GPU-BRKGA: A GPU Accelerated Library for Optimization using The Biased Random-Key Genetic Algorithm

Derek Alves, Davi R. C. Oliveira, Ermeson Andrade, Bruno Nogueira

***Abstract*—This paper presents a CUDA/C++ library, called GPU-BRKGA, for accelerating the biased random-key genetic algorithm (BRKGA). Our library features an easy to use API that allows designers with little experience in GPU development to accelerate the solution of their optimization problems. It automatically handles the problem-independent modules of BRKGA, and leaves the designer in charge with only the decoding procedure, that is, to convert a random-key vector into a solution to the optimization problem. We compared our solution with two other BRKGA API implementations: the standard CPU-based BRKGA API implementation (brkgaAPI) and another GPU-based BRKGA API from the literature (brkgaCuda). Experimental results show that GPU-BRKGA is up to 28x faster than brkgaAPI and up to 3x faster than brkgaCuda.**

***Index Terms*—GPU, Genetic algorithms, Metaheuristics, Optimization.**

I. Introdução

Devido ao paralelismo massivo proporcionado por sua arquitetura, uma GPU (*graphics processing unit*) é capaz de fornecer uma vazão (número de instruções executadas por unidade de tempo) e uma largura de banda de memória muito superiores a uma CPU (central processing unit) [1]. Estas diferenças entre a CPU e a GPU existem porque elas foram projetadas com diferentes objetivos. A CPU foi desenvolvida com o objetivo de executar uma sequência de operações, que chamaremos de *thread*, da forma mais rápida possível. Já uma GPU foi projetada para executar milhares de *threads* em paralelo [2]. Embora a performance por *thread* da GPU seja inferior em comparação à CPU, a capacidade de executar milhares de *threads* simultaneamente torna a GPU mais eficiente nas situações em que a paralelização é viável. Um exemplo trivial de paralelização é somar os elementos de dois vetores em um terceiro vetor, em que em uma CPU utilizamos estruturas de repetição, e em uma GPU cada posição dos vetores seria computada por uma *thread* independente.

Meta-heurísticas são *frameworks* que permitem a construção de algoritmos aproximados para resolver problemas de otimização difíceis [3]. Embora as meta-heurísticas já tenham se provado úteis, elas são computacionalmente custosas, o que frequentemente impede o uso delas em aplicações onde uma resposta rápida é demandada. Desta forma, a implementação de meta-heurísticas em arquiteturas paralelas, como GPUs [1], [4], [5], [6] e FPGAs [7], é uma interessante e atrativa alternativa para acelerá-las.

Desde a sua introdução, a meta-heurística BRKGA (*biased random-key genetic algorithm*) [8] vem sendo aplicada com sucesso em uma grande variedade de aplicações, como gerenciamento de projetos [9], telecomunicações [10], [11], redes elétricas [12], [13], problemas de agendamento [14], [15] e gerenciamento de inventário [16]. Desta forma, uma implementação eficiente em GPU de BRKGA pode aumentar ainda mais este conjunto de aplicações.

Este artigo apresenta uma biblioteca em CUDA/C++, chamada de GPU-BRKGA, para acelerar a meta-heurística BRKGA. A biblioteca proposta possui uma API de fácil uso, permitindo que desenvolvedores com pouca experiência em GPU possam usá-la. GPU-BRKGA automaticamente gerencia os módulos de BRKGA que não são específicos ao problema, e deixa para o usuário apenas a implementação da parte específica, ou seja, converter um vetor de chaves aleatórias em uma solução do problema de otimização [8]. Nossa biblioteca está publicamente disponível em https://sites.google.com/site/nogueirabruno/software. Assim, além de apresentar as principais características da biblioteca, este artigo também tem como propósito explicar, por meio de um exemplo, como usá-la.

Até onde os autores conhecem, na literatura só existe uma implementação em GPU para BRKGA, chamada de brkgaCuda [17]. Neste trabalho, nós conduzimos um extenso conjunto de experimentos e comparamos GPU-BRKGA com brkgaCuda. Uma comparação com a biblioteca padrão de BRKGA em CPU, chamada de brkgaAPI [18], também é conduzida. Os resultados experimentais demonstram que GPU-BRKGA é até 28x mais rápida que brkgaAPI e até 3x mais rápida que brkgaCuda.

O restante deste trabalho é organizado da seguinte maneira. Seção II detalha o que é e como funciona um BRKGA. Seção III detalha a implementação proposta. Seção IV explica como GPU-BRKGA pode ser utilizado na prática para otimizar uma determinada função ou problema definido pelo usuário. Seção V apresenta nossos resultados experimentais, e a Seção VI conclui o trabalho apresentando nossas últimas considerações.

II. Biased Random-Key Genetic Algorithm (BRKGA)

Um algoritmo genético (AG) é uma meta-heurística que se baseia nos conceitos de seleção natural para resolver problemas difíceis de otimização [8]. Nos AGs, cada indivíduo de uma população é uma solução codificada para o problema. Além disso, cada indivíduo possui um cromossomo, que é então associado a uma função de aptidão. Esta função indica

o quanto determinado indivíduo está adaptado ao seu contexto, isto é, o quanto a solução é boa. Um cromossomo é um conjunto de alelos ou características, e as características dos indivíduos mais adaptados são passadas de geração em geração por meio de cruzamento entre pares de indivíduos. Para garantir variabilidade nas soluções, utiliza-se o conceito de mutação, que consiste em modificar aleatoriamente o cromossomo de um indivíduo. Seguindo esses conceitos, os AGs clássicos começam com uma uma população inicial que pode ser gerada aleatoriamente ou induzida por alguma técnica computacional, e então evoluem usando os operadores de cruzamento e mutação para assim gerar novos indivíduos que irão compor as novas gerações.

