UNIVERSIDADE DO ESTADO DA BAHIA
SISTEMAS DE INFORMAÇÃO
ALLAN ARIEL LEITE MENEZES SANTOS

UTILIZAÇÃO DA TECNOLOGIA CUDA PARA PROCESSAMENTO PARALELO DE ALGORITMOS GENÉTICOS

Salvador
2012

ALLAN ARIEL LEITE MENEZES SANTOS

UTILIZAÇÃO DA TECNOLOGIA CUDA PARA PROCESSAMENTO PARALELO DE ALGORITMOS GENÉTICOS

Monografia apresentada ao curso de Sistemas de Informação da Universidade do Estado da Bahia - UNEB, como pré-requisito para obtenção do grau de bacharel, sob orientação do Prof. Cláudio Alves de Amorim.

Salvador
2012

ALLAN ARIEL LEITE MENEZES SANTOS

UTILIZAÇÃO DA TECNOLOGIA CUDA PARA PROCESSAMENTO PARALELO DE ALGORITMOS GENÉTICOS

Monografia apresentada ao curso de Sistemas de Informação da Universidade do Estado da Bahia - UNEB, como pré-requisito para obtenção do grau de bacharel.

COMISSÃO EXAMINADORA

Prof. Dr. Cláudio Alves de Amorim

Prof. Dr. Eduardo Manuel de Freitas Jorge

Prof. Antônio Marcos Brito de Cerqueira

Salvador, 10 de agosto de 2012

Agradecimentos

À minha família, principalmente ao meus pais Lourivaldo e Cristiany, que sempre me apoiou e me deu condições de poder chegar aonde eu cheguei até hoje.

Ao meu orientador Cláudio Alves de Amorim por ter acreditado em mim, pela valiosa contribuição na monografia e pela excelente qualidade de ensino que presta em suas disciplinas, assim como os demais professores da UNEB.

Ao meu amor Fernanda por me entender e me motivar na conclusão do curso.

Aos meus colegas da UNEB, principalmente a Marcos Cézar, Bartira de Oliveira, Felipe Pineiro e Raul Cezar, entre tantos outros que contribuíram de alguma forma tanto em minha formação acadêmica, profissional e como pessoa, além dos momentos de descontração.

Por fim, agradeço aos demais funcionários da UNEB que pude ter o prazer de conviver, como Ana e Márcia, além de Débora. Todos se fizeram importantes de alguma forma e guardarei a lembrança dos momentos vividos.

Resumo

Este trabalho apresenta uma validação da utilização da tecnologia CUDA para otimização de problemas combinatórios. Para tal, foram realizadas duas abordagens de solução com uso de algoritmos genéticos para o problema do caixeiro viajante (campo de testes e resultados). Foram elaborados dois procedimentos para realizarem o cálculo do *fitness*, função que representa o maior gargalo em tempo de execução: um sequencial, com rotinas executadas pela CPU; um paralelo, o qual faz uso de uma GPU para os cálculos referentes à aptidão dos cromossomas através do recurso tecnológico da NVidia chamado CUDA (*Compute Unified Device Architecture*). Os resultados obtidos levaram em consideração o tamanho da população de cromossomas e a quantidade de cidades presentes no Algoritmo Genético e no Problema do Caixeiro Viajante: para até 500 cidades e população de 100 cromossomas, abordagem sequencial apresenta melhor desempenho e, para valores consideravelmente maiores, como 10.000 cidades e 500 cromossomas, a abordagem paralela é a mais indicada.

Palavras-chave: Algoritmos genéticos. Problema do caixeiro viajante. Otimização. NVidia CUDA.

Abstract

This paper presents a validation of the use of CUDA technology for combinatorial optimization problems. For this, there were two approaches to the solution with the use of genetic algorithms for the traveling salesman problem (field tests and results). Two procedures to perform the calculation of fitness function that represents the biggest bottleneck at runtime: a sequentially routines executed by the CPU; a parallel, which makes Use of a GPU for calculation concerning the ability of chromosomes through the use of the NVidia technology called CUDA (Compute Unified Device Architecture). The results take into consideration the size of the population of chromosomes and the number of cities present in the AG and the Traveling Salesman Problem: Up to 500 cities and population of 100 chromosomes, sequential approach has best performance. To values considerably higher, as 10.000 cities and 500 chromosome, the parallel approach is more appropriate.

