CUDABC: Algoritmo de otimização baseada em colônia de abelhas paralelo sob paradigma GPGPU

Mário Tasso Ribeiro Serra Neto
Centro Universitário do Estado do Pará, Belém, Pará, Brasil. Grupo de Estudos Temático em Computação Aplicada - Laboratório de Inteligência do Centro Universitário do Estado do Pará (GET-COM - LINCOMP CESUPA)
mariotnxd@gmail.com

Daniel Leal Souza
Universidade Federal do Pará, Belém, Pará, Brasil. Instituto de Tecnologia (ITEC) – Laboratório de Computação Natural e Bioinspirada (LCNBio - UFPA)
daniel.leal.souza@gmail.com

Rodrigo Lisbôa Pereira
Centro Universitário do Estado do Pará, Belém, Pará, Brasil. Grupo de Estudos Temático em Computação Aplicada - Laboratório de Inteligência do Centro Universitário do Estado do Pará (GET-COM - LINCOMP CESUPA)
lisboa@cesupa.br

Roberto Célio Limão de Oliveira
Universidade Federal do Pará, Belém, Pará, Brasil. Instituto de Tecnologia (ITEC) – Laboratório de Computação Natural e Bioinspirada (LCNBio - UFPA)
limao@ufpa.br

RESUMO

Este artigo apresenta uma versão paralela do algoritmo Artificial Bee Colonoy (ABC) baseado na arquitetura CUDA. O foco principal desta abordagem é a execução concorrente das abelhas sob um esquema competitivo, sendo executado sobre o paradigma da Computação de Uso Geral em Unidades de Processamento Gráfico (GPGPU). A implementação proposta traz vantagens que aliam desempenho do ABC na busca global e as capacidades de computação de alto desempenho, oferecidos pela arquitetura CUDA. Para a análise de desempenho, a solução proposta foi submetida a quatro problemas de benchmark bem conhecidos (Sphere, Rastrigin, Bi-dimensional Rosenbrock, Griewank).

PALAVRAS CHAVE. CUDA, GPU, METAHEURÍSTICA.

Tópicos SS3 – Algoritmos Paralelos em CPU e GPU; MH – Metaheuristicas.

ABSTRACT

This paper presents a parallel version of Artificial Bee Colony (ABC) algorithm based on CUDA architecture. The main focus of this approach is the concurent execution of the bees under a competitive scheme and being executed over the paradigm of General Purpose Computing on Graphics Processing Units (GPGPU). The proposed implementation takes advantages from both ABC's performance in global search as well as CUDA's high performance computing capabilities. For performance analysis, the proposed solution was submitted to four well-known benchmark problems (Sphere, Rastrigin, Bi-dimensional Rosenbrock, Griewank).

KEYWORDS. CUDA, GPU, METAHEURISTIC.

Paper topics SS3 – Parallel Algorithms in CPU and GPU; MH – Metaheuristics.

1. Introdução

Nos últimos doze anos, os algoritmos de busca e otimização baseados em metaheurísticas bio-inspiradas tem experimentado um amplo reconhecimento e utilização, em parte, graças a sua facilidade de implementação e eficiência na obtenção de soluções de busca global [Parsopoulos e Vrhatis 2010]. Em um tópico mais específico, soluções baseadas no paradigma de inteligência de enxames (*Swarm Intelligence*) como o *Particle Swarm Optimization* (PSO), *Genetic Algorithms* (GA) e *Artificial Bee Colony* (ABC) vem se tornando uma alternativa robusta para busca e otimização nos mais variados campos do conhecimento, como exploração de óleo e gás, automação industrial e modelagem operacional [Parsopoulos e Vrhatis 2010, Souza et al. 2014].

Essa classe de algoritmos apresenta vantagens quando o problema em questão requer uma resposta global aproximada sem a necessidade de complexas heurísticas matemáticas ou mesmo de métodos específicos para um determinado problema (esses podem ser inclusos em metaheurísticas como mecanismos de busca local) [Haupt e Haupt 1998].

Apesar dos bons resultados obtidos e documentados na literatura [Parsopoulos e Vrhatis 2010, Engelbrecht 2007], tais algoritmos possuem limitações quanto a velocidade de execução do programa, bem como na interpretação e mimetização de um determinado processo natural em arquiteturas computacionais. O uso de metaheurísticas bio-inspiradas em ambientes de programação sequencial, apesar de bem-sucedido, no que tange o processo de busca e otimização em si [Karaboga 2005, Karaboga e Akay 2009, Karaboga et al. 2007], não permitem uma implementação que simule com maior fidelidade um processo natural (e.g. abelhas que aguardam o término da exploração de outras abelhas no algoritmo ABC, processo sequencial de cruzamento entre pares no algoritmo GA) [Tan 2016].

