Learning and Nonlinear Models - Journal of the Brazilian Society on Computational Intelligence (SBIC), Vol. 13, Iss. 1, pp. 3-20, 2015

Paralelização de Algoritmos de Busca baseados em Cardumes para Plataformas de Processamento Gráfico

Anthony José da Cunha Carneiro Lins, Fernando Buarque Lima Neto, Carmelo José Albanez Bastos Filho

Escola Politécnica da Universidade de Pernambuco - Universidade de Pernambuco

ajccl@ecomp.poli.br, fbln@ecomp.poli.br, carmelofilho@poli.br

Resumo – Busca por cardumes (FSS, *Fish School Search*) é uma técnica de inteligência computacional usada para resolver problemas de otimização em espaços de busca multimodais com alta dimensionalidade. FSS é inspirada no comportamento social de cardumes, na qual a posição de cada peixe no espaço de busca representa uma possível solução para o problema. Como FSS é uma técnica de inteligência de enxames e sua função objetivo pode ser avaliada para cada peixe individualmente, esta se torna uma potencial candidata para implementações em plataformas de processamento paralelo. A utilização de Unidades de Processamento Gráfico (GPU, *Graphic Processing Units*) vem se mostrando bastante vantajosa em aplicações que requerem computação paralela intensiva. Com a popularização da plataforma CUDA da NVIDIA, tornou-se possível o desenvolvimento de aplicações de propósito geral em plataformas com GPUs visando atingir processamento de alto desempenho. Neste trabalho é apresentada uma análise de como adaptar o algoritmo original do FSS utilizando os benefícios da computação paralela. Esta versão foi nomeada de pFSS (*parallel Fish School Search*). Além de mostrar as vantagens do pFSS em relação à abordagem para execução em CPU, também é analisado o desempenho em termos de tempo de execução e convergência nos modos de processamento síncrono e assíncrono. Ainda é apresentada uma análise do impacto da utilização de múltiplas plataformas GPUs operando em colaboração para problemas de alta dimensionalidade. Os resultados desta análise mostram que o pFSS pode atingir *speedups* de até 127 vezes em relação ao processamento seqüencial em CPU, mantendo a qualidade dos resultados retornados pelo algoritmo de busca por cardumes.

Palavras-chave – Inteligência de Enxames, Busca por Cardumes, Unidades de Processamento Gráfico, Computação Paralela.

Abstract – *Fish School Search* (FSS) is a computational intelligence technique developed to solve optimization problems in high dimensional multimodal search spaces. FSS was inspired by the social behavior of fish schools. In the FSS, the position of each fish within the search space represents a possible solution for the problem. Since the FSS is a swarm intelligence technique and the fitness function can be evaluated individually for each fish, FSS becomes a potential candidate for parallelization in Graphic Processing Units (GPUs). The use of GPUs has shown a great performance for applications that require intensive parallel computation. Due to the popularization of the NVIDIA's CUDA cards, many general purpose applications were adapted for GPU-based platforms, enabling high-performance processing. We propose in this paper to adapt the original FSS algorithm for GPU-based parallel platforms. We named this version pFSS (*parallel Fish School Search*). We show the feasibility of our approach in terms of execution time and convergence robustness. We show results for two different processing modes, synchronous and asynchronous. We also present an analysis on the impact of the use of multiple GPU platforms executing in a collaborative approach in order to solve high dimensional optimization problems. We achieved a speedup of 127 when compared to the ordinary sequential CPU approach, but maintaining the quality of the results returned by the optimization process.

Keywords – Swarm Intelligence, Fish School Search, Graphic Processing Unit, Parallel Computing.

1. INTRODUÇÃO

Nos últimos anos tem crescido o número de aplicações de computação de propósito geral que utilizam Unidades de Processamento Gráfico (GPUs - *Graphic Processing Units*) [1] [2]. O uso de GPUs vem se mostrando bastante vantajoso em aplicações que requerem computação paralela intensiva, pois o processamento em paralelo de operações com ponto flutuante, realizado pelas GPUs, permite obter altas taxas de processamento, aumentando assim o desempenho nestas aplicações [3] [4].

Neste trabalho é apresentada uma análise sobre os benefícios da computação paralela em GPUs e como estes benefícios podem ser obtidos para uma técnica de inteligência computacional chamada de Busca por Cardumes (*Fish School Search* - FSS), geralmente usada para otimização em problemas com alta dimensionalidade e alto custo computacional. O objetivo deste trabalho consiste em propor uma abordagem para o FSS que permita a diminuição do tempo de processamento, mantendo a qualidade dos resultados obtidos em diversos problemas de otimização. Além disso, são discutidas as dificuldades de paralelização de algoritmos de Inteligência de Enxames, devido à sincronização do processamento, apresentando um comparativo de testes realizados nos modos síncrono e assíncrono

Também são tratadas questões inerentes ao processamento paralelo tais como: (a) determinar a melhor configuração dos blocos de processamento em GPU; (b) configurar memória para alocações de variáveis e objetos; e (c) analisar o uso de barreiras

de sincronização de processos. Além disso, são discutidas as técnicas para melhorar o desempenho do processamento, como por exemplo, a alocação de memória compartilhada e a comunicação entre os blocos de processamento, bem como o processamento utilizando Múltiplas Plataformas de Processamento Gráfico em Paralelo (*MultiGPUs*). São apresentados os resultados dos experimentos realizados com a técnica de busca baseada em cardumes, em dois problemas de otimização bem conhecidos, um monomodal e outro multimodal. Nesta análise, resultados obtidos em processamento *MultiGPU* apresentaram um *speed-up* de 127 vezes, em relação ao processamento realizado em CPU.

Este trabalho está dividido em nove seções, que são apresentadas a seguir: na segunda seção estão apresentados os conceitos sobre computação paralela; a terceira seção descreve o FSS; a quarta seção apresenta trabalhos sobre técnicas de Inteligência de Enxames com processamento paralelo; na quinta seção têm-se a proposta deste trabalho com o FSS sendo paralelizado em GPUs; a sexta seção apresenta a proposta do FSS paralelizado em múltiplas GPUs; na sétima seção são descritos os arranjos experimentais; na oitava seção são apresentados os resultados obtidos nas duas funções de teste; e na nona seção são apresentadas as conclusões deste trabalho, bem como algumas propostas para trabalhos futuros.

2. COMPUTAÇÃO PARALELA

A computação paralela é, de forma simples, a utilização simultânea dos múltiplos recursos computacionais para resolver um problema computacional. Os recursos computacionais podem ser desde um simples computador com múltiplos processadores a um número arbitrário de computadores conectados por uma rede, ou mesmo a combinação de ambos. A computação paralela é uma evolução da computação serial. Na computação serial, as instruções são executadas em um computador com uma CPU, e são processadas uma após a outra, uma de cada vez.

Em computação paralela, a configuração do hardware utilizado para realizar os múltiplos processamentos torna-se uma questão chave na obtenção do melhor resultado com um desempenho satisfatório. Moore [5] previu que a quantidade dos transistores que compõem os microprocessadores dobraria a cada 18 ou 24 meses, e mesmo com pesquisas recentes sobre redução do tamanho dos componentes e consumo de energia, essa lei ainda é válida. Os processadores conhecidos como *many-cores* (muitos-núcleos) tiveram uma trajetória focada mais no rendimento da execução de aplicações paralelas. Os *many-cores* iniciaram com um grande número de núcleos muito pequenos, tendo mais uma vez, o número de núcleos dobrado a cada geração. Neste grupo de *many-cores*, encontram-se as GPUs, que lideram a corrida pelo desempenho em operações com ponto flutuante desde 2003 [6]. Enquanto a melhoria no desempenho de microprocessadores de propósito geral estagnou de forma significativa, as GPUs têm continuado a melhorar incessantemente. O desempenho pode ser avaliado pelo número de operações de ponto flutuante por segundo (*Floating Point Operations per seconds* - FLOPS) [1]. A diferença de desempenho entre o processamento em GPU com múltiplos núcleos (*many-core*) de propósito geral e a CPU está relacionada com a arquitetura das duas plataformas, onde têm-se um número maior de Unidades Lógica-Aritméticas (ULA) na GPU em relação ao encontrado na CPU. Isto torna possível a execução simultânea de um número muito maior de cálculos de operações vetoriais nas plataformas baseadas em GPUs [7]. Procurando encontrar maior área no processador para realizar estas operações, bem como reduzir o custo de energia, pesquisadores verificaram que poderiam resolver esses problemas otimizando a execução de operações e a taxa de transferência de dados, aumentando o número de *threads* de processamento. Esse aumento do número de *threads* fez com que o hardware diminuísse a espera durante o acesso à memória, reduzindo também a utilização das funções da unidade de controle lógico requisitada por cada *thread*. Pequenas unidades de memória cache ficam disponíveis para controlar a largura de banda, e as múltiplas *threads* que acessam a mesma área de memória não necessitam acessar a memória principal. Como resultado, têm-se muito mais área dedicada no processador para efetuar cálculos de ponto-flutuante.

Diante do exposto, pode-se verificar que as GPUs são projetadas para computar dados númericos, e elas não desempenham bem algumas tarefas específicas para as quais as CPUs foram projetadas. Assim, pode-se esperar que muitas aplicações sejam desenvolvidas utilizando ambos, CPU e GPU. Isto pode ser conseguido executando ações de controle na CPU e cálculo vetorial intensivo na GPU. Atualmente, várias GPUs suportam o padrão IEEE para ponto-flutuante [3] comparável ao existente nas CPUs, inclusive utilizando-se de variáveis de precisão simples e dupla. No caso de utilização de dupla precisão, o desempenho pode atingir metade da velocidade da precisão simples. As CPUs atuais, também já conseguem atingir desempenho similar no processamento para cálculos em precisão dupla. Como consequência do aumento da velocidade conseguida pelas GPUs, esse tipo de hardware vêm se tornando cada vez mais adequado para aplicações numéricas.

2.1 CUDA

O CUDA™(*Computer Unified Device Architecture*) [7] [8] é uma arquitetura para computação paralela de propósito geral, com um modelo de programação paralela e um conjunto arquitetural de instruções, que permite elevar o mecanismo de computação paralela das GPUs NVIDIA para resolver problemas computacionais complexos de forma mais eficiente que uma CPU (mesmo com muitos núcleos). Em CUDA existem três abstrações principais: uma hierarquia de grupos de *threads*, compartilhamento de dados em memórias e a sincronização de dados e processamentos [7]. Estas abstrações permitem dividir o problema em sub-problemas, que podem ser resolvidos de forma independente em paralelo. Cada sub-problema pode ainda ser dividido em procedimentos mínimos que podem ser resolvidos de forma cooperativa paralelamente, usando todos as linhas de processamento (*threads*) dentro de um bloco na GPU. Dessa forma, cada bloco de *threads* pode ser programado para ser utilizado em qualquer um dos núcleos de processamento disponíveis, independentemente da ordem de execução.