Os RKGAs (*random key genetic algorithms*) foram propostos em [19], e se caracterizam por classificar a população em dois grupos: o elite (mais adaptados) e o não-elite. Além disso, o operador de mutação dos AGs clássicos é substituído pela adição de novos indivíduos, chamados mutantes, e gerados de maneira aleatória. No RKGA, ao se gerar uma nova população, os indivíduos elite são copiados para a próxima geração, e o restante da população é substituída por mutantes ou novos indivíduos gerados através de cruzamentos. O cromossomo de cada indivíduo de uma população é representado por um conjunto de chaves aleatórias com valores reais no intervalo [0, 1). Por último, no RKGA está presente o decodificador, que é o procedimento que depende do problema e que tem a função de traduzir um cromossomo em uma solução real do problema. Com os valores dos cromossomos decodificados, é possível então aplicar a função de aptidão para assim avaliamos o quão apto é cada indivíduo.

O BRKGA difere do RKGA na maneira como os indivíduos são selecionados para cruzamento. No BRKGA, um dos pais sempre é selecionado do conjunto elite, enquanto no RKGA esta restrição não existe. Ao obrigar que um dos pais seja necessariamente do conjunto elite, a busca tende a se intensificar em soluções próximas das melhores encontradas previamente. Além disso, durante o cruzamento, existe um viés para que o filho herde mais caracteristicas (alelos) do pai elite.

III. A Biblioteca GPU-BRKGA

O fluxograma na Figura 1 ilustra o funcionamento GPU-BRKGA. É possível observar que GPU-BRKGA é bastante similar ao BRKGA. Uma das diferenças é que, no primeiro, a população é inicializada na memória da GPU. Além disso, em GPU-BRKGA os processos independentes do problema rodam em GPU. O decodificador, que é parte dependente do problema, pode ser especificado pelo usuário para executar em GPU ou CPU.

Conforme mostrado no Algoritmo 1, GPU-BRKGA recebe como entrada os seguintes argumentos: `n` valor natural que representa o tamanho do vetor de chaves aleatórias, `p` valor natural que representa o tamanho da população, `pe` valor real que representa a fração de indivíduos elite presentes na população, `pm` valor real que indica a fração de indivíduos mutantes na população, `rhoe` valor real que representa a probabilidade para o descendente receber alelo do parente elite durante o cruzamento, `decoder` é a referência para um objeto

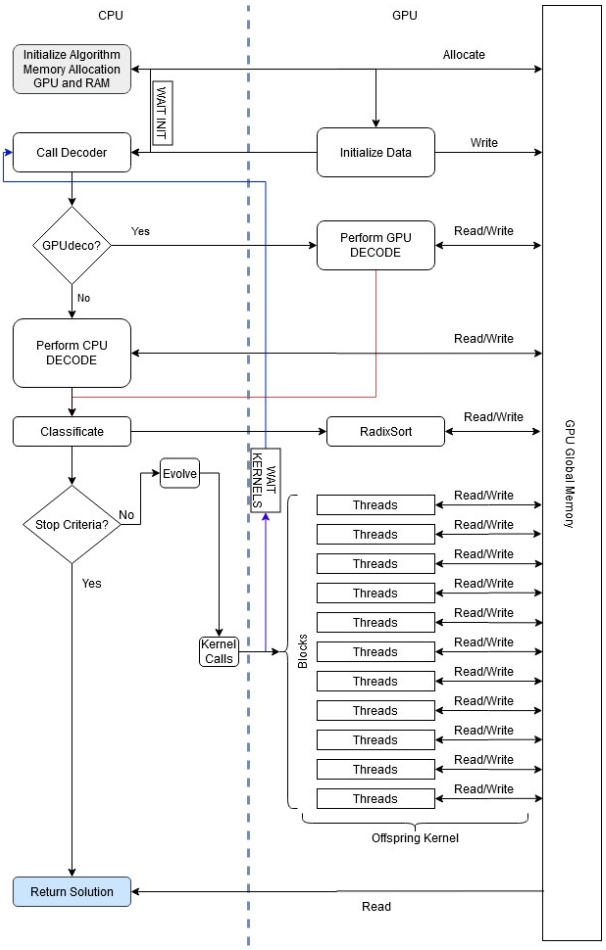

Fig. 1. Diagrama de fluxo do GPU-BRKGA.

decodificador que irá calcular o valor de aptidão de um dado cromossomo, `seed` valor inteiro que representa a semente para inicializar o gerador de números aleatórios, `gpudecode` valor booleano que irá indicar se o decodificador é executado em GPU ou CPU, e por último `K` valor natural indicando o número de populações independentes.

O Algoritmo 1 começa inicializando as estruturas de dados (vetores) que representam as populações atuais `Pop` e as próximas populações `NextPop` na memória da GPU (linha 2). Nestas estruturas, cada população possui p indivíduos, em que cada indivíduo é representado por um vetor de n alelos (cromossomo), e um par chave-valor. Nos pares chave-valor, a chave indica a aptidão do indivíduo, e o valor, seu respectivo *id*. Os alelos são representados por valores reais (*float*) no intervalo [0,1). Além de inicializar estas estruturas, a função *InitGPU* gera e decodifica as populações iniciais com os primeiros indivíduos. Em seguida, ela classifica cada indivíduo gerado em elite e não-elite.

