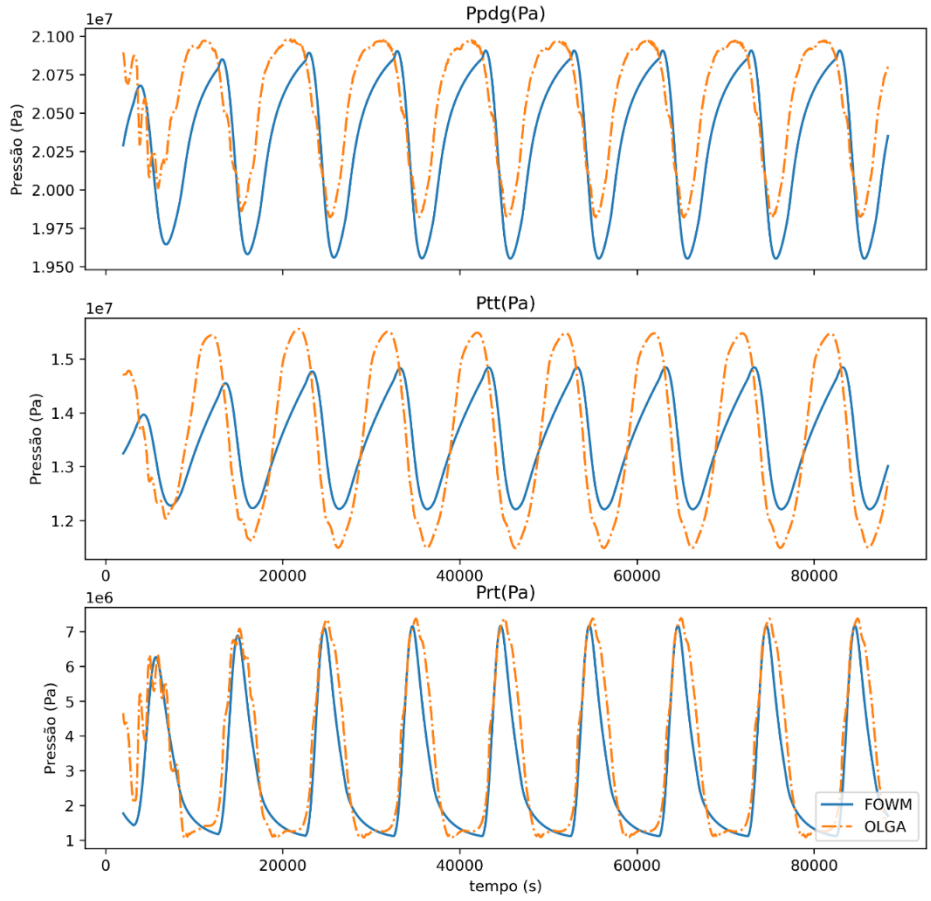

Figura C.2 – Ajuste dos parâmetros do FOWM utilizando P_{pdg}, P_{tt} e P_{rt} para abertura da *choke* de 5%

Tabela C.3 – Parâmetros para abertura de 8%

Parâmetro (Unid)	**Valor**
$m_{L,still}$ (Kg)	0.0000000000
C_g (m*s)	0.0000045268
C_{out} (m^2)	0.0065026172
V_{eb} (m^3)	118.8032760620
E (-)	0.0000000000
K_w (Kg/m)	0.0007741348
K_a (Kg/m)	0.0026430897
K_r (Kg/s)	181.4314727783
ω_u (-)	0.7286875844
F_{obj}	**1476204298240.000000**