2.2 Modelagem de Memória

Plataformas CUDA apresentam uma hierarquia de memória bem definida, a qual inclui tipos distintos de memória na plataforma GPU. É importante ressaltar que o tempo de acesso a estes tipos distintos de memória variam. Cada *thread* de processamento tem uma memória local privada e cada bloco de *threads* tem uma memória compartilhada (*shared memory*), que por sua vez é acessível por todas as *threads* dentro do bloco. Além disso, as *threads* podem ter acesso à mesma memória global. Todos estes espaços de memória estão organizados de acordo com a velocidade de acesso e o tamanho da memória. A memória local é mais rápida e a memória global mais lenta. Por outro lado, a memória local geralmente tem um tamanho menor e a memória global tem tamanho maior. Então, se há dados que devem ser acessados por todos as *threads*, a memória compartilhada pode ser a melhor escolha. No entanto, a memória compartilhada só pode ser acessada pelos *threads* pertencentes a um mesmo bloco e seu tamanho é limitado. Ao realizar a programação dos dados em memória, deve-se verificar a relação entre os espaços de memória dentro de uma GPU baseada em CUDA, tendo o processamento executado em CPU, conhecido como *Host*, responsável por realizar as chamadas das funções e controle dos dados a serem processados em GPU (denominado *Device*) [7].

As principais características dos diversos tipos de memória que podem ser manipuladas dentro de uma GPU compatível com CUDA estão relacionados na Tabela 1.

Tabela 1: Características das Memórias existentes numa GPU CUDA.

Memória	na Placa	Cache	Acesso	Escopo	Tempo de Vida
Registradores	Sim	n/d	L/G	1 *thread*	*Thread*
Local	Não	v. 2.0	L/G	1 *thread*	*Thread*
Compartilhada	Sim	n/d	L/G	Todas as *threads* (Bloco)	Bloco
Global	Não	v. 2.0	L/G	Todas as *threads* + *Host*	Alocação no *Host*
Constante	Não	Sim	L	Todas as *threads* + *Host*	Alocação no *Host*
Textura	Não	Sim	L	Todas as *threads* + *Host*	Alocação no *Host*

Os tipos de memória local e global, que aparecem na Tabela 1, apenas utilizarão de processamento em cache nas GPU com compatibilidade com a versão 2.0 de CUDA. As memórias de registradores e compartilhada não possuem cache disponíveis. As memórias global, local e de textura possuem grande latência no seu acesso, seguida por memória constante, memória de registradores e memória compartilhada. As plataformas CUDA podem facilmente ser classificadas para as GPUs de acordo com a sua capacidade de computação [7]. As placas gráficas com processamento de ponto flutuante com precisão dupla tem capacidade de computação nas versões 1.3 e 2.x. As placas com capacidade 2.x, em diante, podem executar até 1024 *threads* em um bloco e ter 48 KB de espaço para memória compartilhada. As outras versões, com capacidade inferiores, podem executar até 512 *threads* por bloco e ter 16 KB de espaço para memória compartilhada.

2.3 Processamento com CUDA

A fim de processar o algoritmo em paralelo, o número de cópias paralelas das funções *kernel* (que são executadas na GPU) a serem realizadas deverá ser informado à plataforma CUDA. Essas cópias são também conhecidas como blocos paralelos e são divididos em um número de segmentos de execução (*threads*). Os conjuntos desses blocos são chamados de *Grids* (grades). As *threads* existentes dentro de um bloco podem cooperar entre si através do compartilhamento de dados, podendo utilizar-se de alguma memória compartilhada e da sincronização de sua execução, a fim de coordenar os acessos à memória.

No caso de uma função *kernel* ser chamada pela CPU, ela será executada em *threads* separadas dentro do bloco correspondente. As estruturas definidas pelos *grids* podem ser divididas em blocos de duas dimensões (nomeadas por *gridDim.x* e *gridDim.y*). Já os blocos, são endereçáveis em duas dimensões (nomeadas por *blockIdx.x* e *blockIdx.y*), e são divididos em um número de *threads*, limitados pela quantidade máxima de *threads* por bloco, de acordo com cada configuração de GPU. O número de *threads* será constante para todos os blocos e poderá ser acessado de acordo com cada dimensão do bloco através das variáveis internas (*built-in*) do CUDA (nomeadas por *blockDim.x*, *blockDim.y* e *blockDim.z*). Já as *threads*, podem ser estruturadas em 1 a 3 dimensões (nomeadas por *threadIdx.x*, *threadIdx.y* e *threadIdx.z*) [7].

Um conceito importante neste estudo é a medida da ocupação de um multiprocessador dentro da GPU, que pode ser computada usando uma planilha fornecida pela NVIDIA. Para essa medida de ocupação são considerados o número máximo de *threads* por bloco, a quantidade de registradores e o tamanho da memória compartilhada existentes em cada função *kernel*. No caso de uma arquitetura mais robusta, como a Fermi [9] [10], a medida de ocupação é definida como a razão entre o número ***warps*** (cada *warp* é um conjunto de 32 *threads*) residentes utilizados por cada *kernel* e o número máximo de *warps* residentes nos processadores.

2.3.1 Técnicas de Otimização

Como o principal argumento para se utilizar processamento paralelo é aumentar a velocidade de processamento, o CUDA possui técnicas para melhorar o desempenho dos algoritmos paralelizados. Neste trabalho, foram pesquisadas e aplicadas técnicas de otimização disponíveis na API do CUDA [11], que são:

- **Otimização do uso da memória** - Para maximizar o uso da GPU através do aumento da largura da banda. A largura de banda será melhor aproveitada se houver aumento no acesso à memória rápida e diminuição no acesso à memória lenta. O ideal é aproveitar a largura de banda entre a memória e o processador da GPU, ao invés de explorar a comunicação da memória da CPU com a memória da GPU. Então, tem-se como alta prioridade minimizar a transferência de dados entre o processamento *Host* e o processamento na GPU. Para isso, pode-se usar a alocação de dados na memória da GPU, onde estruturas intermediárias serão criadas e destruídas sem a necessidade de serem mapeadas pelo *Host* ou copiadas para a memória da CPU. Tem-se então uma maior largura de banda entre o processo *Host* e o processo na GPU quando for utilizada a memória pinada (*pinned* ou *page-locked*) [6] [7]. Memória pinada é uma área de memória no processador *Host* que possui um endereço que é acessado pelo processamento na GPU, fornecendo um acesso assíncrono aos dados sem uma requisição de cópia explicita ser iniciada [7] [8].

- **Execução Assíncrona e Uso de *Streams*** - Em CUDA, é possível executar operações de forma concorrente entre o *host* e o *device*, neste caso denominadas **assíncronas**. Essas operações assíncronas, permitirão que o controle do processamento retorne para o *host* antes mesmo que o *device* tenha finalizado uma tarefa solicitada. Algumas dessas tarefas assíncronas são [7]: as chamadas de funções *Kernel*, as cópias de dados entre *Host* e o *Device* (vice-versa), e as cópias de dados em memória entre *Devices* (por exemplo múltiplas placas com GPUs). As transferências de dados entre *host* e *device*, utilizando *cudaMemcpy()* são transferências bloqueadas (síncrona). Para operações de transferências de dados em modo assíncrono, tem-se o comando *cudaMemcpyAsync()*, que é uma variante não bloqueada do comando original, e passará o controle do processamento ao *host* sem ter que aguardar o final do processo de cópia. Para realizar a transferência de dados assíncrona, é importante que seja utilizada uma memória pinada no *host*, e que seja passado como argumento de cópia um identificador de *stream* [8]. Uma ***stream***, em CUDA, representa uma fila de operações na GPU que são executadas seguindo uma ordem [12]. Cada *stream* pode ser percebida como uma tarefa na GPU e podem haver oportunidades em que essas atividades sejam executadas em paralelo.

- **Acesso de Memória Coalescente** - De acordo com Farber [6], a memória global está sujeita a regras de união (*coalescing*) de dados, ou agrupamento, que combina múltiplas transações de memória dentro de largas operações de carregamento ou armazenamento únicas que podem atingir altas taxas de transferência de/para memória. Utilizando-se de ***warps***, para acessar dados dentro de uma região contígua, pode-se conseguir a operação de memória mais ampla e altamente eficiente, pois cada byte recuperado é utilizado. Dentre algumas estratégias adotadas para acessar memória coalescente [1] [7], a utilização de memória compartilhada permitirá a implementação de acesso único agrupado dos *warps* com taxa de largura de banda sem perdas, pelo fato de reduzirem as chamadas à memória global. Por outro lado, a utilização de serialização de *warps*, pode reduzir problemas de conflitos de acesso aos bancos de memória, muito constantes neste tipo de memória. É importante utilizar uma barreira de sincronização (*__syncthreads()*) para que todas as *threads* do *warp* acessem os dados da memória compartilhada corretamente.

- ***Loop Unrolling*** - Desenlaçar o laço é uma técnica que permite realizar o processo contido dentro de um laço do tipo *for* de forma que as instruções sejam carregadas na *thread* já prontas para serem processadas. Segundo Murthy *et al*. [13], um dos principais benefícios obtidos ao realizar *Loop Unrolling* está na redução na contagem de instruções dinâmicas, devido ao menor número de operações de comparação e ramos (*branch*) para a mesma quantidade de trabalho realizado.

2.4 Paralelismo em Múltiplas GPUs

Modelar um problema que explore de forma eficiente o paralelismo de *hardware*, mesmo para o problema intrinsecamente paralelo, não é uma tarefa trivial. Problemas com alta complexidade, e com diferentes granularidades, serão cada vez mais submetidos à paralelização do seu processamento. Com o aumento da popularidade, na utilização de hardware com múltiplos processadores gráficos, é importante que sejam analisadas as técnicas para paralelização do processamento em mais de uma placa com GPUs sem perdas de desempenho e qualidade de resultados. O aumento de transferências de dados da memória, diminuirá os benefícios do uso de GPUs múltiplas. Algumas formas de se contornar esse problema necessitam de operações assíncronas para cobrir latências. As propostas apresentadas na versão de CUDA 4.0 são:

- **Zero-Copy:** Para realizar o processamento em MultiGPU, a alocação da memória deverá ser feita utilizando conceito de memória pinada (ver seção 2.3.1 sobre Técnicas de Otimização). Na alocação da memória pinada, passando a *flag cudaHostAllocMapped*, tem-se que a memória alocada nunca será paginada ou realocada dentro da memória física. Esse tipo de alocação no *host* pode ser utilizada em alternativa ao processo de cópia entre processadores *host* e *device*, pois vai permitir que a memória alocada em *Host* seja acessada diretamente de dentro das funções *kernel* no *Device*, sendo conhecida como memória de cópia zero (*Zero-Copy*) [12].

- **UVA:** *Unified-Virtual Address* (UVA) é um endereçamento único de espaço em memória a ser utilizado tanto pelo *host* quanto pelos dispositivos GPU em um sistema. UVA não está disponível em sistemas baseados em 32 bits [6]. Esse endereço de memória único é usado por todas as alocações feitas na memória *host* via *cudaHostAlloc()* e em qualquer memória das GPUs através *cudaMalloc()* e suas variações [7].

- **P2P:** Outro aspecto para obter um melhor desempenho com múltiplas GPUs consiste em realizar a comunicação (cópia de dados) entre pontos de duas GPUs vizinhas. A essa abordagem é dado o nome de *Peer-to-Peer* (ponto a ponto - P2P). Essa

tecnologia está disponível apenas em sistemas de 64 bits com dispositivos compatíveis com a capacidade de cálculo 2.0 ou superior, tais como as arquiteturas da série Tesla da NVIDIA. Com isso, pode-se endereçar a memória de outro dispositivo (isto é, um função *kernel* sendo executada em uma GPU pode referenciar um ponteiro para a memória de outra GPU).

