Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 7109-7121 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

PARALELIZAÇÃO DO ALGORITMO HARMONY SEARCH UTILIZANDO UNIDADE DE PROCESSAMENTO GRÁFICO

Marlon H. Scalabrin[a], **Rafael S. Parpinelli**[a,b] **e Heitor S. Lopes**[a]

[a]*Laboratório de Bioinformática, Universidade Tecnológica Federal do Paraná, Av. 7 de setembro 3165, 80230-901, Curitiba, Brasil , E-mails: marlonscalabrin@yahoo.com.br, hslopes@utfpr.edu.br*

[b]*Grupo de Computação Cognitiva Aplicada, Universidade do Estado de Santa Catarina, Campus Universitário Prof. Avelino Marcante s/n, 89223-100, Joinville, Brasil , E-mail: parpinelli@joinville.udesc.br*

Palavras Chave: Algoritmo Harmony Search, Unidades de Processamento Gráfico (GPU), CUDA.

Resumo. O algoritmo *Harmony Search* (HS) é uma metaheurística baseada em conceitos musicais, sendo inspirada na observação de que o objetivo de um músico é a busca por uma harmonia perfeita. Este trabalho propõe uma abordagem paralela para o algoritmo HS utilizando a arquitetura de programação paralela CUDA (*Compute Unified Device Architecture*) em uma Unidade de Processamento Gráfico (*Graphic Processing Units* - GPU). O algoritmo HS foi modificado de modo a permitir que processos implicitamente paralelos inerentes ao algoritmo pudessem ser implementados na arquitetura paralela, mais especificamente, o improviso de novas harmonias e a geração dos valores iniciais na memória harmônica. Duas abordagens de paralelização foram implementadas: primeiro transferindo apenas a função de avaliação para a GPU; e segundo, transferindo todos os passos do algoritmo para a GPU. A última abordagem evita o *overhead* devido às transferências entre as memórias. Experimentos foram realizados com o algoritmo HS sendo executado tanto em GPU quanto em CPU, para a otimização de várias funções de *benchmark*. Os resultados mostram que o tempo de execução do algoritmo HS utilizando GPU é significativamente menor quando comparado com os tempos de execução em CPU, ambos com a mesma qualidade de soluções. Observou-se que a influência do número de variáveis sobre o tempo de execução é menos significativa na GPU do que na CPU. Também foi observado que, quanto maior a complexidade dos cálculos envolvidos no processamento, maior é o *speed-up* proporcionado pelo uso da GPU.

1 INTRODUÇÃO

Metaheurísticas baseadas em otimização estocástica são estratégias que buscam o melhor elemento de um conjunto de alternativas possíveis. A identificação deste elemento é realizada através da utilização de uma função objetivo específica, que apresenta ou se aproxima do valor desejado (Luke, 2009).

Diversas são as estratégias metaheurísticas que utilizam otimização estocástica. Podem ser citados como exemplo: Algoritmos Genéticos (Holland, 1975) (Goldberg, 1989), *Ant Colony Optimization* (Dorigo e Stützle, 2004), *Bees Algorithm* (Pham et al., 2006), *Particle Swarm Optimization* (Kennedy e Eberhart, 1995), entre outros.

Outra estratégia metaheurística relativamente recente é a *Harmony Search* (HS), introduzida em 2001 por Geem et al. (2001). Esta metaheurística é inspirada em conhecimentos musicais. HS é uma analogia a experimentos de músicos com o improviso de Jazz, que buscam combinações que são esteticamente agradáveis, desde um determinado ponto de vista (Geem et al., 2001; Geem, 2009). Como o *Harmony Search* (HS) é um método de otimização inspirado em música, o aprendizado ocorre através do processo de improviso e memorização. Ele é amplamente empregado em processos de otimização em diferentes áreas, tendo obtido soluções competitivas com outros métodos semelhantes (Geem, 2010).

Embora HS seja bastante eficaz, ele precisa de um tempo relativamente grande para encontrar soluções para problemas com um número elevado de variáveis, pois seus passos e avaliações são realizados diversas vezes. Pela natureza implicitamente paralela da execução de seus passos, a utilização de GPU pode reduzir consideravelmente os requisitos de tempo.

Neste trabalho são apresentadas duas novas implementações do algoritmo HS com a arquitetura CUDA, utilizando o poder de processamento massivamente paralelo da GPU. A primeira implementação executa apenas a função de avaliação na GPU, enquanto na segunda, todos os passos do algoritmo são executados na GPU. Para fins de comparação, uma outra versão sequencial roda exclusivamente na CPU de um computador *desktop*.

As placas gráficas com *chips* GPU estão se tornando cada vez mais presentes em computadores pessoais. Num futuro próximo esta tecnologia pode permitir que um número maior de pessoas seja capaz de resolver problemas reais de grande porte utilizando algoritmos massivamente paralelos.

Na seção 2 é apresentado o algoritmo *Harmony Search*. Na Seção 3 é explanada a tecnologia CUDA empregada para o processamento massivamente paralelo na GPU. A implementação do HS em GPU é apresentada na Seção 4. Os experimentos realizados e os resultados experimentais são relatados na Seção 6. Finalmente, as conclusões do trabalho e direcionamentos futuros são apresentados na Seção 7.