Depois de gerar as populações iniciais, as iterações do processo evolutivo são executadas (linhas 3-4) até que a condição de parada seja atingida. A cada iteração (evolução) os conteúdos das populações atuais são atualizados com os valores das próximas populações. Assim, as próximas populações (`NextPop`) se tornam as populações atuais (`Pop`).

```
1  GPU-BRKGA(n, p, pe, pm, rhoe, decoder, seed, gpudecode, K)
       // inicializa o GPU-BRKGA
2      [Pop, Fit, NextPop, NextFit] ← InitGPU(n, p, pe, pm, rhoe, decoder, seed, gpudecode, K)
3      while stop criteria not met do // execute o processo evolutivo até o critério de parada
4          [Pop, Fit] ← Evolve(k, p, pe, pm, rhoe, n, gpu_decode, Min(n, 512), Pop, Fit, NextPop, NextFit)
5      sol ← GetBestIndividual(Pop, Fit)// melhor solução encontrada no GPU-BRKGA
6      return sol// a solução é retornada para o usuário
```

Algorithm 1: Pseudo-código do GPU-BRKGA

```
1  Evolve(k, p, pe, pm, rhoe, n, gpu_decode, thr, Pop, Fit, NextPop, NextFit)
2      for i = 1...k do
           // Calcula-se os ponteiros para cada uma das populações e aptidões i, para
              que possa acontecer a geração de descendentes através do kernel offspring
3          [Pop_i, NextPop_i] ← getPop(i, Pop, NextPop)
4          [Fit_i, NextFit_i] ← getFit(i, Fit, NextFit)
           // o kernel para gerar uma nova população é lançado
5          Offspring<<<p, thr>>>(Pop_i, Fit_i, NextPop_i, NextFit_i, p, pe, pm, rhoe)
6          if gpu_decode is true then // decodifique a população em GPU
7              NextFit_i ← Decode(NextPop_i)// decodifique a população i em gpu
8          else // senão, a decodificação será executada em CPU
9              HostPop_i ← CudaCopyCPU(NextPop_i) // a população i é copiada para RAM
10             HostPopFit_i ← Decode(HostPop_i) // decodifique a população i em CPU
11             NextPopFit_i ← CudaCopyGPU(HostPopFit_i) // aptidões calculadas irão para VRAM
           // Classificando os individuos
12         NextFit_i ← RadixSort-SortFitnessPairs(NextFit_i)// ordenação Radix-Sort
13     Swap(NextPop_i, Pop_i)// é efetuada uma troca entre as populações, de modo que, a
14     Swap(NextFit_i, Fit_i)// próxima população seja a atual, e a atual seja a próxima.
```

Algorithm 2: Pseudo-código do método de evolução do GPU-BRKGA

Após ser atingida a condição de parada, o resultado obtido é transferido para a memória da CPU (linha 5) para que possa ser retornado ao usuário (linha 6). A condição de parada pode ser estabelecida pelo usuário por uma das seguintes formas: número de gerações, tempo de execução, quantidade de iterações no processo evolutivo sem melhora nos resultados, ou caso um determinado valor de aptidão seja alcançado.

O processo evolutivo das populações é descrito no Algoritmo 2. Neste algoritmo, para cada população i (linha 2), os seguintes passos são executados: Primeiro, são obtidos os ponteiros para as populações e aptidões i (linhas 3-4). Estes ponteiros indicam onde os cromossomos e pares chave-valor da população i começam nos vetores de população e aptidão. Em seguida, o *kernel* emphOffspring é lançado com `p` blocos de `thr` *threads* (linha 5). Este *kernel* tem por objetivo gerar os descendentes das populações atuais. O valor de `thr` é baseado em dois fatores: o tamanho do cromossomo (`n`) e a quantidade máxima de *threads* por bloco suportada pela arquitetura de GPU adotada pelo usuário (`MAX_THREADS`). Caso o cromossomo tenha mais que `MAX_THREADS` alelos, uma ou mais *threads* irá operar em mais de um alelo. Dessa forma, o valor `thr` é definido da seguinte maneira: `thr = Min(n, MAX_THREADS)`.

Depois da execução do *kernel Offspring*, os novos descendentes são decodificados usando a GPU (linhas 6-7) ou CPU (linhas 8-11). Em seguida, os indivíduos são classificados (linha 12) utilizando o procedimento *RadixSort* da biblioteca CUB [20], que executa na GPU e ordena os pares chave-valor de todos os indivíduos de cada população. Por último, os vetores de próximas populações e aptidões são definidos como vetores atuais (linhas 13-14).

O Algoritmo 3 detalha o *kernel Offspring*, que é lançado para gerar os descendentes da geração atual, e assim criar a próxima geração. Primeiro, são calculados os índices que identificam cada indivíduo que será gerado (linha 2-3). Cada indivíduo é tratado por um bloco de *threads*, e cada *thread* é responsável por operar em um ou mais alelos. Para simplificar o pseudocódigo, tratamos apenas o caso onde uma thread opera em um alelo, isto é, `thr = n`. Se o índice do bloco de *threads* for menor que a quantidade de indivíduos elite, o respectivo bloco de *threads* copia o indivíduo (elite) para a próxima população (linhas 4-6). Caso contrário, se o índice do bloco for menor que a quantidade de indivíduos menos a quantidade de mutantes, o bloco de *threads* inicia o procedimento de cruzamento (linhas 7-13).