3. FISH SCHOOL SEARCH

Busca por Cardumes (FSS, *Fish School Search*) é uma técnica de otimização global estocástica. O FSS foi inspirado no comportamento gregário encontrado em algumas espécies de peixes, especificamente para gerar proteção mútua e sinergia ao desempenhar tarefas coletivas, onde ambas permitem aumentar a capacidade de sobrevivência de todo o grupo [14].

O algoritmo FSS possui quatro operadores, que podem ser agrupados em duas classes: a alimentação e a natação. Cada peixe do cardume deverá executar todos os operadores ao longo de todas as iterações. Assim sendo, os operadores do FSS são: (i) o movimento individual, o responsável por realizar uma busca local; (ii) a alimentação, responsável pela atualização do peso do peixe, indicando, assim, se houve ou não sucesso durante a busca; (iii) o movimento coletivo-instintivo, realiza o deslocamento do cardume influenciado pelos peixes que tiveram sucesso individualmente; e (iv) o movimento coletivo-volitivo, responsável por realizar o controle da granularidade do processo de busca. O pseudocódigo da versão básica está apresentado no Algoritmo 1 com os operadores em sua versão original.

Algorithm 1: Pseudocódigo do algoritmo FSS - versão básica.

Inicializar o vetor de posições com valores aleatórios dentro do intervalo permitido;
Atribuir valores iniciais referentes aos pesos dos peixes de acordo com regra de inicialização;
enquanto *não atingir a condição de parada - para cada peixe* **faça**

/* Operador Movimento Individual - verifica posições vizinhas $\vec{n}_i(t)$ e desloca o peixe para $\vec{x}_i(t+1) = \vec{n}_i(t)$ se a nova posição for melhor, senão o peixe mantém a posição $\vec{x}_i(t+1) = \vec{x}_i(t)$ */

$$\vec{n}_i(t) = \vec{x}_i(t) + \vec{U}[-1,1] step_{ind} \tag{1}$$

/* Operador Alimentação - atualiza o peso do peixe de acordo com a variação de fitness $\Delta f = f[\vec{x}_i(t+1)] - f[\vec{x}_i(t)]$ normalizado */

$$W_i(t+1) = W_i(t) + \frac{\Delta f_i}{modulo[max(\Delta f)]} \tag{2}$$

/* Operador Movimento Coletivo-Institivo - Calcula e aplica um deslocamento $\vec{d}(t)$ em todos os peixes */

$$\vec{d}(t) = \frac{\sum_{i=1}^{N} \{\vec{x}_i(t+1) - \vec{x}_i(t)\}\{f[\vec{x}_i(t+1)] - f[\vec{x}_i(t)]\}}{\sum_{i=1}^{N} \{f[\vec{x}_i(t+1)] - f[\vec{x}_i(t)]\}} \tag{3}$$

/* Operador Movimento Coletivo-Volitivo - Calcula o baricentro e aplica deslocamento em relação ao baricentro */

$$\vec{B}(t+1) = \frac{\sum_{i=1}^{N} \vec{x_i}(t+1) W_i(t+1)}{\sum_{i=1}^{N} \vec{x_i}(t+1)} \tag{4}$$

$$\vec{x}_i(t+1) = \vec{x}_i(t+1) \pm step_{vol} \vec{U}[0,1] \frac{\vec{x_i}(t+1) - \vec{B}(t+1)}{distancia[\vec{x_i}(t+1), \vec{B}(t+1)]} \tag{5}$$

Avaliar a função objetivo em cada nova posição;
Atualizar memória de cada indivíduo caso a nova posição seja melhor;
Atualizar passos individual e volitivo;

retorna *A melhor solução guardada na memória do cardume.*

- **Movimento individual**: aplicado a cada peixe no cardume em cada iteração, escolhendo aleatoriamente uma nova posição em sua vizinhança e essa nova posição é avaliada por uma função de aptidão (*fitness*) antes de mover o peixe para a posição calculada. No pseudocódigo 1, a equação (1) do operador individual descreve a posição candidata $\vec{n}_i$(t) do peixe i na iteração t em função da posição atual do peixe $\vec{x}_i(t)$, um passo e um vetor $\vec{U}[-1, 1]$ de números aleatórios gerados a cada iteração por uma distribuição uniforme no intervalo $[-1, 1]$ [15]. O $step_{ind}$ é um percentual da amplitude (tamanho) do espaço de busca na dimensão e é limitado por dois parâmetros ($step_{ind\_min}$ e $step_{ind\_max}$). O $step_{ind}$ decresce linearmente durante as iterações a fim de que todo o cardume realize inicialmente busca em amplitude e altere para busca em profundidade de forma gradativa.

- **Operador de Alimentação**: cada peixe pode aumentar ou diminuir de peso, dependendo do seu sucesso ou insucesso na busca por comida. O peso do peixe é atualizado uma vez a cada iteração do algoritmo usando a equação (2). $W_i(t)$ é o peso do peixe i na iteração t, Δf_i é a diferença entre o valor do *fitness* calculado para a nova posição e o valor do *fitness* calculado para a posição atual de cada peixe. $modulo[max(\Delta f)]$ é o valor máximo absoluto das diferenças de *fitness* na iteração considerando todos os peixes. Uma escala de peso (W_{scale}) é definida para limitar o peso do peixe.

- **Movimento coletivo-instintivo**: após todos os peixes terem se movimentado individualmente, suas posições são atualizadas de acordo com a influência dos peixes que foram mais bem sucedidos na busca por alimentos durante a movimentação individual. O objetivo é utilizar a informação da direção onde há maior proporção de alimentos, ou seja, as melhores soluções para o dado problema. A direção a ser tomada pelo cardume é definida conforme a equação (3), baseando-se na avaliação do *fitness* dos peixes que tiveram melhores resultados. Pode-se observar que o deslocamento depende da variação de posição e da variação de *fitness* no movimento individual. Deve-se observar também que $\{\vec{x}_i(t+1) - \vec{x}_i(t)\} = 0$, para os peixes que não executaram o movimento individual. O valor de $\vec{d}(t)$ é utilizado para deslocar a posição de todos os peixes do cardume.

- **Movimento coletivo-volitivo**: ocorre após os outros dois movimentos. Se a busca feita pelo cardume estiver sendo bem sucedida, o cardume deverá contrair em relação ao baricentro; se não, o cardume deverá se espalhar. Este operador controla a granularidade da busca realizada pelo cardume, aumentando a capacidade de auto-regulação do tipo de busca realizado (exploração em profundidade ou exploração em amplitude). A dilatação e contração do cardume é aplicada a cada posição dos peixes em relação ao baricentro do cardume, conforme a equação (4). Dependendo do sucesso do cardume dentro da iteração, será realizada a expansão (nesse caso usando o sinal $+$) ou a contração (nesse caso usando o sinal $-$) do cardume em relação ao baricentro, usando a equação (5). Nesta equação, $distancia[\vec{x_i}(t), \vec{B}(t)]$ retorna o valor da distância Euclidiana entre o peixe i e o baricentro. $step_{vol}$, chamado de passo volitivo, controla o tamanho do passo do peixe. O $step_{vol}$ é definido como um percentual do espaço de busca e é limitado por dois parâmetros ($step_{vol\_min}$ e $step_{vol\_max}$). $step_{vol}$ é decresce linearmente no decorrer das iterações do algoritmo. Esse processo faz com que o algoritmo tenha um comportamento inicial de busca em amplitude e altere seu comportamento dinamicamente, para realizar busca em profundidade.

4. INTELIGÊNCIA DE ENXAMES COM PROCESSAMENTO PARALELO

Alguns algoritmos de Inteligência de Enxames já foram adaptados para serem executados em plataformas com processamento paralelo baseado em GPU. Zhou e Tan [16] propuseram variações do algoritmo PSO, para serem executadas utilizando o paralelismo de uma plataforma com GPUs. Nesse trabalho, os autores fizeram um comparativo do desempenho das propostas paralelas em relação a uma versão sendo executado em CPU, apresentando uma análise do desempenho em termos de tempo de execução. A métrica usada é a relação do tempo de processamento da CPU e a GPU, conhecida como *speedup*. A análise foi realizada variando o número de partículas e também o número de dimensões. Como resultado, a proposta do algoritmo paralelizado conseguiu atingir melhor desempenho em termos de tempo de execução, obtendo valores similares para as funções de testes calculadas com a versão sequencial. Nos diversos experimentos realizados, os valores de *speedup* obtidos variaram entre $3,7$ a $11,4$ a depender da função de teste e número de partículas e dimensões.

A proposta apresentada por Bastos-Filho *et al.* [17] analisaram o desempenho de variações do algoritmo do PSO com a geração de números aleatórios sendo realizada na CPU e na GPU. Essa proposta utiliza-se do gerador de números aleatórios com o operador *XORshift* proposto por Marsaglia [18], e demonstrou resultados bastante eficientes nas proposições com PSO. Também foram feitas análises comparativas com diferentes versões do PSO baseados em GPU (processamento síncrono e assíncrono) em relação à versão do algoritmo baseado em CPU. Mussi *et al*. [19] apresentaram uma arquitetura para paralelização da versão padrão do PSO (SPSO) em GPU utilizando CUDA. Foram feitas comparações com uma versão em CPU do algoritmo, utilizando diversas funções de testes conhecidas, a fim de analisar o desempenho e a escalabilidade da arquitetura, demonstrando melhorias significativas em relação à versão executada em CPU. Zhu & Curry [20] adaptaram o algoritmo de otimização por colônia de formigas para otimizar funções de teste em GPUs. Pospíchal *et al.* [21] implementaram um Algoritmo Genético paralelo para arquitetura CUDA, com versões síncronas e assíncronas do algoritmo, fazendo com que grupos de indíviduos fossem mapeados em blocos de processamento na memória da GPU.

5. PROPOSTA DE PARALELIZAÇÃO DO FSS PARA PLATAFORMAS GPU

As técnicas propostas na área de Inteligência de Enxames geralmente são estruturadas de forma que estas sejam processadas em paralelo, mesmo que a maioria das implementações sejam implementadas sequencialmente. A proposta de realizar uma análise do algoritmo baseado em cardumes de peixes, FSS, apresentado na seção 3, tem por finalidade demonstrar a estruturação do algoritmo, utilizando a arquitetura CUDA da NVIDIA [7] [8], para ser processado em paralelo em múltiplas placas de processamento gráfico (GPU). É importante ressaltar que até onde é de conhecimento dos autores, nenhuma adaptação do FSS para GPU foi apresentada na literatura.

A proposta deste artigo é apresentar uma versão do FSS com processamento paralelo em GPU (pFSS - *parallel Fish School Search*), utilizando a arquitetura de desenvolvimento paralelo CUDA. Neste trabalho é analisado a utilização dinâmica dos recursos, bem como a organização da memória de dados dentro da GPU, durante o processamento do algoritmo. Também é apresentado um estudo sobre o impacto do uso de barreiras de sincronização, comparando ao desempenho do FSS sendo executado na CPU.

5.1 Síncrono x Assíncrono

A arquitetura CUDA realiza paralelismo de diversas maneiras, sendo a primeira delas através da implementação de um modelo de execução síncrono ou assíncrono, de acordo com o grau de paralelismo e o tipo do problema a ser resolvido. O paralelismo pode ser conseguido entre *threads*, que são linhas de processamento de uma dada funcionalidade, entre blocos de *threads*, entre chamadas a funções *kernel* (que são executadas nas GPUs) e entre GPUs, quando mais de uma GPU é utilizada no mesmo processamento, e por último quando há transferência de dados de/para processador *host* (CPU) e *device* (GPU).