A implementação de metaheurísticas bio-inspiradas sob o paradigma GPGPU através da arquitetura CUDA, proporciona maior viabilidade para uma aproximação significativa dos modelos naturais os quais essas metaheurísticas são inspiradas [Tan 2016, Souza et. al 2014]. O uso de multiplas *threads* executando individualmente as tarefas das abelhas de forma concorrente, bem como a divisão das mesmas em múltiplas colmeias (ou múltiplos *blocks*) aproximam os algoritmos de seu modelo natural original, além de permitir a inserção de novas abordagens e mecanismos de busca locais com um menor impacto no tempo de execução total. Algumas versões multi-populacionais do algoritmo PSO em CUDA podem ser encontradas em [Tan 2016, Souza et al. 2014, Souza et al. 2013, Mussi et al. 2011].

Este trabalho apresenta uma versão do algoritmo ABC sob a arquitetura CUDA. O algoritmo proposto visa a diminuição do tempo de processamento para problemas de alta dimensionalidade em relação a sua versão sequencial na CPU, além de uma aproximação mais adequada ao sistema de colônias encontrada na natureza.

2. Artificial Bee Colony (ABC)

Proposto por Karaboga (2005), o ABC é classificado como uma heurística que imita a estratégia da natureza. Logo, consiste em um algoritmo bioinspirado capaz de resolver problemas diversificados [Binitha e Sathya 2012]. Assim sendo, processos biológicos podem ser como processos de otimização com limites, sendo este, o comportamento de abelhas produtoras de mel.

Karaboga (2005) aplica fontes de alimento, como soluções para problemas, abelhas empregadas e desempregadas. Este modelo segue o mínimo do processo de forragem das abelhas, o qual carrega a emergência de uma inteligência coletiva congruente ao comportamento das abelhas ao procurar por novas candidatas para serem alocadas a novas fontes de comida. Este procedimento possui base conceitual na tese de [Tereshko e Loengarov 2005], que propõem um modelo matemático para o processo de forragem das abelhas.

O ABC é divido em quatro etapas, sendo as três etapas finais inseridas em um ciclo de repetições com critério de parada: 1) Inicialização das abelhas; 2) Envio das abelhas empregadas; 3) Envio das abelhas espectadoras; 4) Envio das abelhas exploradoras.

2.1. Abelhas Empregadas

Esta etapa corresponde ao envio de abelhas empregadas para encontrar novas e melhores fontes de comida – soluções para o problema –. Para cada solução X_m em um conjunto de *SN* soluções candidatas, (1) é aplicada em uma busca local em conjunto com a seleção gulosa: *m*.

$$X_m^{(i)} = X_m^{(t)} + \varphi * \left(X_m^{(t)} - X_k^{(t)}\right) \quad (1)$$

onde *m* corresponde ao índice da solução, $X_m^{(i)}$ equivale ao parâmetro da solução *m* a ser modificado, uma fonte aleatória é selecionada no interior do conjunto *SN* e designada a $X_k^{(t)}$, por fim φ corresponde a um valor aleatório entre os limites [−α, α], normalmente assumindo o valor de 1 [Akay & Karaboga (2012].

2.2. Abelhas Espectadoras

Durante este ciclo, as abelhas no interior da colmeia recebem informação proveniente das abelhas empregadas sobre as melhores fontes de comida disponíveis. Portanto, a decisão para exploração da vizinhança é determinada pelas empregadas [Karaboga 2005] e o método de seleção canônica é semelhante a uma roleta proporcionada pela *fitness*, em que a probabilidade de uma solução ser escolhida é descrita por (2),

$$p_m = \frac{F_m}{\sum_{i=1}^{n} F_i} (2)$$

onde, F_m equivale o valor da fonte x_m obtido pela função objetivo. F_m é dado por (3),

$$F_m(x) = \begin{cases} \frac{1}{1+f(x_m)}, & se: F_m(x) \geq 0 \\ |1 + f(x_m)|, & se: F_m(x) < 0 \end{cases} \quad (3)$$

Para contemplar as abelhas, o método de seleção torneio pode ser aplicado, definindo previamente um número *n* de indivíduos afim de participar do mesmo torneio.

2.3. Abelhas Exploradoras

O último ciclo das abelhas consiste no movimento aleatório pela procura aleatória de soluções vizinhas com proposito de encontrar fontes promissoras. Se uma solução candidata x_m não desenvolver um resultado melhor durante uma serie finitas de iterações representada pela variável *limit*, o valor é modificado por (4) [Karaboga e Akay 2009].

$$x_i^j = x_{min}^j + rand[0,1]\left(x_{max}^j - x_{min}^j\right) \quad (4)$$

onde *i* representa o parâmetro a ser modificado e *j* o índice da abelha.