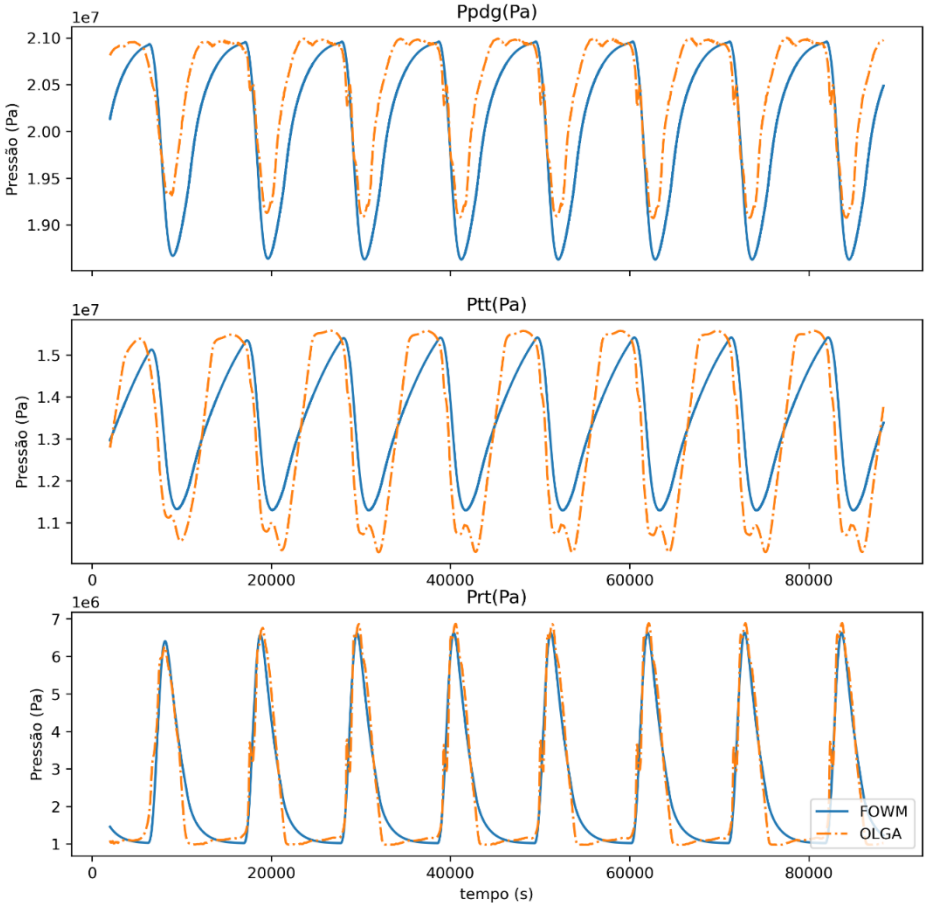

Figura C.3 – Ajuste dos parâmetros do FOWM utilizando P_{pdg}, P_{tt} e P_{rt} para abertura da *choke* de 8%

Tabela C.4 – Parâmetros para abertura de 12%

Parâmetro (Unid)	**Valor**
$m_{L,still}$ (Kg)	38.9986457825
C_g (m*s)	0.0000020571
C_{out} (m^2)	0.0067034205
V_{eb} (m^3)	113.9146118164
E (-)	0.0000000000
K_w (Kg/m)	0.0011603849
K_a (Kg/m)	0.0012493735
K_r (Kg/s)	219.7340698242
ω_u (-)	0.7488170266
F_{obj}	**1097772302336.000000**

