

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
FACULDADE DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

PAULO VICTOR DE LIMA SFAIR ALVARES

IMPLEMENTAÇÃO DO ALGORITMO PSO EM CUDA UTILIZANDO TÉCNICAS DA CAPACIDADE DE COMPUTABILIDADE 6.1 PARA OTIMIZAÇÃO DE PROBLEMAS DE ENGENHARIA COM RESTRIÇÕES.

Belém
2018



UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
FACULDADE DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

PAULO VICTOR DE LIMA SFAIR ALVARES

IMPLEMENTAÇÃO DO ALGORITMO PSO EM CUDA UTILIZANDO TÉCNICAS DA CAPACIDADE DE COMPUTABILIDADE 6.1 PARA OTIMIZAÇÃO DE PROBLEMAS DE ENGENHARIA COM RESTRIÇÕES.

Trabalho de Conclusão de Curso apresentado como requisito para a obtenção do grau de Bacharel em Ciência da Computação, pela Faculdade de Computação, Campus Universitário do Guamá, Universidade Federal do Pará.

Orientador: Prof. Dr. Dionne Monteiro Cavalcante

Belém
2018

PAULO VICTOR DE LIMA SFAIR ALVARES

IMPLEMENTAÇÃO DO ALGORITMO PSO EM CUDA UTILIZANDO TÉCNICAS DA CAPACIDADE DE COMPUTABILIDADE 6.1 PARA OTIMIZAÇÃO DE PROBLEMAS DE ENGENHARIA COM RESTRIÇÕES.

Trabalho de Conclusão de Curso apresentado como requisito para a obtenção do grau de Bacharel em Ciência da Computação, pela Faculdade de Computação, Campus Universitário do Guamá, Universidade Federal do Pará.

Data da Defesa: 12 de Dezembro de 2018
Conceito: Bom

Banca Examinadora

Prof. Dr. Dionne Monteiro Cavalcante
Faculdade de Computação - UFPA
Orientador

Prof. Dr. Josivaldo Araújo
Faculdade de Computação - UFPA
Membro da Banca

Prof. Dr. Claudomiro Sales
Faculdade de Computação - UFPA
Membro da Banca

Belém
2018

RESUMO

O presente trabalho trata da implementação e análise do algoritmo de PSO (Otimização por Enxame de Partículas), algoritmo que abre possibilidade para a paralelização, em CUDA, utilizando um dos recursos da capacidade de computabilidade 6.1, a *shared memory* permitindo a diminuição da latência de comunicação entre o core CUDA e a memória. Com este trabalho, objetivou-se implementar o algoritmo de PSO de forma que o resultado obtido após a execução deste, fosse mais preciso, obtendo-se mais casas decimais de precisão no resultado, permitindo que aplicações que necessitem de uma precisão maior possam ser executadas com este algoritmo, além de melhorar o tempo de execução, utilizando a nova arquitetura das placas de vídeo NVidia GTX série 10, Pascal. Para a análise e comparação de resultados, foram utilizados resultados do trabalho produzido por Daniel Souza em 2014, e nos resultados obtidos, foi alcançada uma melhora, tanto na precisão, onde foi aumentado de 6 casas decimais confiáveis para 16 casas decimais confiáveis, quanto no tempo de execução do algoritmo, o que abre a possibilidade para que aplicações que necessitam de precisão extrema, como aplicações médicas e da biomedicina, possam ser executadas sem que o tempo de execução seja afetado.

Palavras-chave: PSO. Pascal. Otimização. CUDA.

ABSTRACT

The present work is on the implementation and analysis of the PSO (Particle Swarm Optimization) algorithm, an algorithm that opens possibility for the parallelization in CUDA, using a technique of the Compute Capability 6.1, shared memory, allowing the decrease of the latency of communication between the CUDA core and memory. This work aimed to implement the PSO algorithm so that the result obtained after its execution was preciser in such ways that it could obtain more precise decimal results, besides improving the execution time of it using the new architecture of the NVidia GPUs GTX 10 series, Pascal. For the analysis and comparison of results, it has been used results of the work produced by Daniel Souza in 2014, and in the obtained results an improvement was reached both in the precision, where it was increased from 6 decimal places to 16 decimal places, and at run time of the algorithm, which opens the possibility for applications that require extreme precision, such as medical applications and biomedicine, can be executed without the execution time being affected.

Keywords: PSO. Pascal. Optimization. CUDA.

LISTA DE ILUSTRAÇÕES

Figura 1 – Operações de Ponto Flutuante por Segundo na GPU e CPU 15
Figura 2 – Representação gráfica de comparativa entre arquitetura da CPU e GPU . . . 16
Figura 3 – Pico na largura de banda da memória GPU e CPU 18
Figura 4 – Malha (grid) de *Threads* e Blocks 21
Figura 5 – Escalabilidade automática na execução dos blocos 22
Figura 6 – Escalabilidade automática na execução dos blocos 24
Figura 7 – Representação gráfica da condição de *Damping* 28
Figura 8 – Escalabilidade automática na execução dos blocos 28
Figura 9 – Esquema do Peso da Tensão/Compressão sobre Mola 30
Figura 10 – Representação Gráfica de um Redutor de Velocidade 31
Figura 11 – Diagrama de estado de uma partícula (*thread*) 33
Figura 12 – Representação gráfica do cálculo do melhor global 36
Figura 13 – Gráfico de Convergência MWTCS 320 Partículas 39
Figura 14 – Gráfico de Convergência MWTCS 640 Partículas 40
Figura 15 – Média do Fitness MWTCS 40
Figura 16 – Média de Gerações MWTCS 41
Figura 17 – Melhor Fitness Encontrado MWTCS 41
Figura 18 – Tempo de Execução MWTCS 42
Figura 19 – Gráfico de Convergência SRD 320 Partículas 43
Figura 20 – Gráfico de Convergência SRD 640 Partículas 43
Figura 21 – Zoom ao Final da Convergência SRD 640 Partículas 44
Figura 22 – Média do Fitness SRD 44
Figura 23 – Média de Gerações SRD 45
Figura 24 – Melhor Fitness Encontrado SRD 45
Figura 25 – Tempo de Execução SRD 46

LISTA DE QUADROS

Quadro 1 – Tipos de Memória, e suas declarações. 23

LISTA DE TABELAS

Tabela 1 – Resultados MWTCS 42
Tabela 2 – Resultados SRD 46

LISTA DE ALGORITMOS

Algoritmo 1 – PSO clássico com fator de inércia e condição de contorno 29

Algoritmo 2 – PSO implementado na GPU 35

LISTA DE ABREVIATURAS E SIGLAS

PSO	*Particle Swarm Optimization* - Otimização por Enxame de Partículas
AG	Algoritmo Genético

LISTA DE SÍMBOLOS

ϕ	Letra grega Phi, ou o Número de ouro
Λ	Lambda
ζ	Letra grega minúscula zeta
$\in$	Pertence
$\mathcal{O}$	Big O

SUMÁRIO

1 Introdução 12
1.1 Os Algoritmos de Otimização e a Computação Bioinspirada 13
1.2 GPU para Processamento Geral 15
1.3 Motivação 17
1.3.1 Objetivo Geral 17
1.3.2 Objetivos Específicos 18
1.4 Metodologia 18
1.4.1 Funções Teste 19
1.4.2 Estrutura do Trabalho 19
2 Fundamentação Teórica 20
2.1 Processamento Geral Implementado em GPUs 20
2.1.1 Estrutura 20
2.1.2 O Papel da Memória na GPU 21
2.2 PSO 23
2.2.1 Implementação 24
2.3 Descrição dos Problemas de Engenharia 29
2.3.1 Peso da Tensão/Compressão sobre Mola (MWTCS) 30
2.3.2 Projeto de Redutor de Velocidade (SRD11) 31
3 Desenvolvimento do sistema de otimização do PSO em GPU 33
3.1 Projeto do Software 33
3.1.1 Descrição dos Algoritmos Implementados 34
3.1.2 Otimização e Desempenho 36
4 Análise de Desempenho 38
4.1 Problemas de Engenharia Implementados 38
4.1.1 Peso da Tensão/Compressão sobre Mola (MWTCS) 39
4.1.2 Projeto de Redutor de Velocidade (SRD11) 42
5 Considerações Finais 47
5.1 Limitações 47
5.2 Trabalhos Futuros 47

Referências 49

1 INTRODUÇÃO

Sempre foi do instinto humano observar a natureza, fazer anotações e se inspirar nela para desenvolver as suas criações. Segundo Ching (2013) essa observação remonta da época de Pitágoras o qual afirmava que "tudo está organizado segundo os números e as formas matemáticas". Pitágoras descreveu relações matemáticas no sistema musical grego, que posteriormente foi expandida por Platão como a "estrutura harmônica do universo".

Ainda em Ching (2013), o autor discorre sobre a retomada das teorias de Pitágoras e Platão quando comenta que

> Os arquitetos da Renascença, acreditando que suas edificações tinham que pertencer a uma ordem mais elevada, retomaram o sistema matemático grego das proporções. Assim como os gregos concebiam a música como sendo a geometria traduzida em som, os arquitetos renascentistas acreditavam que a arquitetura era matemática traduzida em unidades espaciais. (CHING, 2013, p. 314)

Na busca pela perfeição, e no resgate dos pensamentos Pitagóricos, estas relações numéricas foram amplamente estudadas e utilizadas na arquitetura renascentista que se utilizava das proporções perfeitas, calculadas por Pitágoras e Platão, nas fachadas, na relação dos espaços ou mesmo na totalidade da planta baixa da construção. Um dos matemáticos que estudou estas relações foi Leonardo Fibonacci e as reproduziu em seu livro *"Liber Abaci"* (PISANO; FIBONACCI'S, 2002) publicado em 1202, com sua segunda edição publicada em 1228, contendo esta os estudos que deram visibilidade à tão famosa sequencia de Fibonacci, amplamente conhecida dentro da programação por se tratar de um problema de recursão de fácil implementação.

Segundo Pinheiro (2011), a segunda edição do *"Liber Abaci"* descreve o problema do crescimento de uma população hipotética de coelhos durante um ano, com condições ideais com as seguintes regras:

- Nenhum coelho morre durante o ano;
- Cada casal de coelhos fica fértil depois de 2 (dois) meses;
- Cada casal fértil gera um segundo casal por mês;

Iniciando-se a hipótese com apenas 1 casal imaturo teremos a sequência de Fibonacci:

$$1,1,2,3,5,8,13,21,... \tag{1.1}$$

Tal problema pode ser resumido em uma equação consideravelmente simples,

$$a_n = a_{n-1} + a_{n-2} \tag{1.2}$$

Outro fato curioso sobre a sequência de Fibonacci é que a divisão de qualquer número desta sequência pelo seu antecessor resulta em um número aproximado ao número de ouro ($\phi \approx 1,618$), tal que quanto maior for o número da sequência utilizado, mais próximo fica o resultado do valor irracional de ϕ

$$\phi \approx \frac{a_n}{a_{n-1}} \tag{1.3}$$

Sabendo-se que a existência do número de ouro remonta dos tempos egípcios, época na qual foram construídas as pirâmides de Gizé, que contém a razão áurea (razão entre a altura de uma face e metade do lado da base da grande pirâmide) esta razão também aparece nas construções gregas antigas, tendo como um dos maiores exemplos o Partenom Grego (BELUSSI; BARISON, 2016). Esta razão é tomada como a razão que representa a beleza e perfeição pelos renascentistas.

Tendo estes argumentos em mente, podemos armar que o ser humano sempre está desenvolvendo e se deparando com representações matemáticas da natureza. E foram a partir de observações como estas que se teve a ideia de desenvolver algoritmos computacionais que representam a natureza, a forma como os animais se comportam, a sua organização, o seu desenvolvimento, e até mesmo a sua estrutura neurológica e reprodutora.

Neste trabalho de conclusão de curso, falar-se-a da classe dos algoritmos de Computação bioinspirada, que imitam as técnicas de evolução humana e animal, mais especicamente da técnica da Otimização por Enxame de Partículas (PSO), além do processo de desenvolvimento e implementação dos algoritmos para GPU.

1.1 Os Algoritmos de Otimização e a Computação Bioinspirada

Segundo Tanomaru (1995) "uma grande parte dos problemas científicos pode ser formulada como problemas de busca e otimização". Geralmente todo problema envolve uma série de fatores que podem ser interpretados numericamente sendo que estes mesmos valores sujeitos ou não as restrições.

> O objetivo é encontrar a melhor combinação dos fatores, ou seja, a combinação de fatores que proporcione o melhor desempenho possível para o sistema em questão. Em termos técnicos, o conjunto de todas as combinações possíveis para os fatores constitui o chamado *espaço de busca*. Não é difícil perceber que existe uma dualidade entre os conceitos de busca e otimização, de tal modo que todo problema de busca pode ser considerado um problema de otimização e vice-versa. (TANOMARU, 1995, p. 2)

Ou seja, a otimização pode ser resumida em uma busca sistemática do melhor prático e não de um melhor absoluto. Um problema de otimização pode ser generalizado da seguinte

maneira: dada uma função $f : \Re^n \rightarrow \Re$ e um espaço de busca $S \subseteq \Re^n$, o problema de otimização pode ser escrito como:

$$\text{minimizar} f(x) \mid x \in S \tag{1.4}$$

Sendo que pode-se considerar apenas a minimização pois um problema de minimização pode ser convertido em um problema de maximização usando artifícios como multiplicar a função (f) por -1 (TANOMARU, 1995).

Dependendo da complexidade da função $f(x)$, diferentes algoritmos de otimização podem ser utilizados. Tal que segundo Tanomaru (1995) os processos de otimização podem ser divididos em 3 (três) categorias: os métodos probabilísticos, numéricos e enumerativos.