Neste artigo propõe-se uma análise do desempenho do FSS durante a execução com sincronização de processamento, dentro de cada função *kernel* que é carregada em blocos de memória na GPU. Para que seja feito o controle do sincronismo entre as *threads* de um mesmo bloco de execução, utiliza-se de barreiras de sincronização feitas pela função do CUDA chamada *__syncthreads()* [7], no ponto dentro da função *kernel*, onde se deseja aguardar que todas as *threads* tenham sido executadas. O uso de barreira de sincronização tende a diminuir o desempenho do processamento paralelo a depender do tipo de problema a ser resolvido e a configuração da memória a ser utilizada na GPU, pois cada barreira de sincronização faz com que cada bloco avalie se todas as suas linhas de processamento foram concluídas, e isso geralmente gera impacto negativo no *speedup* do algoritmo em execução.

As chamadas às funções *kernel* são realizadas de forma assíncrona, o que faz com que o processador *host* carregue uma função *kernel* numa fila de execução na GPU, e não aguarde pelo término dela, mas continue a realizar algum outro trabalho. Algum tempo depois de estar na fila da GPU, a função *kernel* é executada. Entretanto, devido a esse mecanismo assíncrono de chamada das funções *kernel*, não há como retornar um valor resultante dessas funções, mas pode-se utilizar uma variável global para receber o resultado deste processamento. Por questões de eficiência, é criado um *pipeline* para enfileirar um número de funções *kernel*, de modo que a GPU permaneça ocupada a maior quantidade de tempo possível. Com este cenário, outra forma de sincronização pode ser necessária para que o processador *host* possa determinar quando o processamento de uma *kernel* ou o *pipeline* foi concluído. Na implementação do pFSS Síncrono, foi introduzida a barreira de sincronização para as *threads* antes do processamento do cálculo da função objetivo do passo *volitivo*, fazendo com que todos os blocos de processamento na GPU sejam executados por completo e o cálculo do *fitness* seja realizado com os valores mais atuais encontrados pelos peixes em todas as dimensões do problema analisado.

Para que o FSS pudesse ser implementado e testado numa versão assíncrona, alguns aspectos foram considerados e analisados, de forma a evitar um baixo desempenho e perdas na qualidade dos resultados. A retirada das barreiras de sincronização, nos processamentos das *threads*, a alocação de memória, e a comunicação entre CPU e GPU, foram os aspectos analisados e implementados em duas diferentes versões do pFSS em modo assíncrono. De uma forma geral, o processamento em uma abordagem paralela assíncrona é bem mais rápida que a versão síncrona, devido a ausência de barreiras de sincronização. Entretanto, apesar da velocidade ser maior, os resultados em termos de *fitness* podem ser piores, pois as informações adquiridas não necessariamente são as mais atuais do processamento em curso.

As versões síncrona e assíncronas do pFSS apresentadas foram implementadas sem nenhuma técnica de otimização de processamento, que estão disponíveis em CUDA, em sua versão 4.0, e a arquitetura Fermi [7] [8] [9]. Entretanto, outras versões de código do pFSS, utilizando técnicas de otimização, serão apresentadas nas próximas seções.

5.2 Modelagem de memória

Com relação às variáveis de memória em CUDA nas versões do pFSS, foram utilizadas variáveis num escopo de memória global, quando usadas nas funções *kernel* e também num escopo de memória local, para declarar índices de dimensões, blocos e *threads*, e para realizar a computação das funções de teste utilizadas como funções objetivo nas simulações, por exemplo. Outras variáveis, também foram declaradas como sendo de memória compartilhada, como no caso da realização do cálculo do baricentro utilizado no operador volitivo, que é uma variável global do algoritmo.

A estrutura das variáveis, a serem manipuladas durante a execução do FSS, foi modelada de forma que o processamento de cada peixe, em cada dimensão do problema, estivesse associado a uma *thread* dentro de um dado bloco na GPU. O agrupamento dos peixes foi feito como uma matriz, com P (peixes) e D (dimensões), ficando uma matriz $m(P, D)$, e foi utilizada para armazenar as posições dos peixes em todas as dimensões. Dependendo do tipo de atributo do FSS, tal como o peso e o valor da

aptidão (*fitness*), que serão únicos para cada peixe, esse valores serão computados e armazenados em uma matriz linha do tipo $m(1, P)$, ou coluna do tipo do tipo $m(D, 1)$, dependendo da forma como os dados forem organizados na memória.

Inicialmente foram adotadas a alocação de grande parte das variáveis como sendo do tipo Global, de forma que todas as *threads*, independente do bloco que a estivesse, pudessem ter acesso ao conteúdo destas variáveis, de acordo com o esquema apresentado na Figura 1. Nas versões iniciais do pFSS (síncrona e assíncrona), a operação de alocação de memória dessas variáveis globais, foram implementadas utilizando o comando *cudaMalloc* [6].

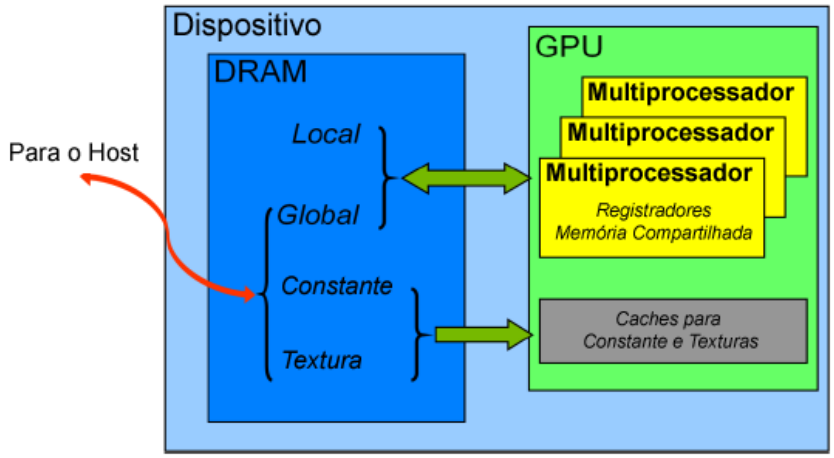


Figura 1: Representação dos espaços de memória existentes numa GPU CUDA. Figura adaptada de [8].

Cada variável alocada em memória deverá ser cuidadosamente desalocada quando o processamento for finalizado, a fim de que o espaço ocupado por ela seja liberado para outros fins. Assim sendo, da mesma forma que o padrão C utiliza *free()* para liberar a memória ocupada por um *malloc()*, em CUDA e consequentemente no pFSS deverá ser chamado *cudaFree()* para liberar a memória alocada na GPU [7] [12]. Essas variáveis alocadas, serão passadas para cada uma das funções *kernel* a serem executadas, de acordo com o algoritmo do pFSS. As variáveis utilizadas no pFSS alocadas em memória compartilhada, dentro de uma determinada função *kernel* na GPU, foram utilizadas de acordo conforme descrito em [12].

Neste caso, a variável criada na memória compartilhada poderá ser acessada por todas as *threads* que compõem o mesmo bloco, e serviram para armazenar o valor absoluto da maior diferença entre os *fitness* calculados. Após ser manipulada, a variável na memória compartilhada deverá ter o seu conteúdo passado para uma variável declarada globalmente, a fim de que o seu valor seja utilizado em outras operações quando a função *kernel* for finalizada [6].

5.3 Comunicação CPU x GPU

A comunicação feita entre a GPU (*Device*) e a CPU (*Host*) pode ser feita em duas vias, e dependendo da forma como é realizada, deve haver um processo de sincronização, que pode impactar no aumento do custo computacional existente desta transação. Na versão síncrona do pFSS, utilizam-se os comandos de cópia de dados de memória do CUDA (*cudaMemcpy*), da CPU para a GPU, e vice-versa. Esse processo de comunicação, gera um gargalo no processamento e deverá ser evitado sempre que possível.

Com o uso de memória compartilhada, as *threads* poderão se comunicar dentro do bloco, sem a necessidade de utilizar barreiras de sincronização para efetuar essa comunicação. Entretanto, se estiver transferindo dados entre os registradores e a memória compartilhada, a barreira de sincronização deve ser chamada para assegurar que todos os elementos da memória compartilhada tenham sido copiados. Dessa forma, a redução no custo computacional do pFSS torna-se considerável e a dinâmica conseguida pela paralelização do processamento aumenta sua eficiência em problemas de otimização hiperdimensionais, melhorando assim sua escalabilidade.

5.4 pFSS com otimização

As versões iniciais do pFSS, estão baseadas em implementações básicas, apenas para demonstrar a viabilidade do processo de paralelização e da utilização da arquitetura CUDA. Nesta seção serão demonstrados pontos de processamento da técnica, utilizando as abordagens para otimização de processamento paralelo, existentes em CUDA versão 4.0, visando aumento de desempenho, maior *speedup* no processamento, evitando perdas na qualidade dos resultados ao final de cada processamento, de acordo com as abordagens descritas em na subseção 2.3.1.

Inicialmente foi feita uma análise referente à estrutura de dados utilizada para armazenar os dados das posições, pesos, *fitness*, dentre outras informações referentes aos peixes durante o processamento do algoritmo, bem como foram re-estruturadas as configurações da grade de processamento (*grid*), com relação ao número de blocos, *threads*, número de *warps* e utilização de memória compartilhada para melhorar a taxa de ocupação na GPU, aumentar a largura de banda e reduzir a latência durante o

processamento do pFSS. Num segundo momento, foi feita também uma análise para utilização de acesso de memória coalescente, mudanças no controle de fluxo das instruções em algumas funções *kernel* para evitar divergências no processamento das *threads*, melhor utilização de barreiras de sincronização, juntamente com processos assíncronos e uso de *streams* e desenlace de laços de repetição. Serão apresentadas a seguir as análises realizadas em cada técnica de otimização aplicadas ao pFSS.

5.5 Estruturas de dados

A estrutura de dados no pFSS é do tipo Estrutura de *Arrays* (SOA). Nas versões iniciais do pFSS, a estrutura de dados SOA foi implementada com todos os dados utilizados no processamento do algoritmo em uma estrutura única. Consequentemente, com todos esses dados alocados em memória global, aumenta a possibilidade de haver uma carga alta de dados na memória, e não há também uma garantia de se obter coalescência no acesso aos dados. Atributos da estrutura relativos ao posicionamento dos peixes, seus pesos e valores de aptidão calculados fazem parte de uma estrutura única. Porém, ter uma estrutura única com atributos de tamanhos diversos (ocupação de memória) pode levar a uma alta latência e a um baixo consumo de largura de banda, impactando negativamente no desempenho do pFSS.

Visando obter um melhor aproveitamento da memória, com baixa latência e aumento da largura de banda, uma nova organização dos dados foi preparada, de acordo com suas caraterísticas e tamanho dos dados no processo de alocação em memória. Uma mudança importante consistiu na retirada de alguns atributos da estrutura original como global, passando para um escopo local a cada função *kernel*, de acordo com a necessidade, como por exemplo o atributo que armazena o novo valor da posição de cada peixe. Um exemplo da mudança da estrutura dos dados está apresentada na Listagem 1.