2 *HARMONY SEARCH*

O algoritmo *Harmony Search* (HS) inicia com uma memória harmônica de tamanho HMS, cada posição da memória é ocupada por uma harmonia de tamanho N. A cada passo de improviso, uma nova harmonia é gerada a partir das harmonias presentes na memória harmônica. Se a nova harmonia gerada for melhor que a pior harmonia da memória harmônica, esta é substituída pela nova.

Os passos de improviso e atualização da memória harmônica são repetidos até que o número máximo de improvisos (MI) seja alcançado. O pseudo-código do algoritmo HS é apresentado na Tabela 1 (Geem et al., 2001).

Como apresentado por Geem et al. (2001) e Mahdavi et al. (2007), o algoritmo pode ser

Tabela 1: Pseudo-código do algoritmo Harmony Search.

Parâmetros: $HMS,\ HMCR,\ PAR,\ MI,\ FW$

Início
$Funcao\ Objetivo\ \ f(\vec{x}),\ \ x = [x_1, x_2, ..., x_d]^T$
$Inicializa\ a\ memoria\ harmonica\ \vec{x}_i\ \ i = 1, 2, ..., HMS$
$Avalia\ cada\ harmonia\ na\ memoria\ harmonica:\ \ f(\vec{x}_i)$
$Ciclo = 1$
Enquanto $Ciclo\ <\ MI$
Para $cada\ variavel\ j = 1, 2, ..., N$
Se $aleatorio \leq HMCR\ \{Consideracao\ da\ Memoria\}$
$x_j^{'} = x_j^i,\ com\ i\ \in\ [1, HMS]\ escolhido\ aleatoriamente$
Se $aleatorio \leq PAR\ \{Consideracao\ de\ Ajuste\}$
$x_j^{'} = x_j^{'} \pm r \times FW,\ com\ r\ aleatorio$
fim se
Senão $Selecao\ Aleatoria$
$Gera\ x_j^{'}\ aleatoriamente$
fim Para
$Avalia\ a\ nova\ harmonia\ gerada:\ \ f(\vec{x}^{'})$
Se $f(\vec{x}^{'})\ eh\ melhor\ que\ a\ pior\ harmonia\ na\ HMS$
$Atualiza\ a\ memoria\ harmonica$
$Ciclo = Ciclo + 1$
fim Enquanto
$Resultados\ e\ visualizacoes$
fim

descrito por cinco passos principais, detalhados a seguir. Outras informações sobre HS e aplicações podem ser encontradas em seu repositório (*HS Repository: http://www.hydroteq.com*).

1. **Inicialização do Problema e Parâmetros do Algoritmo**: No primeiro passo, como em todo problema de otimização, o problema é definido como uma função objetivo a ser otimizada, a qual pode ou não possuir um conjunto de restrições. Originalmente *Harmony Search* foi desenvolvido para a resolução de problemas de minimização (Geem et al., 2001).

   Neste passo também são definidos os parâmetros do algoritmo. Os quatro principais parâmetros são o tamanho da memória harmônica (*Harmony Memory Size* – HMS), a taxa de escolha de um valor da memória (*Harmony Memory Considering Rate* – $HMCR$), a taxa de ajustes dos valores (*Pitch Adjusting Rate* – PAR) e número máximo de improvisos (*Maximum Improvisation* – MI).

2. **Inicialização da Memória Harmônica**: No segundo passo é inicializada a Memória Harmônica (HM) com um número de harmonias geradas aleatoriamente. A Memória Harmônica é o vetor no qual são armazenadas as melhores harmonias encontradas durante uma determinada execução. Cada harmonia é um vetor que representa uma possível solução para o problema.

3. **Improviso de uma nova Harmonia**: No terceiro passo é improvisada uma nova harmonia baseada nas harmonias existentes na HM, sendo que a nova harmonia é uma combinação de várias outras harmonias. Para cada variável da nova harmonia seleciona-se arbitrariamente uma harmonia da HM, verificando a probabilidade deste valor ser ou não utilizado ($HMCR$). Se for utilizado um valor de outra harmonia, o valor desta variável pode sofrer pequenos ajustes (*Fret Width* – FW) segundo uma probabilidade (PAR). Se não for utilizado o valor de outra harmonia, um valor aleatório dentro do intervalo de valores permitidos é atribuído. Sendo assim, os parâmetros $HMCR$ e PAR do algoritmo HS são responsáveis por estabelecer um balanço entre a busca global e a busca local no espaço de busca.

4. **Atualização da Memória Harmônica**: No quarto passo, a cada nova harmonia improvisada é verificado se ela é melhor do que a pior harmonia da HM. Caso confirmada esta condição, a nova harmonia substitui a pior harmonia da HM.

5. **Verificação do critério de parada**: No quinto passo, ao término de cada iteração é verificado se a melhor harmonia satisfaz o critério de parada, normalmente número máximo de improvisos MI. Em caso positivo, a execução é terminada. Caso contrário, retorna para o segundo passo enquanto não atingir o critério de parada.

3 COMPUTAÇÃO COM GPU

Nas últimas três décadas os processadores de computadores *desktop* (*Central Processing Unitiy* – CPU) têm melhorado significativamente seu desempenho. De acordo com a Lei de Moore, o seu desempenho tem dobrado a cada 18 meses. Por outro lado, o desempenho das GPUs está aumentando em uma taxa extraordinária, muito mais rápido do que o previsto por esta lei (Mohanty, 2009). Consequentemente, as GPUs tendem a ser atualmente o *hardware* mais poderoso para processamento de alto-desempenho. Assim, muitos pesquisadores e desenvolvedores têm se interessado em aproveitar o seu poder de processamento paralelo para computação científica e de propósito geral (Owens et al., 2005).