Se múltiplas soluções ultrapassarem o limite durante o mesmo ciclo de iterações, apenas uma obtém seus valores renovados [Karaboga 2005]. Por outro lado, caso o mesmo evento ocorra durante outro ciclo, seu valor necessita ser modificado [Karaboga e Akay 2009].

De acordo com Karaboga e Akay (2009), o ABC clássico é regulado através de três parâmetros: 1) A quantidade de fontes de comida, a qual é equivalente a quantidade de abelhas empregadas, *SN*; 2) a variável limite; 3) e o máximo de ciclos/iterações. O algoritmo repete os processos principais – empregadas, espectadoras e exploradoras – até atender ao critério de parada.

3. Computer Unified, Device Architeture (CUDA)

A arquitetura CUDA proporciona uma interface de programação onde o usuário é capaz de aplicar instruções a fim de, designar funções ou procedimentos a serem executados via GPU [Kirk e Hwu, 2010]. Tal ambiente permite um modelo de programação heterogênea entre CPU e GPU [Stringhini et al. 2012].

CUDA proporciona a possibilidade de escrever programas de propósito geral para serem executados nos processadores de fluxo contidos na GPU. Tais programas são escritos através da linguagem CUDA-C, que introduz novas nomenclaturas e recursos a linguagem de programação C, possibilitando a compatibilidade de programas heterogêneos [Schatz et al. 2007]. A implementação aqui proposta fora desenvolvida sob a versão 8.0.61 referente a arquitetura CUDA.

Está arquitetura requer do programador um conhecimento apurado sobre a organização de dados sob as estruturas de *threads*, além de outras funcionalidades que requerem do programador habilidades em paralelismo explícito [NVIDIA 2017].

3.1 Modelo Organizacional e Estrutura de Threads

Implementações sob a arquitetura CUDA podem ser abordadas em diferentes níveis organizacionais de *threads*, como apresentado por [Souza et. al. 2013]. Kirk e Hwu (2010), definem três níveis organizacionais:

- ***Threads***: Unidade básica de fluxo de instruções paralela, é responsável por uma cópia do programa para uma determinada quantidade de dados. Podem representar diferentes elementos em vetores, matrizes ou estruturas de dados sem dependência funcional.;
- ***Blocks* (Matriz de *threads*)**: Suportam uma quantidade especifica de *threads*. Arquiteturas inferiores a 2.0 abrigavam 512 *threads* por *block*, enquanto que arquiteturas mais modernas apresentam 1024 *threads* por *block*;
- ***Grid* (Matriz de *blocks*)**: Conjunto de todos os *blocks* utilizados por uma função na GPU.

A hierarquia de *threads* presentes na arquitetura CUDA estão representados pela Figura 1.

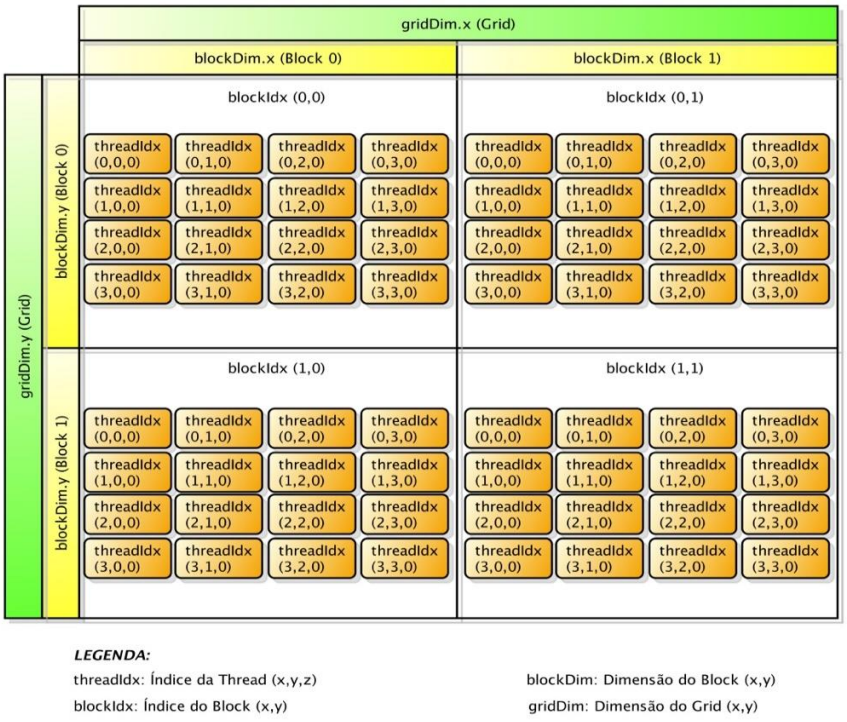

Figura 1. Hierarquia bidimensional de *threads*, *blocks* e *grid* em CUDA [Souza et. al., 2014].