Listagem 1: Exemplo de Estrutura de *Arrays* para o pFSS - codificado na linguagem C.

```
typedef struct {
        double *positions;
        double *deltaX;
} TPositions;

typedef struct {
        double  *weight;
        double  *fitness;
        double *oldWeight;
        double *deltaFitness;
        double *fitnessNeighbor;
} TOutputs;

typedef struct {
        double *gbestFitness;
        unsigned int *gbest;
        unsigned int *cycleGbest;
        double *stepInd;
        double *stepVol;
        double *school_barycentre;
} TData;
```

Com base nessa nova organização de dados, tem-se que os dados serão alocados em suas respectivas estruturas de *arrays*, de acordo com a quantidade de memória a ser utilizada, de modo a melhorar o acesso aos dados na memória da GPU. Vale ressaltar que todos os dados a serem alocados na memória da GPU deverão ser declarados como ponteiros, de forma que os endereços dos dados sejam manipulados durante o processamento na memória global da GPU [1].

5.6 Grades de Processamento e Ocupação da GPU

A configuração de grades de processamento (*grades*, blocos e *threads*), para a versão do pFSS com otimização, considera a análise da taxa de ocupação da GPU, visando reduzir a latência e aumentar a largura de banda no processamento das funções *kernel*. Com foco no processamento a ser feito em GPUs, como por exemplo o modelo Tesla C2070, que possui *warps* com 32 *threads*, cada *streaming multiprocessor* (SM) contendo 48 *warps* ativos, fazendo um total de 1536 *threads* ativas. Neste modelo de placa de GPU, o qual possui 14 SMs, tem-se um total de 21.504 *threads*.

A fim de aproveitar a maior taxa de ocupação possível, foram seguidos os seguintes passos para a melhor configuração de blocos e *threads* de processamento:

- **Analisar o arquivo PTX:** Através do resultado da compilação das funções kernel, utilizando os parametros pra ativar a saída com a codificação PTX para cada função kernel, onde são obtidas as informações referentes ao número de registradores e quantidade memória compartilhada utilizada em cada uma das funções;
- **Calculadora de Ocupação:** De posse desses dados, aplicá-los na planilha que realiza os cálculos da taxa de ocupação. Analisar que nem sempre o número máximo de *threads* por bloco fornecerá a melhor taxa de ocupação (relação da latência com largura de banda). Outro ponto importante é verificar o número de *warps* que serão utilizados com essa configuração, quanto maior o número, maior será a capacidade de processamento. A planilha fornecerá o número de blocos, de acordo com as configurações fornecidas (número de registradores, memória compartilhada) e o número de *threads* por bloco escolhido ao analisar o gráfico de impacto da variação do tamanho do bloco [8] [22];

- **Configurar Grades:** Com os dados calculados pela planilha de ocupação, o número de blocos e de *threads* por bloco, o próximo passo é codificar a configuração dos *grids*. Neste ponto foram criadas diferentes configurações de *grid*, para cada função *kernel*, conforme sua capacidade de processamento. Também foi feito o cálculo do número de blocos da grade, de acordo com a quantidade de dados a ser processada por cada grade.

5.7 Alocação de dados e uso de *Streams*

Na versão inicial do pFSS, todas as variáveis de memória utilizadas para receber algum resultado de processamento realizado na CPU, ou seja variáveis *host*, foram alocadas utilizando comando de alocação em memória padrão do C, *malloc()*, conforme exemplo apresentado em [6] e também em [7]. Buscando otimizar esse processo de alocação em memória *host*, foi utilizada abordagem de memória pinada, neste caso usando *cudaMallocHost*, para efetuar alocação na memória *host*, mas que pode ser acessada mais rapidamente quando da cópia do dado proveniente da GPU. A outra forma de alocação de memória pinada foi utilizada em versão do pFSS para múltiplas GPUs, apresentada em detalhes na seção 6 deste trabalho. O exemplo mostrando a alocação de memória pinada no pFSS, está descrito em [6] e [12].

Ao utilizar memória pinada, dois pontos importantes devem ser observados. O primeiro deles é que deve ser verificada quantidade de memória a ser alocada, para evitar comprometimento da largura de banda. O segundo ponto é que todo processo de transferências de dados deve ser assíncrono (*cudaMemcpyAsync*), de forma que seja aproveitada a capacidade de tratar com múltiplos *buffers* da memória na GPU [1].

5.8 Uso de desenlace

Sendo o FSS um algoritmo com vários pontos de iteração de seus agentes através de laços de repetição, além da quantidade de laços existentes no algoritmo, o tamanho desses laços conforme o número de dimensões e de peixes influenciam no desempenho do algoritmo para para resolver determinado problema. Para o pFSS, a utilização de *Loop Unrolling* a fim de aumentar o desempenho do algoritmo foi feito conforme descrito em [12].

Com o uso da diretiva *pragma* com valor igual a 4, por exemplo, têm-se uma operação de desenlace onde o compilador irá montar uma estrutura para que o conteúdo do laço seja preparado para processamento de 4 em 4 passos, ou seja, ao invés de um processamento por vez dentro do laço, haveriam 4 linhas de processamento por vez, tornando o laço mais rápido.

5.9 Barreiras de sincronização

O uso de barreiras de sincronização em aplicações de computação paralela permitem obter um controle do processo de execução das *threads*, de forma que possíveis divergências ou dados incoerentes sejam considerados ao longo da execução do algoritmo. Como mencionado previamente, pode haver um impacto no desempenho do algoritmo quando comparadas às versões *com* e *sem* barreiras de sincronização. Porém, para melhorar a qualidade dos resultados relativos ao processo de otimização, foi necessário o uso de barreiras de sincronização em alguns pontos do algoritmo e o impacto no desempenho foi mínimo, não prejudicando o processamento mesmo em problemas com alta dimensionalidade.

Para o otimizar o processamento pFSS com barreiras de sincronização, foi utilizada uma barreira de sincronização de *streams* ao final de cada ciclo de processamento dos operadores, além de barreiras de sincronização ao final das funções *kernel* que gerenciavam o processamento da função objetivo.

6. pFSS EM MÚLTIPLAS GPUs

Uma versão do FSS paralelizado foi projetada para problemas de otimização com configurações (número de dimensões e peixes) elevadas para execução em múltiplas placas com GPUs. As múltiplas placas GPUs podem dividir o processamento, aumentando o desempenho do processamento desde que o volume de dados seja justificável para processamento em mais de uma GPU ao mesmo tempo. Para que o pFSS seja processado em mais de uma plataforma GPU, sem perda de desempenho, utilizam-se de abordagens específicas do CUDA para otimizar o processamento sem perdas de largura de banda durante a comunicação entre o *device* e o processador *host* e entre as GPUs disponíveis na configuração do sistema utilizado. Foi utilizada uma abordagem usando o conceito de subenxames, ou subcardumes, para dividir o volume de dados a serem processados nas GPUs disponíveis no sistema, como estratégia de execução buscando aumento de desempenho do pFSS.

A comunicação entre os subcardumes é realizada a cada n iterações, onde o peixe de melhor *fitness* de cada subcardume, e a informação deste peixe será enviada para o subcardume processado na placa de GPU vizinha e será inserido no lugar de um peixe que estiver com o pior valor de *fitness*.

6.1 UVA & Peer To Peer

A versão do pFSS para múltiplas placas GPUs utiliza duas abordagens existentes em CUDA, para aumentar o desempenho na comunicação entre *device-host* (e vice-versa) e entre múltiplas placas GPUs. Para comunicação entre *device-host* usando UVA (*Unified Virtual Access*) no pFSS, copiando o conteúdo de uma variável no *device* para o *host* diretamente, com um custo computacional muito baixo, pois também se utiliza de uma variável no *host* alocada como descrito na abordagem *Zero-Copy*, conforme apresentado na Listagem 2.

Listagem 2: Exemplo do uso de UVA no pFSS.

```
cudaHostAlloc( (void**)&gbestLeader, sizeof(float), cudaHostAllocWriteCombined);

cudaMemcpyAsync(gbestLeader, fssDataStruct[j].gbestFitness, sizeof(float), cudaMemcpyDefault);
```

No trecho de código apresentado na Listagem 2, uma variável é declarada no *host* para acesso direto do *device*, de forma a armazenar o valor da melhor posição do melhor peixe do subcardume e seu valor do *fitness* resultante da execução da função objetivo. A cópia do dado, neste exemplo, é realizada em modo assíncrono.

Para realizar processamento em múltiplas GPUs, o pFSS inicialmente realiza a verificação da quantidade de GPUs disponíveis no sistema, e indica quais as GPUs deverão estar habilitadas para comunicação P2P (*Peer-to-Peer*) entre as GPUs. Além de indicar os pares de GPUs que estarão conectados, o pFSS fará uso de UVA para realizar as cópias assíncronas entre cada *device*, conforme apresentado na Listagem 3.

Listagem 3: Exemplo do uso de P2P com UVA no pFSS.

```
//variaveis para indicar permissao de acesso entre pares
int can_access_peer_0_1, can_access_peer_1_0;
//verifica se acesso entre pares de GPUs permitido
cutilSafeCall(cudaDeviceCanAccessPeer(&can_access_peer_0_1, 0, 1));
cutilSafeCall( cudaSetDevice(0) );
cudaDeviceEnablePeerAccess(1, 0);
//realizando copias entre GPU 0 e 1
cudaMemcpy(fssDataStruct[1].refPosition, fssDataStruct[0].gbestPosition, matrixDim, cudaMemcpyDefault);
cudaMemcpy(fssDataStruct[1].refFitness, fssDataStruct[0].gbestFitness, sizeof(double), cudaMemcpyDefault);
```

No exemplo de código na Listagem 3 o processo de cópia entre os dispositivo não está usando o modo assíncrono, mas uma versão com cópia assíncrona foi avaliada durante os experimentos. Após o processamento ser concluído, o pFSS desabilita o acesso P2P entre os pares de *devices* configurados.

7. EXPERIMENTOS

Nesta seção são apresentados os detalhes das configurações para testes e comparações entre as análises implementadas para o pFSS, usando as métricas empregadas. Em seguida, serão apresentados os resultados obtidos pelos testes do algoritmo pFSS em diversas configurações. As análises implementadas para o pFSS, apresentadas neste trabalho são:

- análise do uso de barreiras de sincronização;
- análise do número de peixes/dimensões;
- uso de técnicas de otimização;
- utilização de MultiGPUs.

Os experimentos são realizados de acordo com cada análise a ser feita para o pFSS, utilizando funções de testes e variações nas configurações das grades de processamento, conforme as diferentes proposições estudadas. Para a primeira análise, o objetivo é realizar um estudo comparativo entre versões com e sem barreiras de sincronização, no processamento síncrono e assíncrono de funções *kernel* na GPU. Na segunda análise, os experimentos têm por objetivo, verificar o impacto do aumento do número de dimensões e peixes nas configurações das grades de processamento das funções *kernel* em GPU. Os algoritmos utilizados nos experimentos da segunda análise são implementados em modo assíncrono, sem nenhuma barreira de sincronização durante o processamento. Para a terceira análise, os experimentos são realizados em uma versão do pFSS com otimizações referentes à mudança nas configurações de grades de processamento conforme a taxa de ocupação atingida por cada função *kernel* na GPU, bem como a utilização de *streams* de processamento, comunicação assíncrona entre CPU e GPU, mudanças na estrutura de código para realizar *Loop Unrolling* e também para evitar divergências no processamento das *threads*. Os experimentos realizados para a quarta e última análise do pFSS proposta demonstram o processamento do algoritmo em múltiplos dispositivos gráficos, considerando o processamento utilizando a técnica P2P de CUDA para a comunicação entre mais de uma GPU e acesso direto a memória *host*, para configurações de grade processamento de alta dimensionalidade.