- **Métodos Probabilísticos:** são métodos que usam a probabilidade para definir o resultado do problema, diferenciando estes de processos totalmente aleatórios.
- **Métodos Numéricos:** são métodos que se baseiam em cálculos numéricos e analíticos, e são aplicáveis basicamente quando a função f é completamente conhecida e derivável, ou pode ser aproximada por alguma outra função derivável até o grau de precisão desejado. Neste método também pode ser usado técnicas de gradiente em funções não-lineares e técnicas de pesquisa operacional quando a função é linear.
- **Métodos enumerativos:** examinam cada ponto do espaço de busca, a procura dos pontos ótimos, o que torna este método inviável quando o espaço de busca é infinito.

Os algoritmos bioinspirados fazem uso dos métodos probabilísticos, pois se baseiam na probabilidade de o resultado que está sendo calculado estar correto ou não, devido a este fato os algoritmos bioinspirados entregam um resultado aproximado, e poucas vezes entregam resultados exatos.

Sendo assim vista de forma global, a computação bioinspirada implementa diversos mecanismos adaptativos de otimização que simulam as diversas observações da natureza. Apesar desta incluir processos aleatórios estão longe de serem algoritmos de busca aleatória, pois, implementam métodos inteligentes para nortear a evolução e desenvolvimento na busca de um resultado satisfatório.

Uma vertente da computação evolucionária que pode ser citada são os Algoritmos Genéticos (AG), onde este é caracterizado por um sistema onde uma população de indivíduos é submetida a sucessivas seleções adaptativas ocorre um processo de especialização e espalhamento de indivíduos. Cada indivíduo da população é uma solução em potencial do problema $f(x)$, e trocam informações entre si durante a replicação e geração dos próximos indivíduos caracterizando o processo de *crossover* (*exploitation*), e espalham a busca dentro do *espaço de busca* através da mutação dos indivíduos gerados (*exploration*), sendo que o processo de

mutação ocorre durante a seleção, onde ao final toda a população é substituída por uma nova, que provavelmente está mais próxima do resultado.

Neste trabalho, é visado o estudo e otimização do desempenho do algoritmo de Otimização por Enxame de Partículas (PSO), sendo constituído por um enxame de partículas, além de também possuir métodos de *exploration* e *exploitation* de indivíduos. O PSO canônico trabalha com números reais, onde estes são a representação da posição de um indivíduo dentro do espaço de busca.

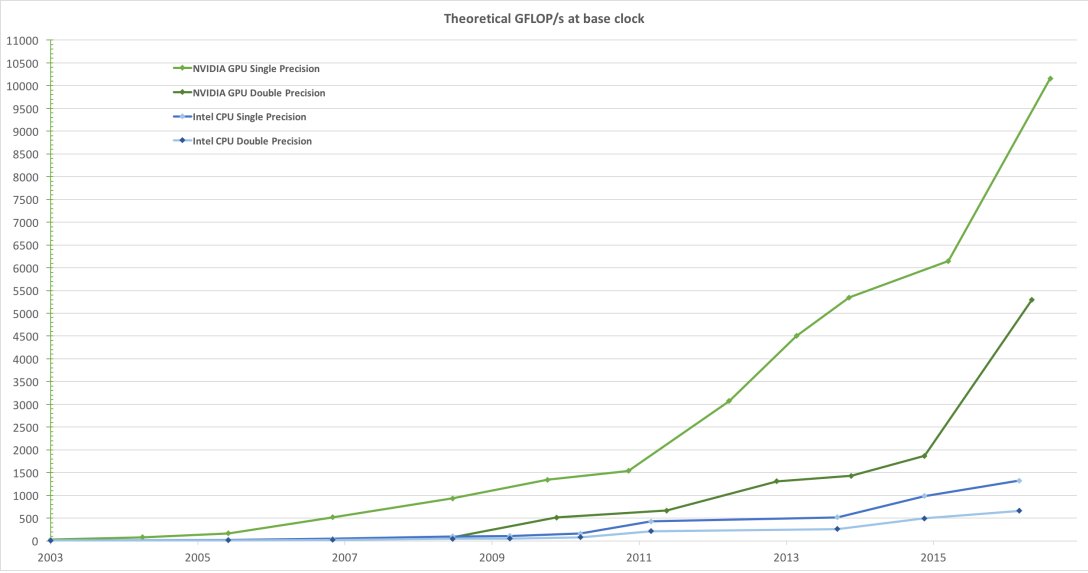
Considerando que o algoritmo de PSO trabalha individualmente com cada partícula e o processamento mais intenso deste se dá no cálculo da função de *fitness*, insere-se a possibilidade de trabalhar-se em um ambiente totalmente paralelizado.

1.2 GPU para Processamento Geral

A partir da concepção das primeiras GPUs no início dos anos 80, uma série de mudanças significativas ocorreram em seu desenvolvimento. Foram descobertas novas funcionalidades, e principalmente, um grande aumento na sua capacidade de processamento, que culminou na primeira década do século XXI em sua utilização para outros propósitos que não fosse processamento gráfico (SOUSA, 2010).

Logo no cenário mais recente impulsionada pela demanda insaciável do mercado por gráficos 3D de alta definição em tempo real, a unidade de processador gráfico programável ou GPU evoluiu para um processador *manycore* altamente paralelo, *multithreaded*, com grande potência computacional e largura de banda de memória muito alta, como pode ser visualizado na Figura 1 (NVIDIA, 2018).

Figura 1 – Operações de Ponto Flutuante por Segundo na GPU e CPU

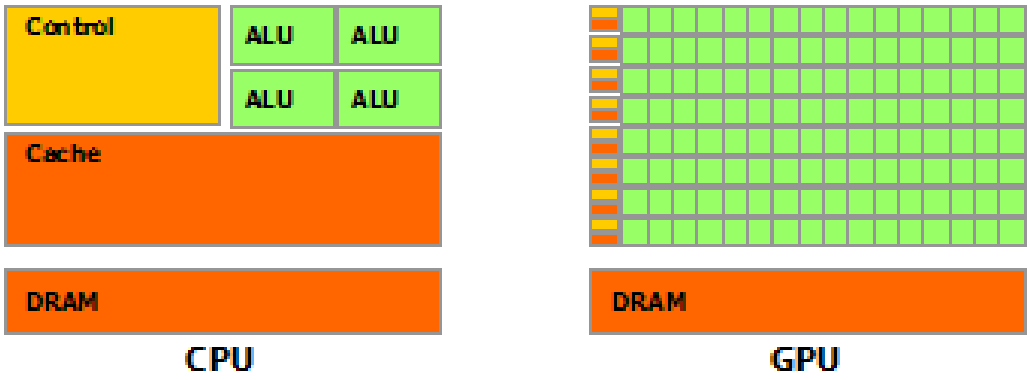


Fonte: (NVIDIA, 2018)

Este crescimento exponencial acentuado se dá pela estruturação dos núcleos de processamento na arquitetura da GPU. Categorizado como processador *manycore*, a GPU trabalha focada em processamento paralelo massivo, onde pode-se dividir a tarefa em pedaços menores que não possuam dependência funcional ou de dados, e executá-los simultâneamente.

Comparado com a CPU, categorizada como *multicore*, que busca acelerar o processamento de programas sequenciais, através da divisão de instruções entre núcleos. Esta divisão pode ser melhor visualizada na Figura 2

Figura 2 – Representação gráfica de comparativa entre arquitetura da CPU e GPU



Fonte: (NVIDIA, 2018)

Inicialmente a abordagem do uso da GPU para processamento geral era muito restrita, pois a GPU foi imaginada e desenvolvida para executar somente o *pipeline* do processamento gráfico. Então a única maneira de se utilizar desde hardware era através de APIs (*Application Programming Interface*) gráficas como o OpenGL e o DirectX, tal que a execução do processamento geral estaria dependente das funcionalidades destas APIs, contudo mesmo com tais limitações, os pesquisadores se utilizaram destas APIs para processamento de propósito geral, como se estas fossem parte do *pipeline* gráfico (SANDERS; KANDROT, 2010; SOUZA et al., 2014).

Apesar das limitações que as APIs traziam, os processadores *manycore* se mostravam superiores na velocidade de processamento, principalmente envolvendo operações de ponto flutuante simples. Com o aumento no uso de GPUs no desenvolvimento de aplicações gerais, em 2007 a NVIDIA lançou oficialmente a arquitetura CUDA, que trouxe uma flexibilidade muito maior na programação de GPUs para processamento geral, bem como a consolidação desta tecnologia em vários centro de pesquisa ao redor do mundo (HWU; KEUTZER; MATTSON, 2008; SOUZA et al., 2014).

Como pode ser visualizado na Figura 1, o aumento no desempenho das GPUs desde 2007 no calculo de pontos flutuantes simples foi aproximadamente multiplicado por 20, tal que desde meados de 2008 foi implementado o cálculo de números de pontos flutuantes de precisão dupla (*Double Precision*) na linha Tesla de GPUs da NVIDIA, funcionalidade que só viria a chegar para o grande público em meados de 2010 na série da GeForce GTX 465, com velocidades de

processamento ainda muito baixas. Antes disso só era possível realizar cálculos utilizando-se de dupla precisão nas placas quadro, da arquitetura Tesla de 2008 (NVIDIA, 2018).

Atualmente com a implementação de novas funcionalidades no CUDA, e com o aumento exponencial no poder de processamento das GPUs, tem-se a possibilidade de se executar códigos com mais precisão e velocidade em uma GPU recente, que é compatível com a computabilidade do CUDA mais recente.

É importante notar a definição do termo 'capacidade de computabilidade' (*compute capability*) cunhado pela NVIDIA, que se refere às especificações gerais e características disponíveis de uma determinada GPU.

1.3 Motivação

Com os novos lançamentos da série GeForce 10 da NVIDIA em 17 de maio de 2016, as novas GPUs, contendo a arquitetura Pascal que trouxe um grande aumento de desempenho comparado com a sua arquitetura antecessora (Maxwell), apesar do aumento no desempenho, o preço destas era comercialmente competitivo em comparação com a geração anterior. Juntamente com estes novos hardwares no mercado, novas versões da biblioteca CUDA foram disponibilizadas, permitindo a implementação, ou a melhora de desempenho em alguns detalhes da programação, como o calculo dos números de ponto flutuante com precisão dupla, quase alcançando o valor em GFLOP/s do calculo de ponto flutuantes de precisão simples da geração anterior, como pode ser visualizado na Figura 1 além na melhora da banda de acesso à memória (Figura 3) o que tem um grande impacto na avaliação do desempenho do algoritmo.

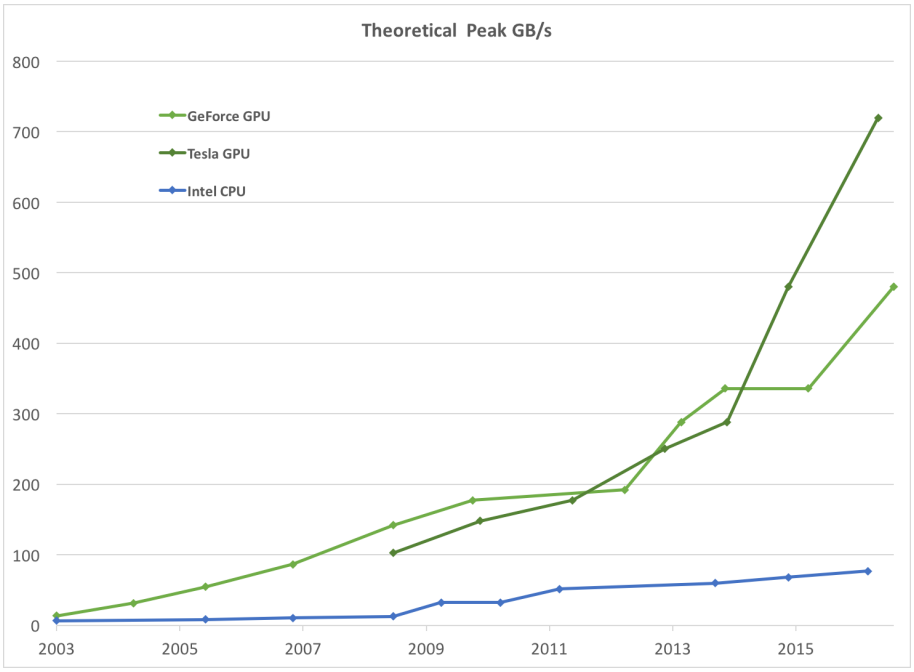
Levando em conta as grandes melhoras que as novas GPUs tem trazido para o mercado, neste capítulo será apresentado os objetivos que se procura alcançar com o desenvolvimento deste trabalho.

1.3.1 Objetivo Geral

O principal objetivo deste trabalho é buscar implementar as novas funcionalidades da *compute capability 6.1*, que possam trazer alguma melhora de desempenho na implementação do algoritmo de PSO com variáveis de dupla precisão, possibilitando assim um melhor desempenho no cálculo de resultados mais precisos, com confiabilidade que podem chegar a 16 dígitos decimais ($2^{-52} = 2.220446^{e-16}$).

As contribuições que este trabalho pode proporcionar é um código completo, bem estruturado e documentado, que pode ser reutilizado em projetos futuros, sendo assim disponibilizado como código aberto online.

Figura 3 – Pico na largura de banda da memória GPU e CPU



Fonte: (NVIDIA, 2018)

1.3.2 Objetivos Específicos

Este trabalho possui os seguintes objetivos específicos:

- A partir do algoritmo canônico do PSO, implementar a versão para o paradigma paralelizado do CUDA.
- Alcançar resultados que possuam mais casas decimais, permitindo a obtenção de mais precisão.
- Diminuir o tempo de processamento do algoritmo por completo, mesmo que se esteja utilizando variáveis de dupla precisão.