4. CUDABC (CUDA + ABC)

A versão sequencial do algoritmo ABC, aplica as formulações matemáticas e etapas logicas de cada abelha de modo linear durante a execução. Cada ciclo representa um modelo síncrono, visto que, uma abelha necessita aguardar o termino das instruções de outra abelha para executar seus ciclos.

A implementação do CUDABC visa a otimização no tempo de execução, quando comparado a versão sequencial do algoritmo. A arquitetura CUDA proporciona uma abordagem assíncrona do algoritmo, que proporciona uma analogia a dinâmica da colmeia, visto que, uma abelha não necessita aguardar por outra para iniciar o seu trabalho. Devido à baixa necessidade de processamento nas etapas de inicialização e ciclo de espectadoras, estas continuam executadas via CPU enquanto que, os ciclos de empregadas e espectadoras são processados via GPU. A Figura 2 apresenta o fluxograma geral da implementação do CUDABC.

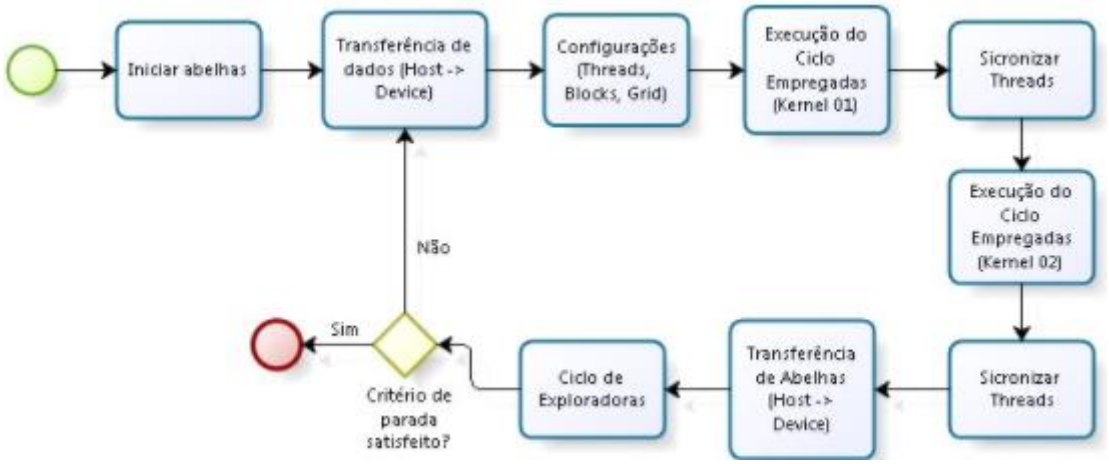

Figura 2. Fluxograma geral do CUDABC.

Após o termino de cada ciclo executado via GPU, é necessário que as *threads* sejam sincronizadas. Este mecanismo de sincronização é dado por uma função nativa da arquitetura CUDA. Os pseudocódigos do *kernel* para ambos os ciclos em paralelos implementados neste trabalho, estão representados respectivamente nos Algoritmos 1 e 2. O primeiro ciclo implementado consiste em três etapas principais: Designar cada *thread* para representar uma abelha especifica; em seguida, cada *thread* desenvolve o processo matemático aplicado para cada abelha; por fim, cada *thread* calcula o novo valor encontrado para a função objetivo de sua respectiva abelha e aplica uma busca gulosa para encontrar e atualizar o melhor valor local.

Algoritmo 1 - *Kernel* Ciclo Empregadas
1: Calcular índice da *thread* (TID) via *ThreadIdx.x*
2: **Se**, TID < QUANTIDADE_DE_ABELHAS, **faça:**
2.1: Copiar valores da abelha de índice TID para uma abelha temporária
2.2: Selecionar aleatoriamente uma abelha de índice != de TID (*T*)
2.3: Modificar solução da abelha temporária por intermédio de (1) utilizando (*T*)
2.4: Calcular *fitness da abelha temporária*
2.5: **Se**, *fitness* temporária < *fitness* da abelha TID, **faça:**
2.5.1: Modificar abelha de índice TID por abelha temporária
2.6: **fim se**
3: **fim se**

Para o ciclo de espectadoras, inicialmente, um número de *threads* correspondente ao número de metade da quantidade de abelhas é alocado; posteriormente, cada *thread* aplica o torneio, método de seleção selecionado pelos autores; por fim, aplica uma solução gulosa para selecionar a melhor entre as escolhidas para o torneio. Após o termino da execução do ciclo de espectadoras, as *threads* são sincronizadas novamente e em seguida, a memória atrelada a GPU é devolvida para CPU, onde a função de busca pela melhor solução global é executada.