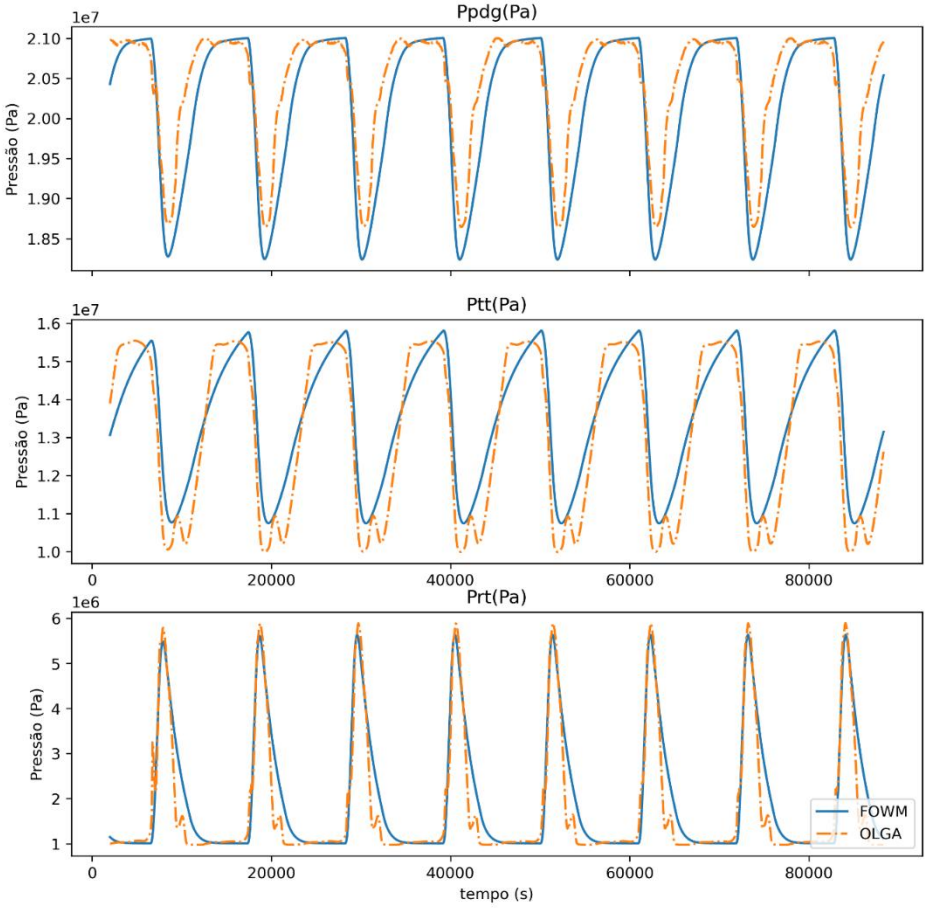

Figura C.4 – Ajuste dos parâmetros do FOWM utilizando Ppdg, Ptt e Prt para abertura da choke de 12%

Tabela C.5 – Parâmetros para abertura de 18%

Parâmetro (Unid)	**Valor**
$m_{L,still}$ (Kg)	1745.1672363281
C_g (m∗s)	0.0000022865
C_{out} (m^2)	0.0054538455
V_{eb} (m^3)	110.0677413940
E (-)	0.0170637984
K_w (Kg/m)	0.0048913779
K_a (Kg/m)	0.0030283371
K_r (Kg/s)	199.1116180420
ω_u (-)	1.0029321909
F_{obj}	**1133733871616.000000**