Com o surgimento do NVIDIA Tesla (©Nvidia Corporation, USA) tornou-se possível a programação direta da GPU, como processadores massivamente paralelos, ao invés de simplesmente como API de aceleradores gráficos. A partir disto, a NVIDIA desenvolveu o modelo de programação CUDA (*Compute Unified Device Architecture*) para permitir a programadores escrever programas paralelos escaláveis usando uma extensão simples da linguagem C (Garland et al., 2008).

CUDA é a arquitetura de computação massivamente paralela de propósito geral que utiliza uma Unidade de Processamento Gráfico (*Graphic Processing Units* - GPU), proporcionando um significativo aumento da velocidade de processamento (*speed-up*). Desta forma, é possível utilizar a GPU para resolver problemas computacionalmente complexos em velocidade maior do que em uma CPU *multicore* (NVIDIA CUDA Team, 2010).

A plataforma de desenvolvimento CUDA foi formalmente introduzida em fevereiro de 2007 com a disponibilização do SDK (*Software Developers Kit*) com compiladores para os sistemas operacionais Windows e Linux (NVIDIA Corporation, 2007). Atualmente a tecnologia se encontra na versão 3.0.

A arquitetura CUDA é composta, principalmente, pelo conjunto de instruções CUDA ISA (*Instruction Set Architeture*) e pelos mecanismos de computação paralela da GPU, sendo que a codificação CUDA normalmente é feita em linguagem C.

A execução de uma tarefa na GPU é realizada através de chamadas de funções denominadas *kernels*. Estas chamadas inicializam várias *threads* idênticas na GPU que, por sua vez, são executadas paralelamente sobre um grande conjunto de dados. Cada *thread* é um mecanismo responsável pelo processamento de uma pequena porção de dados (NVIDIA CUDA Team, 2009).

A inicialização de um *kernel* é configurável conforme a necessidade, podendo ser alocado um grande número de *threads* para a sua execução. A distribuição das *threads* na GPU também é definida na inicialização do *kernel*, sendo que as *threads* estão distribuídas em *blocks* que, por sua vez, são englobados em um *grid* computacional.

A distribuição dos *blocks* dentro do *grid* pode ser realizada em até duas dimensões e a distribuição dos *kernels* dentro de cada *block* pode ser realizada em até três dimensões. Na Figura 1 é mostrado um exemplo da distribuição espacial de um *grid* detalhando seus *blocks* e *threads* de forma bidimensional e o acesso do *host* (CPU) ao *device* (GPU).

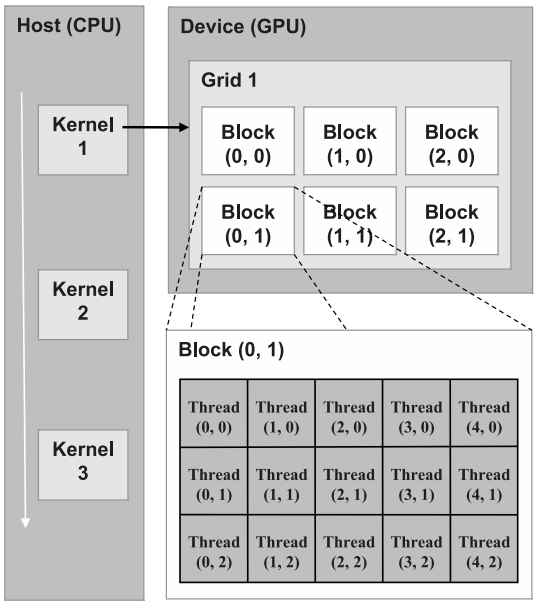


Figura 1: Distribuição de *threads* e *blocks* dentro de um *grid*. Baseado em Kirk e Hwu (2009a).

Cada *thread*, assim como cada *block*, possui um índice tridimensional próprio (`threadIdx` e `blockIdx` respectivamente) que permite acessar ou computar os índices de memória no qual estão disponíveis os dados necessários para a sua execução. Logo, o número total de *threads* está diretamente ligado à dimensão dos dados a serem processados.

As *threads* de um mesmo *block* podem se comunicar utilizando espaços de memória compartilhados e podem ser sincronizadas. Com a sincronização, todas as *threads* de um *block* executarão até um mesmo ponto do fluxo de execução. Só após todas atingirem o ponto de sincronismo é que poderão prosseguir a execução.

As GPUs atuais são construídas com muitos multiprocessadores de vários *cores*, denominados *Streaming Multiprocessors* (SM), sendo que existem limitações quanto ao número de *threads* em cada bloco. Atualmente o número de *threads* por bloco pode chegar em até 512 (NVIDIA CUDA Team, 2010), desde que não ultrapasse a limitação de memória e recursos compartilhados dentro de cada SM. Há, também, uma limitação física do *hardware*, a qual limita o número de *threads* por SM em 768. Como cada *block* é alocado por inteiro em um SM, o seu tamanho deve ser tal que minimize o desperdício computacional em cada SM (Kirk e Hwu, 2008).