No início do processo de cruzamento, os parentes são selecionados, sendo um elite e um não-elite. A seleção de ambos os parentes (linhas 8-10) é decidida por meio da utilização de um gerador de números aleatórios. Geramos um número entre `0` e `pe`, selecionando assim o parente elite, e outro número aleatório para o parente não-elite, dessa vez entre `pe` e `p`. Após selecionados os parentes, o processo de

```
1  Kernel Offspring(Pop, PopFit, NextPop, NextPopFit, p, pe, pm, rhoe)
       // BlockIdx, BlockDim e ThreadIdx são conhecidos pela thread
2      alele_idx ← ThreadIdx // calcula o índice para o alelo operado por essa thread
3      i_idx ← getIdx(BlockIdx, BlockDim) // calcula o índice para o novo indivíduo
4      if BlockIdx < pe then // copie pe indivíduos elite para a próxima população
5          j_idx = PopFit[BlockIdx] // o índice é calculado, utilizando o id de cada
               indivíduo que está armazenado no par chave-valor do fitness
6          NextPop[i_idx + alele_idx] ← Pop[j_idx + alele_idx]// copia o alelo do indivíduo elite
               deste bloco
7      else if BlockIdx < p − pm then // gere p-pe-pm novos indivíduos realizando o cruzamento
8          if ThreadIdx = 0 then // selecione os parentes aleatoriamente para o
9              shmemo elitePar ← RNG(pe) // cruzamento, utilizando memória compartilhada
10             shmemo nonElitePar ← pe+ RNG(p − pe)// entre as threads deste bloco.
11         SyncThreads() // sincronize as threads do mesmo bloco para o cruzamento
           // Jogue a moeda viciada e defina qual será o alelo herdado, do parente
               elite ou não elite levando em conta o viés definido em rhoe.
12         sel_par ← RNG-Real() < rhoe ? eliteParent : nonEliteParent
           // característica do parente selecionado é passada para o descendente
13         NextPop[i_idx + alele_idx] ← Pop[(sel_par ∗ BlockDim) + alele_idx]
14     else if BlockIdx < p then // insira pm mutantes na nova população
           // um alelo aleatório no intervalo [0,1) é gerado por thread deste bloco,
               gerando assim um mutante por bloco
15         NextPop[i_idx + alele_idx] ← RNG-Real ()
16     if ThreadIdx is 0 then
           // atribua o índice do bloco (id do indivíduo) para o fitness
17         NextPopFit[BlockIdx] ← BlockIdx
```

Algorithm 3: Pseudo-código do kernel *Offspring*

cruzamento é iniciado com o lançamento da moeda viciada (linha 12). Esta moeda viciada faz com que a probabilidade de selecionar as características do indivíduo elite seja `rhoe`. Depois de selecionar de qual pai o alelo será herdado, este alelo é atribuído ao novo filho (linha 13).

Caso o índice do bloco seja maior que a quantidade de indivíduos menos a quantidade de mutantes, o bloco de *threads* faz a inserção de mutantes (linhas 14-15). Os mutantes são inseridos utilizando o gerador de números aleatórios. Por último (linhas 16-17), no par chave-valor do indivíduo gerado (aptidão), é atribuído ao campo valor o índice do bloco de *threads* que gerou o indivíduo.

O GPU-BRKGA foi implementado em CUDA/C++ e está estruturado a partir de basicamente três classes, sendo estas: *GPU-BRKGA*, *Decoder* e *Individual*.

1) *GPU-BRKGA* é a classe base da nossa API e contém todos os métodos e variáveis necessários para a execução da parte independente do problema.
2) *Decoder* é a classe a ser implementada pelo usuário e precisa conter pelo menos dois métodos: *Decode* e *Init*, que serão apresentados adiante. Outros métodos podem ser especificados pelo usuário se necessário.
3) *Individual* é usada para representar um indivíduo na memória da CPU.

Além dessas classes, utilizamos o arquivo *kernels.cuh* que contém algumas funções e *kernels* necessários para a API, como por exemplo o *kernel Offspring*.

A seguir, iremos detalhar alguns métodos da classe GPU-BRKGA, que podem ser utilizados pelo usuário. Um método bastante importante para o funcionamento da biblioteca é o método `evolve` que pode receber ou não um valor natural e maior que zero representando a quantidade de evoluções que serão efetuadas. O método é encarregado por evoluir as populações por uma ou mais gerações (caso não seja estipulado parâmetro, por padrão, a população é evoluída uma vez).

O método `reset` é utilizado para reinicializar todas as populações, ele executa o mesmo procedimento que a linha 2 do Algoritmo 1. O usuário pode utilizá-lo caso não obtenha resultados satisfatórios com uma população, e assim reinicializar todas as populações pode se obter uma melhor solução a partir dos novos indivíduos.

Um outro método útil da biblioteca é o método `exchangeElite`, que troca indivíduos elite entre todas as populações (caso exista mais de uma população independente), substituindo os indivíduos menos aptos de uma população pelos indivíduos elite de outras populações.

Uma maneira de analisar a melhor solução encontrada até então é utilizando o método `getBestIndividual` da biblioteca, que retorna o melhor indivíduo entre todas as populações. Com isto o usuário pode verificar se uma condição de parada foi atingida, ou reiniciar a população caso não seja obtido um resultado melhor em determinado número de iterações, como já citado anteriormente.

Adicionalmente, o usuário pode utilizar o método `getPopulations` da biblioteca para obter todas as populações. Recomendamos que este método seja utilizado apenas em situações em que a condição de parada é atingida, pois ele é bastante custoso. Como é necessário mover muitos dados em memória da GPU para a CPU, o uso constante deste método irá aumentar o tempo de execução. De forma similar, a implementação de um decodificador em CPU também é custosa, pois a cada geração temos que transferir dados da memória da GPU para a CPU, e vice-versa.