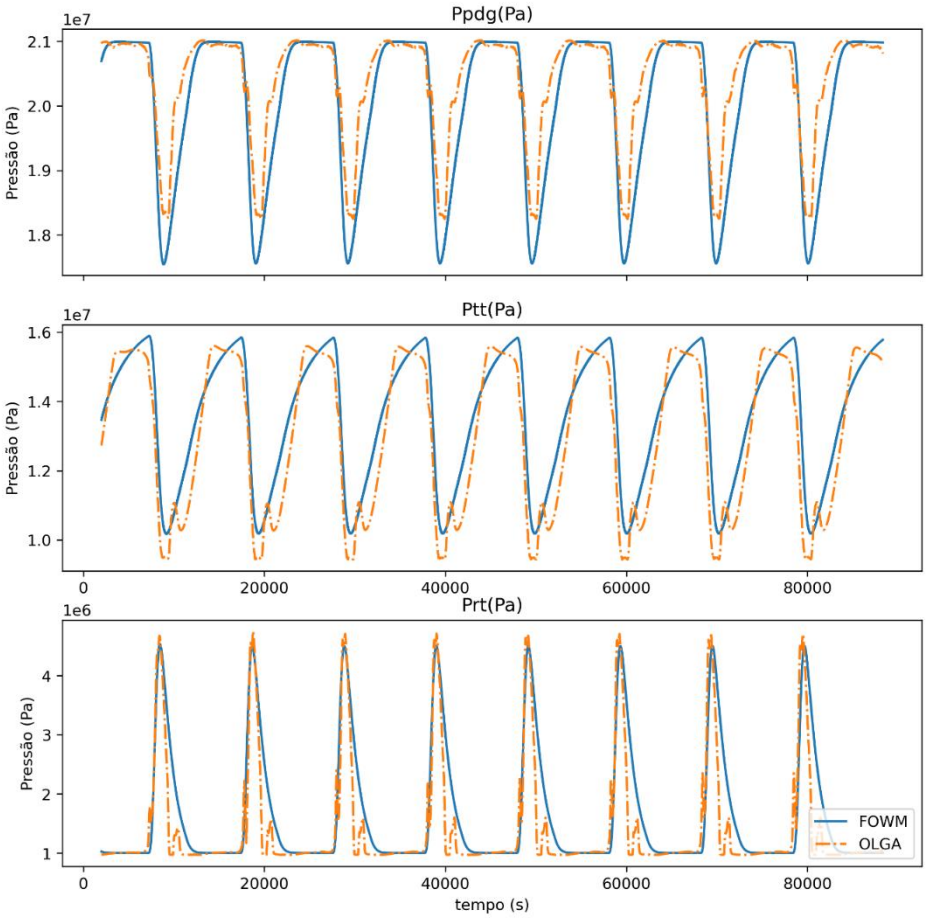

Figura C.5 – Ajuste dos parâmetros do FOWM utilizando Ppdg, Ptt e Prt para abertura da choke de 18%

Tabela C.6 – Parâmetros para abertura de 22%

Parâmetro (Unid)	Valor
$m_{L,still}$ (Kg)	0.2853584290
C_g (m*s)	0.0000012885
C_{out} (m^2)	0.0064817765
V_{eb} (m^3)	96.6741790771
E (-)	0.0092883222
K_w (Kg/m)	0.0011252579
K_a (Kg/m)	0.0000554353
K_r (Kg/s)	261.0945129395
ω_u (-)	0.7650133371
F_{obj}	**1004471713792.000000**