1.4 Metodologia

Na elaboração deste trabalho, foi feita uma pesquisa sobre as funcionalidades que beneficiariam o desenvolvimento e o aprimoramento da implementação do PSO, tomando por base os dados de teste gerados em (SOUZA et al., 2014). O pseudo-código desenvolvido, para guiar a implementação do algoritmo final teve como base o pseudo-código clássico do PSO com fator de inércia (SOUZA et al., 2014, p. 44), tal que a partir deste algoritmo foi implementada a versão do mesmo para o paradigma paralelizado do CUDA, e após a implementação deste os parâmetros de cada teste foram ajustados para que se obtivesse o melhor resultado e desempenho.

1.4.1 Funções Teste

Os problemas sujeitos a restrição são considerados problemas NP-difíceis, pois quando um problema é utilizado, sua representação matemática é representada por uma função que deve satisfazer uma série de restrições representada por outro conjunto de funções, segundo Silva, Teixeira e Oliveira (2012), tais restrições são importantes ao representar problemas de projeto de engenharia pois elas podem representar as restrições de espaço, força, dentre outras que se tem no mundo real, sendo que estas estão inseridas na descrição inicial do problema, e que por vezes, são difíceis de satisfazer.

As funções de teste restritivas (problemas de engenharia) utilizadas na avalização do desempenho destes algoritmos foram as seguintes:

- Minimização do Peso da Tensão/Compressão Sobre Molas (MWTCS);
- Projeto de Redutor de Velocidade de 11 Restrições (SRD11);

1.4.2 Estrutura do Trabalho

Este trabalho é dividido em capítulos que abordam o processo de pesquisa, desenvolvimento e avaliação de desempenho dos resultados obtidos e está dividido da seguinte maneira:

- **Capítulo 2:** Apresenta a fundamentação teórica comentando sobre o algoritmo do PSO, a estrutura da GPU e como as funcionalidades das novas versões que serão úteis no projeto.
- **Capítulo 3:** Comenta sobre a implementação, os pseudo-códigos apresentados, e a documentação do código implementado;
- **Capítulo 4:** Na análise de desempenho tem-se a descrição mais detalhada de cada um dos problemas de engenharia implementados no código. A demonstração de cada um dos resultados, e o comparativo com os outros testes previamente implementados em outras publicações.
- **Capítulo 5:** Apresenta as conclusões sobre a análise dos dados, as limitações que foram impostas pelo projeto e pelo objeto de estudo e comenta sobre os trabalhos finais que podem ser desenvolvidos a partir deste trabalho;

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta toda a fundamentação teórica que dá embasamento à este trabalho. Está contido neste capítulo tópicos relacionados à GPU para processamento geral, as versões e atributos analizados para a implementação neste trabalho, a estrutura da GPU, e por fim o funcionamento do PSO e como que ele foi modelado de forma a otimizar o seu funcionamento.

2.1 Processamento Geral Implementado em GPUs

Tendo como base o pensamento demonstrado pela NVIDIA no trecho:

> Com o advento das CPUs *multicore* e das GPUs *manycore* resignificou os processadores como sistemas paralelos, tal que, o paralelismo continua a escalar com a lei de Moore. O desafio é desenvolver softwares que escalam transparentemente este paralelismo juntamente com o crescente numero de processadores, como as aplicações de gráfico 3D que escalam transparentemente seu paralelismo para GPUs *manycore* com uma grande variação de numero de *cores*.
>
> O modelo de programação paralela em CUDA é desenvolvido para superar o desafio mantendo uma curva de aprendizagem baixa para programadores familiarizados com linguagens de programação padrões como C.(NVIDIA, 2018)

A utilização deste modelo de programação paralela CUDA, quando se tem o hardware necessário é um ótimo caminho a se seguir, em se tratando de programação paralela, devido a grande quantidade de documentação disponível e da baixa curva de aprendizado.

Concebida em 2006 e oficialmente lançada pela NVIDIA em 2007, a arquitetura CUDA revolucionou a programação paralela para processamento geral em GPUs, não fazendo mais uso de APIs com instruções relacionadas ao *pipeline* gráfico, mas sim utilizando-se das estruturas de hardware, permitindo que cada Unidade Lógica e Aritmética (ULA), possa ser utilizada por aplicações voltadas para cálculos e operações de propósito geral (SANDERS; KANDROT, 2010).

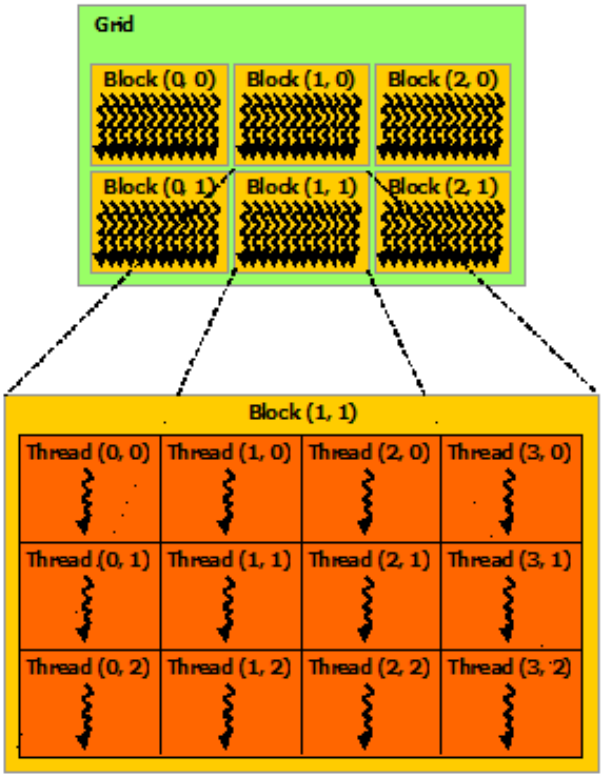
O ambiente CUDA em sua versão mais atual permite uma grande flexibilidade para o programador, que apesar de ter um inicio rápido na programação precisa de conhecimento técnico mais detalhado para produzir um código que de fato seja eficiente e faça uso de todas as funcionalidades disponíveis no hardware (NVIDIA, 2017).

2.1.1 Estrutura

Na organização da estrutura CUDA se têm 3 (três) níveis estruturais que podem ser visualizados na Figura 4.

Estes níveis são definidos da seguinte maneira:

Figura 4 – Malha (grid) de *Threads* e Blocks



Fonte: (NVIDIA, 2018)

- ***Thread*:** É a unidade básica da execução da arquitetura CUDA, sendo que cada *thread* é responsável por executar uma cópia das instruções, com uma parte dos dados, tal que várias *threads* juntas executem a mesma cópia das instruções para a totalidade dos dados (NVIDIA, 2017);

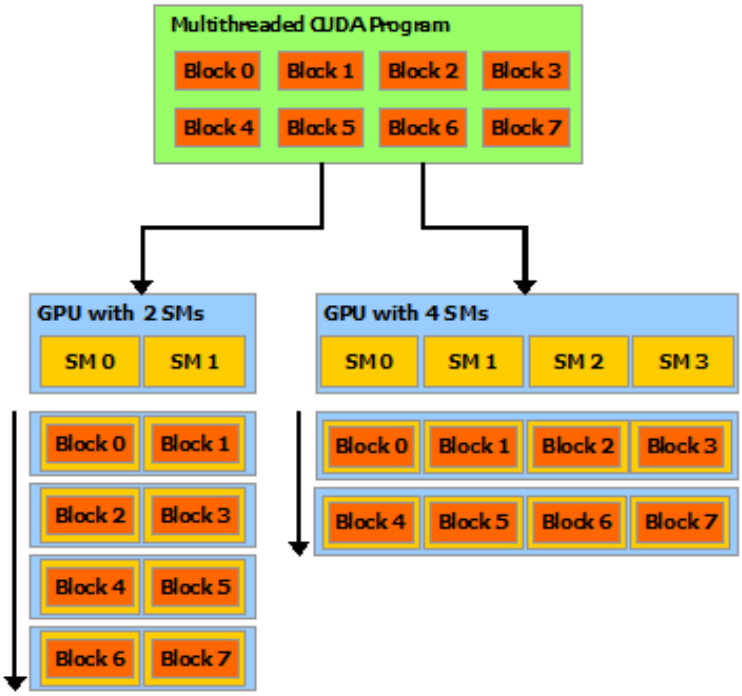
- ***Block*:** Armazena um vetor (podendo ser multidimensional suportando até 3 dimensões) de *threads*, tal que este agrupamento permite a sincronização das *threads* através de um comando. Um bloco de *threads* é sempre executado em um multiprocessador de fluxo (SM). Um bloco pode conter até 1024 *threads* sendo a dimensão máxima do bloco de *threads*(x,y,z): (1024, 1024, 64);

- ***Grid*:** Armazena um conjunto de blocos que executam a mesma instrução dentro da GPU. Uma grid distribui dinamicamente os blocos nos multiprocessadores de fluxo (SM) disponíveis na GPU como é demonstrado na Figura 5. Uma grid pode ter a dimensão máxima de(x,y,z): (2147483647, 65535, 65535);

2.1.2 O Papel da Memória na GPU

Além da velocidade de processamento a GPU precisa ter uma gerência de memória avançada que consiga acompanhar o paralelismo e a velocidade de processamento, evitando

Figura 5 – Escalabilidade automática na execução dos blocos



Fonte: (NVIDIA, 2018)

assim que ocorram gargalos no programa. Sendo assim, divide-se a memória da GPU CUDA em 6 partes, classificadas abaixo por ordem de velocidade:

1. ***Register*** *memory*
2. ***Shared*** *memory*
3. ***Constant*** *memory*
4. ***Texture*** *memory*
5. ***Local*** e ***Global*** *memory*

Vale ressaltar que os únicos tipos de memória que residem no chip da GPU são a *Register* e a *Shared memory*, tal que as outras *Constant*, *Texture*, *Local* e *Global memory* ficam fora do chip. Logo, seria o mais lógico que em um programa desenvolvido em CUDA somente se utilizasse a *Register* e a *Shared memory*, entretanto algumas outras variáveis devem ser levadas em conta quando se trata do uso da memória em CUDA como o escopo e o tempo de vida da mesma.

- A ***Register Memory*** tem o escopo da *thread*, e só existe durante o tempo de vida desta, ou seja, dados alocados dentro da *Register memory* não podem ser compartilhados com outras *threads*, e vão deixar de estar alocados quando a *thread* finalizar seu processamento.

- A ***Local Memory*** não existe fisicamente, pois é uma abstração do escopo de uma *thread*. Contudo ela é armazenada na *Global Memory* e tem o mesmo escopo e tempo de vida que a *Register Memory*, contudo o seu desempenho é muito mais lento (mesmo desempenho da *Global Memory*).

- A ***Shared Memory*** possui o escopo de um bloco, e o tempo de vida deste, sendo assim, os dados alocados na *shared memory* podem ser compartilhados entre as *threads* de um mesmo bloco, possibilitando que as *threads* possam se comunicar umas com as outras de maneira rápida.

- A ***Global Memory*** é visivel à toda aplicação, inclusive ao *host*, e tem o tempo de vida igual ao da aplicação do *host*.

- A ***Costant Memory*** é utilizada para dados constantes, que não serão alterados no curso de uma execução do kernel, e é uma memória somente leitura, como a *Constant Memory*, a ***Texture Memory*** é um outro tipo de memória somente leitura, contudo ela é desenvolvida para leituras sequenciais de grande escala, onde se pode ter uma redução no tráfego de memória e aumento na performance comparado à *Global Memory*.

Pode-se visualizar na Figura 6 e no Quadro 1 a disposição e interação dos vários tipos de memória, seus escopos e declarações, de forma resumida.

Quadro 1 – Tipos de Memória, e suas declarações.

Declaração da Variável	Memória	Escopo	Tempo de Vida
Variáveis que não são vetores	Registrador	Thread	Kernel
Vetores	Local	Thread	Kernel
__device__ __shared__ int sharedVar;	Shared	Block	Kernel
__device__ int globalVar	Global	Grid	Application
__device__ __constant__ int constantVar;	Constant	Grid	Application

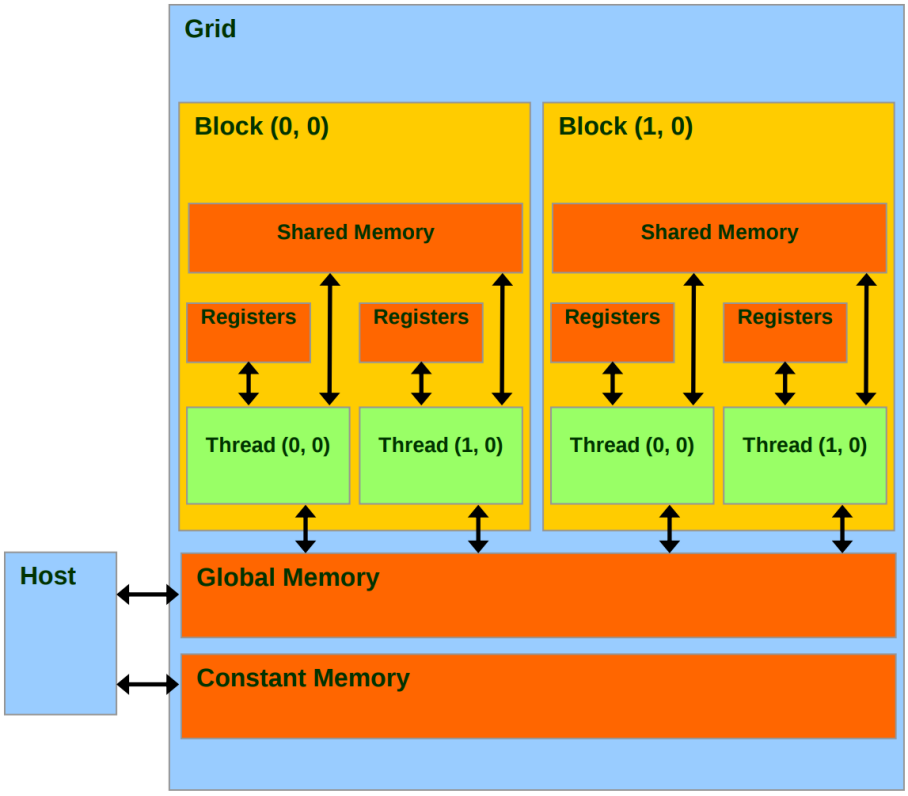
Fonte: Nvidia (2018)

Neste projeto a memória constante e a memória de textura não foram utilizadas, pois os dados constantes durante um algoritmo do PSO não são significativos o suficiente para se fazer necessária a utilização destas memórias em específico.