IV. Usando a Biblioteca

Esta seção descreve como nossa biblioteca pode ser utilizada para otimizar qualquer problema definido pelo usuário. Primeiro, o usuário precisa definir um decodificador para o problema a ser otimizado. O decodificador pode ser implementado em CPU ou GPU. Após a definição do decodificador, é necessário implementar um arquivo *main*, que fará as chamadas para biblioteca.

As Listagens 1 e 2 exibem exemplos dessas definições, considerando que o objetivo é otimizar a função de teste Schwefel [21], definida a seguir: $f(x) = 418,9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$, $x_i \in [-500, 500]$ para todo $i = 1, \ldots, d$. Neste exemplo, cada dimensão $i \in \{1, \ldots, d\}$ da função de teste será representada por um alelo do cromossomo. Assim, o papel do decodificador é converter cada chave aleatória no intervalo $[0, 1)$ para os valores do domínio da função Schwefel. Isto pode ser feito por meio da seguinte função: $decKey(x) = (x - 0,5) \cdot 1000$.

Como exemplificado na Listagem 1, no arquivo *main*, o usuário pode utilizar algumas funções já definidas pelo GPU-BRKGA, como o método `evolve`. No exemplo, foi adotado como critério de parada o número de gerações.

Conforme a Listagem 2, para implementar um decodificador é necessário a especificação de uma classe que implemente a interface *Decoder*. Esta classe deverá possuir dois métodos como base:

- `void Decoder::Init()` este método é chamado na inicialização de GPU-BRKGA e pode ser utilizado para ler uma instância do problema a ser otimizado e/ou inicializar estruturas de dados.
- `void Decoder::Decode(float* population, float* fitnessKeys)` este método é chamado para decodificar uma população, isto é, obter a aptidão associada a cada cromossomo. Ele recebe como parâmetro dois ponteiros, o primeiro ponteiro indica onde começa a população de chaves aleatórias que está sendo decodificada (cada população possui `p` cromossomos, e cada cromossomo possui `n` alelos). O segundo parâmetro é o ponteiro que indica o início do vetor onde cada valor de aptidão calculado deve ser armazenado.

No decoder implementado para o exemplo, cada cromossomo é decodificado usando um bloco de 64 *threads*. Considerando que cada população independente tem 512 cromossomos, são lançados 512 blocos. Além disso, o *kernel* de decodificação faz uso do método *Blockreduce* da biblioteca CUB [20] para calcular o somatório da função Schwefel.

V. Resultados Experimentais

Nesta seção apresentamos os experimentos computacionais utilizados para avaliar o desempenho de GPU-BRKGA. Nossa plataforma de experimentos usa o sistema operacional Ubuntu 18.04.4 LTS e é composta por um processador Intel Core i5-7200U @ 2.50GHz - 2.71GHz, com 8GB de memória DDR4 @ 2133MHz e uma GPU NVIDIA GeForce 940MX possuindo 2GB de memória de vídeo GDDR5. Nosso GPU-BRKGA foi compilado com o NVIDIA CUDA 10.1.243 e g++ 7.5.0 utilizando a *flag* de otimização *'-O3'*. Os tempos de execução foram obtidos utilizando *CUDA events*. Esses tempos incluem o *overhead* de comunicação entre CPU e GPU, a inicialização da memória da GPU, e a transferência de dados da memória da CPU para a memória da GPU e vice-versa.

O GPU-BRKGA foi avaliado nas seguintes funções multi-dimensionais bastante adotadas na literatura para testar algoritmos de otimização: Rastrigin, Schwefel e Styblinski-Tang [21]. Para cada uma das funções, conduzimos três grupos de experimentos considerando os seguintes pares de valores para *p* (número de indivíduos na população) e *n* (número de alelos): *(p=256, n=32)*, *(p=512, n=64)* e *(p=1024, n=128)*. Executamos o GPU-BRKGA em cada um destes cenários por 1000, 10000 e 20000 gerações. Em todos os experimentos, a seguinte configuração de parâmetros foi adotada : $K = 1$, $pe = 15,625$, $pm = 15,625$, $rhoe = 0,7$ e $gpu\_deco = true$.

GPU-BRKGA foi comparada com duas outras bibliotecas: a biblioteca padrão baseada em CPU dos autores originais de

Listing 1. Exemplo para arquivo main

```
    ...
// Inicialize o GPU-BRKGA
  GPU-BRKGA<Decoder> algorithm(n, p, pe, pm, rhoe, RefDecoder, se, ipt, thr, gpu_deco
      , K);
  std::cout << "Executando por" << MAX_GENS << "geracoes..." << std::endl;
  do {
    algorithm.evolve(); // Evolui a populacao por uma geracao
  }while(generation++ < MAX_GENS);
  Individual ind = algorithm.getBestIndividual(); // Retorna o melhor individuo
      encontrado pelo GPU-BRKGA
  printf("Fitness:%.3f", ind.fitness.first);
```