O multiprocessador cria, gerencia, agenda e executa as *threads* em grupos de 32 *threads* paralelas chamados *warps*. Embora todas as *threads* que compõem um *warp* comecem juntas no mesmo endereço do programa, elas executam de forma independente. A forma como cada *block* é dividido em *warps* é sempre a mesma. Os *warps* contém *threads* consecutivas e, desta forma, o primeiro *warp* contém sempre a *thread* identificada como 0 (NVIDIA CUDA Team, 2010).

A execução de um *warp* é mais eficiente quando as 32 *threads* executam simultaneamente a mesma instrução, possuindo o mesmo caminho de execução. Se o caminho das *threads* diverge, isto é, possuem fluxos de execuções diferentes, que podem ser originados por desvios de estruturas condicionais, o *warp* os executa separadamente de forma sequencial (NVIDIA CUDA Team, 2010).

3.1 Memória

Na arquitetura CUDA existem seis tipos diferentes de memória, representados na Figura 2, cada qual com suas características próprias de velocidade, permissões, visibilidade e tempo de vida (NVIDIA CUDA Team, 2008, 2010; Kirk e Hwu, 2008):

1. **Registradores**: são posições de memória de acesso rápido nas quais são armazenados valores de tipos primitivos. Os registradores possuem permissões de leitura e escrita, visibilidade restrita a cada *thread* e seu tempo de vida é equivalente ao tempo de vida de sua *thread*.

2. **Memória Local**: é o espaço de memória no qual são armazenadas as variáveis endereçáveis, como ponteiros para valores na memória global e vetores, além dos tipos primitivos, quando não há espaço suficiente para eles nos registradores. Esta memória possui as mesmas restrições e visibilidade dos Registradores.

3. **Memória Compartilhada**: possui permissões de leitura e escrita, visibilidade restrita ao bloco e acesso rápido. Ela permite a troca de informações entre as *threads* em execução dentro de um mesmo bloco. Para ter acesso a ela utiliza-se o qualificador de variável `__shared__`, informando que todos os valores armazenados naquela variável serão alocadas na memória compartilhada. Seu tempo de vida é equivalente ao tempo de vida de seu bloco.

4. **Memória Global**: possui permissões de leitura e escrita e, como o próprio nome diz, visibilidade global. Esta área de memória possui elevado tempo de acesso e os valores armazenados nela são independentes do tempo de execução das *kernels*, podendo-se armazenar nela informações a serem compartilhadas por *kernels* distintos ou execuções distintas do mesmo *kernel*, além do próprio *host*. Este espaço de memória é gerenciável e pode ser alocado ou liberado a qualquer momento. Para ter acesso a ela utiliza-se o qualificador de variável `__global__`.

5. **Memória de Constantes**: tem permissão de leitura e possui visibilidade global. Como a memória global, possui elevado tempo de acesso. Seus valores, após carregados, não podem sofrer alteração. Para ter acesso a ela utiliza-se o qualificador de variável `__constant__`. Atualmente, o tamanho total das variáveis de constante está limitado a 64K Bytes;

6. **Memória de Textura**: semelhante à memória de constantes, exceto por possuir um mecanismo de *cache* automático.

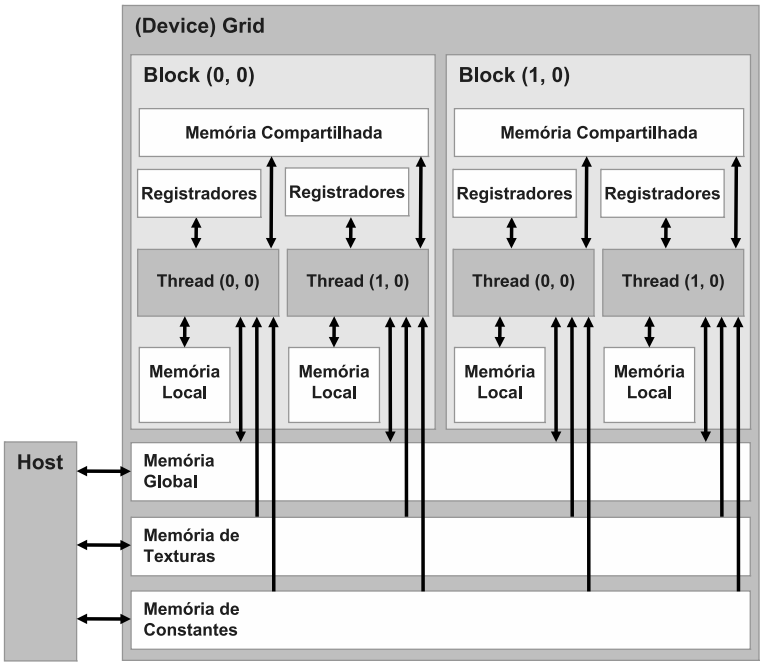


Figura 2: Acesso e visibilidades das memórias. Baseado em Kirk e Hwu (2009b).

As diferentes possibilidades de utilização dos recursos (*blocks*, *threads* e memórias) podem causar grande variação no desempenho da execução do *kernel*. Assim, é usual a cópia dos trechos mais acessados da memória global para as memórias compartilhadas antes do início do processamento (Kirk e Hwu, 2008), além da adequação dos tamanhos dos blocos e *threads* (Pamplona, 2008).