2.2 PSO

Introduzido por Kennedy J. e Eberhart R. em 1995, o algoritmo de otimização por enxame de partículas (PSO) é um método de otimização de funções contínuas e não lineares. Este método foi descoberto pela simulação de um modelo de interação social simplificado (KENNEDY, 1995). Descrito por Kennedy (1995), o PSO é embasado por dois elementos complementares, a vida

Figura 6 – Escalabilidade automática na execução dos blocos



Fonte: (NVIDIA, 2018)

artificial, e cardumes de peixes, bando de pássaros, ou nas teorias de enxames em particular. Mas, também, está relacionado com a computação evolucionária, e tem ligações tanto com algoritmos genéticos quanto com programação evolucionária.

O PSO se assemelha com outras técnicas evolucionárias por também fazer uso de funções de avaliação (*fitness*), assim como os algoritmos genéticos (GA), que tem o objetivo de medir o desempenho de cada indivíduo dentro da população, contudo estas se diferenciam no tocante a evolução, sendo que esta (GA) substitui os indivíduos através dos processos evolutivos (i.e. seleção natural, mutação) fazendo surgir novos elementos que trazem a totalidade, parte ou até mesmo novas informações comparado com a geração anterior a primeira; já no PSO recalcula-se as velocidades e o direcionamento de cada partícula e avalia a posição dentro do espaço de busca através de processos de mecânica básica, para assim encontrar o melhor global, compartilhando estas informações com a totalidade das partículas ou com a sua vizinhança (pode variar de acordo com a implementação empregada).

2.2.1 Implementação

Como descrito por Fulcher (2008), no algoritmo do PSO a posição de uma partícula codifica uma possível solução para o problema a se resolver, idealmente uma solução ótima, mas uma solução subótima, dentro de uma margem de erro, também pode ser aceita. Estas partículas

se movem regidas por uma série de fatores, de maneira que elas tendem a convergir para a solução ótima.

Cada partícula do enxame é responsável por manter registro da sua posição e velocidade, seu *fitness*, e a melhor posição que esta já encontrou dentro do espaço de busca (*pbest*), sendo que o enxame fica responsável por armazenar a melhor posição já encontrada pelo conjunto de partículas (*gbest*). Estas informações são importantes para o cálculo do reposicionamento de cada partícula tal que o usuário define previamente constantes (C_1 e C_2) que dão peso à estas informações.

Se considerarmos que todas as partículas tem peso, então elas não podem sofrer alterações de direção bruscas, logo a atualização da velocidade deve levar em conta um componente que contem a mesma direção que a velocidade calculada anteriormente, isso efetivamente provê uma suavização na movimentação das partículas pelo espaço de busca. Pode-se chamar esta variável de fator de Inércia (w).

Nesta seção será descrita as mais importantes constantes, os cálculos do fator de inércia, de atualização de velocidade e posição, os métodos de contorno e por fim o algoritmo que serviu de base para a implementação do código. A Figura 8 ilustra a movimentação de uma partícula dentro do espaço de busca

Constantes Individual e Social

As constantes Individual (C_1) e Social (C_2) tem a função de valorizar mais os melhores valores encontrados individualmente por cada partícula ou então o melhor valor já encontrado pelo enxame, tal que numa proporção $C_1 > C_2$ os valores individuais tem mais peso, e importância no resultado do que os melhores valores globais, mas por outro lado se $C_1 < C_2$ as importâncias se invertem, fazendo com que o algoritmo tenha outro comportamento na busca do resultado.

Fator de Inércia

O fator de inércia (w) traz para o algoritmo algo parecido com as constantes social e individual, ela define a importância da velocidade anterior sobre o próximo movimento, ou seja, ela define a inércia no movimento da partícula. Existem meios lineares (2.1) e estocásticos (2.2) de se calcular o valor e w:

$$w = \frac{(k-1)}{(I_{MAX}-1)(-W_{MAX}+W_{MIN})+W_{MAX}} \tag{2.1}$$

$$w = W_{MAX} * rand + W_{MIN}(1-rand) \tag{2.2}$$

As variáveis descritas em 2.1 e 2.2 são: Fator de Inércia (w); Iteração atual (k); Número

máximo de iterações (I_{MAX}); Limite máximo e mínimo para o fator de inércia (W_{MAX} e $-W_{MIN}$); Número gerado uniformemente entre 0 e 1 (*rand*).

É importante frisar que neste trabalho o cálculo linear não pode ser utilizado pois não foi utilizado um valor fixo de iterações, pois era necessário trazer mais flexilidade aos testes, fazendo com que a busca pelo melhor global pudesse acontecer livremente até o ponto em que o algoritmo não fosse mais passível de evolução.

Atualização da Velocidade e Posição

As partículas se movimentam pelo espaço de busca, de forma que são guiadas pelo melhor global que já foi encontrado pelo enxame, pela melhor posição que já foi encontrada pela própria partícula e pela velocidade anterior desta partícula. Tal que os dois primeiros fatores controlados pelas constantes C_2 e C_1 respectivamente que são utilizadas para controlar a importância da informação pessoal, em detrimento da informação social, e o último controlado pela variável w, além de variáveis aleatórias que ajudam a distribuir as partículas pelo espaço, permitindo assim uma melhor exploração do mesmo.

Sabendo disso, podemos considerar que a avaliação da evolução do algoritmo do PSO pode ser analisada através da trajetória do enxame pelo espaço de busca, e do tempo utilizado para que a solução ótima (ou subótima) seja encontrada.

Com isso temos as equações 2.3 e 2.4 que representam o cálculo da atualização da velocidade e da posição de cada partícula, respectivamente:

$$V_i(k+1) = wV_i(k) + R_1C_1(pbest - X_i(k)) + R_2C_2(gbest - X_i(k)) \tag{2.3}$$

$$X_i(k+1) = X_i(k) + V_i(k+1) \tag{2.4}$$

As variáveis descritas em 2.3 e 2.4 são: Iteração atual da partícula (k); Fator de inércia (w); Índice da partícula (i); Valor atual de velocidade ($V_i(k)$); Novo valor de velocidade ($V_i(k+1)$); Valor atual de posição ($X_i(k)$); Novo valor de posição ($X_i(k+1)$); Melhor solução encontrada pela partícula (*pbest*); Melhor solução encontrada pelo enxame (*gbest*); Fatores de interação individual e social (C_1 e C_2); Números aleatórios gerado uniformemente entre 0 e 1 (R_1 e R_2).

Para melhor entendimento da equação 2.3, abaixo se têm as 3 porções do cálculo da velocidade sendo 2.5 a porção que calcula a inércia da partícula, considerando a velocidade anterior, e o fator de inércia; 2.6 a porção que considera o melhor pessoal (*pbest*) e utiliza a Constante Individual (C_1) e uma variável aleatória (R_1) para calcular a importância da "opinião pessoal"da própria partícula na direção que ela deve seguir; e 2.7 a porção que considera o melhor

global (*gbest*) e utiliza a Constante Social (C_2) e outra variável aleatória (R_2) para calcular a importância da "opinião do enxame"sobre a direção que a partícula deve seguir;

$$inercia = wV_i(k) \tag{2.5}$$

$$individual = R_1C_1(pbest - X_i(k)) \tag{2.6}$$

$$social = R_2C_2(gbest - X_i(k)) \tag{2.7}$$

$$V_i(k+1) = inercia + individual + social \tag{2.8}$$

O resultado da soma dos valores de *inercia*, *individual*, *social* é igual à nova velocidade da partícula que será somada com a posição anterior, para que seja calculada a nova posição da partícula na função 2.4.

Condições de Contorno de Posição e Velocidade

Cada partícula no PSO, deve estar dentro de um espaço de busca, definido na inicialização do algoritmo 2.9, tal que após a atualização da posição de cada partícula, estas são submetidas à verificação de localização, e caso esta esteja fora deste espaço de busca são aplicadas as regras de contorno ("*Damping*") para velocidade 2.10 e para posição 2.11

$$X_{MIN} \leq X_i(k) \leq X_{MAX} \tag{2.9}$$

$$V_i(k+1) = \begin{cases} -V_i(k+1)R_1 & se\ X_i(k+1) < X_{MIN} \\ V_i(k+1) & se\ X_{MIN} \leq X_i(k+1) \leq X_{MAX} \\ -V_i(k+1)R_1 & se\ X_i(k+1) > X_{MAX} \end{cases} \tag{2.10}$$

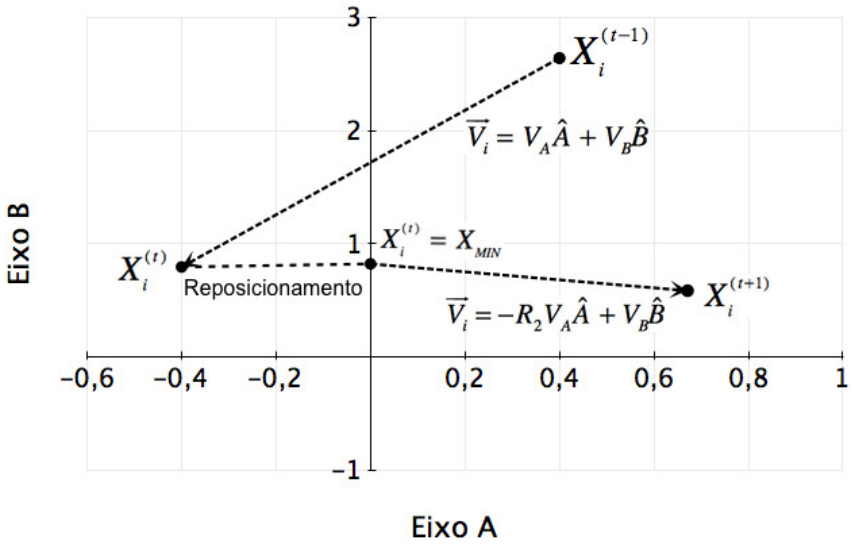
$$X_i(k+1) = \begin{cases} X_{MIN} & se\ X_i(k+1) < X_{MIN} \\ X_i(k+1) & se\ X_{MIN} \leq X_i(k+1) \leq X_{MAX} \\ X_{MAX} & se\ X_i(k+1) > X_{MAX} \end{cases} \tag{2.11}$$

As variáveis descritas em 2.10 e 2.11 são: Iteração atual da partícula (k); Índice da partícula (i); Novo valor de velocidade ($V_i(k+1)$); Novo valor de posição ($X_i(k+1)$); Valores mínimo e máximo que a variável pode assumir (X_{MIN}); Número aleatório gerado uniformemente entre 0 e 1 (R_1).

A Figura 7 ilustra a função de reposicionamento no espaço de busca "*Damping*"

Algoritmo e Ilustração da Movimentação das Partículas

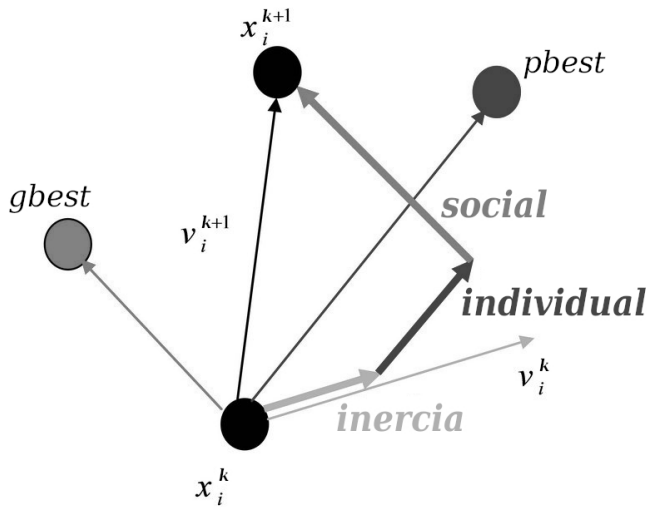
Figura 7 – Representação gráfica da condição de *Damping*



Fonte: (SOUZA et al., 2014)

A Figura 8 demostra a movimentação de uma partícula segundo as funções 2.3 e 2.4, pode-se visualizar a influência das constantes C_1 e C_2 e variáveis w, R_1 e R_2, através da diferença entre o vetor original (*gbest*, *pbest* ou $V_i(k)$) e o vetor resultado (*social*, *individual* ou *inercia*).