TABLE I
COMPARATIVO CONSIDERANDO A FUNÇÃO RASTRIGIN

p: 256 - n: 32	GPU-BRKGA		brkgaAPI		brkgaCuda		Speedup GPU-BRKGA		Utilização API	
Gerações	t(s)	erro	t(s)	erro	t(s)	erro	brkgaAPI	brkgaCuda	GPU-BRKGA	brkgaCuda
1000	0.054	2.230	0.373	2.388	0.171	2.242	6.898	3.159	81.10%	94.02%
10000	0.471	0.024	3.624	0.022	1.475	0.023	7.693	3.131	78.11%	93.01%
20000	0.937	0.006	7.258	0.006	2.935	0.006	7.748	3.133	79.09%	93.33%
p: 512 - n: 64	GPU-BRKGA		brkgaAPI		brkgaCuda		Speedup GPU-BRKGA		Utilização API	
Gerações	t(s)	erro	t(s)	erro	t(s)	erro	brkgaAPI	brkgaCuda	GPU-BRKGA	brkgaCuda
1000	0.150	5.430	1.448	6.000	0.236	5.934	9.632	1.573	82.14%	88.65%
10000	1.346	0.060	14.034	0.058	2.134	0.059	10.423	1.585	80.98%	88.00%
20000	2.674	0.016	28.124	0.014	4.219	0.016	10.518	1.578	80.96%	87.93%
p: 1024 - n: 128	GPU-BRKGA		brkgaAPI		brkgaCuda		Speedup GPU-BRKGA		Utilização API	
Gerações	t(s)	erro	t(s)	erro	t(s)	erro	brkgaAPI	brkgaCuda	GPU-BRKGA	brkgaCuda
1000	0.563	14.351	5.814	15.016	0.570	15.666	10.329	1.013	83.86%	84.06%
10000	4.667	0.181	56.343	0.157	5.449	0.177	12.073	1.168	80.30%	83.13%
20000	9.228	0.047	112.331	0.032	11.090	0.045	12.173	1.202	80.06%	83.41%

BRKGA (brkgaAPI [18]), e outra implementação em GPU da literatura (brkgaCuda [17]). A biblioteca brkgaAPI foi escolhida para a comparação por ser a implementação em CPU mais amplamente adotada na literatura, já a biblioteca brkgaCuda foi considerada porque, até onde os autores conhecem, ela é a única implementação em GPU de BRKGA da literatura. Fizemos uma modificação em brkgaCuda para permitir a decodificação paralela de uma população, assim como acontece no GPU-BRKGA, de modo a tornar a comparação justa. O mesmo protocolo de execução de GPU-BRKGA (número de gerações e parâmetros) foi adotado nos outros algoritmos.

As Tabelas I, II e III apresentam os resultados do comparativo entre GPU-BRKGA, brkgaAPI e brkgaCuda. As implementações são comparadas em termos de média do

Listing 2. Definição do decodificador em GPU

```
//Kernel utilizado para decodificacao em gpu
__global__ void dec(float* d_next, float* d_nextFitKeys){
  unsigned int idx = threadIdx.x + blockIdx.x * blockDim.x; //o indice da thread foi
      calculado
  float xi = (d_next[idx]-0.500)*1000; //xi foi adequado ao dominio da funcao
      schwefel
  float value = xi*sinf(sqrtf(fabsf(xi))); //o valor foi calculado para o somatorio
  //Block reduce foi executado para efetuar o somatorio de todos values entre as
      threads deste bloco
  typedef cub::Blockreduce<float, 64> Blockreduce;
  __shared__ typename Blockreduce::TempStorage temp_storage;
  float sum = Blockreduce(temp_storage).reduce(value, cub::Sum());
  //Se o indice da thread for zero, a aptidao calculada sera atribuida
  if(!threadIdx.x)
    d_nextFitKeys[blockIdx.x] = 418.9829 * blockDim.x - sum;
}

// Exemplo de inicializacao de um decodificador usado pelo GPU-BRKGA, em que temos
    uma populacaoo com 512 individuos cada um possuindo 64 alelos.
void Decoder::Init(){
    blocks = 512;
    threads = 64;
}

// Exemplo de definicao de um decodificador.
void Decoder::Decode(float* next, float* nextFitnessKeys){
  dec<<<blocks, threads>>>(next, nextFitnessKeys);
  return;
}
```

TABLE II
COMPARATIVO CONSIDERANDO A FUNÇÃO SCHWEFEL

p: 256 - n: 32	GPU-BRKGA		brkgaAPI		brkgaCuda		Speedup GPU-BRKGA		Utilização API	
Gerações	t(s)	erro	t(s)	erro	t(s)	erro	brkgaAPI	brkgaCuda	GPU-BRKGA	brkgaCuda
1000	0.054	13.720	0.417	14.437	0.173	15.300	7.755	3.215	81.84%	94.35%
10000	0.468	0.163	4.131	0.177	1.460	0.111	8.831	3.120	79.03%	93.28%
20000	0.929	0.047	8.234	0.060	2.953	0.036	8.859	3.178	79.49%	93.54%
p: 512 - n: 64	GPU-BRKGA		brkgaAPI		brkgaCuda		Speedup GPU-BRKGA		Utilização API	
Gerações	t(s)	erro	t(s)	erro	t(s)	erro	brkgaAPI	brkgaCuda	GPU-BRKGA	brkgaCuda
1000	0.147	38.171	1.628	36.207	0.230	36.934	11.111	1.571	84.39%	90.07%
10000	1.307	0.382	15.966	0.451	2.082	0.435	12.215	1.593	83.37%	89.56%
20000	2.594	0.102	31.896	0.159	4.195	0.123	12.297	1.617	83.37%	89.70%
p: 1024 - n: 128	GPU-BRKGA		brkgaAPI		brkgaCuda		Speedup GPU-BRKGA		Utilização API	
Gerações	t(s)	erro	t(s)	erro	t(s)	erro	brkgaAPI	brkgaCuda	GPU-BRKGA	brkgaCuda
1000	0.541	104.594	6.507	101.136	0.542	103.433	12.033	1.003	86.84%	86.88%
10000	4.451	1.057	63.127	1.342	5.184	0.978	14.184	1.165	85.47%	87.53%
20000	8.794	0.312	125.985	0.392	10.354	0.296	14.326	1.177	84.67%	86.95%