Keywords: Genetic algorithms. Travelling salesman problem. Otimization. NVidia CUDA.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Quadros

1 Introdução

O Problema do Caixeiro Viajante (ou *Travelling Salesman Problem* - TSP ou PCV, em português) é um clássico da Inteligência Artificial e atrai a atenção de diversos pesquisadores e resume-se em fazer o caixeiro visitar todas as cidades envolvidas na problemática percorrendo a menor distância possível e sem passar por uma cidade que já foi visitada, a não ser no fim, em que retorna-se à cidade de origem. Muitos estudiosos tentam buscar formas de solucionar este problema, principalmente quando as instâncias das variáveis possuem valores cada vez maiores. Sob a ótica de otimização, o PCV pertence à categoria conhecida como NP-difícil (do inglês "NP-hard"), o que significa que possui ordem de complexidade exponencial. Em outras palavras, o esforço computacional para a sua resolução cresce exponencialmente com o tamanho do problema (CUNHA et al, 2002).

Algoritmo Genético (cuja sigla é A.G.) é uma técnica de busca altamente paralelizada pertencente à área da Inteligência Artificial que se baseia nos princípios da Teoria da Evolução de Charles Robert Darwin (1809 – 1882), encontrando soluções aproximadas para problemas complexos de otimização (GOLDBERG, 1989). Surgiu graças ao americano John Holland e seus colegas de trabalho, que desenvolveram este tipo específico de algoritmo englobando conceitos da biologia evolutiva. A utilização de algoritmos genéticos no problema do caixeiro viajante culmina em soluções aproximadas da solução ideal e que satisfazem a problemática. O uso de Algoritmos Genéticos pode exigir muito do processamento da CPU (*Central Processing Unit*) de um computador, já que as rotinas de um A. G. podem ser executadas diversas vezes e as variáveis envolvidas podem possuir altos valores.

Mesmo com o surgimento de processadores multi-núcleos, estes ainda não são suficientes para atender os casos mais críticos de uso de A. G, exigindo novas estratégias de processamento. Tais limitações de hardware se apresentaram, portanto, como grandes desafios a serem enfrentados. Uma solução está na forma como essas informações são trabalhadas. A alternativa em questao é o uso da Programação Paralela, podendo reduzir o tempo de processamento de algoritmos complexos, obtendo resultados mais precisos e satisfatórios. A estratégia desenvolvida para lidar com esta problemática é a de retirar a sobrecarga de processamento na CPU e

transferí-la para uma GPU (*Graphics Processing Unit*) multi-núcleo, ou seja, entra-se no conceito de GPGPU (*General-Purpose Computing on Graphics Processing*), em outras palavras processamento de propósito geral sobre unidades de processamento gráfico.

Comparando-se o processamento das GPU's da NVIDIA e dos processadores Intel em relação às operações de ponto flutuante, as GPU's apresentam um poder computacional maior: um processador Core2 Duo de 3.0 GHz corresponde a 50 GFLOPS, enquanto uma GPU NVIDIA GT200 chega a 900 GFLOPS (YANO, 2010). Devido a este comparativo onde são apresentadas algumas das vantagens de GPUs em cima de CPUs, a NVidia desenvolveu a tecnologia CUDA para permitir o uso das centenas (em alguns casos, dos milhares) de núcleos existentes nas placas de vídeo como processadores paralelos permitindo a execução paralela das instruções que lhe são enviadas. Desta forma, tais placas não ficam restritas à área dos jogos eletrônicos e outros gráficos, passando a serem utilizadas com propósitos mais gerais (GPGPU), auxiliando, por exemplo, no desempenho dos resultados de Algoritmos Genéticos em ação.

Este trabalho tem como objetivo avaliar o potencial da arquitetura CUDA para a otimização de problemas combinatórios, utilizando como campo de teste os Algoritmos Genéticos e o Problema do Caixeiro Viajante. Para isso, serão utilizadas implementações sequencial e paralela de Algoritmos Genéticos buscando otimizar o procedimento de obtenção do fitness, maior gargalo.