Algoritmo 2 - *Kernel* Ciclo Espectadoras
1: Calcular índice da *thread* (TID) via *ThreadIdx.x*
2: **Se**, TID < (QUANTIDADE_DE_ABELHAS / 2), **faça:**
2.1: Selecionar 2 abelhas (*i, j*) para torneio
2.2: **Se**, *fitness* abelha_*i* < *fitness* abelha_*j*, **faça:**
2.2.1: Atribuir valores da abelha_*i* para abelha_*j*
2.3: **Se não**, **faça**:
2.3.1: Atribuir valores da abelha_*j* para abelha_*i*
3: **fim se**

5. Configurações

No seguinte experimento, foram utilizadas quatro funções de *benchmark*s, apresentadas na Tabela 1 para análise dos testes de desempenho, processadas pelo computador, cujo especificações estão inseridas na Tabela 2. Estas funções são consideradas representativas pela comunidade cientifica afim de testar a eficiência de qualquer algoritmo com proposito de otimização numérica.

Índice	**Funções**	**Limite**
1	$f(x) = \sum_{i=1}^{5} x_i^2$	$-100 \leq x_i \leq 100$
2	$f(x) = 100\,(x_2 - x_1^2)^2 + (x_1 - 1)^2$	$-2.048 \leq x_i \leq 2.048$
3	$f(x) = \sum_{i=1}^{10} (x_i^2 - 10\cos(2\pi x_i) + 10)$	$-600 \leq x_i \leq 600$
4	$f(x) = \frac{1}{4000} * \sum_{i=1}^{N} x_i^2 - \prod_{i=1}^{N} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	$-600 \leq x_i \leq 600$

Tabela 1. Funções de *Benchmark*.

Processador	Intel® Core™ i7-6700HQ (Quad Core 2.6GHz)
Memória RAM	16GB DDR4-2400MHz
GPU NVIDIA	NVDIA® GeForce® GTX 1060 6GB DDR5 (Pascal)
CUDA CORES	1280

Tabela 2. Configurações do Computador.

6. Resultados

A Tabela 3 apresenta os seguintes dados: as gerações aplicadas em cada teste; a quantidade de abelhas alocadas para cada teste (quantidade de *threads alocados)*; o tempo médio de execução das arquiteturas GPU e CPU respectivamente representadas por TM GPU e TM CPU; o desvio padrão, DP GPU e DP CPU, apresentado pelos experimentos; o melhor tempo de execução representado por MTE GPU e MTE CPU; e por fim marcações grifadas que representam os melhores valores de cada experimento.

A quantidade de parâmetros necessários para satisfazerem cada função são equivalentes a 100. Cada experimento varia entre: quantidade de abelhas por colônia; e a quantidade de gerações. Devido a limitação de hardware, a capacidade máxima de threads que podem ser alocadas corresponde a 1024, para outros testes, onde a quantidade de abelhas excede o limite disponível, outros blocos precisam ser alocados para desenvolver o paralelismo, o que vai contra o modelo proposto neste artigo de uma colônia por bloco. A quantidade de 5 execuções para cada função foi adotada afim de realizar qualquer análise estatística entre os diferentes experimentos. Enquanto que os Gráficos 1, 3, 5, 7 apresentam as medias de cada experimento e gráficos 2, 4, 6, 8 correspondem ao melhor tempo de execução encontrado durante os testes.

N°	Função	Gerações	Abelhas	TM GPU (s)	TM CPU (s)	DP GPU	DP CPU	MTE GPU (s)	MTE CPU (s)
1	Sphere (1)	200	100	0,125	**0,090**	**0,021**	**0,013**	0,100	**0,078**
2		400	500	0,254	**0,187**	**0,008**	**0,016**	0,242	**0,171**
3		800	1.000	**0,509**	0,556	**0,010**	**0,030**	**0,492**	0,531
4		5.000	100	2,366	**2,286**	**0,135**	**0,054**	**2,230**	2,239
5		7.000	500	3,279	**3,171**	**0,019**	**0,042**	3,252	**3,129**
6		10.000	1.000	**5,376**	6,389	**0,109**	**0,079**	**5,287**	6,297
1	Rosenbrock (2)	200	100	0,133	**0,106**	**0,018**	**0,007**	0,111	**0,093**
2		400	500	**0,262**	0,775	**0,030**	**0,014**	**0,211**	0,764
3		800	1.000	**0,501**	2,845	**0,010**	**0,043**	**0,484**	2,798
4		5.000	100	2,608	**2,359**	**0,022**	**0,038**	2,580	**2,312**
5		7.000	500	**3,426**	12,973	**0,117**	**0,487**	**3,220**	12,440
6		10.000	1.000	**5,671**	34,805	**0,057**	**0,295**	**5,602**	34,495
1	Rastrigin (3)	200	100	0,127	**0,123**	**0,027**	**0,006**	**0,087**	0,115
2		400	500	**0,254**	0,882	**0,007**	**0,022**	**0,248**	0,861
3		800	1.000	**0,512**	3,393	**0,009**	**0,032**	**0,499**	3,353
4		5.000	100	**2,613**	2,827	**0,050**	**0,018**	**2,548**	2,796
5		7.000	500	**3,348**	14,791	**0,223**	**0,124**	**3,049**	14,693
6		10.000	1.000	**5,655**	41,444	**0,105**	**0,353**	**5,528**	41,060
1	Grienwank (4)	200	100	**0,131**	0,142	**0,014**	**0,003**	**0,110**	0,139
2		400	500	**0,250**	1,057	**0,032**	**0,012**	**0,214**	1,040
3		800	1.000	**0,522**	4,056	**0,013**	**0,037**	**0,511**	4,030
4		5.000	100	**2,607**	3,276	**0,019**	**0,023**	**2,580**	3,240
5		7.000	500	**3,523**	18,378	**0,022**	**0,408**	**3,491**	18,040
6		10.000	1.000	**5,651**	50,796	**0,127**	**0,863**	**5,506**	50,021