Figura 8 – Escalabilidade automática na execução dos blocos



Algoritmo 1: PSO clássico com fator de inércia e condição de contorno

```
Inicializar as partículas do enxame(posição(X), velocidade(V), melhorLocal(pbest));
Inicializar as variáveis do enxame(melhorGlobal(gbest));
for i ← 0 to (QUANTIDADE DE PARTÍCULAS − 1) do
    for j ← 0 to (QUANTIDADE DE VARIÁVEIS − 1) do
        Inicializar os valores de Posição com valores randômicos;
        Inicializar os valores de Velocidade com valores randômicos;
    fitness ← Avaliar fitness da partícula(P_i);
    pbest ← Inicializar o melhor local da partícula(P_i);
gBest ← Obter o melhor global do enxame();
for iter ← 0 to (QUANTIDADE DE ITERAÇÕES) do
    w ←Atualiza o fator de inércia(Equação 2.1 ou 2.2);
    for i ← 0 to (QUANTIDADE DE PARTÍCULAS − 1) do
        for j ← 0 to (QUANTIDADE DE VARIÁVEIS − 1) do
            Atualiza a velocidade de P_i na variável X_j (Equação 2.3);
            Atualiza a posição de P_i na variável X_j (Equação 2.4);
            Atualiza condições de contorno na posição da variável X_j (Equação 2.11);
            Atualiza condições de contorno na velocidade da variável X_j (Equação 2.10);
        fitness ← Avaliar fitness da partícula(P_i);
        pbest ← Atualizar o melhor fitness encontrado pela partícula(P_i);
    gBest ← Obter o melhor global do enxame();
```

2.3 Descrição dos Problemas de Engenharia

Os problemas de otimização restritiva na área da engenharia envolvem o uso de condições restritivas comumente associadas a um intervalo de valores que uma variável pode assumir, ou as condições também podem refletir relação existente entre as variáveis, tal que a não satisfação se enquadra em uma violação (TEIXEIRA et al., 2012).

O exemplo abaixo retrata a abstração dos problemas restritivos mostrando um exemplo de restrição cujo resultado tende a valores menores ou iguais a 0 (zero) e a função:

$$\text{Minimizar} f(X_i) \tag{2.12}$$

Sujeito à:

$$g \leq 0 \tag{2.13}$$

Onde:

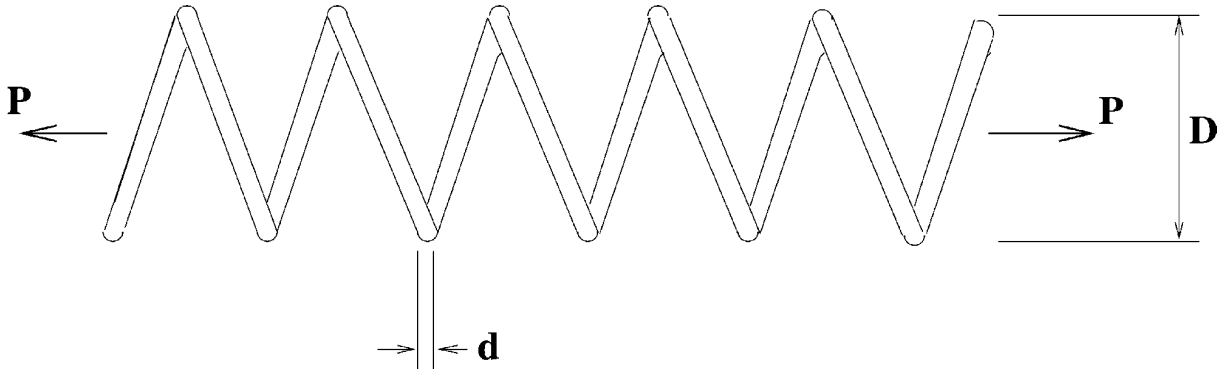
- X_i: Valores de parâmetro da função;
- $f(Xi)$: Resultado da função;
- g: função de restrição;

Vale também ressaltar que a quantidade de funções de restrição, pode variar de acordo com o problema em questão.

2.3.1 Peso da Tensão/Compressão sobre Mola (MWTCS)

O esquema de Peso da Tensão/Compressão sobre Mola (MWTCS), segundo Coello e Montes (2002) se resume na minimização do peso da tensão/compressão sobre uma mola, que está sujeita a algumas restrições, tais como deflexão mínima, tensão do cisalhamento, frequência de onda, limites de diâmetro externo e variáveis de projeto. As variáveis do problema MWTCS são: Diâmetro do fio (d), diâmetro da bobina (D), número máximo de bobinas ativas (P), a Figura 9 representa graficamente o esquema do MWTCS.

Figura 9 – Esquema do Peso da Tensão/Compressão sobre Mola



Fonte: (COELLO; MONTES, 2002)

Abaixo a formalização matemática do MWTCS:

$$\text{Minimizar} f(X_i) = (P+2)Dd^2 \tag{2.14}$$

Sujeito a:

$$g1 = 1 - \frac{D^3 P}{7178 d^4} \leq 0 \tag{2.15}$$

$$g2 = \frac{4D^2 - dD}{12566(Dd^3 - d^4)} + \frac{1}{5108d^2} - 1 \leq 0 \tag{2.16}$$

$$g3 = 1 - \frac{140,45d}{D^2 P} \leq 0 \tag{2.17}$$

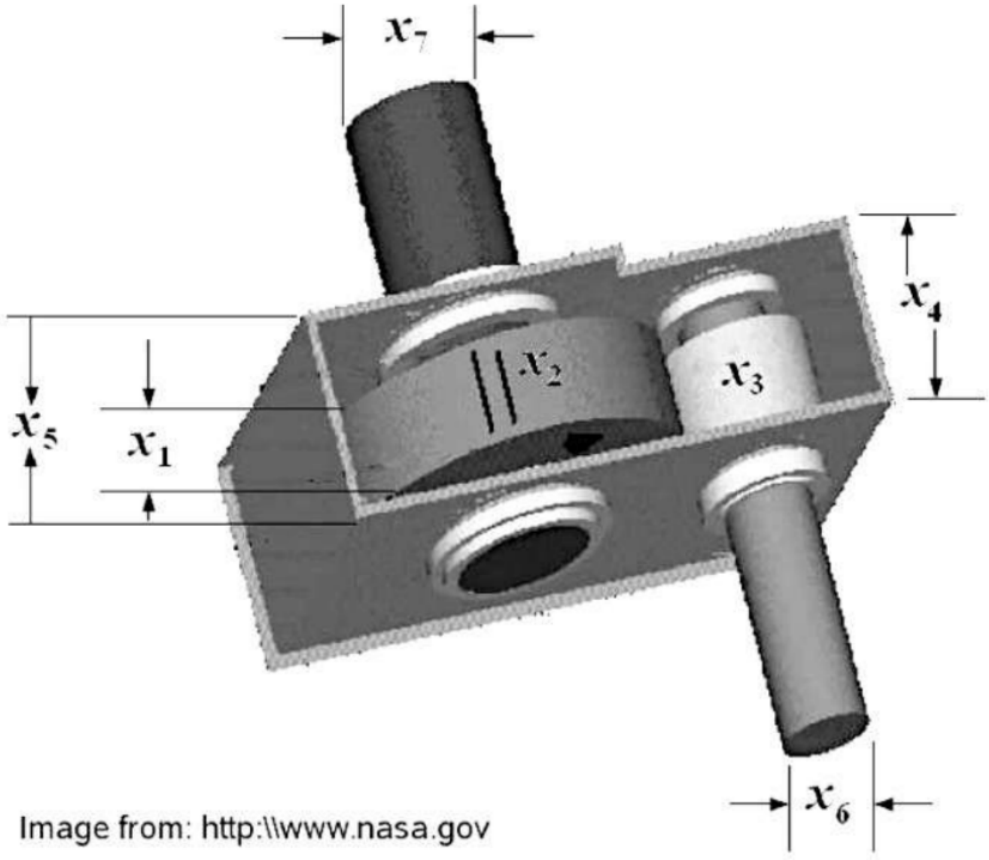
$$g4 = \frac{D+d}{1,5} - 1 \leq 0 \tag{2.18}$$

Os limites das variáveis d, D e P são: $0,05 \leq d \leq 2,0$; $0,25 \leq D \leq 1,3$; $2,0 \leq P \leq 15,0$.

2.3.2 Projeto de Redutor de Velocidade (SRD11)

O Projeto de Redutor de Velocidade (SRD) consiste em um sistema de redução de velocidade através da minimização do peso obedecendo certas restrições como: Flexão de estresse dos dentes da engrenagem, tensão superficial, desvios transversais das hastes, e as tensões no eixo Teixeira et al. (2011). As variáveis do problema SRD são: Largura do Rosto (x1); Módulo de dentes (x2); Número de dentes do pinhão (x3); Comprimento do primeiro eixo entre os rolamentos (x4); Comprimento do segundo eixo entre os rolamentos (x5); Diâmetro do primeiro eixo (x6); Diâmetro do segundo eixo (x7).

Figura 10 – Representação Gráfica de um Redutor de Velocidade



Este é o mais complexo dos problemas, contudo, para adicionar este ao código não foi necessário tanto esforço pois o primeiro foi desenvolvido com o intuito de que se adicionassem outros testes posteriormente.

Abaixo a formalização matemática do SRD:

$$\begin{aligned}\text{Minimizar} f(X_i) &= 0,7854X_1X_2^2(3,3333X_3^2+14,933X_3-43,0934)\\ &-1,508X_1(X_6^2+X_7^2)+7,4777(X_6^3+X_7^3)\\ &+0,78054(X_4X_6^2+X_5X_7^2)\end{aligned} \tag{2.19}$$

Sujeito a:

$$g1 = \frac{27}{X_1X_2^2X_3} - 1 \leq 0 \tag{2.20}$$

$$g2 = \frac{397,5}{X_1X_2^2X_3^2} - 1 \leq 0 \tag{2.21}$$

$$g3 = \frac{1,93X_4^3}{X_2X_3X_6^4} - 1 \leq 0 \tag{2.22}$$

$$g4 = \frac{1,93X_5^3}{X_2X_3X_7^4} - 1 \leq 0 \tag{2.23}$$

$$g5 = \frac{1}{110X_6^3}\sqrt{\left(\frac{745X_4}{X_2X_3}\right)^2+1,69\times10^6} - 1 \leq 0 \tag{2.24}$$

$$g6 = \frac{1}{85X_7^3}\sqrt{\left(\frac{745X_5}{X_2X_3}\right)^2+157,5\times10^6} - 1 \leq 0 \tag{2.25}$$

$$g7 = \frac{X_2X_3}{40} - 1 \leq 0 \tag{2.26}$$

$$g8 = \frac{5X_2}{X_1} - 1 \leq 0 \tag{2.27}$$

$$g9 = \frac{X_1}{12X_2} - 1 \leq 0 \tag{2.28}$$

$$g10 = \frac{1,5X_6+1,9}{X_4} - 1 \leq 0 \tag{2.29}$$

$$g11 = \frac{1,1X_7+1,9}{X_5} - 1 \leq 0 \tag{2.30}$$

Os limites das variáveis $X1, X2, X3, X4, X5, X6$ e $X7$ são: $2,6 \leq X1 \leq 3,6$; $0,7 \leq X2 \leq 0,8$; $17 \leq X3 \leq 28$; $7,3 \leq X4 \leq 8,3$; $7,8 \leq X5 \leq 8,3$; $2,9 \leq X6 \leq 3,9$; $5,0 \leq X7 \leq 5,5$.

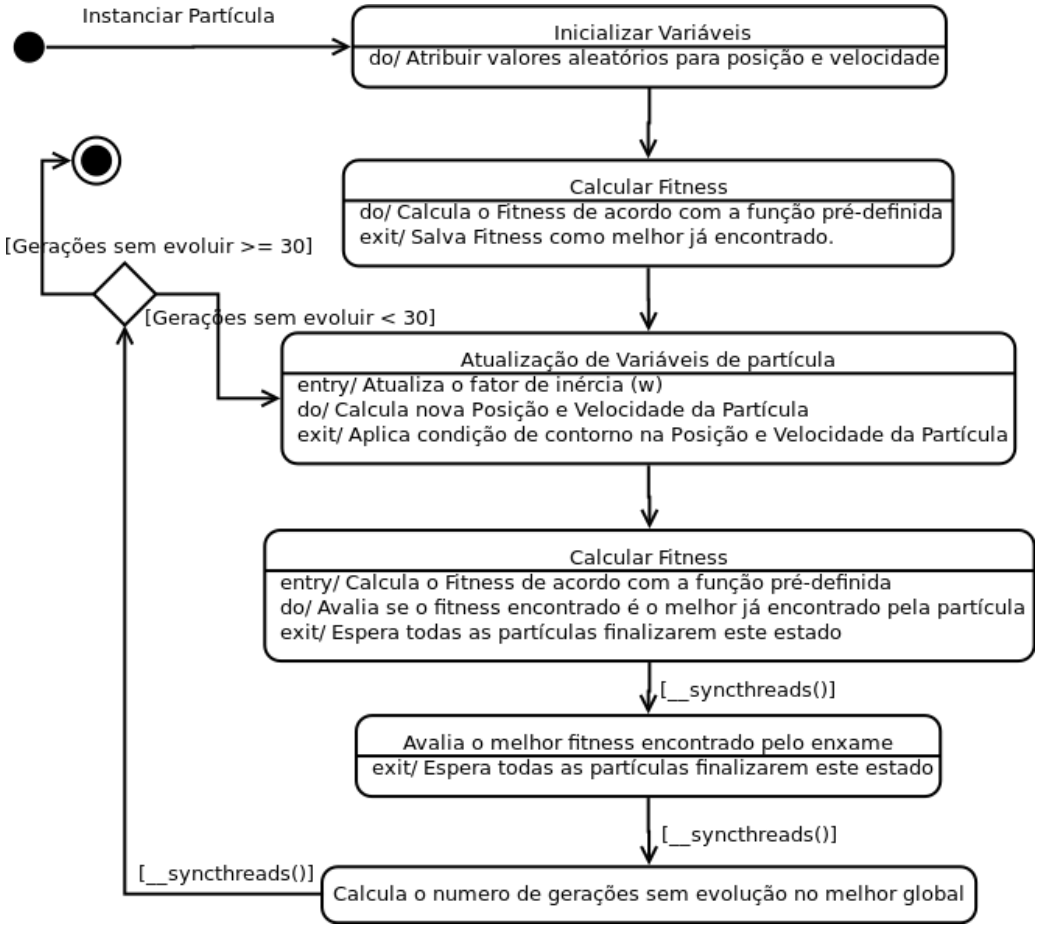
3 DESENVOLVIMENTO DO SISTEMA DE OTIMIZAÇÃO DO PSO EM GPU

Neste capítulo será apresentado o projeto que deu base à implementação, com o diagrama de estados, a descrição do algoritmo que foi de fato implementado, e como foio planejamento para obter uma melhora no desempenho.