Esta monografia está organizada em cinco capítulos. O capítulo 1 apresenta uma breve contextualização e introdução ao problema de otimização de algoritmos genéticos, em relação à forma de processamento da informação.

O capítulo 2 expõe brevemente o problema do caixeiro viajante, algoritmos genéticos, programação paralela e a tecnologia CUDA da NVidia, componentes do referencial teórico de pesquisa.

O capítulo 3 aborda sobre trabalhos relacionados na mesma área de otimização e com paralelismo e uso de GPUs.

A implementação do algoritmo genético, em suas duas abordagens: sequencial e paralela, é descrita no capítulo 4.

Em seguida esta a conclusão, assim como trabalhos futuros.

2 *TSP, Algoritmos Genéticos, Programação Paralela e CUDA*

2.1 O Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante (ou *Travelling Salesman Problem* - TSP) é um dos mais conhecidos na área de otimização combinatória (HOFFMAN e WOLFE, 1985; MELAMED et al., 1990). São dadas as cidades e as distâncias entre elas, e o caixeiro precisa visitar todas as cidades na menor rota possível. Em outras palavras, busca-se a sequência de cidades que possui a menor distância percorrida possível em um grafo que envolve N nós.

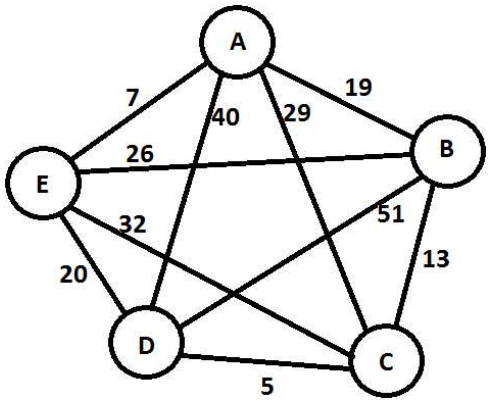

Figura 1: Grafo de cidades e distâncias.

A figura 1 apresenta graficamente como seria uma organização das cidades, representadas por letras. Os números correspondem às distâncias entre elas. Sem passar duas ou mais vezes por uma mesma cidade, o caixeiro viajante precisa visitar cada uma, formando assim uma rota. A soma das distâncias percorridas entre as cidades culmina no total percorrido e, dentre as possíveis rotas existentes, o caixeiro viajante busca aquela que possuir o menor gasto possível de esforço.

O PCV é um problema de fácil entendimento, porém de difícil solução, e a dificuldade do problema se torna visível quando consideramos o número de possíveis tours (HELSGAUN,

Tabela 1: Comparativo entre a possibilidades de rotas e o número de cidades

Nº de Cidades	Possibilidades	Total de Rotas Possíveis
10	10!	3,62 x 10^{6} / 2
25	25!	1,55 x 10^{25} / 2
50	50!	3,04 x 10^{64} / 2
75	75!	2,48 x 10^{109} / 2
100	100!	9,33 x 10^{157} / 2
125	125!	1,88 x 10^{209} / 2
150	150!	5,71 x 10^{262} / 2

1999 apud GAZOLLA, 2010). A tabela 1 apresenta o número possível de rotas, dependendo do valor total de cidades envolvidas:

Está presente no conjunto que engloba os problemas NP-Completos, não possuindo solução através de algoritmos polinomiais. Contudo, com o uso de Algoritmos Genéticos, é possível obter soluções aproximadas para este problema e que satisfazem como resultado. Segundo Papadimitriou e Steiglitz (1998):

- Problemas NP-Completos não podem ser exatamente solucionados por nenhum algoritmo polinomial conhecido, dado um problema cuja população inicial é realisticamente grande.
- Se existe uma solução para um problema NP-Completo, então existem algoritmos polinomiais para todos os problemas NP-Completo.

Aplicar métodos exatos isoladamente para sua solução é inviável para instâncias grandes, sendo, portanto utilizados métodos heurísticos ou aproximados (GAZOLLA, 2010).