Tabela 3. Parâmetros adotados para os testes de desempenho e respectivos resultados.

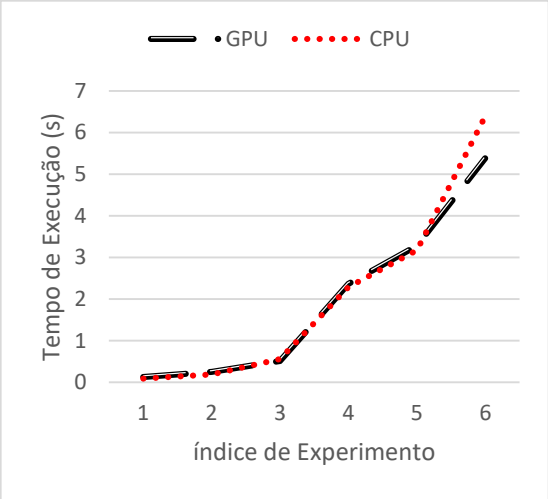

Gráfico 1. Representação gráfica das medias referentes a função *Sphere* (1).

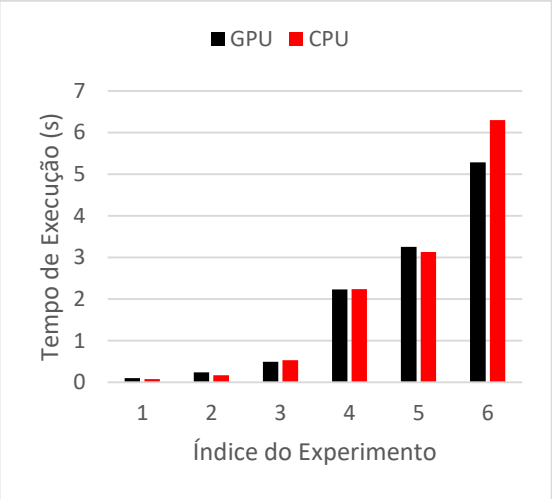

Gráfico 2. Representação gráfica dos melhores tempos de execução para função *Sphere* (1).

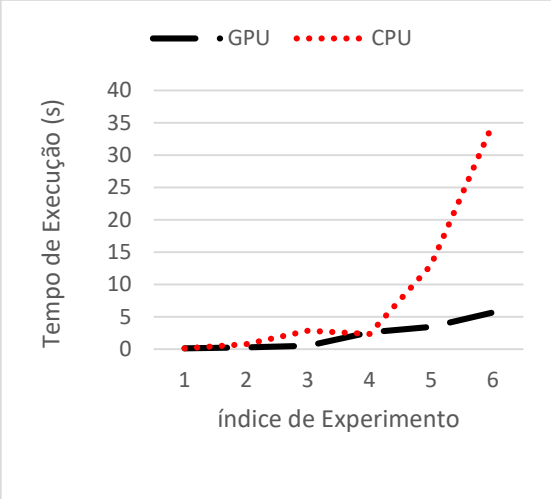

Gráfico 3. Representação gráfica das medias referentes a função *Rosenbrock* (2).

