Uma Interface de Programação de Aplicações para o BRKGA na plataforma CUDA

Eduardo C. Xavier[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
13083-852 – Campinas – SP – Brazil

Abstract. *In this paper we present the development of an Application Programming Interface (API) for the algorithmic framework Biased Random-Key Genetic Algorithms (BRKGA) that executes in the CUDA platform. We compare the performance of our BRKGA API against the standard BRKGA API developed by Toso and Resende, and show that even using a simple entrance level GPGPU, significant speedup can be achieved. In the same spirit of the standard BRKGA API, we developed our API in such a way that all central aspects of the BRKGA logic are dealt by the API and little effort is required by an user of the API. The task of the user is to implement a problem-dependent procedure to convert vectors of random keys into a solution to the specific problem being solved. We provide an example of use of the API for the Traveling Salesman Problem (TSP) and show that the CUDA implementation outperforms the standard API even when this last one is executed in parallel using OpenMP with several threads.*

Resumo. *Neste artigo apresentamos o desenvolvimento de uma Interface de Programação de Aplicações (IPA) para o framework Biased Random-Key Genetic Algorithms (BRKGA), para execução na plataforma CUDA. Nós comparamos a performance da IPA para BRKGA proposta contra uma IPA padrão para BRKGA proposta por Toso e Resende, e mostramos que mesmo usando uma GPGPU de entrada, é possível obter um speedup significativo. No mesmo espírito da IPA padrão para BRKGA, nós desenvolvemos a nossa IPA de tal forma que os aspectos lógicos principais do BRKGA são considerados na IPA e pouco esforço de um usuário é requerido para usar a IPA para implementar soluções para problemas específicos. O trabalho do usuário é a implementação de uma função dependente do problema, que dado um vetor de chaves aleatórias computa uma solução para o problema sendo considerado. Nós apresentamos um exemplo de uso da IPA para o problema Traveling Salesman Problem (TSP) e mostramos que a execução da IPA em CUDA é mais rápida do que a execução da IPA padrão mesmo quando esta última é executada em paralelo com uso de OpenMP com várias threads de processamento.*

1. Introdução

Nos últimos anos várias técnicas foram desenvolvidas no intuito de se resolver problemas de otimização combinatória NP-difíceis com tempos aceitáveis na prática. Existem técnicas exatas, onde se busca encontrar soluções ótimas, dentre elas destacamos métodos enumerativos como *branch-and-bound*, métodos baseados em programação linear inteira, e métodos com programação dinâmica, e existem métodos onde busca-se encontrar soluções não necessariamente ótimas, mas soluções com garantia de qualidade

como algoritmos de aproximação. Ainda que tenha havido muito progresso no desenvolvimento destas técnicas, a resolução de problemas combinatórios NP-Difíceis muitas vezes requer soluções que sejam encontradas de forma rápida e para instâncias de tamanho muito grande. Neste caso o uso de heurísticas tem sido promissor. As heurísticas encontram soluções de boa qualidade, muitas vezes soluções ótimas, com o benefício de serem executadas rapidamente. Mesmo quando se busca encontrar soluções ótimas, o uso de heurísticas é muitas vezes utilizado em conjunto com técnicas exatas, visando encontrar limitantes para o ótimo, acelerando assim o processo de resolução dos problemas de forma exata.

Uma meta-heurística é um procedimento de alto nível que coordena heurísticas mais simples, como um heurística de busca local, no intuito de encontrar soluções de melhor qualidade do que estas heurísticas isoladamente. Muitas meta-heurísticas foram desenvolvidas nos últimos anos, e destacamos a busca tabu [Glover and Laguna 1998], algoritmos genéticos [Holland 1992], *simulated annealing* [Kirkpatrick et al. 1983], GRASP [Feo and Resende 1995] e *Biased Random-Key Genetic Algorithms (BRKGA)* [Gonçalves and Resende 2011]. Esta última pertence a classe de algoritmos evolutivos com princípios de funcionamento parecidos com algoritmos genéticos tendo como grande vantagem a fácil aplicação a diferentes tipos de problemas, bastando ser especificado uma função que mapeia vetores de valores reais entre 0 e 1 em soluções para o problema sendo resolvido em questão. Toda a parte de evolução, cruzamento, mutação, etc, comumente considerada em algoritmos genéticos, fica transparente para o usuário do método.