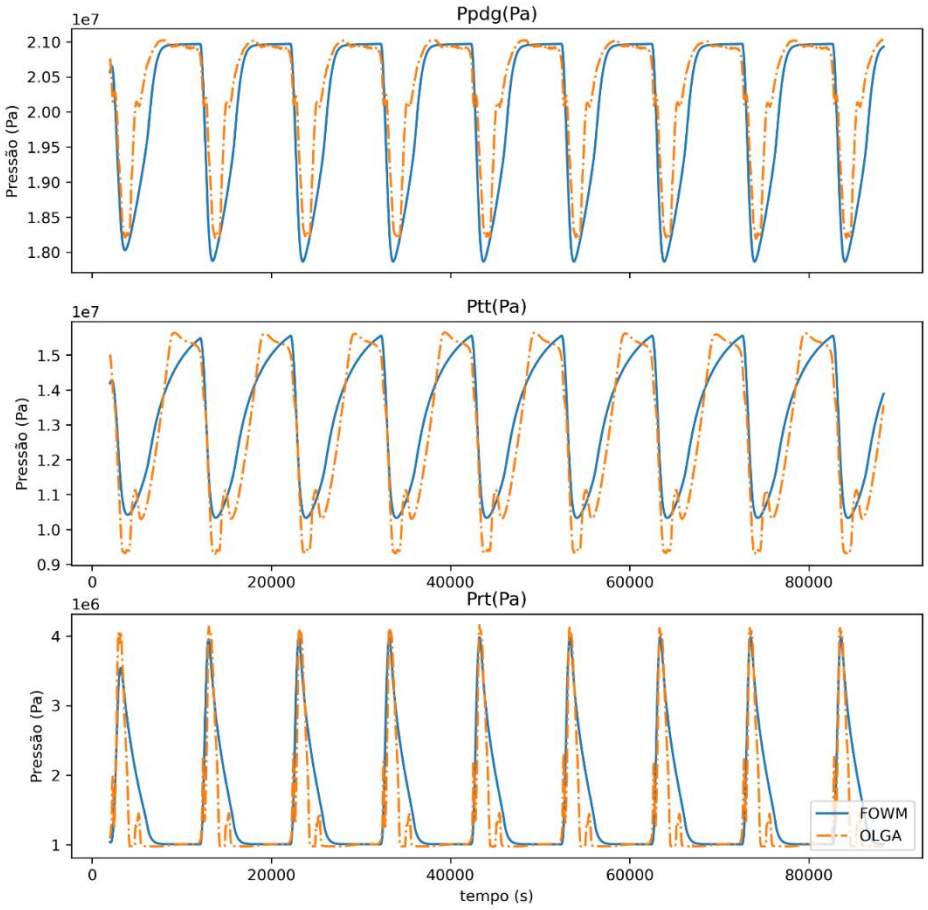

Figura C.6 – Ajuste dos parâmetros do FOWM utilizando Ppdg, Ptt e Prt para abertura da choke de 22%

Tabela C.7 – Parâmetros para abertura de 30%

Parâmetro (Unid)	**Valor**
$m_{L,still}$ (Kg)	19.6914081573
C_g (m*s)	0.0000013804
C_{out} (m^2)	0.0055124983
V_{eb} (m^3)	109.0522613525
E (-)	0.0162889212
K_w (Kg/m)	0.0017496019
K_a (Kg/m)	0.0002571717
K_r (Kg/s)	297.0666503906
ω_u (-)	0.8954259753
F_{obj}	**977026547712.000000**