Inúmeros problemas reais são modelados como problemas do tipo caixeiro viajante ou suas variantes. Consequentemente, existe uma importante necessidade de novos algoritmos de solução (DA CUNHA, 2002). Entre esses problemas pode-se citar o problema de produção que corresponde ao sequenciamento de n tarefas em uma única máquina, de forma a minimizar o tempo total de execução das mesmas. Em linhas de montagem de componentes eletrônicos busca-se encontrar, por exemplo, o roteiro de mínima distância para um equipamento cuja tarefa é soldar todos os componentes de uma placa eletrônica. O menor percurso total do equipamento para percorrer todos os pontos da placa está diretamente associado à produtividade da linha (SOUZA, 1993 apud DA CUNHA, 2002, p. 1).

2.2 Algoritmos Genéticos

Os Algoritmos Genéticos seguem o princípio da seleção natural e sobrevivência do mais apto (LACERDA e CARVALHO, 1999). De acordo com o inglês **Charles Darwin** em seu livro "A Origem das Espécies"(1859), quanto melhor um indivíduo se adaptar ao seu meio ambiente, maior é sua chance de sobreviver e gerar descendentes. Tais algoritmos estão precisamente na área da Computação Evolucionária, criada por **I. Rechenberg** em 1960 com seu trabalho *Estratégias de Evolução* (*Evolutionsstrategie* no original) (OBITKO, 1998).

Os elementos existentes no problema do caixeiro viajante podem ser associados aos fundamentos biológicos de Darwin, entre os quais estão: gene, cromossoma, população, seleção, cruzamento, elitismo, mutação e geração. Genes são blocos de DNA responsáveis pelas características de um ser vivo. Os genes de um Algoritmo Genético podem armazenar informações sobre as cidades envolvidas: cada gene representa uma cidade a ser visitada, sem repetição. O conjunto de genes representa um Cromossomo, que armazena todas as cidades, representando uma possível solução do problema, não necessariamente a melhor delas. Cada Cromossomo possui um tempo de custo referente à distância entre as cidades (genes) envolvidas, sem repetição, o que representa o *Fitness*. Uma população reúne diversos Cromossomas, aglomerando diferentes soluções para o problema, desta forma.

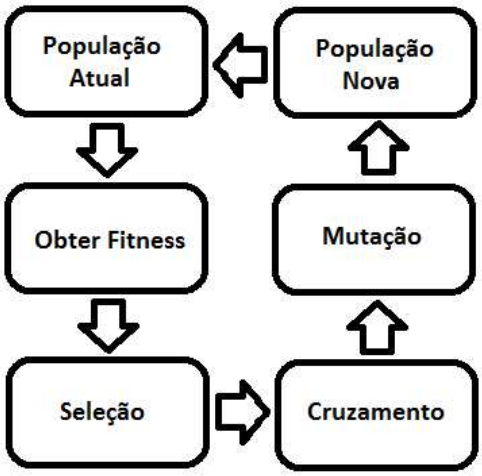

Figura 2: Ciclo de um Algoritmo Genético.

A figura 2 revela o ciclo padrão de um Algoritmo Genético, em que instancia-se a população inicial com cromossomas aleatórios e calcula-se a aptidão (*Fitness*) de cada cromossoma. A seleção dos Cromossomos acontece levando em consideração os mais aptos para a problemática em questão. Armazena-se o(s) Cromossomo(s) com maior aptidão (*Fitness*), fenômeno conhecido como Elitismo, e os demais sofrem o processo chamado Cruzamento, que consiste em dividir cada um deles em partes que serão trocadas dois a dois, obterendo rotas diferentes e,

consequentemente, novos valores de *Fitness*, ampliando assim as chances de que uma solução melhor possa aparecer. A mutação pode ocorrer em cada um dos membros de uma População, onde um Gene sofre alteração do seu valor, contribuindo assim para o surgimento de novas soluções.

Uma geração está representada por um ciclo do Algoritmo Genético: dada uma População com possíveis soluções, calcula-se a distância total entre as cidades de acordo com cada configuração de rota que cada Cromossoma possui, obtém-se o valor da menor distância e cruzam-se os demais Cromossomas. Com o avanço das gerações, as soluções que serão encontradas se aproximarão da melhor solução possível, que é a rota com a menor distância percorrível.