3.1 Projeto do Software

Após a implementação da base do algoritmo do PSO em CUDA, foi possível tomar um conhecimento maior sobre o funcionamento da linguagem e como era possível organizar o código de maneira que ele fosse facilmente expansível e testável. Com esse requisito em mente, de permitir que este algoritmo fosse facilmente reutilizado, de forma que adicionar novos testes à este seja trivial, foi montado o diagrama de estados demontrado na Figura 11, representando os estados que a partícula passa durante o processo do PSO já considerando a estrutura do CUDA.

Figura 11 – Diagrama de estado de uma partícula (*thread*)



Fonte: (SOUZA et al., 2014)

3.1.1 Descrição dos Algoritmos Implementados

Como descrito na sessão 2.1.2, foi utilizada a "*shared memory*"para manter as informações do enxame compartilhadas entre todas as *trheads*. Caso a "*shared memory*"não tivesse sido utilizada, as informações somente poderiam ser compartilhadas através da "*global memory*", contudo, esta é 10 vezes mais lenta do que a "*shared memory*"como explicado na sessão 2.1.2. Pode-se assim afirmar que a banda de memória é o *"bottleneck"* (gargalo) neste processo, pois como todas as variáveis sempre precisam estar registradas em memória, a velocidade de acesso à esta limita a velocidade de execução do algoritmo, e por este motivo, esta foi o principal foco neste trabalho, melhorar o desempenho no acesso à memória durante o algoritmo.

Além das informações sobre o *fitness* da partícula, a "*shared memory*"também foi utilizada para que pudesse ser implementado um elemento de parada no algoritmo, fazendo com que este buscasse a solução até que esta solução não melhorasse por uma quantidade determinada de iterações, fazendo com que não seja necessário que se saiba de antemão a quantidade de iterações ótima para que a solução ótima daquele problema seja encontrado. Esta alteração no funcionamento do algoritmo terá grande impacto na análise dos resultados que será feita mais adiante neste trabalho no Capítulo 4.

Abaixo a descrição do algoritmo implementado:

Algoritmo 2: PSO implementado na GPU

```
1  Inicialização das variáveis para posterior análise gráfica;
2  Alocação de memória na GPU das variáveis(X_MAX, X_MIN, e outras variáveis de análise);
3  Inicializar função de aleatoriedade do cuda(initCuRand());
4  for g ← 0 to (QUANTIDADE DE EXECUÇÕES − 1) do
5      Cálculo do tamanho para a sharedMemory;
6      Inicializar e dividir a sharedMemory para as variáveis necessárias;
7      Inicializar as partículas do enxame(posição(X), velocidade(V), melhorLocal(pbest);
8      Inicializar as variáveis do enxame(melhorGlobal(gbest));
9      for j ← 0 to (QUANTIDADE DE VARIÁVEIS − 1) do
10         Inicializar os valores de Posição com valores randômicos;
11         Inicializar os valores de Velocidade com valores randômicos;
12     fitness ← Avaliar fitness da partícula(P_i);
13     pbest ← Inicializar o melhor local da partícula(P_i);
14     gBest ← Obter o melhor global do enxame();
15     Sincroniza GPU;
16     while geracoes_sem_evoluir < 30 do
17         w ←Atualiza o fator de inércia(Equação 2.2);
18         for j ← 0 to (QUANTIDADE DE VARIÁVEIS − 1) do
19             Atualiza a velocidade de P_i na variável X_j (Equação 2.3);
20             Atualiza a posição de P_i na variável X_j (Equação 2.4);
21             Atualiza condições de contorno na posição da variável X_j (Equação 2.11);
22             Atualiza condições de contorno na velocidade da variável X_j (Equação 2.10);
23         fitness ← Avaliar fitness da partícula(P_i);
24         pbest ← Atualizar o melhor fitness encontrado pela partícula(P_i);
25         Sincroniza GPU;
26         gBest ← Obter o melhor global do enxame();
27         Sincroniza GPU;
28         if bestViolacao < bestViolacaoItAnterior ou
             (bestViolacao = bestViolacaoItAnterior e gBest < gBestItAnterior) then
29             geracoes_sem_evoluir ← 0;
30             bestViolacaoItAnterior ← bestViolacao;
31             gBestItAnterior ← gBest;
32         else
33             geracoes_sem_evoluir ← geracoes_sem_evoluir + 1;
34         Sincroniza GPU;
35     Copia o melhor fitness encontrado para a memória global;
36 Imprime os resultados na saída da execução;
```

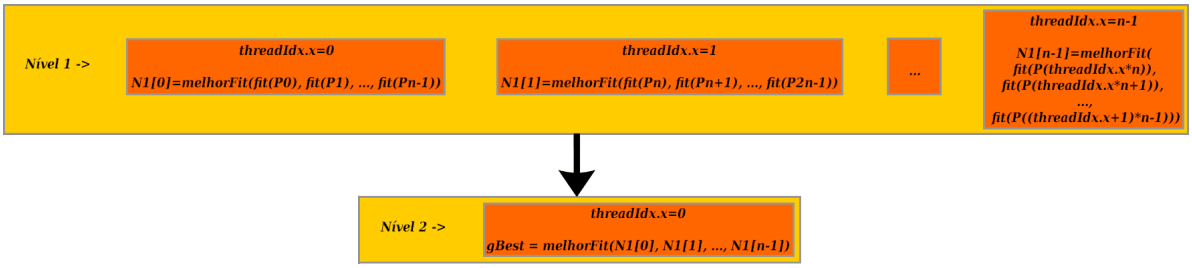
3.1.2 Otimização e Desempenho

Após a análise do Algoritmo 1, pode-se facilmente visualizar a sua falta de eficiência computacional, tendo um comportamento de $\mathcal{O}(n^3)$ no pior e no melhor caso. A partir disso o objetivo de se implementar o PSO em uma GPU vem da grande capacidade de paralelização deste algoritmo, pois apesar de a informação ser interdependente em certo nível, esta pode ser particionada e paralelizada no momento mais crítico onde ocorrem os cálculos mais onerosos do algoritmo (linhas 11 à 17 e 20 do Algoritmo 1).

A ideia para a paralelização deste algoritmo em um ambiente GPGPU é de utilizar uma *thread* para calcular uma partícula, tal que pode-se sempre utilizar o número máximo de *trheads* disponíveis em um bloco, sendo o número máximo, de acordo com a versão CUDA utilizada de 1024 *threads* dentro do bloco, contudo na placa gráfica utilizada só se dispõe de 640 *CUDA cores* sendo assim foi preferido utilizar o número máximo de 640 partículas calculadas simultaneamente, cada uma individualmente por uma *thread*. Caso se tivesse escolhido utilizar mais de 640 *threads* simultaneamente, seria causado um *overhead* no cálculo fazendo-se necessário o escalonamento dos processos dentro dos cuda *CORES*.

Após o cálculo de cada iteração de uma partícula, estas se sincronizam e aguardam o cálculo do melhor global, que também pode ser paralelizado utilizando-se um grupo de *threads* da forma representada na Figura 12, onde cada uma das n *threads* no primeiro nível (n definido antes do início do algoritmo) encontra o melhor entre um grupo de $n-1$ partículas, e o nível subsequente somente precisa trabalhar na busca entre os n resultados do nível anterior.

Figura 12 – Representação gráfica do cálculo do melhor global



Realizando estas melhoras no algoritmo pode-se considerar ele agora com comportamento de $\mathcal{O}(n^2)$ pois as iterações nas partículas (linhas 3 e 12 do algoritmo 1) foram paralelizadas.

Sendo assim, o desempenho do algoritmo pode melhorar bastante, contudo ainda se encontra a barreira da velocidade de acesso à memória, contudo como descrito em 2.1.2, pode-se ter acesso à *Shared Memory* durante a execução de uma *thread* do *kernel* sendo este acesso mais rápido do que o acesso à Memória Global, pois a *Shared Memory* se encontra dentro do bloco, tornando-a mais próxima da *thread* e por ser compartilhada apenas com as *threads* e por estar no mesmo nível de acesso de um registrador, a latência se torna menor fazendo com que a *thread* fique menos tempo ociosa aguardando por um dado.

Outros pontos possíveis de otimização que não foram implementados neste trabalho são a utilização de uma variável de precisão simples para realizar a aproximação do resultado e depois utilizar uma variável de precisão dupla para aproximar as casas decimais restantes, este procedimento tem o poder (em teoria) de aumentar em 32 (trinta e duas) vezes o poder de processamento, pois nos chips GeForce da série 10 se têm 32 cores CUDA de precisão simples (Single Precision CUDA Core) para 1 core CUDA de precisão dupla (Double Precision CUDA Core) tal que a placa gráfica utilizada neste trabalho consegue realizar 1.733 GFLOPS em precisão simples e 54 GFLOPS em precisão dupla (NVIDIA, 2017).

4 ANÁLISE DE DESEMPENHO

Objetivando-se avaliar o desempenho do algoritmo proposto no decorrer deste trabalho comparando com a versão previamente publicada em SOUZA et al. (2014), foram escolhidos 2 (dois) casos de teste também utilizados por SOUZA et al. (2014), pois pode-se realizar uma comparação na melhor evolução. Foram escolhidos os algoritmos mais simples e mais complexo dentre os 4 (quatro) utilizados em SOUZA et al. (2014), sendo que o mais simples foi escolhido para provar que o algoritmo funciona e que ele consegue chegar em resultados satisfatórios, e o mais complexo para garantir que em um processamento mais complexo, com mais variáveis e um espaço de busca maior, o algorítimo também obtém um desempenho de acordo com o previsto (tempo reduzido e resultado com o dobro de casas decimais de precisão).

A máquina utilizada nos testes foi um Samsung Odyssey modelo 800G5W com as conFigurações de hardware descritas abaixo:

- **Sistema Operacional:** Linux Ubuntu 16.02
- **CPU:** Intel® Core™ i7-7700HQ com clock base de 2.80GHz;
- **Memória RAM:** 8GB de Memória RAM DDR4 com clock base de 2133MHz;
- **GPU:** Geforce GTX 1050 com 640 núcleos CUDA e velocidade máxima de clock 1493MHz;
- **Memória VRAM:** 4042MB de memória Global GDDR5;
- **Arquitetura GPU:** Pascal;
- **Capacidade de Computabilidade:** 6.1;
- **Versão do Driver CUDA:** 9.2;

4.1 Problemas de Engenharia Implementados

Tendo em mente não apenas achar o melhor resultado, mas permitir que o algoritmo busque este resultado até que não hajam mais melhoras neste resultado encontrado, não foi definido um número máximo de iterações em nenhum dos dois testes realizados neste trabalho, permitindo assim que o algoritmo continue a busca por uma melhora mesmo quando o resultado já está estabilizado, podendo assim achar o melhor numero de iterações e obter uma variância e uma média de gerações que o algoritmo executou.

Vale também ressaltar que os testes foram reexecutados 500 vezes para que fossem geradas médias e variâncias entre os resultados encontrados, e assim melhor validar a consistência nos resultados deste algoritmo. Foram utilizadas as seguintes constantes: quantidade de partículas

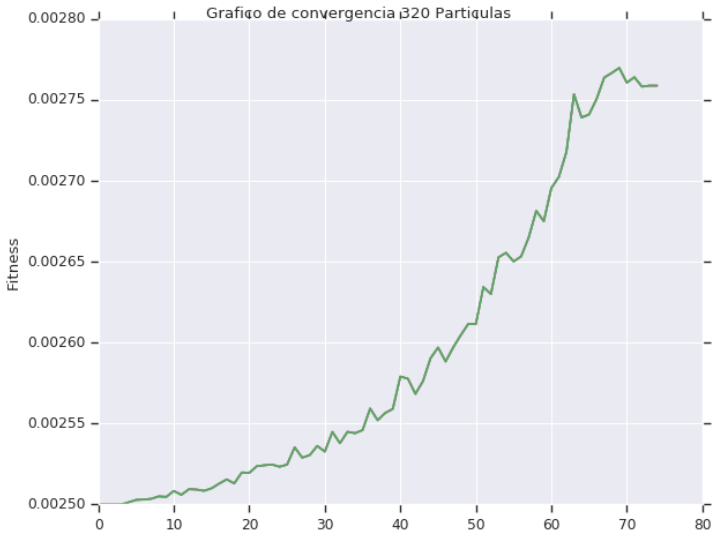
no enxame = 320, 640, estes valores foram usados como referência devido a uma limitação da GPU utilizada, pois para que se aumentasse a contagem de partículas no enxame, seria necessário criar um escalonamento das partículas por *thread*, causando assim um overhead no algoritmo.

4.1.1 Peso da Tensão/Compressão sobre Mola (MWTCS)

A seguir são descritos os valores de teste do algoritmo:

Constante individual ($C1$): 1,0; constante social ($C2$): 1,5; valor máximo para o fator de inércia (w_{MAX}): 0,3; valor mínimo para o fator de inércia (w_{MIN}): 0,0. O fator de inércia é recalculado a cada iteração de acordo com a equação 2.2.

Figura 13 – Gráfico de Convergência MWTCS 320 Partículas



Pode-se notar a estabilização do resultado nas Figuras 13 e 14, onde no primeiro o melhor resultado é encontrado antes da 70^a geração (comparado com a 80^a geração fixada no artigo de comparação), contudo o algoritmo continuou a busca por mais 20 gerações para garantir que um melhor local não havia sido atingido. Vale ressaltar que nestas Figuras são representados os valores de média, e a variância para mais e para menos, contudo devido ao espaço de busca ser bastante restrito, e a variância do resultado ter sido bem pequena também não é possível visualizar claramente estas linhas nas Figuras.