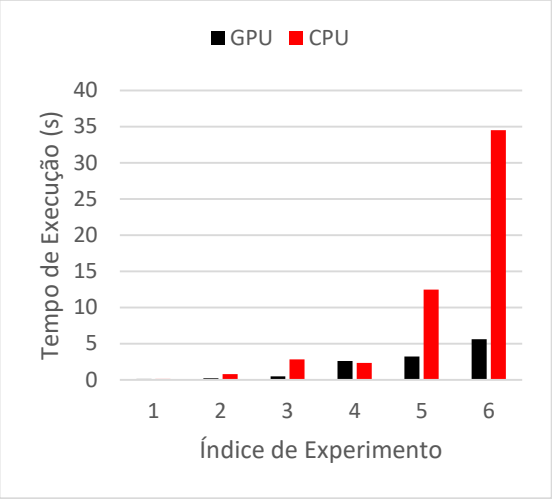

Gráfico 4. Representação gráfica dos melhores tempos de execução para função *Rosenbrock* (2)

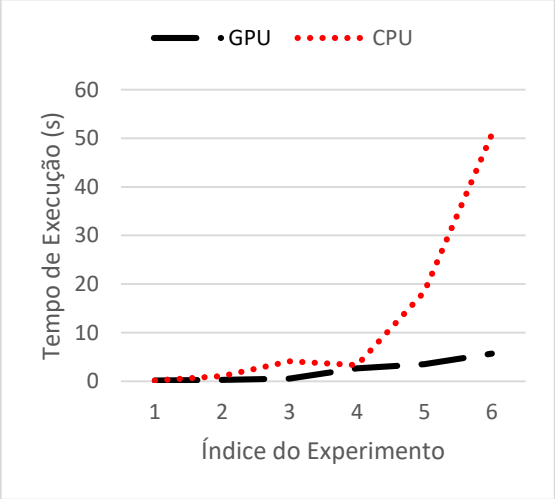

Gráfico 5. Representação gráfica das medias referentes a função *Rastrigin* (3).

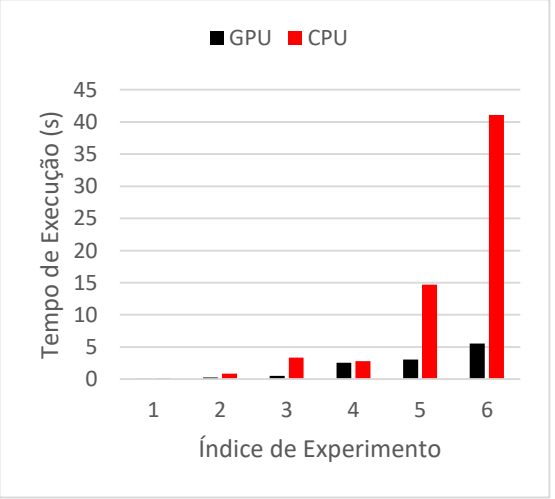

Gráfico 6. Representação gráfica dos melhores tempos de execução para função *Rastrigin* (3).

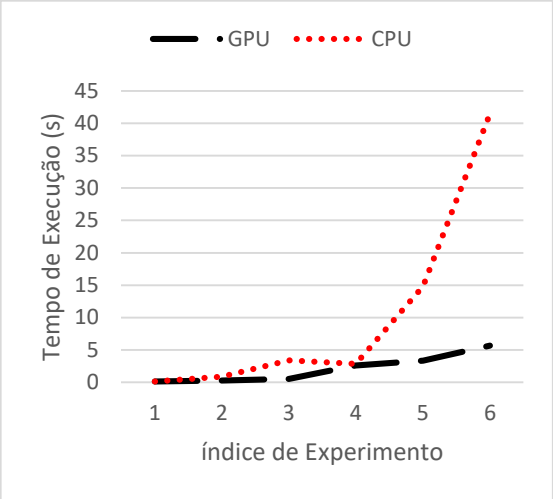

Gráfico 7. Representação gráfica das medias referentes a função *Grienwank* (4).

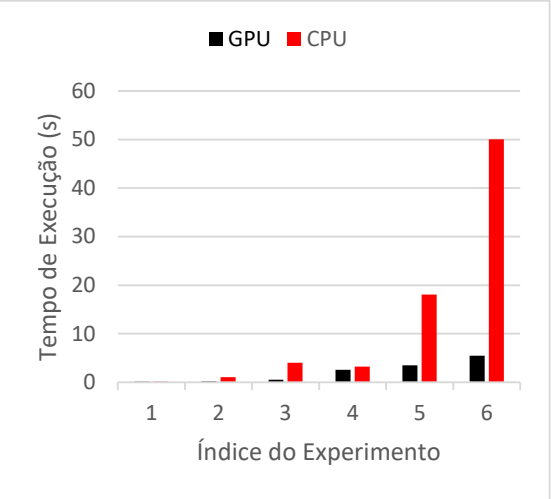

Gráfico 8. Representação gráfica dos melhores tempos de execução para função *Grienwank* (4).

Os resultados obtidos mostraram que o CUDABC obteve menor valor de media em dezessete experimentos abordados, enquanto que o desvio padrão mostrou-se inferior em quatorze destes experimentos. A perda de desempenho nos casos que o tempo de desempenho do CUDABC foi inferior ao ABC sequencial, podem ser atribuídos quatro fatores: 1) A função *Sphere*, é delimitada a um limite máximo de 5, logo, não necessita do mesmo poder de processamento que as outras funções apresentadas; 2) A necessidade de alocação de memória para as variáveis *devices* antes de serem submetidas a GPU; 3) O aguardo da sincronização das *threads* entre os ciclos; 4) ou o gargalo encontrado na troca de memória entre a GPU e CPU.