Algoritmos baseados em BRKGA foram aplicados com sucesso para os mais diversos tipos de problemas, como problemas de empacotamento [Gonçalves and Resende 2012, Gonçalves and Resende 2013], problemas de projeto de redes [Gonçalves and Resende 2015, Noronha et al. 2011] problemas de escalonamento [Gonçalves et al. 2011], dentre outros. Como o framework de BRKGA define uma série de passos que são independentes do problema, e apenas uma função que é dependente do problema, a saber, a função de decodificação (veja mais detalhes sobre BRKGA na Seção 2), é razoável pensar em uma implementação independente do algoritmo de BRKGA onde usuários precisam apenas criar a função de decodificação. Visando então criar uma interface reaproveitável para o BRKGA, Toso e Resende [Toso and Resende 2015] criaram uma Interface de Programação de Aplicações (IPA) em C++ para o BRKGA, de tal forma que toda a lógica de um BRKGA está já implementada, restando ao usuário estabelecer a função de decodificação. Desta forma torna-se extremamente conveniente criar heurísticas para um problema de otimização basedas em BRKGA pois o esforço de implementação fica substancialmente reduzido.

Considerando implementações de heurísticas para GPGPU, destacamos que diversas heurísticas para os mais diversos problemas foram criadas nos últimos anos, em geral trazendo ganhos significativos de *speedup* quando comparadas com heurísticas sequenciais. Heurísticas baseadas em *Simulated Annealing* foram implementadas em GPGPU para o problema quadrático de alocação em [Takemoto et al. 2018], e para o problema da mochila multi-dimensional [Dantas and Cáceres 2016, de Almeida Dantas and Cáceres 2018] por exemplo. Heurísticas baseadas em busca tabu também foram implementadas em GPGPU para diferentes problemas como escalonamento flow-shop [Czapiński and Barnes 2011] e o

problema quadrático de alocação [Czapiński 2013]. Implementações de heurísticas baseadas em algoritmos genéticos também foram implementadas em GPGPU [Pospichal et al. 2010, Arora et al. 2010, Zhang and He 2009]. Para mais informações sobre implementações de heurísticas para problemas de otimização com algoritmos paralelos o leitor pode ver a resenha [Alba et al. 2013].

Apesar destas heurísticas baseadas em meta-heurísticas encontrarem soluções de muito boa qualidade, a reutilização de código para tratar de diferentes problemas requer um trabalho significativo de re-implementação. Até houve avanços em se criar pacotes reutilizáveis, por exemplo em [Meffert et al. 2012] é proposto um pacote para implementação de algoritmos genéticos. Porém, ainda assim o uso do pacote é complicado, onde o usuário deve estabelecer como será feita a codificação de soluções em cromossomos, como serão os operadores de evolução e mutação etc [Toso and Resende 2015]. Desta forma a meta-heurística BRKGA traz um avanço significativo ao simplificar o processo de criação de novas heurísticas especialmente quando se usa uma IPA para a mesma.

Neste trabalho nós então propomos uma nova IPA para o BRKGA utilizando CUDA. Desta forma o esforço de implementação de heurísticas para os mais diversos problemas é reduzido ao mesmo tempo em que temos ganhos do uso de processamento paralelo em GPGPU, em comparação com a implementação padrão da IPA de Toso e Resende para CPU.

Este artigo está organizado da seguinte forma: na Seção 2 descrevemos o funcionamento do BRKGA, na Seção 3 descrevemos a IPA padrão de Toso e Resende, na Seção 4 apresentamos a implementação da nossa IPA com CUDA e finalmente na Seção 5 apresentamos experimentos computacionais comparando as duas IPAs.

2. Biased Ramdom-Key Genetic Algorithms

Os algoritmos genéticos [Holland 1992] foram desenvolvidos como métodos para resolução de problemas de otimização onde é simulado o processo de seleção natural de indivíduos. Cada indivíduo é representado como um cromossomo, que por sua vez representa uma solução do problema sendo considerado. Cada cromossomo é composto por n genes. Os algoritmos genéticos evoluem uma população de indivíduos por um determinado número de gerações. A cada geração uma nova população é criada combinando indivíduos da população corrente para gerar descendentes para a próxima geração, processo conhecido como *crossover*. Apesar do processo ser aleatório, como forma de passar boas características para a próxima geração, no crossover é favorecido a passagem das características de indivíduos que representam melhores soluções. Os indivíduos da nova geração também podem sofrer mutações, e isto é utilizado como uma forma de diversificação, que sob a ótica de otimização serve para escapar de mínimos locais. O conceito de sobrevivência é aplicado como uma forma de selecionar os indivíduos representando as melhores soluções para a próxima geração. Note que é preciso uma função de *fitness* que avalia quão bom cada cromossomo é, e em geral ela representa o valor da solução obtida através do cromossomo em questão.