7.1 Configurações de testes

São realizadas simulações com o pFSS usando configurações de *hardware* específicas, compatíveis com a arquitetura CUDA, mas com diferentes capacidades de processamento. Os computadores utilizados foram baseados em Sistemas Operacionais baseados em Linux como: MacOSX versão 10.6.8 (*Snow Leopard*) e o Ubuntu 10.10.

O PSC (*Personal Super Computer*) possui 4 placas gráficas compatíveis com a arquitetura CUDA, e 4 processadores Intel, totalizando 16 núcleos de processamento e 24 GBytes de memória RAM. Para efeito de comparação com o processamento, foram realizados testes em um computador pessoal (*notebook*) da marca *Macbook Pro*, da Apple, que é equipado com uma placa gráfica compatível com CUDA, modelo GeForce 320M, processador Intel com 2 núcleos de processamento e 4 GBytes de memória RAM.

Tabela 2: Configurações das GPUs utilizadas nos experimentos.

Computador	Modelo	Versão CUDA	Número de Placas	Núcleos por GPU	Threads por Bloco	Memória Global
PSC	Tesla C2070	4.0	4	448	1024	5375 MB
MacBook Pro	GeForce 320M	3.2	1	48	512	265 MB

Os experimentos utilizam variações do número de dimensões e peixes para configurar as grades de processamento na GPU. As configurações de dimensões e peixes também foram utilizadas em simulações executadas na versão sequencial do FSS para CPU. Em cada experimento foram executadas 50 simulações e cada uma com 10000 iterações em cada conjunto de teste, sendo cada teste organizado de forma a executar uma abordagem proposta para o pFSS com diferentes números de dimensões (30, 100, 500 e 1000), mantendo o número de peixes fixo igual a 30. Os resultados apresentam os valores dos tempos de execução de cada simulação e o melhor valor global para a função de teste utilizada.

Além do número de dimensões e peixes, o FSS utiliza-se da faixa de valores do espaço de busca que é inferido no problema, e dos parâmetros utilizados nos passos individual e volitivo do FSS conforme descrito na Tabela 3.

Tabela 3: Valores inicial e final para os passos Individual e Volitivo do FSS.

Operador	Valor do passo	
	Inicial	Final
Individual	$10\%(2 \cdot maximo(\text{espaço de busca}))$	$1\%(2 \cdot maximo(\text{espaço de busca}))$
Volitivo	$10\%(Individual_{Inicial})$	$10\%(Individual_{final})$

As funções de teste estão descritas na Tabela 4 com a faixa de valores para cada uma delas. Com a função Rosenbrock, o valor para o atributo α foi igual a 100 em todas as simulações com esta função. As Figuras 2(a) e 2(b) demonstram a descrição dos espaços de busca das funções de teste para 2D. É importante observar que a função Rosenbrock é uma função com plateau e é difícil para algoritmos que utilizam informação de gradiente. Por outro lado, a função Rastrigin é uma função com muitos mínimos. Estas funções de teste são amplamente utilizadas para avaliação de algoritmos de intelogência de enxames.

Tabela 4: Funções de teste com os respectivos espaços de busca.

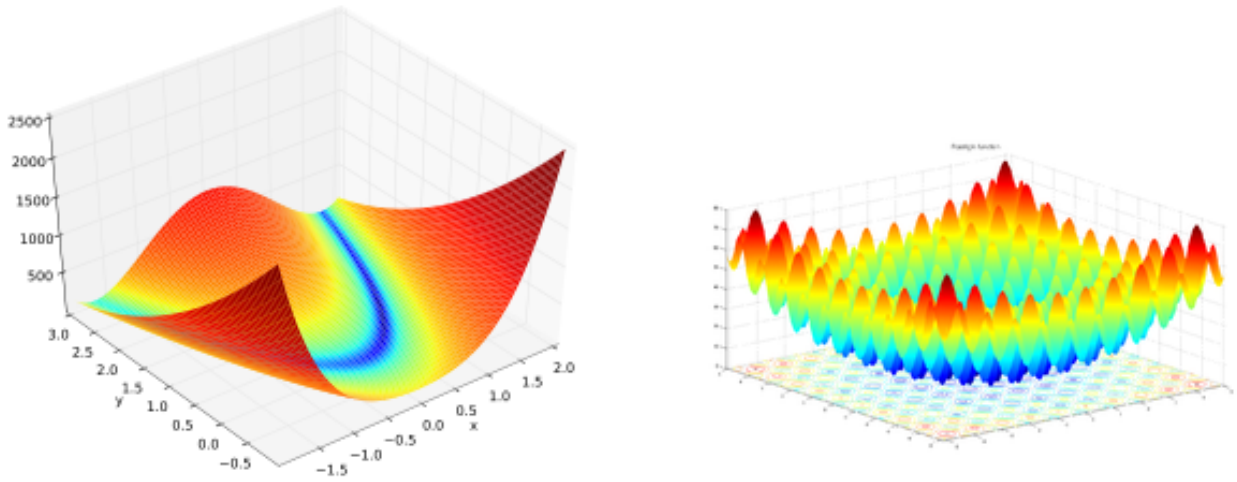
Função	Espaço de Busca
$F_{Rosenbrock}(\vec{x}) = \sum_{i=1}^{n}[\alpha(x_{i+1} - x_i^2)^2 + (1 - x_i)^2]$	$[-30; 30]$
$F_{Rastrigin}(\vec{x}) = 10n + \sum_{i=1}^{n}[x_i^2 - 10cos(2\pi x_i)]$	$[-5,12; 5,12]$

Para os experimentos com múltiplas placas com GPUs, são realizadas análises de desempenho (tempo de execução) e dos valores das funções objetivos, obtidos com processamento em 2, 3 e 4 placas gráficas executando em paralelo, em um supercomputador pessoal com arquitetura Fermi. No processamento em múltiplas placas com GPUs, os números de dimensões e peixes são aumentados de forma a verificar a taxa de ocupação máxima em cada GPU. Também é avaliada a utilização de múltiplas *threads* de processamento em CPU, de forma a realizar a distribuição do controle do processamento realizado em mais de uma placa gráfica para mais de um núcleo de CPU ativo. Assim sendo, o algoritmo é analisado de acordo com as seguintes versões implementadas neste trabalho, apresentadas na Tabela 5.

Para cada versão, é executado o conjunto de funções de teste com diferentes números de dimensões, gerando amostras de dados a serem analisados usando diferentes funções estatísticas. O valor do *speedup* obtido através da análise do processamento sequencial em CPU e paralelo em GPU, é calculado para cada conjunto de teste. Ao realizar os experimentos com multiGPUs, a cada 200 iterações há um processo de sincronização entre os processos que estão executando em cada placa gráfica, de forma a compartilhar a informação do melhor peixe, a posição e valor da aptidão calculada pela função objetivo. Com isso, o novo cardume com ótimas posições é influenciado de forma a colaborar com a aceleração do processo de convergência e melhorar a qualidade dos resultados.

8. RESULTADOS DOS EXPERIMENTOS

Nesta Seção são apresentadas comparações entre o desempenho do FSS original e as abordagens paralelizadas propostas neste trabalho, utilizando os experimentos descritos na Seção 7.1. Os experimentos realizados foram divididos de acordo com o



(a) Rosenbrock (b) Rastrigin

Figura 2: Gráficos das funções de teste (a) Rosenbrock e (b) Rastrigin.

Tabela 5: Versões de teste para o pFSS.

Versão	Descrição
CPU-MAC	sequencial com processamento em CPU do MAC
GPU-MAC	paralelizada com processamento em GPU do MAC
CPU-PSC	sequencial com processamento em CPU do PSC
GPU-PSC	paralelizada com processamento é uma única placa gráfica do PSC
MGPU2-PSC	paralelizada com processamento em duas placas gráficas do PSC
MGPU4-PSC	paralelizada com processamento em quatro placas gráficas do PSC

tipo do dispositivo a ser utilizado no teste, utilizando as siglas PSC e MAC para identificar as configurações de processamento em CPU e GPU, no *Personal Super Computer* e no *MacbookPro*, respectivamente. Cada experimento também é referenciado pelo nome da função de teste e o número de dimensões processado. A configuração do MAC é o referencial para processamento sequencial do FSS, a ser comparada com as abordagens paralelas propostas. Essa comparação foi pensada de forma a mostrar que o algoritmo é escalável com aumento significativo do número de dimensões, além de demonstrar o comportamento do algoritmo em diferentes configurações de *hardware* para resolver o mesmo tipo de problema.

8.1 Análise de Sincronismo

Inicialmente foram realizados experimentos com uma versão do pFSS para avaliar o uso de barreiras de sincronização de processamento. Uma versão síncrona, com várias barreiras de sincronização nas funções *kernel*, e outras duas versões assíncronas, sendo uma com um número reduzido de barreiras e outra sem nenhuma barreira de sincronização. Os resultados obtidos dessa análise estão representados nas Tabelas 6 e 7, apresentando os valores de *fitness* e tempo obtidos cada função de teste, respectivamente.

Os resultados foram obtidos sem que as versões do pFSS estivessem utilizando qualquer recurso de otimização do processamento disponível no CUDA. Os resultados obtidos demonstram ganho de desempenho sem perdas na qualidade dos resultados. A análise do desempenho das abordagens de acordo com a utilização de barreiras de sincronização, demonstram um ganho no *speedup* de até 6 vezes, nas versões assíncronas de acordo com a Tabela 7.