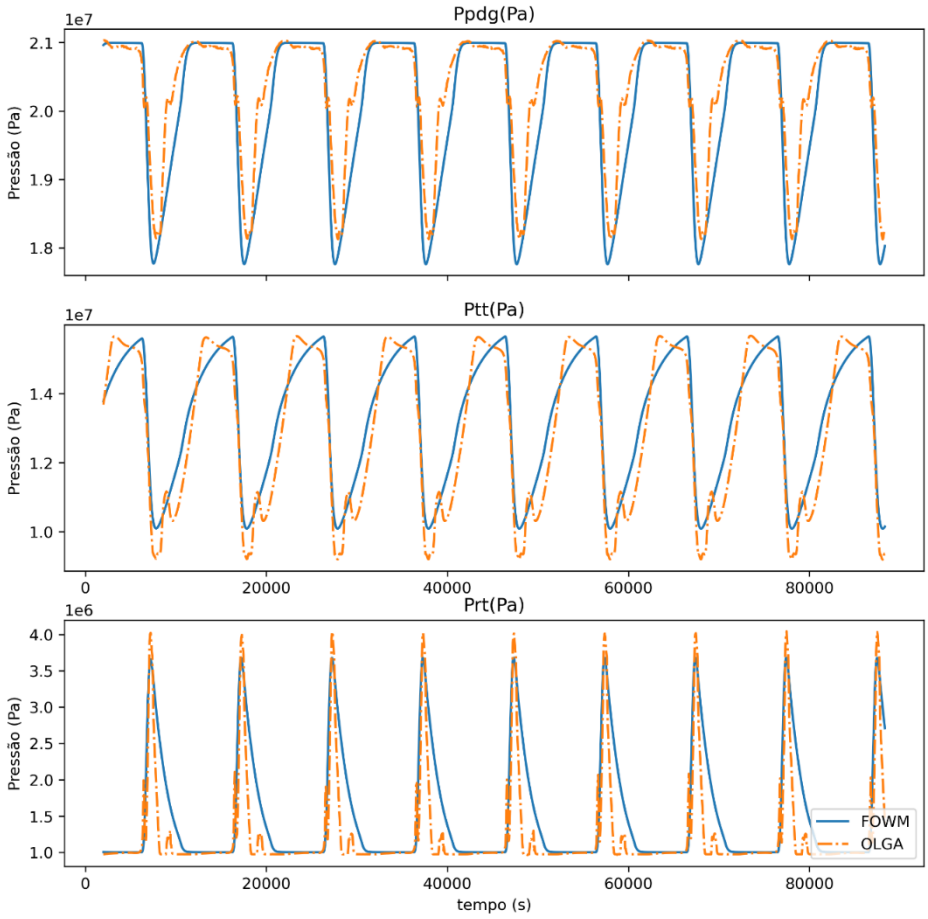

Figura C.7 – Ajuste dos parâmetros do FOWM utilizando Ppdg, Ptt e Prt para abertura da choke de 30%

Tabela C.8 – Parâmetros para abertura de 50%

Parâmetro (Unid)	Valor
$m_{L,still}$ (Kg)	0.0000000000
C_g (m*s)	0.0000007447
C_{out} (m^2)	0.0060968339
V_{eb} (m^3)	90.6493606567
E (-)	0.0308531653
K_w (Kg/m)	0.0011326337
K_a (Kg/m)	0.0001252235
K_r (Kg/s)	306.3644714355
ω_u (-)	1.0344890356
F_{obj}	**812733366272.000000**