Em um BRKGA cada cromossomo é representado por um vetor de n chaves aleatórias que são números reais aleatórios no intervalo $[0, 1]$. Inicialmente uma população com p cromossomos é criada de forma aleatória com cada gene gerado de

forma independente e com valor uniforme em $[0, 1]$. É preciso criar uma função de decodificação, que dado um vetor de chaves aleatórias mapeia este para uma solução do problema sendo tratado, e calcula o valor desta solução, otendo-se assim o fitness deste cromossomo. Após o cálculo de fitness para cada cromossomo na população, esta é particionada em dois grupos, temos p_e soluções elite, com os melhores valores de fitness, e $p - p_e$ cromossomos não-elite. O método de crossover é feito selecionando-se de forma aleatória um cromossomo elite C_e e também de forma aleatória um cromossomo não elite C_{ne}. Seja $\rho > 0.5$ um parâmetro para a operação de crossover. Criamos um novo cromossomo C_n por crossover, onde para cada índice de gene $i = 1, \ldots, n$, o gene $C_n[i]$ é igual a $C_e[i]$ com probabilidade ρ e é igual a $C_{ne}[i]$ com probabilidade $(1 - \rho)$. Note que desta forma um novo indivíduo é gerado favorecendo-se a utilização de características da solução elite. Na nova população também são incluídos cromossomos mutantes que são gerados como um cromossomo inicial, isto é, um cromossomo onde cada gene é setado com um valor aleatório em $[0, 1]$. Uma nova população é construída da seguinte forma, primeiro os p_e melhores cromossomos da geração atual são copiados para a nova população. Depois p_m mutantes são incluídos na nova população. Finalmente $p - p_e - p_m$ novos cromossomos são gerados por crossover como explicado anteriormente.

Desta forma o framework BRKGA é uma meta-heurística para resolução de problemas de otimização onde ela busca encontrar a melhor solução em um hipercubo n-dimensional, com lados de tamanho 1, e a função de decodificação faz o mapeamento de soluções no hipercubo em soluções do problema sendo considerado, devolvendo o valor de cada solução. Na Figura 1 apresentamos um fluxograma indicando o funcionamento do BRKGA. Nesta figura destacamos em azul escuro a única parte do BRKGA que é dependente do problema. Todos os demais componentes são independentes do problema e por isso o BRKGA é uma meta-heurística facilmente aplicada aos mais diversos problemas de otimização combinatória. Um usuário deve se preocupar apenas em criar uma função de decodificação para o problema sendo resolvido.

Considere por exemplo o problema TSP, onde temos como entrada um grafo completo métrico $G = (V, E)$ com custos nas arestas, e cujo objetivo é encontrar um ciclo hamiltoniano de custo mínimo. Podemos considerar cada cromossomo com tamanho $n = |V|$, onde cada gene representa um vértice de G. Dado um vetor de chaves aleatórias, podemos ordená-lo em ordem crescente pelos valores das chaves aleatórias. Esta ordem representa uma permutação dos n vértices de G e podemos considerá-la como uma ordem de visitação dos vértices, formando um ciclo. Considerando este exemplo, uma possível função de decodificação para o TSP pode ser definida como: ordene o vetor de chaves aleatórias anotando os índices de cada gene. Crie um ciclo onde cada vértice é visitado na ordem do seu respectivo índice no vetor ordenado, e o custo do ciclo é a soma das arestas usadas.

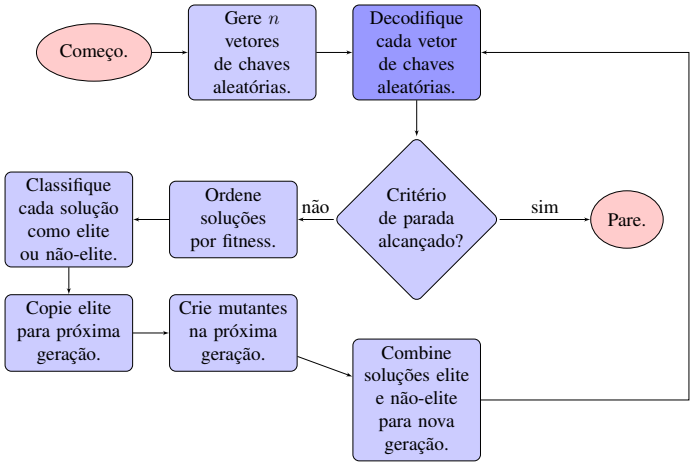

Figura 1. Esquema de funcionamento de um algoritmo BRKGA.

Note ainda que é fácil implementar o conceito multi-populacional no BRKGA onde cada população é criada separadamente e todo o processo de evolução segue em separado das demais. Porém uma operação comum em algoritmos genéticos multi-populacionais é a troca de melhores indivíduos entre as populações visando aumentar a variabilidade de boas soluções em cada população [Branke et al. 2000]. Não é difícil alterar o framework BRKGA para lidar com o caso multi-populacional, bastando setar um determinado número de iterações onde há troca das melhores soluções entre as populações.