É interessante notar também que apesar da função 2.14 necessitar ser minimizada, as Figuras 13 e 14, não mostram exatamente isso, tendo estes um formato crescente ao invés de terem um formato decrescente, o que faria mais sentido. Este comportamento curioso é devido ao fato da função ter restrições, e um *fitness* com menos violações à restrições, mesmo que ele tenha um resultado maior do que os outros, ele é considerado melhor, pois é mais importante se ter o número de violações à regras de restrição das funções o mais próximo de zero do que ter um resultado com valor menor. E este fato pode ser facilmente observado nas Figuras referidos

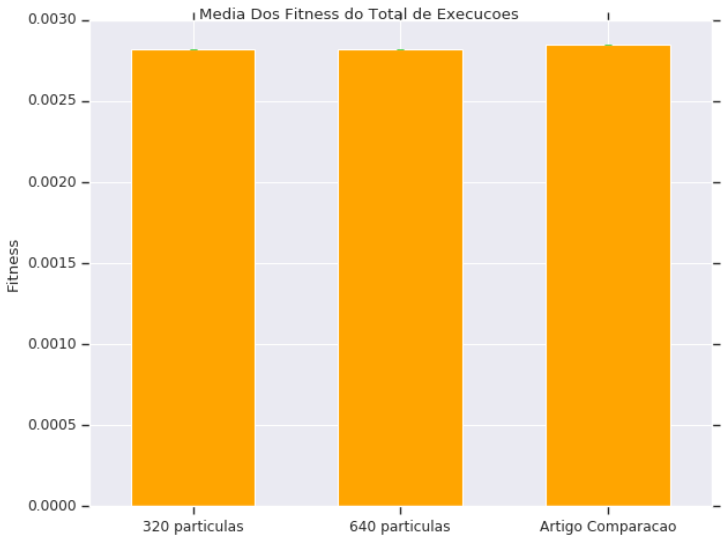
(13 e 14), tal que em cada aumento brusco (pico) das Figuras referidas, seria uma melhora no número de violações.

Figura 14 – Gráfico de Convergência MWTCS 640 Partículas



É possível notar também, no gráfico 15, que o resultado, na média, teve uma variância muito pequena ($5,5782724573062214e^{-35}$), menor do que a encontrada no trabalho que está sendo usado como base para comparação, além disso, devido a uma melhor aproximação do resultado este trabalho conseguir aproximar um valor em média menor do que o encontrado por SOUZA et al. (2014).

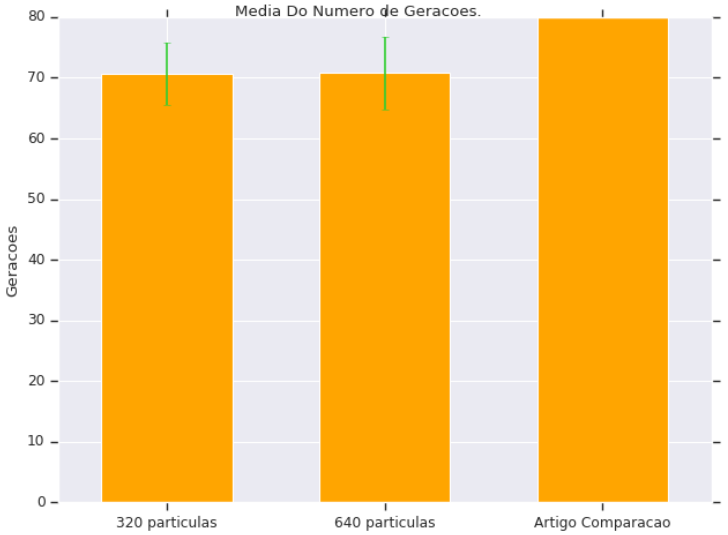
Figura 15 – Média do Fitness MWTCS



Comparando o número de gerações dos testes com 320 partículas e 640 partículas, que podem ser visualizados no gráfico 16, não foi possível observar que o comportamento das execuções com 640 partículas foi mais consistente do que os com 320 partículas, pois nem sempre ter mais partículas no espaço de busca facilita encontrar o melhor global, pois, quando se

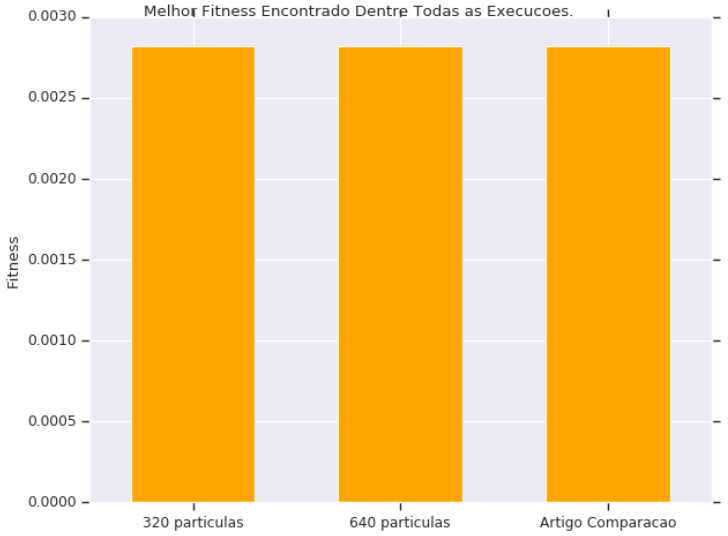
tem um espaço de busca reduzido o espalhamento de uma grande quantidade de partículas pode ser um desperdício de recursos.

Figura 16 – Média de Gerações MWTCS



O melhor *fitness* encontrado foi exatamente o mesmo do artigo que está sendo usado como base de comparação, contudo, neste trabalho, como o resultado foi aproximado utilizando variáveis de dupla precisão, um resultado mais preciso pode ser aproximado.

Figura 17 – Melhor Fitness Encontrado MWTCS



Com o gráfico 18 podemos notar que mesmo passando por menos gerações, a execução de 640 partículas tem um tempo de execução maior do que a execução de 320 partículas, mesmo que esta diferença seja de $\approx 10^{-5}$.

Em conclusão, todos os resultados obtidos nestes testes estão resumidos e demonstrados na Tabela 1 abaixo:

Figura 18 – Tempo de Execução MWTCS

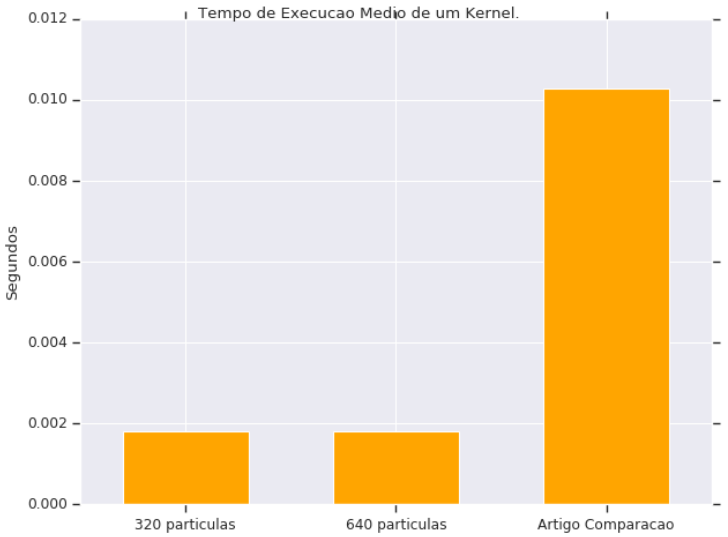


Tabela 1 – Resultados MWTCS

Resultado	320 Partículas	640 Partículas	Artigo Comparação
Média Melhor Fitness	0,0028202297651596	0,0028202297651596	0,0028468
Melhor Fitness	0,0028202297651596	0,0028202297651596	0,002820
Média de Gerações	87,18	80,54	80
Variância Melhor Fitness	$5,5782724573062214e^{-35}$	$9,4039548065783000e^{-36}$	$4,948406e^{-8}$
Tempo de Execução (seg)	0,0015274 seg	0,0020429 seg	0,010299 seg

4.1.2 Projeto de Redutor de Velocidade (SRD11)

Foram utilizadas as seguintes constantes neste teste: quantidade de partículas no enxame = 320, 640; constante individual ($C1$): 1,0; constante social ($C2$): 1,5; valor máximo para o fator de inércia (w_{MAX}): 0,9; valor mínimo para o fator de inércia (w_{MIN}): 0,4. O fator de inércia é recalculado a cada iteração de acordo com a equação 2.2.

Devido a este caso de teste possuir um espaço de busca bem maior do que o anterior, é possível visualizar a média, e a variância diminuindo de acordo com o progresso do algoritmo nas Figuras 19 e 20, onde a linha laranja representa o valor da média + o valor da variância; a linha azul, representa o valor da média; e a linha verde representa o valor da média - o valor da variância.

É fácil notar que o algoritmo consegue encontrar um valor aproximado logo no começo de sua execução, contudo, ainda necessita de um ajuste fino, que pode ser visualizado no gráfico 21, onde o resultado já é aproximado do final, contudo o algoritmo ainda busca ajustes para aumentar a precisão no resultado. O gráfico 21 é uma ampliação do gráfico 20, onde podemos visualizar o mesmo que no gráfico 14, sobre a estabilização e finalização do algoritmo ao fim da linha.

Figura 19 – Gráfico de Convergência SRD 320 Partículas

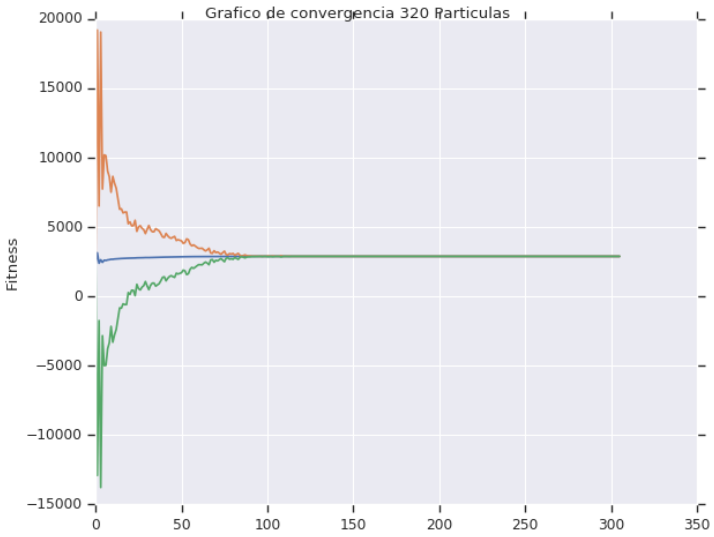
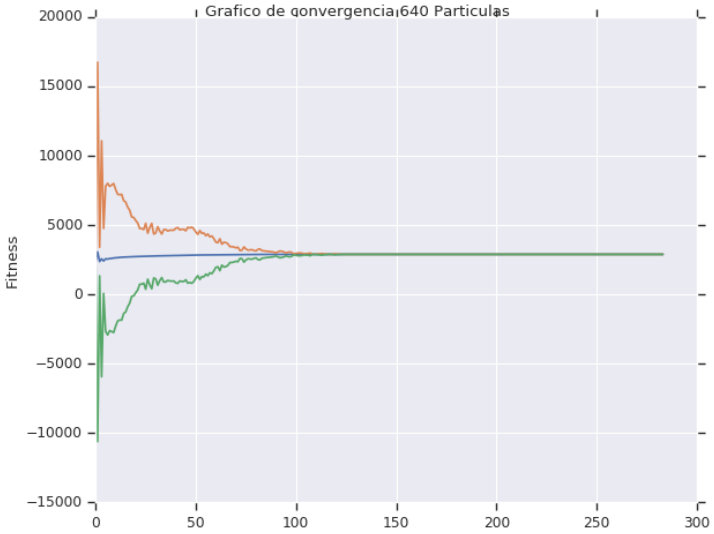


Figura 20 – Gráfico de Convergência SRD 640 Partículas



Comparando o algoritmo base de SOUZA et al. (2014) com o algoritmo desenvolvido neste trabalho através do gráfico 22, pode-se visualizar uma melhora no quesito precisão, neste caso de teste, tal que a variância dos resultados das execuções foram bem menores nos resultados do algoritmo desenvolvido neste trabalho. A média dos resultados também foi mais precisa neste trabalho, assim como no teste anterior, a menor variância destes resultados foi mais visível neste caso de teste do que no anterior.