TABLE III
COMPARATIVO CONSIDERANDO A FUNÇÃO STYBLINKSI-TANG

p: 256 - n: 32	GPU-BRKGA		brkgaAPI		brkgaCuda		Speedup GPU-BRKGA		Utilização API	
Gerações	t(s)	erro	t(s)	erro	t(s)	erro	brkgaAPI	brkgaCuda	GPU-BRKGA	brkgaCuda
1000	0.053	0.185	0.827	0.209	0.171	0.175	15.460	3.203	82.07%	94.40%
10000	0.460	0.002	8.307	0.002	1.460	0.002	18.059	3.174	79.08%	93.41%
20000	0.914	0.001	16.470	0.001	2.879	0.001	18.018	3.149	80.50%	93.81%
p: 512 - n: 64	GPU-BRKGA		brkgaAPI		brkgaCuda		Speedup GPU-BRKGA		Utilização API	
Gerações	t(s)	erro	t(s)	erro	t(s)	erro	brkgaAPI	brkgaCuda	GPU-BRKGA	brkgaCuda
1000	0.148	0.544	3.260	0.578	0.236	0.566	21.962	1.588	83.78%	89.79%
10000	1.322	0.007	32.364	0.005	2.094	0.008	24.475	1.584	82.47%	88.93%
20000	2.622	0.003	64.787	0.001	4.157	0.004	24.706	1.585	82.44%	88.92%
p: 1024 - n: 128	GPU-BRKGA		brkgaAPI		brkgaCuda		Speedup GPU-BRKGA		Utilização API	
Gerações	t(s)	erro	t(s)	erro	t(s)	erro	brkgaAPI	brkgaCuda	GPU-BRKGA	brkgaCuda
1000	0.550	2.028	13.029	1.881	0.555	1.851	23.678	1.008	85.35%	85.46%
10000	4.541	0.017	128.980	0.007	5.319	0.016	28.406	1.171	82.39%	84.97%
20000	8.976	0.009	258.094	0.003	10.562	0.010	28.755	1.177	82.18%	84.85%

tempo de execução *(t(s))* e média do erro *(erro)* (diferença entre o valor ótimo conhecido e o valor obtido pelo algoritmo). Apresentamos também o *speedup* de GPU-BRKGA em comparação às outras duas APIs e, por fim, o percentual do tempo de execução gasto com as funções das APIs em GPU. O percentual gasto com as APIs foi calculado subtraindo do tempo total o tempo gasto com a decodificação dos indivíduos.

A Tabela I contém o comparativo considerando a função Rastrigin. Podemos observar que em todos os cenários os tempos de execução de GPU-BRKGA foram inferiores, e seus erros foram similares às outras APIs. Além disso, o *speedup* sobre brkgaAPI é de aproximadamente **12x** nos cenários de maior população e tamanho de cromossomo. Em comparação ao brkgaCuda, os melhores *speedups* estão na faixa de **3x** e acontecem em populações menores com cromossomos menores.

Nas Tabelas II e III temos o comparativo da função Schwefel e Styblinski-Tang, respectivamente. Novamente podemos observar que os resultados da Tabela I se repetem: melhores tempos de execução para o GPU-BRKGA e erros similares. Da mesma maneira, os *speedups* em comparação à brkgaAPI são melhores quando as populações são maiores e com cromossomos maiores. Em particular, é possível observar que GPU-BRKGA é até **28x** mais rápida que brkgaAPI. Este aumento no *speedup* é esperado visto que com populações/cromossomos maiores, o potencial de paralelização aumenta.

Já em relação à brkgaCuda, os resultados nas Tabelas II e III confirmam que GPU-BRKGA tem *speedups* melhores em populações/cromossomos menores. Isto pode ser explicado pela principal diferença de implementação entre brkgaCuda e GPU-BRKGA. Em brkgaCuda, durante as operações de seleção, cruzamento, e mutação, apenas uma *thread* é lançada para cada cromossomo. Já em GPU-BRKGA, conforme pode ser observado nos Algoritmos 2 e 3, são lançadas `thr` *threads* para cada cromossomo. Devido ao maior número de *threads* em execução, GPU-BRKGA apresenta um melhor aproveitamento do paralelismo da GPU. Note porém que à medida que a população/cromossomo aumenta, o número de *threads* que a GPU é capaz de executar em paralelo entra em saturação e, por isso, os tempos de execução de GPU-BRKGA se aproximam de brkgaCuda.

Para verificar se GPU-BRKGA é de fato uma boa aproximação para brkgaAPI, aplicamos o teste não-paramétrico de Friedman considerando os erros médios

gerados por GPU-BRKGA e brkgaAPI. O *p-value* obtido foi próximo de 1, o que indica que não existem diferenças significativas entre os erros médios dos dois algoritmos. Este resultado demonstra que não é possível rejeitar a hipótese de que GPU-BRKGA é capaz de encontrar as mesmas soluções que o brkgaAPI quando ambos são executados pelo mesmo número de gerações.

VI. Conclusão

Neste artigo, propusemos uma nova paralelização baseada em GPU para o BRKGA. Nós conduzimos um conjunto abrangente de testes considerando vários cenários e três funções bastante conhecidas na literatura. Nossos principais resultados são: (i) um *speedup* de até **28x** em relação a implementação sequencial padrão do BRKGA (brkgaAPI), (ii) desempenho até 3x superior em relação a outra biblioteca em GPU da literatura (brkgaCuda). Nosso GPU-BRKGA está publicamente disponível e possui uma API de fácil uso, permitindo que ele possa ser utilizado para otimizar qualquer problema definido pelo usuário. Em trabalhos futuros, pretendemos usar o GPU-BRGKA para otimizar problemas reais e mais desafiadores, como problemas de otimização combinatória em grafos [1].