3. A Interface de Programação de Aplicações para BRKGA

Dado a facilidade de uso de um algoritmo BRKGA para os mais diversos problemas de otimização, onde a única parte dependente do problema é a função de decodificação, é natural construir uma Interface de Programação de Aplicações (IPA) para o BRKGA com toda a parte de inicialização e evolução (incluindo classificação de cromossomos em elite e não elite, criação de mutantes, e cruzamento) implementadas independente de problemas específicos. Desta forma, resta a um usuário do framework a implementação de uma função de decodificação, que recebe como entrada um cromossomo e devolve o valor da solução que ele representa para o problema específico.

Uma IPA em C++ para o BRKGA foi criada por Toso e Resende [Toso and Resende 2015], e é conhecida como *brkgaAPI*. A brkgaAPI provê duas classes básicas para usuários, a classe *Population* e a classe *BRKGA*. A classe Population implementa métodos para recuperação de informações de uma população específica, como o melhor fitness de um indivíduo da população, o cromossomo com melhor fitness, etc. Já a classe BRKGA implementa o framework em si, e é implementado como uma classe template que recebe na sua criação dois templates de classe específicos, uma classe *Decoder* e uma classe *RNG*. A classe Decoder deve ser implementada pelo usuário com um método público *decoder* que recebe como parâmetro um vetor de *double* como parâmetro, representando um cromossomo, e deve devolver o valor da solução associada com este cromossomo. A classe RNG é utilizada como gerador de números aleatórios e deve prover a implementação do método *rand()*, que devolve um ponto-flutuante aleatório no intervalo $[0, 1)$, e o método *randInt(unsigned N)* que deve devolver um inteiro aleatório no intervalo $[0, N]$. Portanto as únicas classes específicas a serem implementadas pelo usuário

são Population e RNG que são utilizadas na criação de um objeto do tipo BRKGA. A classe BRKGA implementa o algoritmo BRKGA em si, e contém diversos métodos para execução do BRKGA, como método para inicialização de uma população, um método para evoluir as populações por um determinado número de gerações, método para trocar indivíduos elite entre as populações etc. Vale ressaltar que a brkgaAPI é implementada em C++ com o uso de OpenMP para paralelizar a decodificação, visto que na fase de classificação e ordenação precisamos estabelecer o fitness de cada indivíduo de todas as populações e como os cromossomos são independentes, a decodificação destes pode ser feita em paralelo.

4. Uma Interface de Programação para BRKGA utilizando CUDA

Neste trabalho propomos a criação de uma IPA para o BRKGA para execução em GPGPU em CUDA. Chamaremos de brkgaCuda tal IPA que está disponível para download em `https://bitbucket.org/eduardocxavier/brkga-cuda/src/master/`. A principal classe na IPA é a classe `BRKGA`, que implementa toda a lógica do BRKGA, e contém os seguintes métodos públicos:

- `BRKGA`, construtor que recebe como parâmetro os valores de inicialização da classe que são: `unsigned n` o tamanho do cromossomo, `unsigned p` o tamanho da população, `float pe` a proporção de indivíduos elite, `float pm` a proporção de indivíduos mutantes, `float rhoe` o valor do parâmetro para a operação de crossover, `unsigned K` o número de populações independentes, e `unsigned decode_type` que especifica se a função de decodificação será executada na CPU ou na GPGPU.
- `void evolve` com parâmetro `int number_generations=1` que especifica o número de evoluções a se fazer nas populações correntes.
- `void exchangeElite` com parâmetro `unsigned M` especificando o número de cromossomos elite para se trocar entre as populações independentes.
- `std::vector<std::vector <float>> getkBestChromosomes` com parâmetro `unsigned k`, devolve os `k` melhores cromossomos dentre todas as populações.
- `void setInstanceInfo` com parâmetros `void *info`, `long unsigned num`, e `long unsigned size`, que serve para salvar na memória um vetor de `num` dados, onde cada dado tem `size` bytes. Este vetor é utilizado na decodificação de cada cromossomo, e contém portanto informações relevantes para poder ser computado o valor da solução representada por um cromossomo. Caso a decodificação ocorra na GPGPU este vetor é salvo na memória global da mesma. Por exemplo, para o problema TSP este vetor representa uma matriz com valores de distância entre cidades.

A outra classe relevante na IPA é a classe `Decoder` com métodos de decodificação que devem ser implementados pelo usuário. O usuário deve implementar apenas um dos métodos de acordo com a forma que será feita a decodificação. Os métodos da classe são:

- `float host_decode` que possui parâmetros `float *chromosome`, `int n` e `void *h_instance_info`. O método recebe um ponteiro para o início do cromossomo, o tamanho do cromossomo `n` e `h_instance_info`

é um ponteiro para o vetor de informações necessárias para a decodificação. O método deve devolver o valor da solução representada por este cromossomo. Este método é executado na CPU.