4 *HARMONY SEARCH* UTILIZANDO GPU

Neste trabalho são propostos dois modelos de implementação paralela do algoritmo HS em GPU, denominados HS-GPU-Fit e HS-GPU.

Na primeira implementação (HS-GPU-Fit) apenas a função de avaliação (função de *fitness*) é executada na GPU. Todos os demais passos do algoritmo HS são executados na CPU, assim como visto na Tabela 1. Para a execução da função de *fitness*, os dados da nova harmonia improvisada são transferidos para a GPU, para que assim a função de avaliação possa ser executada. Ao término da execução da função de avaliação o valor calculado é transferido novamente para a CPU e o processo continua.

Na segunda implementação (HS-GPU) todos os passos do HS são executados na GPU, de forma a minimizar a transferência de dados entre as memórias do computador *host* e o *device*. No passo de inicialização, após o carregamento dos parâmetros pela CPU, estes são transferidos para a memória da GPU, sendo também alocadas as posições na memória global da GPU. São copiados também os valores da CPU para os vetores que serão utilizados durante a execução do HS.

A inicialização da memória harmônica é realizada com uma chamada de *kernel* para cada posição da memória harmônica, produzindo uma nova harmonia para cada posição. Cada variável da nova harmonia é independente das demais. Desta forma, a obtenção do seu valor é realizado por uma *thread* em um bloco de N *threads*, sendo N o número de variáveis da harmonia. Na GPU a memória harmônica é alocada na memória global e é representada linearmente, de forma que a cada N posições encontra-se uma harmonia diferente.

Estando a memória harmônica carregada com suas harmonias iniciais, começa o processo iterativo de otimização do HS. A cada ciclo da iteração duas chamadas de *kernel* são realizadas. A primeira delas realiza o improviso de uma nova harmonia e a segunda chamada, a atualização da memória harmônica.

No improviso, o processo de seleção de cada variável da nova harmonia é realizado independentemente. Da mesma forma que na inicialização de cada posição da memória harmônica, cada *thread* fica responsável por uma variável da nova harmonia.

A atualização da memória harmônica é realizada substituindo-se a pior harmonia pela nova harmonia improvisada, caso esta seja melhor do que a anterior. Para encontrar a pior harmonia é realizada uma busca sequencial na memória harmônica e posteriormente o processo de atualização é realizado, tratando independentemente cada variável da posição da memória harmônica a ser substituída.

Concluído o processo de otimização, os dados referentes à melhor harmonia encontrada são transferidos do *device* para o *host* para que possam ser utilizados e visualizados.

5 EXPERIMENTOS REALIZADOS

A plataforma experimental utilizada neste artigo para o *Harmony Search* é baseada em computador pessoal com CPU Intel Core2Quad 2.8GHz e placa de vídeo NVIDIA GeForce GTX 285, rodando o sistema operacional Linux.

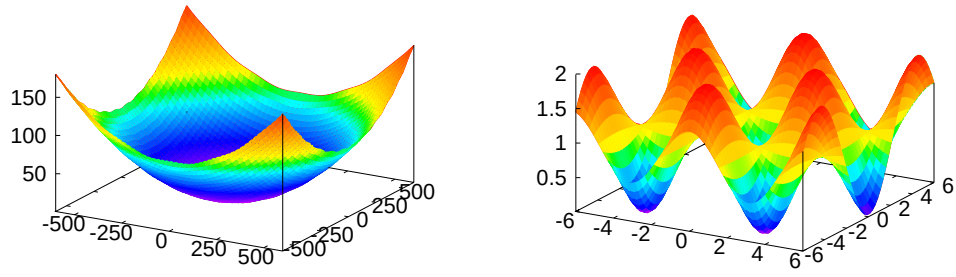
O foco dos experimentos foi realizar medições de tempo e realizar um comparativo de desempenho para identificar em quais circunstâncias as implementações em GPU são mais vantajosas do que uma implementação sequencial rodando em CPU.

Foram realizados diversos experimentos utilizando algumas funções de *benchmark* comumente utilizadas para a análise de desempenho de metaheurísticas (Digalakis e Margaritis, 2002). Mais especificamente, foram utilizadas as funções de Griewank, Rosenbrock e Schaffer. Cada função foi executada nas três versões implementadas do *Harmony Search*: sequencial rodando na CPU (CPU-HS); o algoritmo rodando na CPU, mas a função de *fitness* executada na GPU (HS-GPU-Fit); e o algoritmo completo de HS rodando na GPU (HS-GPU). Para testar a escalabilidade das implementações, todas as funções de teste foram otimizadas com as seguintes dimensões: 100, 250, 500, 1000, 2000, 5000 e 10000.