8.2 Comparações de desempenho

Foi realizada uma análise do desempenho em cada uma das funções, utilizando o processamento sequencial na CPU do MAC como referencial e comparando com as versões do pFSS implementadas com técnicas de otimização de processamento do CUDA. Percebeu-se que os resultados calculados pelas funções objetivas em GPU, apresentam leves diferenças em relação aos obtidos nos processamentos em CPU, devido a dois fatores que são:

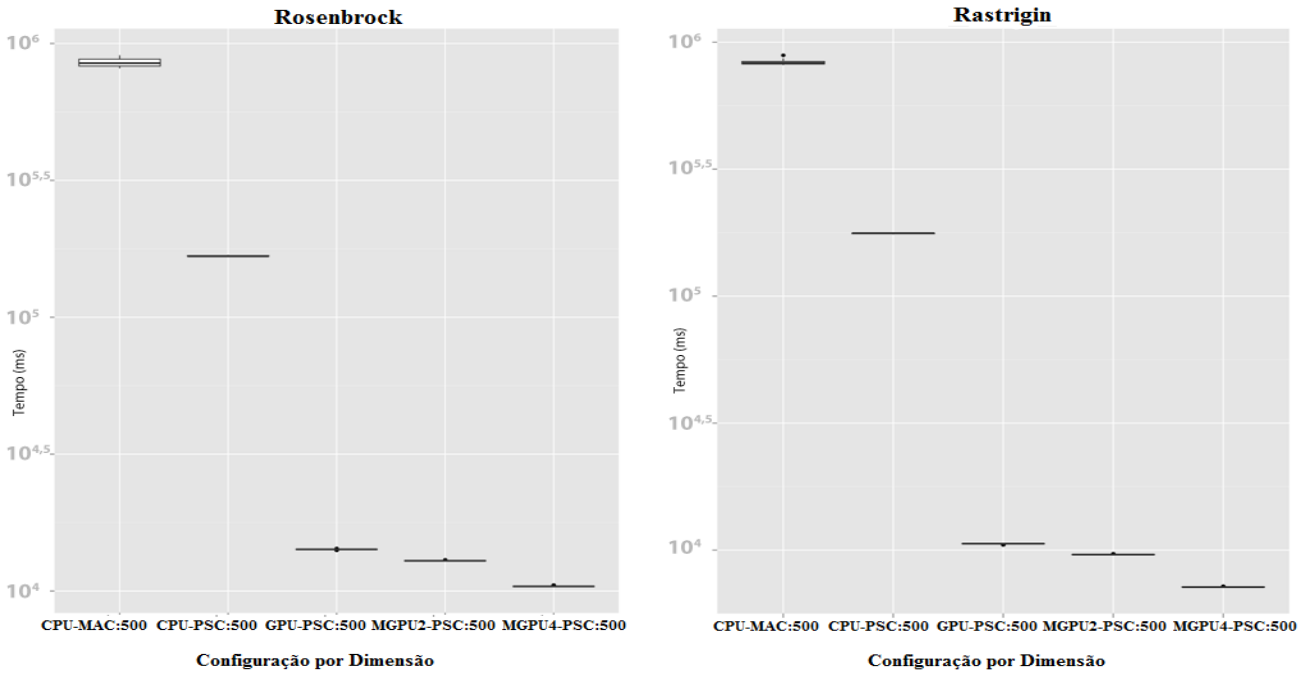
- utilização de processamento assíncrono: fazendo com que as *threads* sejam processadas com maior independência, gerando valores com diferenças durante a execução das funções no *kernel* das GPUs;
- processamento de ponto flutuante em GPU: Segundo Whitehead e Fit-florea [3], as GPUs da NVIDIA possuem diversos problemas com processamento de cálculos com ponto flutuante de precisão simples e dupla. Os autores sugerem algu-

Tabela 6: Média e desvio padrão do valor da função objetivo para as funções Rosenbrock e Rastrigin.

Função	Versão do Algoritmo	*Fitness*	
		Média	Desvio Padrão
Rosenbrock	CPU	2,89e+01	2,00e-02
	GPU Síncrono	2,89e+01	1,00e-02
	GPU Assíncrono A	2,89e+01	1,00e-02
	GPU Assíncrono B	2,89e+01	2,00e-02
Rastrigin	CPU	2,88e-07	5,30e-08
	GPU Síncrono	1,81e-07	4,66e-08
	GPU Assíncrono A	2,00e-07	2,16e-08
	GPU Assíncrono B	1,57e-07	1,63e-08

Tabela 7: Média e desvio padrão do tempo de execução e análise do *Speedup* para as funções Rosenbrock e Rastrigin.

Função	Versão do Algoritmo	Tempo (ms)		
		Média	Desvio Padrão	*Speedup*
Rosenbrock	CPU	6,69e+03	1,02e+03	–
	GPU Síncrono	2,04e+03	6,1e+01	3,27e+00
	GPU Assíncrono A	1,57e+03	9,29e+00	4,26e+00
	GPU Assíncrono B	1,57e+03	7,13e+00	4,27e+00
Rastrigin	CPU	9,60e+03	6,56e+02	–
	GPU Síncrono	2,00e+03	2,75e+00	4,79e+00
	GPU Assíncrono A	1,57e+03	2,11e+00	6,13e+00
	GPU Assíncrono B	1,57e+03	4,40e+00	6,13e+00



(a) Rosenbrock com 500 dimensões.

(b) Rastrigin com 500 dimensões.

Figura 3: *Boxplot* com comparação do *tempo* em milisegundos, obtido pelos algoritmos em execução sequencial e com paralelismo em 500 dimensões.

Tabela 8: Média e desvio padrão dos valores da função objetivo por configuração de *hardware* e número de dimensões.

Função	Configuração	*Fitness*			
		30 dimensões	100 dimensões	500 dimensões	1000 dimensões
Rosenbrock	CPU-MAC	2,88e+01(2,20e-02)	9,86e+01(5,00e-02)	4,99e+02(3,50e-01)	9,97e+02(5,78e-01)
	CPU-PSC	2,88e+01(3,00e-02)	9,86e+01(7,80e-02)	4,97e+02(5,50e-01)	9,94e+02(1,17e+00)
	GPU-MAC	2,90e+01(1,11e-01)	9,90e+01(5,30e-02)	NS	NS
	GPU-PSC	2,90e+01(3,40e-02)	9,90e+01(7,50e-02)	4,99e+02(0,00e+00)	9,99e+02(3,00e-02)
	MGPU2-PSC	NS	NS	4,99e+02(5,80e-01)	9,99e+02(6,44e-01)
	MGPU4-PSC	NS	NS	4,98e+02(8,90e-01)	9,98e+02(1,21e+00)
Rastrigin	CPU-MAC	5,12e-10(2,14e-10)	1,88e-09(3,81e-10)	9,62e-09(8,97e-10)	2,01e-08(1,64e-09)
	CPU-PSC	4,58e-08(1,52e-08)	1,03e-07(3,90e-08)	2,48e-07(4,56e-08)	3,46e-07(3,17e-08)
	GPU-MAC	2,47e-05(2,6e-06)	7,09e-05(1,91e-05)	NS	NS
	GPU-PSC	2,12e-05(5,6e-06)	7,60e-05(1,47e-05)	3,97e-04(4,33e-05)	6,56e-04(2,89e-05)
	MGPU2-PSC	NS	NS	3,21e-04(7,60e-05)	5,89e-04(7,09e-05)
	MGPU4-PSC	NS	NS	2,95e-04(7,31e-05)	5,63e-04(7,34e-05)

mas alternativas, como utilização de *flags* de compilação e mudanças na utilização de funções matemáticas, para tentar contornar essas dificuldades. Mesmo assim, nos resultados deste trabalho ainda permaneceram com algumas diferenças.

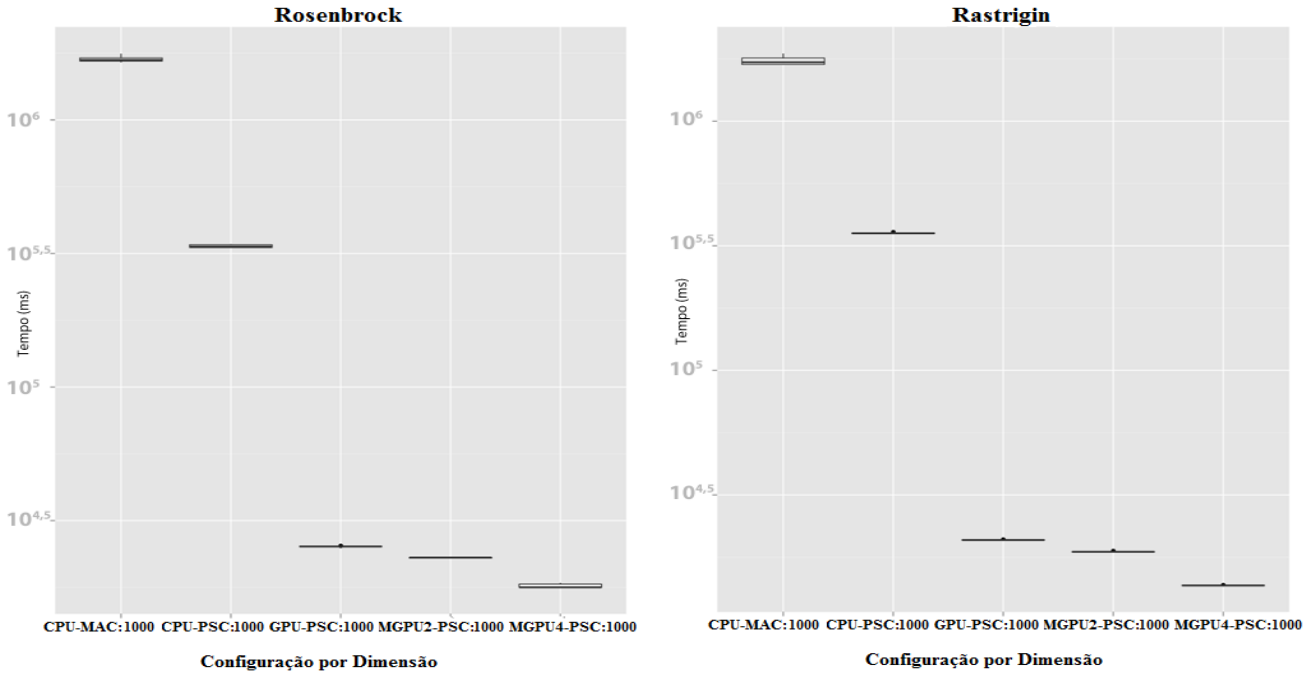
Os resultados com uma única placa com GPUs serão apresentados juntamente com os resultados com múltiplas placas na próxima sub-seção.

8.3 Comparações com Múltiplas Placas

Os resultados desta seção têm como objetivo mostrar o impacto da variação da quantidade de placas gráficas na melhoria de desempenho do algoritmo. Para mostrar o potencial dos mecanismos empregados, foram realizadas simulações em problemas com o número de dimensões igual a 500 e 1000, a fim de justificar o uso de recursos em processamento de grandes quantidades de dados. A Figura 3 mostra os gráficos *boxplot* comparativos entre as versões CPU no MAC e diferentes quantidades de placas gráficas para processamento paralelo no PSC para os problemas considerados com 500 dimensões. A Figura 4 demonstra a análise do desempenho do algoritmo para os problemas considerados com 1000 dimensões, em múltiplas placas gráficas, além do processamento sequencial.

A Tabela 8 apresenta a média e desvio padrão dos valores da função objetivo por configuração de *hardware* e número de dimensões. As análises dos resultados das funções objetivo, com o valor de *fitness*, apresentaram pequenas diferenças para a função Rosenbrock, isto porque o mínimo se localiza próximo de um plateau. Para a função Rastrigin, a diferença é significativa, mas em todos os casos as diferentes versões do pFSS retornaram bons resultados. Neste segundo caso, existem vários mínimos e estes são estreitos no espaço de busca. As diferenças observadas estão relaçionadas à precisão nas variáveis, processamento totalmente assíncrono e problemas de manipulação de valores com ponto flutuante pelas GPUs NVIDIA.

Na Tabela 9 estão apresentados os valores de média e desvio padrão para o tempo de processamento, em milisegundos. Com base nos tempos calculados para cada configuração, a Tabela 10 apresenta os valores de *Speedup* atingidos nos experimentos.



(a) Rosenbrock com 1000 dimensões.

(b) Rastrigin com 1000 dimensões.

Figura 4: *Boxplot* com comparação do *tempo* em milisegundos, obtido pelos algoritmos em execução sequencial e com paralelismo em 1000 dimensões.

Tabela 9: Média e desvio padrão dos tempos de processamento, em milisegundos, por configuração de *hardware* e número de dimensões.