- `__device__ float device_decode` que possui parâmetros `float *chromosome`, `int n`, e `void *d_instance_info`. Este método recebe os mesmos parâmetros que o anterior mas note que este método é executado na GPGPU e portanto `d_instance_info` é um ponteiro para a memória global na GPGPU.
- `__device__ float device_decode_chromosomeSorted` que possui como parâmetros `ChromosomeGeneIdxPair *chromosome`, `int n` e `void *d_instance_info`. É similar aos métodos anteriores só que neste caso o primeiro parâmetro é um vetor de um `struct` com dois campos, um com o valor do gene e outro com seu índice no cromossomo. Além disso o vetor está ordenado em ordem crescente pelos valores da chave. Este método é executado na GPGPU.

Na implementação da IPA, cada cromossomo é processado por uma thread. Para aproveitar melhor o paralelismo da GPGPU fornecemos o terceiro método, onde o cromossomo já está ordenado pelas chaves. Em vários problemas de otimização é comum mapear o cromossomo em uma solução realizando-se primeiramente a ordenação do cromossomo por valores das chaves. Optamos por oferecer este método para um usuário, pois caso o usuário tivesse que realizar por si só a operação de ordenação para cada cromossomo, isso não seria tão eficiente quanto ao uso da biblioteca `Thrust` [Hoberock and Bell 2010] sobre todos os cromossomos de uma só vez, que é como é feita a ordenação dos cromossomos internamente na classe `BRKGA`. Além destas classes temos dois arquivos usados na IPA: um com a definição da classe `ConfigFile` que contém parâmetros de configuração da execução do BRKGA e o arquivo `CommonStructs.h` que contém definições de estruturas usadas pela IPA, como a estrutura `ChromosomeGeneIdxPair` usada na decodificação com cromossomos ordenados.

É importante ressaltar que um usuário da IPA não precisa necessariamente conhecer programação em CUDA para utilizar a IPA, já que será preciso apenas implementar uma das funções de decodificação que podem ser implementadas em C/C++ padrão. Para aproveitar melhor o paralelismo os dois últimos métodos são preferíveis, pois serão executados na GPGPU, porém o usuário deve estar ciente disto pois na GPGPU não será possível acessar outros dados exceto aqueles salvos em `d_instance_info`, não podendo ser acessado um arquivo por exemplo, ou qualquer outra estrutura em memória da CPU.

4.1. Paralelização do BRKGA

Nesta seção descrevemos resumidamente a paralelização feita na IPA brkgaCuda. Mencionamos anteriormente que alocamos uma thread para realizar a decodificação de cada cromossomo, caso o usuário opte por executar esta operação na GPGPU. Além disso várias outras etapas realizadas sequencialmente no BRKGA são paralelizadas na brkgaCuda. Organizamos blocos de threads em uma dimensão. Por padrão usamos 256 threads por bloco, e arredondamos o tamanho da população, p, para um múltiplo de 256. Alocamos uma thread para cada cromossomo, de tal forma que a dimensão do grid será $(p * K)/256$.

Para representar todos os cromossomos de todas as populações utilizamos um vetor de `float` contíguo. Para gerar uma nova população é necessário um outro vetor contíguo para onde copiamos cromossomos elite, criamos cromossomos mutantes e geramos cromossomos por crossover. Por isso apenas dois vetores são alocados representando todos os cromossomos de todas as populações, um para a população corrente e outro para a nova população a ser gerada. Após ser gerada a nova população apenas uma troca de ponteiros é necessária. A fase de inicialização de todos os cromossomos de todas as populações é feita de forma paralela onde utilizamos o método `curandGenerateUniform` do CUDA toolkit para inicializar uma área de memória com valores aleatórios no intervalo $[0, 1]$. Para gerar uma nova população, primeiramente fazemos um processo similar a fase de inicialização setando de forma paralela todos os genes de todos cromossomos da nova população com valores aleatórios. Depois disso, cada thread fará o processamento de um cromossomo. Caso a thread esteja associada com um cromossomo elite, então são copiados os genes para o novo cromossomo, caso esteja associada com um cromossomo mutante nada precisará ser feito pois o cromossomo foi já inicializado com valores aleatórios. Para uma thread associada com um cromossomo resultante de crossover, esta usa o valor do gene corrente como um numéro aleatório (lembre-se que os cromossomos da nova população foram inicializados aleatoriamente), que será testado contra o valor ρ para decidir se o gene deverá ser setado com o valor do pai elite ou do pai não elite. Desta forma toda a geração de uma nova população é feita de forma paralela, com uma thread para cada cromossomo.

As fases de ordenação e classificação dos cromossomos por fitness também são feitas em paralelo. Usamos um algoritmo paralelo de ordenação estável da biblioteca `thrust` para ordenar cromossomos por valor de fitness e depois por índice de sua população de tal forma que para cada população os cromossomos estarão ordenados em ordem crescente de fitness. Note que na ordenação dos cromossomos usamos apenas índices destes no processo de ordenação, não sendo necessário a movimentação dos cromossomos no espaço de memória alocado para os mesmos.