Comparando o número de gerações dos testes com 320 partículas e 640 partículas, que podem ser visualizados no gráfico 23, pode-se visualizar que o número de gerações na execução com 640 partículas é menor, entretanto a diferença não é tão grande quanto no teste anterior. O que pode ser notado neste gráfico é que foi necessário um número muito menor de gerações (iterações) do algoritmo para que se alcançasse um resultado satisfatório, e que fosse mais preciso

Figura 21 – Zoom ao Final da Convergência SRD 640 Partículas

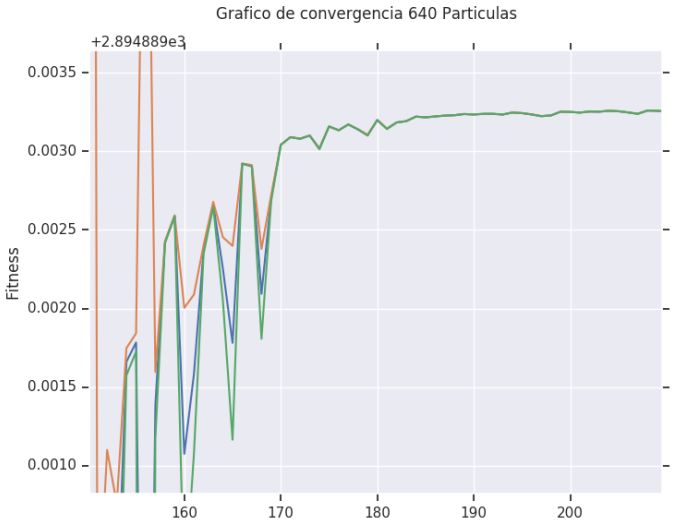
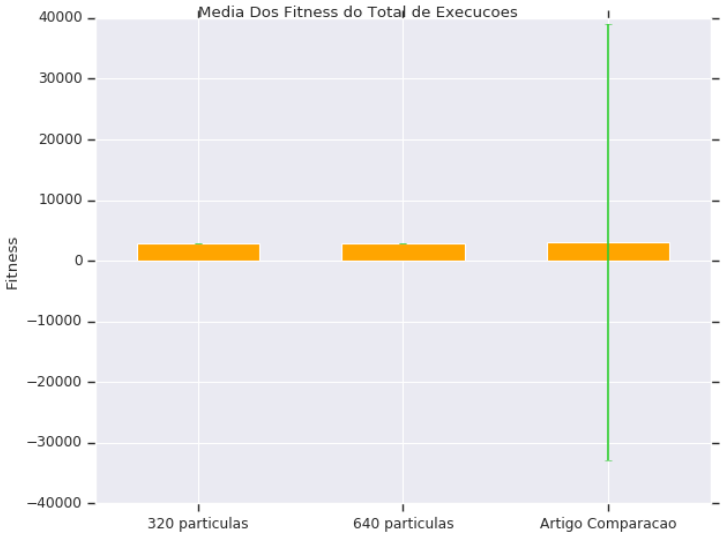


Figura 22 – Média do Fitness SRD



que o encontrado em SOUZA et al. (2014).

O melhor *fitness* encontrado foi exatamente o mesmo do artigo que está sendo usado como base de comparação, contudo, neste trabalho, como o resultado foi aproximado utilizando variáveis de dupla precisão, um resultado mais preciso pode ser aproximado.

Como o aumento do número de partículas no espaço de busca não foi efetivo, podemos notar no gráfico 25 que associado a isso, o tempo de execução do algoritmo com 640 partículas é praticamente o dobro do tempo de execução com 320 partículas, pois são necessários o dobro de cálculos.

Esta não melhora no número de gerações com o aumento do número de partículas é devido ao fato do espaço de busca ser muito grande, logo apenas dobrar a quantidade de partículas não é eficiente o suficiente para melhorar o desempenho do algoritmo e de fato ajudar a chegar a

Figura 23 – Média de Gerações SRD

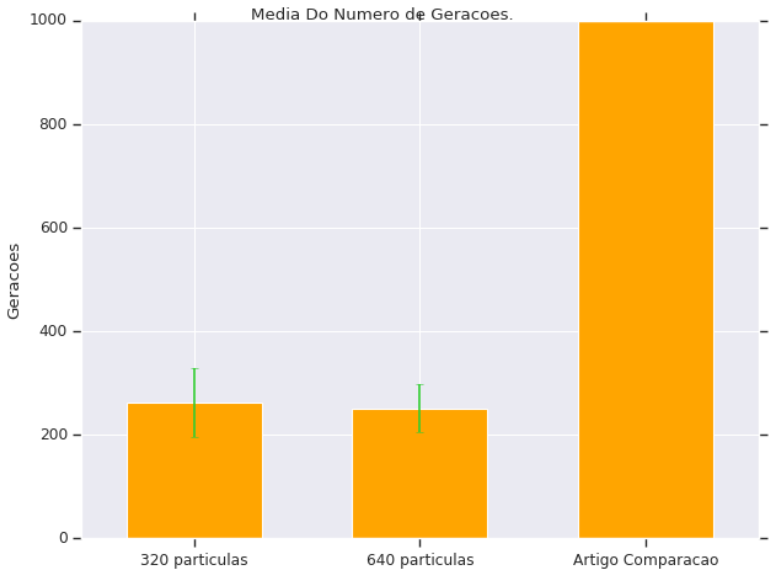
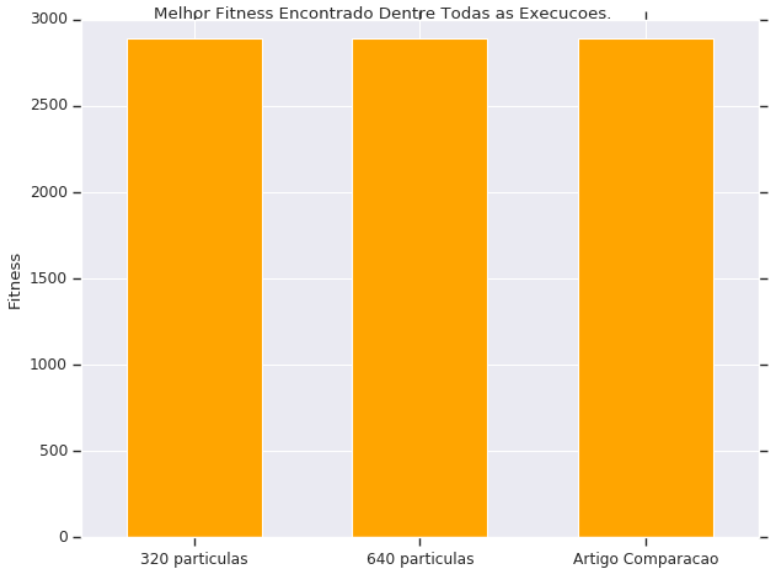


Figura 24 – Melhor Fitness Encontrado SRD



um melhor o resultado.

Em conclusão, todos os resultados obtidos nestes testes estão resumidos e demonstrados na Tabela 2 abaixo:

Figura 25 – Tempo de Execução SRD

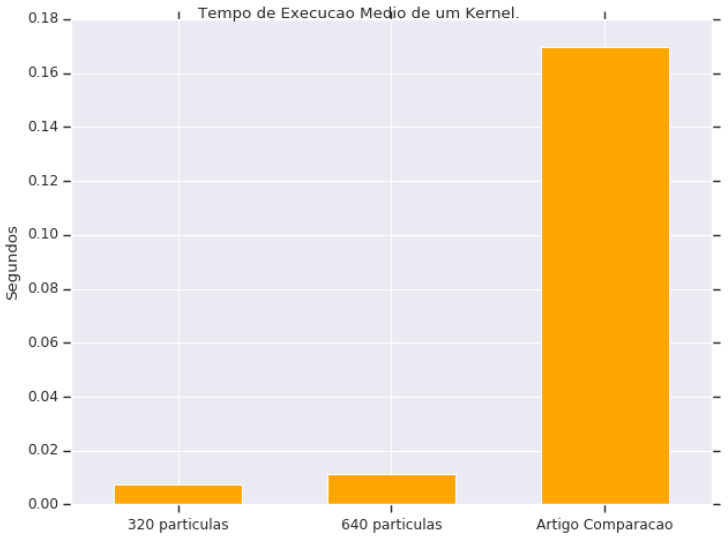


Tabela 2 – Resultados SRD

Resultado	**320 Partículas**	**640 Partículas**	**Artigo Comparação**
Média Melhor Fitness	2894,8922632716171392	2894,8922632716171392	3081,7047524
Melhor Fitness	2894,8922632716307816	2894,8922632716307816	2894,901611
Média de Gerações	262,268	251,622	1000
Variância Melhor Fitness	$1,891185189459538e^{-22}$	$1,8861209875129468e^{-22}$	$3,601403e^{+4}$
Tempo de Execução (seg)	0,0074902 seg	0,011488 seg	0,169510 seg

5 CONSIDERAÇÕES FINAIS

Com este trabalho, objetivou-se alcançar uma nova implementação do algoritmo do PSO dentro da arquitetura CUDA, onde foram alcançados resultados satisfatórios, com uma melhora significativa no desempenho, pois era esperado que o algoritmo fosse executado em um tempo menor do que o do artigo usado como comparação, por estar sendo executado em uma plaga gráfica mais nova, contudo os resultados foram surpreendentemente menores com relação ao tempo de execução. Tal melhora deve-se a utilização de novas arquiteturas de memória disponíveis nos mais recentes chips gráficos NVidia, sendo a *Shared Memory* a grande protagonista.

Vale ressaltar também, que a nova tecnologia e a melhora na velocidade de processamento disponível atualmente não teve grande papel nesta melhora de desempenho, pois o *throughput* (quantidade de dados processados pelo core) quando se usa variáveis do tipo *double* chega a ser 32 vezes menor do que quando se trabalha com variáveis do tipo *float* devido à quantidade de cores que podem processar 64bits ser reduzida na versão da GPU que foi utilizada para estes testes. Sendo assim a grande protagonista de fato foi a melhora na velocidade da memória, e de fato a nova forma de se utiliza-la.

5.1 Limitações

Após concluir o trabalho, e agregar mais experiência sobre a plataforma CUDA, algumas limitações puderam ser notadas. Dentre elas algumas foram destacadas abaixo:

- **Falta de testes sem restrições e poucos testes:** A ausência destes testes se dá pelo fato de o tempo hábil para o desenvolvimento. Contudo, devido ao fato de o trabalho ter sido desenvolvido com o intuito de permitir a fácil inserção de novos testes com apenas algumas mudanças no código fonte, esta limitação deste trabalho abre portas para um trabalho futuro, que permite uma maior exploração deste algoritmo.
- **Ausência de comparação de desempenho do mesmo algoritmo em mais de um hardware:** De início se planejava fazer os testes de desempenho em placas gráficas com mais poder de processamento, para que fosse possível realizar a comparação do desempenho da banda de memória (ponto de ataque principal deste trabalho) das duas plataformas, contudo não foram possíveis realizar estes testes devido a falta de acesso à tal hardware.

5.2 Trabalhos Futuros

Vislumbra-se desenvolver ainda mais este trabalho, podendo assim superar as limitações apresentadas anteriormente e acrescentar ainda mais funcionalidades e melhorar o desempenho

do algoritmo implementado. Sendo assim, abaixo são listados alguns trabalhos futuros:

- **Melhora de desempenho:** Utilização de variáveis de precisão simples para aproximar os resultados e depois passar os resultados para variáveis de precisão dupla para refinar a busca, pode trazer grande aumento na performance como citado em 3.1.2
- **Adição de mais testes:** Adicionar outros testes com e sem restrições para que a gama de resultados seja mais consistente.
- **Transformar em uma biblioteca compilada para reutilização:** Para que possa ser facilmente reutilizado, algumas modificações ao código fonte ainda precisam ser realizadas, permitindo que este código se torne uma biblioteca compilada que possa ser reutilizada e implementada por outros programadores em seus próprios códigos.

REFERÊNCIAS

BELUSSI, G. M.; BARISON, M. M. B. Número de ouro. **Londrina: Universidade Estadual de Londrina, S/D. Disponível em: http://www. mat. uel. br/geometrica/artigos/st-15-tc. pdf. Acesso em**, v. 5, n. 02, 2016.

CHING, F. D. **Arquitetura: forma, espaço e ordem**. [S.l.]: Bookman Editora, 2013.

COELLO, C. A. C.; MONTES, E. M. Constraint-handling in genetic algorithms through the use of dominance-based tournament selection. **Advanced Engineering Informatics**, Elsevier, v. 16, n. 3, p. 193–203, 2002.

FULCHER, L. C. J. J. **Computational Intelligence: A Compendium**. 1. ed. Springer-Verlag Berlin Heidelberg, 2008. (Studies in Computational Intelligence 115). ISBN 978-3-540-78292-6,978-3-540-78293-3. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=befc3384847d92e479541c9bb86b4f8b>.

HWU, W.-m.; KEUTZER, K.; MATTSON, T. G. The concurrency challenge. **IEEE Design & Test of Computers**, IEEE, v. 25, n. 4, 2008.

KENNEDY, J. Particle swarm optimization. In: **Encyclopedia of machine learning**. [S.l.]: Springer, 1995. p. 1942–1948.

NVIDIA. Nvidia tesla p100 whitepaper. **NVIDIA Corporation**, 2017.

NVIDIA. **CUDA C Programming Guide**. 2018. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.

PINHEIRO, F. L. P. Influência das leis de escala sobre a dinâmica de populações. 2011.

PISANO, L.; FIBONACCI'S, L. Liber abaci. **Sigler, Laurence E.(2002)" Fibonacci's Liber Abaci: A Translation into Modern English of Leonardo Pisano's Book of Calculation", Springer-Verlag, New York, Inc**, 2002.

SANDERS, J.; KANDROT, E. **CUDA by example: an introduction to general-purpose GPU programming**. [S.l.]: Addison-Wesley Professional, 2010.

SILVA, D. J. A. da; TEIXEIRA, O. N.; OLIVEIRA, R. C. L. de. Performance study of cultural algorithms based on genetic algorithm with single and multi population for the mkp. In: **Bio-Inspired Computational Algorithms and Their Applications**. [S.l.]: InTech, 2012. p. 385–404.

SOUSA, D. L. Pso-gpu: Implementação da metaheurística de otimização por enxame de partículas (pso) paralela sob a arquitetura cuda. 2010.

SOUZA, D. L. et al. Otimização por multi-enxame evolucionário de partículas clássico e quântico competitivo sob a arquitetura paralela cuda aplicado em problemas de engenharia. Universidade Federal do Pará, 2014.

TANOMARU, J. Motivação, fundamentos e aplicações de algoritmos genéticos. In: **Anais do II Congresso Brasileiro de Redes Neurais**. [S.l.: s.n.], 1995. v. 1.

TEIXEIRA, O. N. et al. Algoritmo genético com interação social na resolução de problemas de otimização global com restrições. **Editora OMNIPAX**, p. 197–223, 2011.

TEIXEIRA, O. N. et al. Algoritmo genético com interação social nebulosa. Universidade Federal do Pará, 2012.