Quanto maior o valor destas variáveis envolvidas, maior é a demanda de processamento e consequentemente maior é o tempo de execução do algoritmo genético. Porém, com o avanço tecnológico dos dias de hoje é possível utilizar diversos recursos capazes de diminuir este tempo. Um deles consiste em paralelizar o algoritmo e executá-lo em uma placa de vídeo que suporte a tecnologia CUDA.

2.3 Programação Paralela

Nesta seção serão abordados os temas de Programação Paralela e GPGPUs.

A taxonomia de Flynn (1966) é uma das metodologias mais difundidas para classificar a arquitetura de computadores, levando em consideração duas dimensões independentes: instruções e dados. Cada uma destas dimensões pode apresentar o valor *single* ou *multiple*, gerando assim uma tabela qualificando quatro arquiteturas, como apresenta a figura 3.

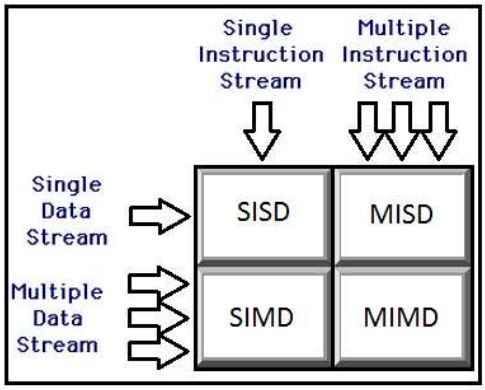

Figura 3: Taxonomia de Flynn (1966).

A programação paralela, por definição, divide a execução de um programa em partes, e cada parte é processada através de uma *thread* (YANO, 2010). Segundo Barney (2009), "*Thread* é

um fluxo de execução independente que possui uma memória compartilhada com o processo pai e que pode ser escalonada pelo sistema operacional". Na taxonomia de Flynn, está representada pelas arquiteturas SIMD (*Single Instruction, Multiple Data*) e MIMD (*Multiple Instruction, Multiple Data*).

Os dispositivos de processamento gráfico possuíam limitação de uso, sendo restritos apenas para renderizar, graficamente, jogos, vídeos ou similares. Porém com o avanço tecnológico tornou-se possível explorar melhor o potencial que as placas gráficas possuem, abrangendo sua utilização para diversas áreas jamais alcançadas. Surge então o conceito de GPGPU (*General-Purpose Computing on Graphics Processing*), que nada mais é do que realizar processamentos (antes possíveis apenas em CPUs) em dispositivos GPUs, eliminando a fronteira gráfica ao qual estavam fadados.

Pode-se analisar a arquitetura das CPUs e GPUs na figura 4. A GPU dispõe de todas as unidades que a CPU dispõe: Unidade de Controle, Unidades de Lógica Aritmética (ALUs), Cache e Dynamic Random Access Memory (DRAM), porém grande parte de toda área do chip foi destinada a inúmeras ALUs, o que faz com que a GPU seja uma máquina essencialmente preparada para muitos cálculos (GAZOLLA, 2010).