5. Resultados dos Experimentos Computacionais

Os experimentos computacionais foram realizados em um laptop com Ubuntu 18.04, 16GiB de memória principal, processador Intel Core i7-8550U 1.8GHz com 4 cores e 8 threads, e com placa gráfica Nvidia MX150 com 4GiB de memória. Tanto a IPA padrão brkgaAPI, quanto a IPA brkgaCuda estão implementadas em C++. Uma limitação do nosso trabalho é a realização de testes em apenas uma máquina (um par CPU/GPU), e como trabalhos futuros pretendemos realizar mais experimentos com diferentes configurações CPU/GPU.

Como problema teste utilizamos o problema TSP e utilizamos as instâncias da TSPLIB disponíveis em `http://www.math.uwaterloo.ca/tsp/data/index.html`. Utilizamos as instâncias da coleção *National TSP* que contém mapas de diversos países contendo as coordenadas das principais cidades de cada país, onde a distância entre cidades é a distância euclidiana. Utilizamos também a coleção *VLSI TSP* que possui coordenadas de pontos em circuitos digitais e a distância entre pontos também corresponde à distância euclidiana. Como precisamos alocar uma matriz de distâncias na memória global (além de outros dados como cromossomos das populações etc.), não foi possível utilizar todas as instâncias nos testes, pois instâncias com mais de 25000 vértices

	brkgaCuda	brkgaAPI CPU-1
zi929	**948092.81**	953056.00
lu980	**126053.04**	138241.00
rw1621	**579833.19**	588307.00
mu1979	**2465718.00**	2623049.00
nu3496	3200197.50	**3181367.00**
ca4663	70621040.00	**70027576.00**
tz6117	20468628.00	**20420208.00**
eg7146	9406424.00	**9379337.00**
ym7663	**12508235.00**	13047365.00
pm8079	7895304.00	**7876657.00**
ei8246	**10367363.00**	10511863.00
ar9152	**53765484.00**	54006324.00
ja9847	**48044316.00**	48508624.00
gr9882	21144074.00	**20969820.00**
kz9976	**86311328.00**	90957584.00
fi10639	30961720.00	**30901924.00**
mo14185	**33993396.00**	34129560.00
ho14473	16695521.00	**16615462.00**
it16862	**53788440.00**	54029448.00
vm22775	94970560.00	**94326544.00**
sw24978	90397880.00	**90250136.00**
Média	**31840933.69**	32068688.19

Tabela 1. Valores das soluções encontradas para cada instância da coleção National TSP.

acabavam estourando o limite de memória da GPGPU. Desta forma utilizamos todas as instâncias com até 25000 vértices na entrada. Na coleção *National TSP* foram usadas 21 instâncias enquanto na coleção *VLSI TSP* foram usadas 81 instâncias.

Ambas IPAs foram executadas com os mesmos parâmetros, população de 256 indivíduos, 3 populações independentes, 1000 gerações, $p_e = 0.1$, $p_m = 0.1$, $\rho = 0.7$, troca de 2 melhores soluções elite entre populações a cada 200 gerações. Estes valores foram utilizados como o padrão da brkgaAPI de Toso e Resende exceto o tamanho da população que arredondamos para o número de threads em um bloco. Na brkgaCuda implementamos o método de decodificação em GPGPU recebendo cromossomos já ordenados. A decodificação da brkgaAPI foi usada como a implementada por Toso e Resende [Toso and Resende 2015].

Primeiramente verificamos se as soluções encontradas pelas duas implementações são significativamente diferentes. Na Tabela 1 apresentamos os valores das soluções encontradas para cada instância da coleção *National TSP*. Podemos observar que os valores das soluções encontradas não são tão diferentes. De fato realizamos um teste de Wilcoxon para verificar se as distribuições de valores são diferentes e obtemos um p-value de 0.68, e portanto as diferenças não são significativas. Este resultado faz sentido tendo em vista que são duas implementações de um mesmo método de solução e portanto não deveria haver diferenças significativas nos valores das soluções encontradas.

Na Tabela 2 apresentamos os tempos de execução da implementação com brkgaCuda (Cuda Time), e com a IPA brkgaAPI com 1 thread (CPU-1), com 4 threads (CPU-4) e finalmente com 8 threads (CPU-8). Também apresentamos os valores de *speedup* obtidos da implementação com a brkgaCuda em comparação com a brkgaAPI com os diferentes números de threads usados nesta última. Notamos que a implementação com brkgaCuda obteve os menores tempos de execução para todas as instâncias, mesmo quando comparado com a implementação de brkgaAPI executando com 8 threads. Na primeira