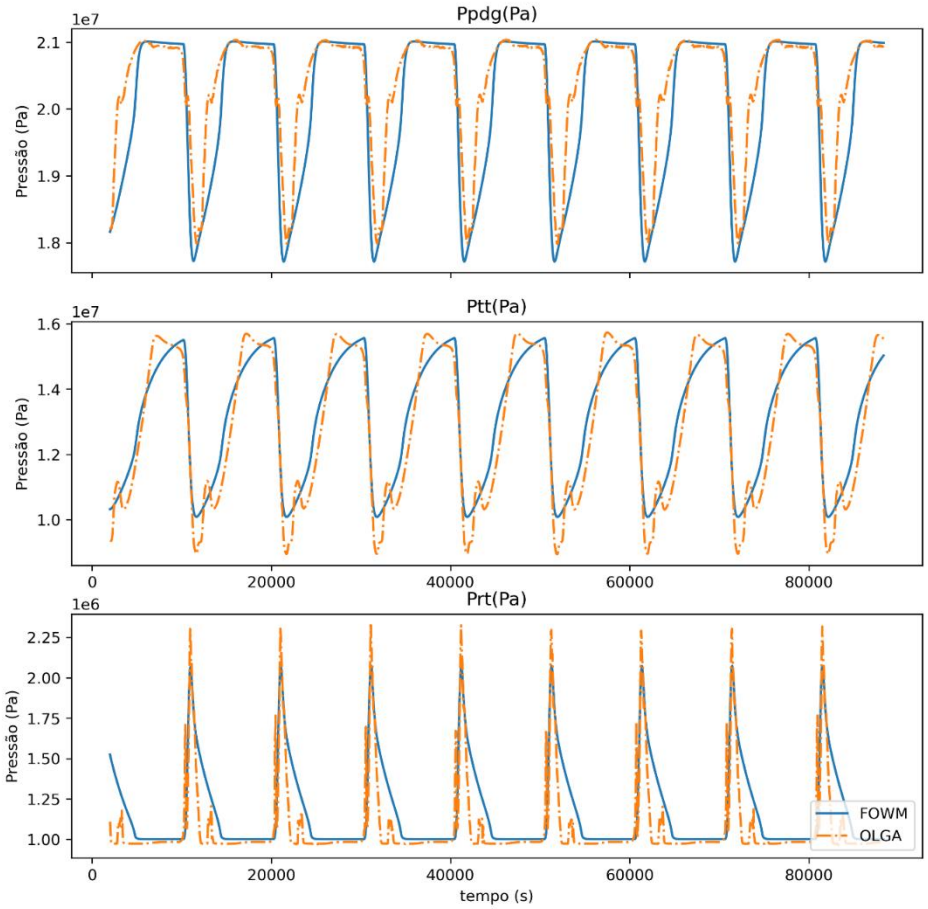

Figura C.8 – Ajuste dos parâmetros do FOWM utilizando Ppdg, Ptt e Prt para abertura da choke de 50%

Tabela C.9 – Parâmetros para abertura de 75%

Parâmetro (Unid)	**Valor**
$m_{L,still}$ (Kg)	112.5619125366
C_g (m*s)	0.0000006773
C_{out} (m^2)	0.0026599723
V_{eb} (m^3)	82.2888107300
E (-)	0.0526351817
K_w (Kg/m)	0.0047039101
K_a (Kg/m)	0.0030386157
K_r (Kg/s)	165.1463470459
ω_u (-)	2.4849076271
F_{obj}	**1161051897856.000000**

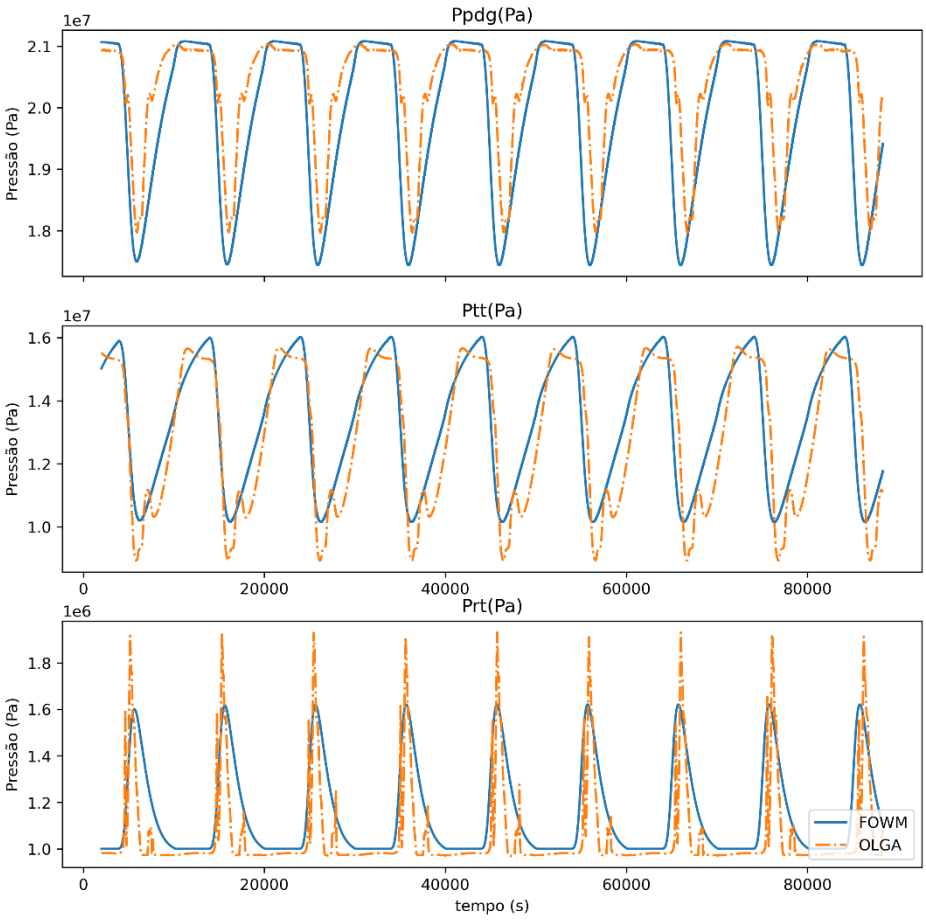

Figura C.9 – Ajuste dos parâmetros do FOWM utilizando P_{pdg}, P_{tt} e P_{rt} para abertura da *choke* de 75%