Acknowledgment

Os autores agradecem ao CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) e a UFAL (Universidade Federal de Alagoas) pela concessão de bolsas e auxílio financeiro que possibilitou a dedicação integral ao programa de iniciação científica e a realização do estudo.

References

[1] B. Nogueira and R. G. Pinheiro, "A cpu-gpu local search heuristic for the maximum weight clique problem on massive graphs," *Computers & Operations Research*, vol. 90, pp. 232–248, 2018.
[2] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.
[3] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009, vol. 74.
[4] B. Nogueira, E. Tavares, J. Araujo, and G. Callou, "Accelerating continuous grasp with a gpu," *The Journal of Supercomputing*, vol. 75, no. 9, pp. 5741–5759, 2019.
[5] B. Nogueira and R. G. Pinheiro, "A gpu based local search algorithm for the unweighted and weighted maximum s-plex problems," *Annals of Operations Research*, vol. 284, no. 1, pp. 367–400, 2020.
[6] M. Essaid, L. Idoumghar, J. Lepagnot, and M. Brévilliers, "Gpu parallelization strategies for metaheuristics: a survey," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 34, no. 5, pp. 497–522, 2019.
[7] B. Nogueira and E. Barboza, "A fpga-based accelerated architecture for the continuous grasp," *Computing*, pp. 1–20, 2020.
[8] J. F. Gonçalves and M. G. Resende, "Biased random-key genetic algorithms for combinatorial optimization," *Journal of Heuristics*, vol. 17, no. 5, pp. 487–525, 2011.
[9] J. F. Gonçalves, M. G. Resende, and J. J. Mendes, "A biased random-key genetic algorithm with forward-backward improvement for the resource constrained project scheduling problem," *Journal of Heuristics*, vol. 17, no. 5, pp. 467–486, 2011.
[10] M. G. Resende, "Biased random-key genetic algorithms with applications in telecommunications," *Top*, vol. 20, no. 1, pp. 130–153, 2012.
[11] A. Lima, R. Pinheiro, B. Nogueira, and R. Peixoto, "Algoritmo genético de chaves aleatórias viciadas para o problema do tempo de transmissão mínimo," Oct 2020.
[12] H. de Faria, M. G. Resende, and D. Ernst, "A biased random key genetic algorithm applied to the electric distribution network reconfiguration problem," *Journal of Heuristics*, vol. 23, no. 6, pp. 533–550, 2017.
[13] L. A. Roque, D. B. Fontes, and F. A. Fontes, "A biased random key genetic algorithm approach for unit commitment problem," in *International Symposium on Experimental Algorithms*. Springer, 2011, pp. 327–339.
[14] G. B. Mainieri, "Meta-heurística brkga aplicada a um problema de programação de tarefas no ambiente flowshop híbrido." Ph.D. dissertation, Universidade de São Paulo, 2014.
[15] S. M. Homayouni, D. B. Fontes, and F. A. Fontes, "A brkga for the integrated scheduling problem in fmss," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2019, pp. 287–288.
[16] F. T. Chan, R. Tibrewal, A. Prakash, and M. Tiwari, "Inventory based multi-item lot-sizing problem in uncertain environment: Brkga approach," in *Advances in Sustainable and Competitive Manufacturing Systems*. Springer, 2013, pp. 1197–1206.
[17] E. Xavier, "Uma interface de programação de aplicações para o brkga na plataforma cuda," in *Anais Principais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*. SBC, 2019, pp. 13–24.
[18] R. F. Toso and M. G. Resende, "A c++ application programming interface for biased random-key genetic algorithms," *Optimization Methods and Software*, vol. 30, no. 1, pp. 81–93, 2015.
[19] J. C. Bean, "Genetic algorithms and random keys for sequencing and optimization," *ORSA journal on computing*, vol. 6, no. 2, pp. 154–160, 1994.
[20] D. Merrill, "Nvidia cub," https://nvlabs.github.io/cub/, 2020.
[21] J. Momin and X.-S. Yang, "A literature survey of benchmark functions for global optimization problems," *Journal of Mathematical Modelling and Numerical Optimisation*, vol. 4, no. 2, pp. 150–194, 2013.

Derek Alves is an undergraduate student in Computer Engineering at Universidade Federal de Alagoas (UFAL). His interests include parallel programming, genetic algorithms and optimization. He is also a member of the Optimization Laboratory (OptLab) based at Instituto de Computação (IC - UFAL).

Davi R. C. Oliveira is an undergraduate student in Computer Engineering at UFAL. He is a member of the Optimization Laboratory at the IC - UFAL. His interests include programming and optimization.

Ermeson Andrade is an associate professor at the Department of Computing at Universidade Federal Rural de Pernambuco (UFRPE), Brazil. In March 2014, he finished his PhD degree in Computer Science at Federal University of Pernambuco. His research interest is focused in Performance, Dependability, Critical infrastructure, Data science and Modeling.

Bruno Nogueira is an assistant professor at the Institute of Computing at Universidade Federal de Alagoas, Brazil. He obtained his BSc (2009), MSc (2010), PhD (2015) in computer science from Federal University of Pernambuco. His research interest is focused in Optimization, Performance and dependability evaluation, and High-performance computing.