Função	Configuração	Tempo(ms)			
		30	100	500	1000
Rosenbrock	CPU-MAC	5,12e+04(1,82e+03)	1,68e+05(4,50e+03)	8,52e+05(2,67e+04)	1,69e+06(3,75e+04)
	CPU-PSC	1,00e+04(1,41e+02)	3,33e+04(4,60e+02)	1,67e+05(7,74e+02)	3,37e+05(4,42e+03)
	GPU-MAC	5,22e+04(1,37e+02)	6,72e+03(4,01e+02)	NS	NS
	GPU-PSC	1,23e+03(2,40e-01)	2,77e+03(3,43e+00)	1,42e+04(2,86e+01)	2,52e+04(5,28e+01)
	MGPU2-PSC	NS	NS	1,29e+04(1,43e+01)	2,30e+04(5,44e+01)
	MGPU4-PSC	NS	NS	1,04e+04(2,77e+01)	1,80e+04(2,40e+02)
Rastrigin	CPU-MAC	5,29e+04(2,68e+03)	1,67e+05(4,31e+03)	8,32e+05(1,62e+04)	1,75e+06(7,29e+04)
	CPU-PSC	1,05e+04(5,03e+02)	3,56e+04(4,95e+02)	1,77e+05(7,18e+02)	3,55e+05(1,80e+03)
	GPU-MAC	5,25e+03(1,42e+02)	6,83e+03(4,80e+02)	NS	NS
	GPU-PSC	1,14e+03(1,27e+00)	2,46e+03(3,02e+00)	1,06e+04(3,91e+01)	2,08e+04(2,86e+01)
	MGPU2-PSC	NS	NS	9,60e+03(1,46e+01)	1,87e+04(2,86e+01)
	MGPU4-PSC	NS	NS	7,16e+03(9,37e+00)	1,37e+04(1,43e+01)

Ainda na Tabela 10 os valores da descritos na coluna CPU-MAC são os valores de referência para o tempo de processamento nesta configuração. Os valores constantes nas outras colunas das outras configurações representam quantas vezes o processamento foi mais rápido em relação ao tempo de processamento de referência. Nas duas funções de teste analisadas houve um aumento no desempenho, chegando a atingir um *speedup* de **127 vezes** na função Rastrigin, com 1000 dimensões, em processamento com 4 placas gráficas em paralelo. Esta mesma configuração para a função Rosenbrock atingiu um *speedup* acima de **93 vezes**. Na configuração com uma única placa gráfica ativa no PSC, o *speedup* foi de **66,8 vezes**. Já em uma placa gráfica de custo mais baixo, utilizada no MAC, o experimento com 100 dimensões atingiu o *speedup* de **25 vezes**.

Tabela 10: Valores de *Speedup* por configuração de *hardware* e número de dimensões.

Função	Dimensões	CPU-MAC	CPU-PSC	GPU-MAC	GPU-PSC	MGPU2-PSC	MGPU4-PSC
Rastrigin	30	5,29e+04	5,02e+00	1,01e+01	4,63e+01	NS	NS
	100	1,67e+05	4,69e+00	2,45e+01	6,80e+01	NS	NS
	500	8,32e+05	4,70e+00	NS	7,86e+01	8,66E+01	1,16e+02
	1000	1,75e+06	4,93e+00 p	NS	8,42e+01	9,37e+01	**127e+02**
Rosenbrock	30	5,12e+04	5,11e+00	9,80e+00	4,17e+01	NS	NS
	100	1,68e+05	5,06e+00	2,51e+01	6,08e+01	NS	NS
	500	8,52e+05	5,09e+00	NS	6,00e+01	6,60e+01	8,18e+01
	1000	1,69e+06	5,01e+00	NS	6,68e+01	7,35e+01	**9,38e+01**

9. CONCLUSÕES

As metaheurísticas bio-inspiradas estão cada vez mais sendo difundidas para aplicações em problemas complexos de alta dimensionalidade. Na área de Inteligência de Enxames, a técnica de busca por cardumes (FSS, Fish School Search) vêm sendo utilizada em problemas de otimização, apresentando grande capacidade de exploração de espaços de buscas, devido à capacidade de auto-adaptação na granularidade da busca empregada. Esta proposta mostra uma boa escalabilidade com o número de dimensões nas funções de teste utilizadas.

Com a necessidade de se obter um alto desempenho, durante o processamento de aplicações de alta complexidade, a proposta apresentada neste artigo visa utilizar computação paralela distribuída, que vem apresentando bons resultados e se tornando cada vez mais popular nos ambientes de pesquisas científicas. Isto ocorre devido ao uso de dispositivos gráficos de alta capacidade de processamento com preços cada vez mais acessíveis e também com a utilização de ambientes de desenvolvimento mais amigáveis como CUDA da NVIDIA, que reduz a complexidade da programação de aplicações de propósito geral com processamento paralelo. Entretanto, esta adaptação para plataformas paralelas deve ser realizada com cuidado, respeitando as restrições relativas a utilização de memória, transferência de dados, compartilhamento de informação e dependência de dados. Além disso, a arquitetura CUDA possui diversas técnicas de otimização de processamento, que podem ser programadas para aumentar o desempenho das aplicações, realizando inclusive processamento em mais de um dispositivo gráfico em paralelo, aumentando a capacidade máxima de processamento de dados.

Neste trabalho foi proposta a primeira adaptação do algoritmo FSS, de forma eficiente para processamento paralelo em unidades de processamento gráfico, utilizando a arquitetura CUDA. Esta versão do algoritmo foi nomeada de pFSS. Inicialmente, foi feito um estudo sobre o impacto das barreiras de sincronização na qualidade dos resultados e no desempenho durante o processamento em modo síncrono e assíncrono. Essa análise mostrou um *speedup* de **6 vezes** do modo totalmente assíncrono em comparação com a versão sequencial do FSS.

Também foram analisadas as técnicas de otimização de processamento disponíveis em CUDA, e uma nova versão foi proposta para processamento em paralelo. Esta nova versão possui melhorias no processamento devido a: modificação no acesso a memória global, utilização de memória compartilhada, transferência de dados em modo assíncrono, utilização de recursos para evitar divergências de *threads*. Os resultados obtidos, em termos de desempenho, com essas modificações atingiram um *speedup* de **84 vezes** com uma única placa de processamento gráfico para a função Rastrigin em 1000 dimensões.

Em outra análise proposta, foram realizados testes com o pFSS utilizando mais de um dispositivo de processamento gráfico em paralelo. Para isso, foram criada uma versão do pFSS com processamento em MultiGPUs, através de comunicação P2P assíncrona. Os experimentos realizados em 2 e 4 placas gráficas modelo NVIDIA Tesla C2070, num *Personal Super Computer*, permitiram aumentar o número de dimensões e, consequentemente, a capacidade de processamento em paralelo. Com esta versão, foi atingido um *speedup* de **127 vezes** e **93 vezes** em relação a versão original com processamento sequencial para as funções Rastrigin e Rosenbrock, respectivamente.

Como sugestões para trabalhos futuros decorrentes desta pesquisa estão: incorporar o processamento utilizando múltiplas *threads* em CPU, de forma a realizar o gerenciamento dos vários dispositivos de processamento gráfico de forma mais independente; implementar as técnicas de paralelização em outros algoritmos da área de Inteligência de Enxames; avaliar o desempenho das propostas através de outras métricas e análises estatísticas; avaliar o desempenho do pFSS e suas variações em outras versões do CUDA; e criar uma versão do pFSS utilizando OpenCL, com o objetivo de comparar com a versão CUDA.

REFERÊNCIAS

[1] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, first edition, 2010.

[2] N. Zhang, Y. shan Chen and J. li Wang. "Image parallel processing based on GPU". In *Advanced Computer Control (ICACC), 2010 2nd International Conference on*, volume 3, pp. 367 –370, 27-29 2010.

[3] N. Whitehead and A. Fit-florea. "Precision & Performance : Floating Point and IEEE 754 Compliance for NVIDIA GPUs". *NVIDIA white paper*, vol. 21, no. 10, pp. 767–75, 2011.

[4] C. Yang, Q. Wu, J. Chen and Z. Ge. "GPU Acceleration of High-Speed Collision Molecular Dynamics Simulation". In *Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on*, volume 2, pp. 254 –259, 11-14 2009.

[5] G. Moore. "Progress In Digital Integrated Electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]". *Solid-State Circuits Newsletter, IEEE*, vol. 20, no. 3, pp. 36 –37, sept. 2006.

[6] R. Farber. *CUDA Application Design and Development*. 2011.

[7] NVIDIA. *NVIDIA CUDA Programming Guide 4.0*. 2011.

[8] C. NVIDIA. "Best Practices Guide". *URL http://developer. download. nvidia. com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Best_PracticesGuide_2*, vol. 3, 2010.

[9] NVIDIA. *TUNING CUDA APPLICATIONS FOR FERMI*. 2011.

[10] A. Heinecke, M. Klemm and H.-J. Bungartz. "From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture". *Computing in Science and Engineering*, vol. 14, pp. 78–83, 2012.

[11] NVIDIA. "NVIDIA CUDA Driver API". *URL http://docs.nvidia.com/cuda/cuda-driver-api/index.html*, 2012.

[12] J. Sanders and E. Kandrot. *CUDA By Example: an introduction to general-purpose GPU programming*. 2010.

[13] G. S. Murthy, M. Ravishankar, M. M. Baskaran and P. Sadayappan. "Optimal loop unrolling for GPGPU programs". *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS*, vol. 46, no. 1, pp. 1–11, 2010.

[14] C. J. A. Bastos-Filho, F. B. Lima Neto, A. J. C. C. Lins, A. I. S. Nascimento and M. P. Lima. "A novel search algorithm based on fish school behavior". In *Systems, Man and Cybernetics, 2008. SMC 2008. IEEE International Conference on*, pp. 2646 –2651, 2008.

[15] C. J. A. Bastos-Filho, F. B. Lima-Neto, M. F. C. Sousa, M. R. Pontes and S. S. Madeiro. "On the influence of the swimming operators in the Fish School Search algorithm". In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pp. 5012 –5017, 2009.

[16] Y. Zhou and Y. Tan. "GPU-based parallel particle swarm optimization". In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pp. 1493 –1500, may 2009.

[17] C. J. A. Bastos-Filho, M. A. C. Oliveira Junior, D. N. O. Nascimento and A. D. Ramos. "Impact of the Random Number Generator Quality on Particle Swarm Optimization Algorithm Running on Graphic Processor Units". *Hybrid Intelligent Systems, 2010. HIS '10. Tenth International Conference on*, pp. 85–90, 2010.

[18] G. Marsaglia. "Xorshift RNGs". *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 7 2003.

[19] L. Mussi, F. Daolio and S. Cagnoni. "Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture". *Inf. Sci.*, vol. 181, no. 20, pp. 4642–4657, October 2011.

[20] W. Zhu and J. Curry. "Particle Swarm with graphics hardware acceleration and local pattern search on bound constrained problems". *Swarm Intelligence Symposium, 2009. SIS '09. IEEE*, pp. 1 –8, 30 2009-april 2 2009.

[21] P. Pospíchal, J. Jaroš and J. Schwarz. "Parallel Genetic Algorithm on the CUDA Architecture". In *Applications of Evolutionary Computation*, LNCS 6024, pp. 442–451. Springer Verlag, 2010.

[22] C. NVIDIA. "CUDA Architecture Overview". *URL http://developer.nvidia.com/category/zone/cuda-zone*, vol. 1, 2009.