Segundo Cho (Cho et al., 2008), a função de Griewank é uma função multimodal que tem sido amplamente utilizada para testar a convergência dos algoritmos de otimização, pois o número de mínimos cresce exponencialmente conforme aumenta o número de dimensões. O intervalo de valores utilizados é de -600 a 600, sendo que o mínimo global está localizado em 0. A Figura 3 ilustra esta função em duas dimensões, sendo que a primeira (3a) foi plotada no intervalo de -600 a 600, mostrando todo o espaço de busca e a segunda (3b) no intervalo de -6 a 6, mostrando em detalhes as oscilações de menor amplitude. A função de Griewank é definida pela Equação 1:

$$f(\vec{x}) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos \frac{x_i}{\sqrt{i}} + 1 \qquad (1)$$

A função de Rosenbrock, ilustrado na Figura 4 é uma função caracterizada por um vale profundo que lembra uma parábola, levando ao mínimo global. O intervalo de busca para este problema é definido entre -5 e 5. Apesar de sua aparência simples, esta é uma função de difícil otimização, particularmente com muitas dimensões, pois no extenso platô é difícil obter uma indicação precisa da direção do ótimo global (Digalakis e Margaritis, 2002). A função de

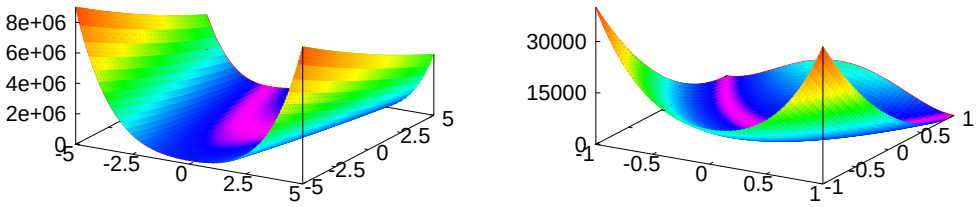


(a) -600 a 600 (b) -6 a 6

Figura 3: Função de Griewank.

Rosenbrock multidimensional é definida pela Equação 2:

$$f(\vec{x}) = \sum_{i=1}^{n-1}[(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2] \tag{2}$$



(a) -5,12 a 5,12 (b) -1 a 1

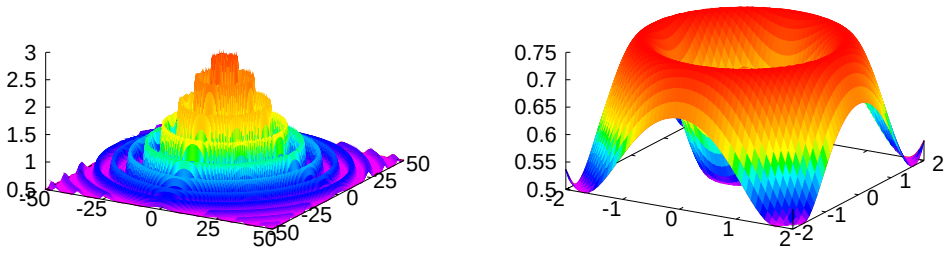
Figura 4: Função de Rosenbrock.

A função de Schaffer (Figura 5) é também uma função fortemente multimodal com muitos ótimos locais distribuídos e concêntricos ao ótimo global. Segundo de Castro e Timmis (2002) o ótimo global é difícil de encontrar porque o valor dos ótimos locais diferem minimamente do ótimo global (em torno de 0,001). A função de Schaffer é definida pela Equação 3:

$$f(\vec{x}) = \sum_{i=1}^{n-1}[0.5 + \frac{\sin(\sqrt{x_{i+1}^2 + x_i^2}) - 0.5}{(0.001 * (x_{i+1}^2 + x_i^2)^2 + 1)}] \tag{3}$$

6 RESULTADOS E DISCUSSÃO

Para a função de Griewank, ambas as implementações em GPU obtiveram melhor desempenho em tempo de processamento do que a implementação em CPU. Isto foi válido para os experimentos com mais de 500 dimensões. Sendo que, a HS-GPU foi significativamente mais eficiente para os experimentos com com mais de 2000 dimensões, como mostrado no gráfico



(a) -50 a 50 (b) -2 a 2

Figura 5: Função de Schaffer.

da Figura 6. Nesta figura, o eixo x mostra o número de dimensões da função sendo otimizada (uma relação direta da complexidade do problema) e eixo y representa o tempo de processamento normalizado considerando os três modelos implementados. Quanto maior o número de dimensões, maior o aumento de velocidade proporcionado pelo uso da GPU, chegando a um *speed-up* de 7x para a HS-GPU e 2,45x para a HS-GPU-Fit, quando comparados com a versão sequencial HS-CPU.

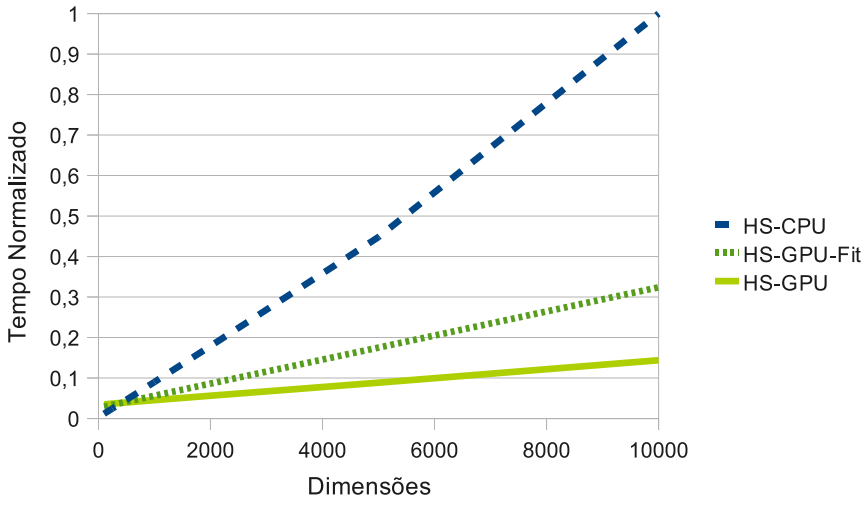


Figura 6: Comparativo de desempenho das implementações sobre a função de Griewank.