Tabela C.10 – Parâmetros para abertura de 100%

Parâmetro (Unid)	**Valor**
$m_{L,still}$ (Kg)	37.9241905212
C_g ($m*s$)	0.0000008393
C_{out} (m^2)	0.0024037922
V_{eb} (m^3)	114.7375946045
E (-)	0.1060981676
K_w (Kg/m)	0.0052288142
K_a (Kg/m)	0.0030205802
K_r (Kg/s)	223.0875701904
ω_u (-)	2.0042479038
F_{obj}	**1242063437824.000000**

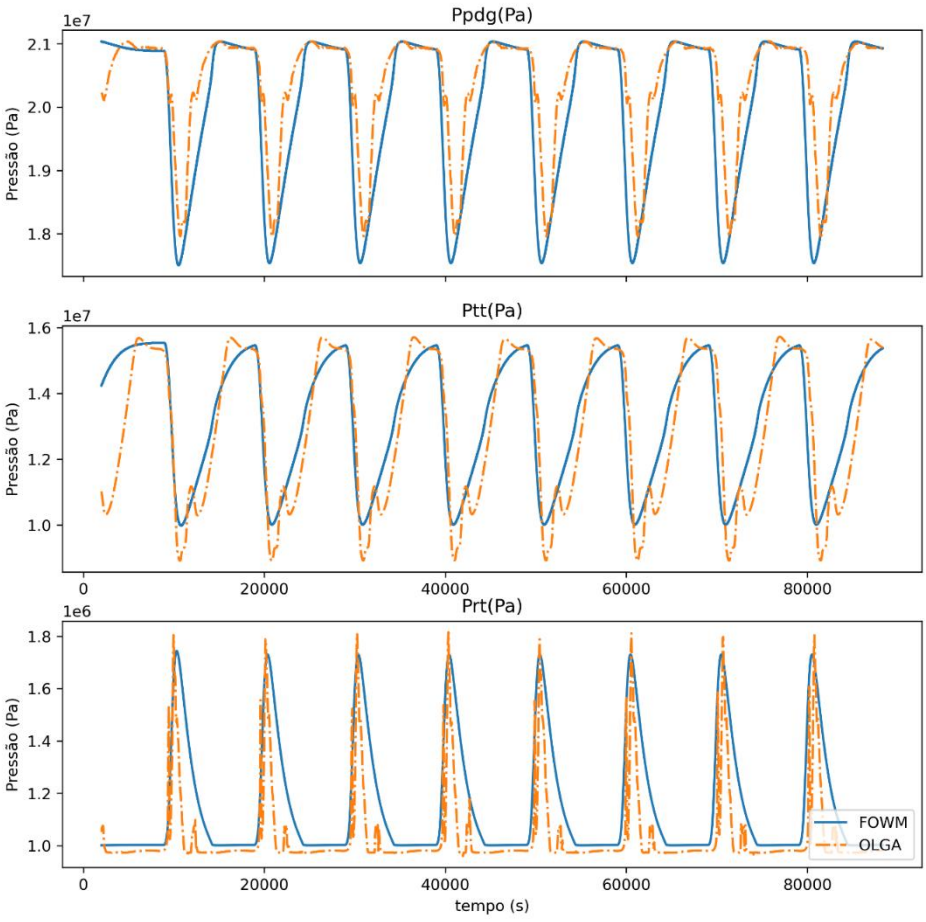

Figura C.10 – Ajuste dos parâmetros do FOWM utilizando P_{pdg}, P_{tt} e P_{rt} para abertura da *choke* de 100%