	Cuda Time	CPU-1 Time (s)	CPU-4 Time (s)	CPU-8 Time (s)	speedup-1	speedup-4	speedup-8
zi929	**14.94**	36.99	19.44	19.88	2.48	1.30	1.33
lu980	**15.49**	39.59	20.82	20.29	2.56	1.34	1.31
rw1621	**28.76**	69.32	35.48	35.47	2.41	1.23	1.23
mu1979	**32.66**	86.37	43.55	44.34	2.64	1.33	1.36
nu3496	**59.61**	160.64	79.78	76.92	2.69	1.34	1.29
ca4663	**81.82**	218.59	108.28	97.42	2.67	1.32	1.19
tz6117	**107.15**	294.05	138.38	126.86	2.74	1.29	1.18
eg7146	**125.61**	347.73	162.28	145.52	2.77	1.29	1.16
ym7663	**134.27**	374.84	174.44	154.76	2.79	1.30	1.15
pm8079	**142.36**	398.41	184.73	162.91	2.80	1.30	1.14
ei8246	**149.81**	404.05	188.10	166.00	2.70	1.26	1.11
ar9152	**166.95**	453.42	209.06	182.92	2.72	1.25	1.10
ja9847	**179.20**	490.24	224.11	198.90	2.74	1.25	1.11
kz9976	**180.48**	499.32	226.84	198.45	2.77	1.26	1.10
gr9882	**187.45**	492.58	225.94	200.13	2.63	1.21	1.07
fi10639	**192.61**	540.63	243.17	214.26	2.81	1.26	1.11
mo14185	**257.66**	729.22	336.51	285.45	2.83	1.31	1.11
ho14473	**281.66**	741.17	334.46	290.99	2.63	1.19	1.03
it16862	**317.45**	906.21	387.06	338.79	2.85	1.22	1.07
vm22775	**430.26**	1226.25	546.55	461.74	2.85	1.27	1.07
sw24978	**472.90**	1361.43	592.81	513.24	2.88	1.25	1.09
Média	**169.48**	470.05	213.42	187.39	2.71	1.28	1.16

Tabela 2. Tempos de execução em segundos para cada instância da coleção National TSP.

coluna temos os nomes das instâncias, e estas estão ordenadas pelo número de vértices em cada uma. Notamos que o valor de speedup é maior para instâncias grandes quando comparamos com a brkgaAPI com 1 thread, porém o mesmo não ocorre para a implementação com 8 threads onde os maiores ganhos parecem ocorrer com instâncias menores. Mas mesmo para as instâncias maiores há um speedup significativo quando comparamos a implementação da brkgaCuda versus brkgaAPI com 8 threads. Nós também realizamos testes estatísticos para verificar se os tempos eram significativamente menores na implementação brkgaCuda versus brkgaAPI com 1, 4 e 8 threads. Para todas versões de números de threads o valor de p-value obtido no teste de Wilcoxon foi menor do que 0.01 e portanto brkgaCuda possui tempos de execução significativamente menores.

Por motivos de espaço não apresentamos a tabela de resultados para a coleção de instâncias *VLSI TSP*, mostrando apenas a Figura 2. Nela temos os tempos de execução para o brkgaCuda (cuda), brkgaAPI com 1 thread (cpu-1), com 4 threads (cpu-4) e com 8 threads (cpu-8), onde à esquerda estão os resultados para a coleção *National TSP* e à direita os resultados da coleção *VLSI TSP*. Vemos que também para a coleção *VLSI TSP*, a versão brkgaCuda é sempre mais rápida, mesmo quando comparada com a versão padrão executada com 8 threads. Os speedups médios obtidos da brkgaCuda contra as três versões da brkgaAPI foram: 2.61 para 1 thread, 1.30 para 4 threads e 1.26 para 8 threads.

6. Conclusões

Neste trabalho apresentamos uma nova IPA para o BRKGA utilizando a plataforma CUDA. O BRKGA é uma meta-heurística baseada em algoritmos genéticos e o uso de uma IPA para ela traz uma grande vantagem para usuários, tendo em vista que o trabalho de implementação de uma solução para um problema de otimização fica significativamente reduzido, bastando ao usuário implementar uma função de decodificação, que transforma um vetor de chaves aleatórias (cromossomo) em uma solução do problema. Nós comparamos a nossa IPA, denominada brkgaCuda, com a IPA padrão para

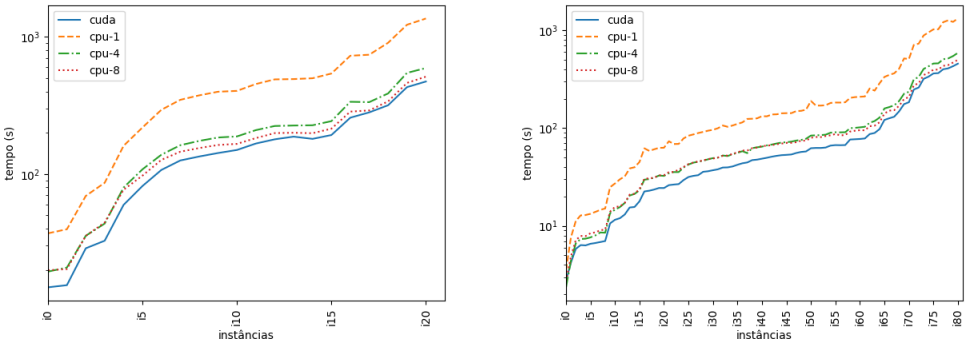

Figura 2. Tempos de execução para a coleção National (esq) e VLSI (dir). Eixo y em escala logarítmica. Instâncias em ordem crescente de tamanho no eixo x.