Para a função de Rosenbrock, a HS-GPU obteve melhores resultados em tempo de processamento para dimensões superiores a 2000 dimensões, enquanto que a implementação HS-GPU-Fit obteve melhores tempos para dimensões acima de 5000 dimensões, como mostrado na Figura 7. O *speed-up* proporcionado pelo uso da GPU chega a 3,29x para a implementação HS-GPU e 1,3x para a implementação HS-GPU-Fit.

Para a função de Schaffer, tanto a implementação HS-GPU quanto a HS-GPU-Fit tiveram tempos de processamento menores do que a HS-CPU, para dimensões maiores que 1000. A partir deste mesmo número de dimensões a implementação HS-GPU apresenta melhores tempos que a HS-GPU-Fit, como mostrado na Figura 8. Para a implementação HS-GPU foi obtido um *speed-up* de 3,95x e 1,55x para a HS-GPU-Fit.

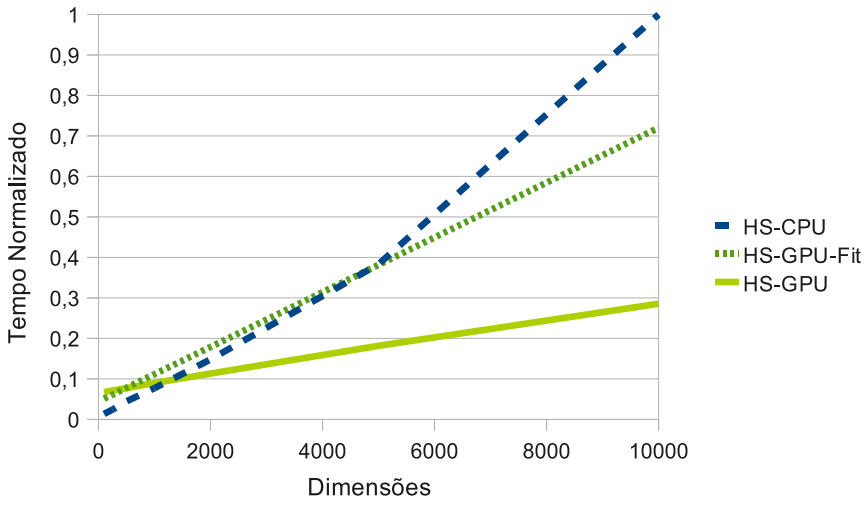


Figura 7: Comparativo de desempenho das implementações sobre a função de Rosenbrock.

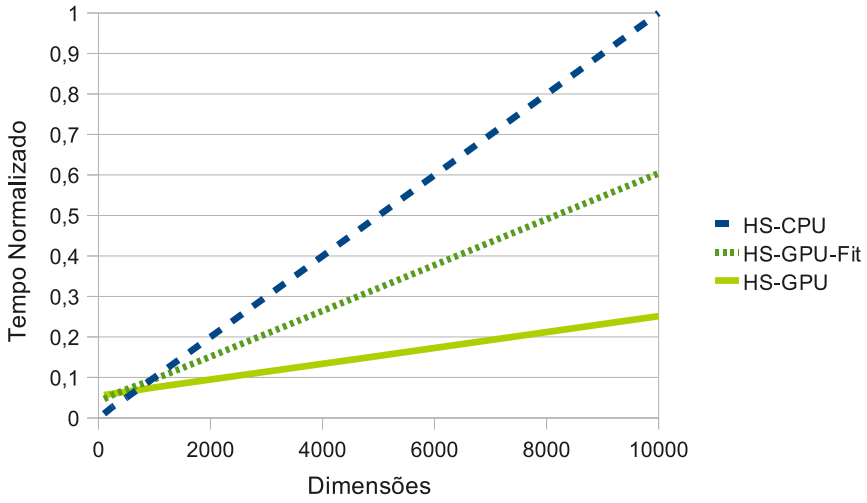


Figura 8: Comparativo de desempenho das implementações sobre a função de Schaffer.

Analisando o grau de complexidade das funções executadas, a função de Griewank é a função mais custosa computacionalmente, pois apresenta várias funções transcendentes, como cosseno e raiz quadrada, além de muitas multiplicações e divisões. Nas implementações, o quadrado foi codificado através de multiplicação. A função de Schaffer é a segunda função mais custosa, utilizando seno e raiz quadrada. A função de Rosenbrock é a menos custosa por utilizar apenas somas e multiplicações. Partindo-se deste critério de complexidade, as Figuras 6, 8 e 7 e os *speed-ups* correspondentes proporcionado pela GPU, sugerem que quanto maior a complexidade computacional da função, maior é o ganho obtido ao se utilizar a GPU em relação à CPU.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este artigo apresentou duas implementações da metaheurística *Harmony Search* baseadas em GPU: GPU-HS-Fit e GPU-HS. A otimalidade dos resultados obtidos foram similares em todos os algoritmos experimentados, tanto utilizando os recursos da GPU quanto utilizando somente

os recursos da CPU. No entanto, verificou-se um ganho expressivo de tempo de processamento com o uso da GPU.