7. Conclusão

O proposito deste estudo foi analisar a performance do *Artificial Bee Colony* sob a arquitetura paralela das GPUs utilizando a linguagem de programação CUDA-C. Para tal, o algoritmo CUDABC desenvolvido, foi testado em *benchmarks* representativos, amplamente utilizados para otimização, e seus resultados foram comparados com uma versão sequencial do ABC. Os resultados apresentados demostraram que a implementação paralela do ABC é superior, em tempo de execução, quando comparada a versão tradicional escrita na linguagem de programação C.

Como proposta futuras, destaca-se uma futura implementação de uma versão multi-colméia do ABC, onde cada *block* seria responsável por uma colméia com diferentes parâmetros. Cabe destacar que a inclusão de mecanismos de busca local (e.g. estratégias evolutivas, análise probabilística baseada na memória de cada abelha) estão sendo desenvolvidos no intuito de aprimorar o processo de otimização.

Referencias

Akay, B. e Karaboga, D. (2012). Artificial Bee Colony Algorithm for Large-Scale Problems and Engineering Design Optimization. Journal of Intelligent Manufacturing 23.4 pp. 1001–1014**,**

Binitha, S. e Sathya, S. (2012). A Survey of Bio inspired Optimization Algorithms, International Journal of Soft Computing and Engineering (IJSCE), ISSN: 2231-2307, Volume-2, Issue-2.

Engelbrecht, A. P. (2007), Computational Intelligence: An Introduction, John Wiley and Sons Inc.

Haupt, R. L. e S. E. Haupt. (1998). *Practical Genetic Algoritms*, primeira edição, John Wiley and Sons Inc.

Karaboga, D., Basturk, B. e Ozturk, C. (2007) "Artificial bee colony (ABC) optimization algorithm for training feed-forward neural networks, Modeling Decisions for Artificial Intelligence", volume 4617/2007 of LNCS: pp. 318-319, Springer, Berlin.

Karaboga, D. (2005) An Idea Based on Honey Bee Swarm for Numerical Optimization. Relatório Técnico TR06, Universidade de Erciyes, Departamento de Engenharia Computacional.

Karaboga, D. e Akay, B. (2009). A comparative study of Artificial Bee Colony algorithm. Applied Mathematics and Computation 214.1, pp. 108 –132.

Kirk, D.B. e W.W. Hwu. (2010). Programming Massively Parallel Processors: A Hands on Approach, 1a edição, Elsevier and Morgan Kaufman.

Mussi, L., Nashed, Y. S. G. e S. Cagnoni. (2011). 'GPU-based asynchronous particle swarm optimization', *GECCO 11 Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* 2, 1555–1562.

NVIDIA 2017. (2017). CUDA C Programming Guide. htttp://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Acessado: 15-04-2017.

Parsopoulos, K. e Vrhatis, M. N. (2010). *Particle Swarm Optimization and Intelligence: Advances and Applications*, primeira edição, IGI Global.

Souza, D. L., Teixeira, O. N., Monteiro, D. C., de Oliveira, R. C. L. de Oliveira e Molinetti, M. A. F. (2014). *A Novel Competitive Quantum-Behaviour Evolutionary Multi-Swarm Opti- mizer Algorithm Based on CUDA Architecture Applied to Constrained Engineering Design*, Springer-Veerlag.

Souza, D. L., Teixeira, O.N., Monteiro, D.C. e de Oliveira, R. C. L. (2013). A new cooperative evolutionary multi-swarm optimizer algorithm based on CUDA archi- tecture applied to engineering optimization, *em* I.Hatzilygeroudis & V.Palade, eds., 'Combinations of Intelligent Methods and Applications', Vol. 23, Springer-Verlag, pp. 95–115.

Schatz, M.C., Trapnell, C., Delcher, A.L. e Varshney, A. (2007). "Highthroughput sequence alignment using graphics processing units ," BMC Bioinformatics, Vol. 8, p.474.

Stringhini, D., R. A. Gonçalves e A. Goldman (2012), Introdução à Computação Heterogênea, Jornada de Atualização de Informática (JAI) da Sociedade Brasileira de Computação (SBC), capítulo 7.

Tan W. (2016). GPU-based Parallel Implementation of Swarm Intelligence Algorithm, primeira edição, Elsevier and Morgan Kaufman

Tereshko, V. e Loengarov, A. (2005). Collective decision-making in honey bee foraging dynamics, Computing and Information Systems, 9 (3): 1-7, University of the West of Scotland, UK.