BRKGA, denominada brkgaAPI. Utilizamos como exemplo de problema a ser resolvido, o problema TSP, e mostramos que a brkgaCuda obteve significativa redução no tempo de execução contra a brkgaAPI, mesmo quando esta última é executada com 8 threads.

7. Agradecimentos

Este trabalho teve apoio financeiro do CNPq (proc. 304856/2017-7, 425340/2016-3) e Fapesp (proc. 2015/11937-9).

Referências

Alba, E., Luque, G., and Nesmachnow, S. (2013). Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research*, 20(1):1–48.

Arora, R., Tulshyan, R., and Deb, K. (2010). Parallelization of binary and real-coded genetic algorithms on gpu using cuda. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE.

Branke, J., Kaußler, T., Smidt, C., and Schmeck, H. (2000). A multi-population approach to dynamic optimization problems. In *Evolutionary Design and Manufacture*, pages 299–307. Springer.

Czapiński, M. (2013). An effective parallel multistart tabu search for quadratic assignment problem on cuda platform. *Journal of Parallel and Distributed Computing*, 73(11):1461–1468.

Czapiński, M. and Barnes, S. (2011). Tabu search with two approaches to parallel flowshop evaluation on cuda platform. *Journal of Parallel and Distributed Computing*, 71(6):802–811.

Dantas, B. D. A. and Cáceres, E. N. (2016). A parallelization of a simulated annealing approach for 0-1 multidimensional knapsack problem using gpgpu. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 134–140. IEEE.

de Almeida Dantas, B. and Cáceres, E. N. (2018). An experimental evaluation of a parallel simulated annealing approach for the 0–1 multidimensional knapsack problem. *Journal of Parallel and Distributed Computing*, 120:211–221.

Feo, T. A. and Resende, M. G. (1995). Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133.

Glover, F. and Laguna, M. (1998). Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer.

Gonçalves, J. F. and Resende, M. G. (2011). Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525.

Gonçalves, J. F. and Resende, M. G. (2012). A parallel multi-population biased random-key genetic algorithm for a container loading problem. *Computers & Operations Research*, 39(2):179–190.

Gonçalves, J. F. and Resende, M. G. (2013). A biased random key genetic algorithm for 2d and 3d bin packing problems. *International Journal of Production Economics*, 145(2):500–510.

Gonçalves, J. F. and Resende, M. G. (2015). A biased random-key genetic algorithm for the unequal area facility layout problem. *European Journal of Operational Research*, 246(1):86–107.

Gonçalves, J. F., Resende, M. G., and Mendes, J. J. (2011). A biased random-key genetic algorithm with forward-backward improvement for the resource constrained project scheduling problem. *Journal of Heuristics*, 17(5):467–486.

Hoberock, J. and Bell, N. (2010). Thrust: A parallel template library. `http://thrust.github.io/`. Accessado: 11-09-2019.

Holland, J. H. (1992). Genetic algorithms. *Scientific american*, 267(1):66–73.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, 220(4598):671–680.

Meffert, K., Rotstan, N., Knowles, C., and Sangiorgi, U. (2012). Jgap-java genetic algorithms and genetic programming package. *URL: http://jgap. sf. net*.

Noronha, T. F., Resende, M. G., and Ribeiro, C. C. (2011). A biased random-key genetic algorithm for routing and wavelength assignment. *Journal of Global Optimization*, 50(3):503–518.

Pospichal, P., Jaros, J., and Schwarz, J. (2010). Parallel genetic algorithm on the cuda architecture. In *European conference on the applications of evolutionary computation*, pages 442–451. Springer.

Takemoto, L. A., de Almeida Dantas, B., and Mongelli, H. (2018). A parallel approach of simulated annealing using GPGPU to solve the quadratic assignment problem. In *Symposium on High Performance Computing Systems, WSCAD 2018, São Paulo, Brazil, October 1-3, 2018*, pages 23–29.

Toso, R. F. and Resende, M. G. C. (2015). A c++ application programming interface for biased random-key genetic algorithms. *Optimization Methods and Software*, 30(1):81–93.

Zhang, S. and He, Z. (2009). Implementation of parallel genetic algorithm based on cuda. In *International Symposium on Intelligence Computation and Applications*, pages 24–30. Springer.