A execução dos passos do *Harmony Search* em paralelo na GPU mostrou-se muito eficiente para os problemas tratados, no que diz respeito à complexidade das funções de avaliação e na elevada dimensionalidade. Com a implementação apresentada, um aumento de velocidade superior a 7x pôde ser obtido, quando comparado com a CPU, para os problemas mais complexos. Isto sugere fortemente a sua aplicabilidade em computação científica e a problemas reais de engenharia.

As principais dificuldades encontradas foram relativas aos conceitos básicos da estrutura CUDA, como acesso a memória, *warps* e limitações. Durante o desenvolvimento percebeu-se que a organização dos blocos e *threads*, bem como a estratégia e o número de acessos à memória influenciam fortemente no desempenho de uma aplicação em GPU. Assim, é possível que a versão implementada possa ser melhorada. Embora a implementação em GPU não seja tão simples quanto em CPU, ainda é uma alternativa muito mais acessível do que outras abordagens para aceleração de processamento, como, por exemplo, implementação em FPGA (Lopes et al., 2007).

A próxima etapa da pesquisa será aperfeiçoar a implementação em GPU, procurando obter ganhos maiores de velocidade tanto para problemas de alta dimensionalidade quanto para problemas com um número reduzido de dimensões. Em termos de aplicações, o próximo passo é aplicar a problemas reais de otimização em engenharia, além de problemas multiobjetivos e problemas de otimização discreta.

Este trabalho mostrou a viabilidade da utilização da GPU para a redução do tempo de processamento do algoritmo *Harmony Search* paralelizado. De maneira geral, esta estratégia parece ser promissora também para outras metaheurísticas populacionais que trabalham com a avaliação simultânea de múltiplas soluções.

REFERÊNCIAS

Cho H., Olivera F., e Guikema S. A derivation of the number of minima of the Griewank function. *Applied Mathematics and Computation*, 204(2):694–701, 2008.

de Castro L.N. e Timmis J. An artificial immune network for multimodal optimisation. In *Proc. of 2002 IEEE World Congress on Computational Intelligence*, páginas 699–704. IEEE Press, Honolulu, Hawaii, USA, 2002.

Digalakis J.G. e Margaritis K.G. An experimental study of benchmarking functions for evolutionary algorithms. *International Journal of Computer Mathematics*, 79(4):403–416, 2002.

Dorigo M. e Stützle T. *Ant Colony Optimization*. MIT Press, Cambridge, USA, 2004.

Garland M., Grand S.L., Nickolls J., Anderson J., Hardwick J., Morton S., Phillips E., Zhang Y., e Volkov V. Parallel computing experiences with CUDA. *IEEE Micro*, 28:13–27, 2008.

Geem Z. Global optimization using harmony search: Theoretical foundations and applications. *Foundations of Computational Intelligence*, 3:57–73, 2009.

Geem Z.W. State-of-the-art in the structure of harmony search algorithm. In Z.W. Geem, editor, *Recent Advances In Harmony Search Algorithm*, volume 270 de *Studies in Computational Intelligence*, páginas 1–10. Springer, 2010.

Geem Z.W., Kim J.H., e Loganathan G.V. A new heuristic optimization algorithm: Harmony search. *Simulation*, 76(2):60–68, 2001.

Goldberg D.E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, USA, 1989.

Holland J.H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, USA, 1975.

Kennedy J. e Eberhart R. Particle swarm optimization. In *Proc. of the IEEE Int. Conf. on Neural Networks*, páginas 1942–1948. IEEE Press, Piscataway, USA, 1995.

Kirk D. e Hwu W. Applied parallel programming, chapter 4 - CUDA memories, 2008. Draft.

Kirk D. e Hwu W. Lectures 8: Threading and memory hardware in G80, 2009a. Draft.

Kirk D. e Hwu W. Lectures 9: Memory hardware in G80, 2009b. Draft.

Lopes H., C.R.Erig Lima, e Murata N. A configware approach for high-speed parallel analysis of genomic data. *Journal of Circuits, Systems, and Computers*, 16(4):1–15, 2007.

Luke S. Essentials of metaheuristics. http://cs.gmu.edu/ sean/book/metaheuristics/, 2009.

Mahdavi M., Fesanghary M., e Damangir E. An improved harmony search algorithm for solving optimization problems. *Applied Mathematics and Computation*, 188(2):1567–1579, 2007.

Mohanty S.P. GPU-CPU multi-core for real-time signal processing. In *Proc. of the 27th IEEE International Conference on Consumer Electronics*, páginas 55–56. IEEE Computer Society, Los Alamitos, USA, 2009.

NVIDIA Corporation. CUDA for GPU computing. 2007.

NVIDIA CUDA Team. CUDA programming model overview. 2008.

NVIDIA CUDA Team. NVIDIA compute PTX: Parallel thread execution, ISA version 1.4. 2009.

NVIDIA CUDA Team. CUDA programming guide version 3.0. 2010.

Owens J.D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A.E., e Purcell T. A survey of general-purpose computation on graphics hardware. In *Proc. of Eurographics*, páginas 21–51. 2005.

Pamplona V. CUDA. 2008.

Pham D., Ghanbarzadeh A., Koç E., Otri S., Rahim S., e Zaidi M. *The Bees Algorithm - A Novel Tool for Complex Optimisation Problems*, páginas 454–459. Intelligent Production Machines and Systems. Elsevier Science, 2006.