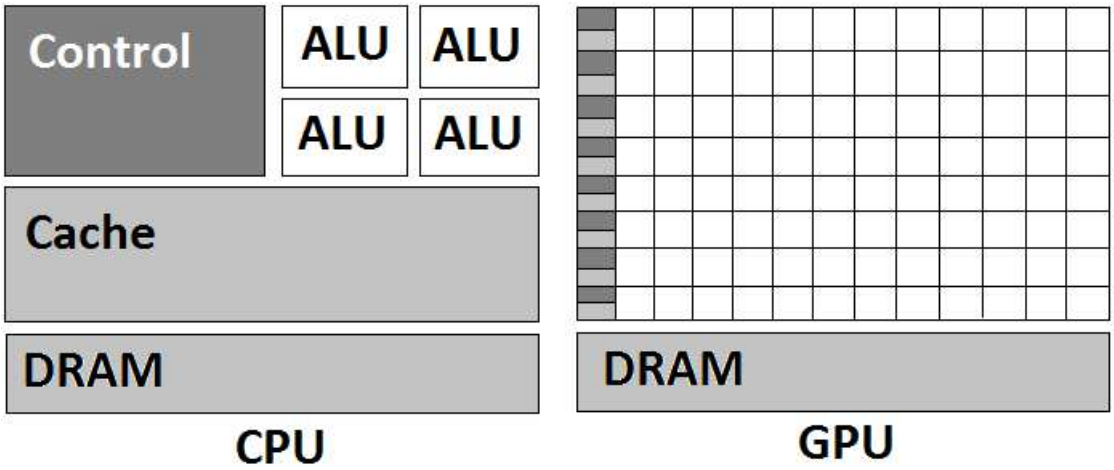

Figura 4: A GPU dedica mais transistores para processamento de dados, adaptado de NVIDIA (2012).

Com programação paralela a aplicação é desenvolvida para aproveitar o paralelismo desde o algoritmo. A vantagem dessa abordagem é que ela fornece maior ganho de desempenho, mas a desvantagem é que exige um maior esforço no desenvolvimento da aplicação (YANO, 2010). As plataformas de programação paralela existentes passaram por avaliação baseada nos sete critérios quantitativos propostos pela APSTC - Asia-Pacific Science and Technology Center (2008). São estes:

- Arquitetura de sistema: pode ser de memória compartilhada que se refere a sistemas na qual o processador utiliza uma área de memória compartilhada (compartilham dos mesmos endereços de acesso a memória), ou de memória distribuída que se refere aos casos em que cada nó de processamento tem seu próprio espaço de memória não compartilhado por outros.

- Metodologia de programação: de que forma os programadores conseguem utilizar os recursos de paralelismo. Por ex.: API, nova linguagem, diretivas especiais.

- Gerenciamento de trabalho: o paralelismo pode ocorrer através de processos ou de *threads*. Caso programador precise cuidar da criação e destruição de *threads* dizemos que o gerenciamento de trabalho é explícito, ou então ele é implícito, e basta especificar a seção de código que vai rodar em paralelo.

- Esquema de particionamento da carga de trabalho: A carga de trabalho que será executada é divida em pequenas porções chamadas tarefas, no critério implícito os programadores precisam apenas especificar qual carga de trabalho vai ser processada em paralelo sem se preocupar em gerenciar isso. Enquanto no critério explícito os programadores precisam decidir manualmente como essa carga de trabalho será divida.

- Mapeamento entre tarefa e a *thread* ou processo: no critério implícito o programador não precisa especificar qual *thread*/processo é responsável pela tarefa. Já no explícito gerenciar isso é responsabilidade do programador.

- Sincronização: define em que sequência as *threads*/processos acessam os dados que compartilham. Na sincronização implícita não há esforço necessário do programador, ou este é mínimo e não é necessário, ou simplesmente basta especificar que naquele trecho de código ocorrerá uma sincronização, enquanto na sincronização explícita os programadores precisam gerenciar como se dará o acesso dos processos e *threads* nesta área compartilhada.

- Modelo de comunicação: Este modelo foca no paradigma de comunicação utilizado por um modelo.

Com GPUs que possuem centenas de núcleos e capacidade de processamento cada vez maiores, aliado ao conceito de GPGPU, APIs que permitem o desenvolvimento de aplicações paralelizáveis surgiram e uma delas será abordada na próxima seção.

2.4 NVidia CUDA

Nesta seção será abordada a tecnologia CUDA da NVidia em aspectos como arquitetura, esquema de memórias e organização das *threads*, blocos e *grids*.

O CUDA é uma plataforma que utiliza software e hardware para computação paralela de alto desempenho de propósito geral que utiliza o poder de processamento dos núcleos das GPU's's da NVIDIA (YANO, 2010). A arquitetura SIMD é a que mais se assemelha com o o SIMT (*Single Instruction Multiple Thread*) utilizada pelo CUDA, onde há apenas um controlador que dispara a instrução para ser executada por todas as unidades de processamento. Há algumas fontes que afirmam o uso do modelo SIMD pelo CUDA, como FATAHALIAN (2009).

Sua arquitetura consiste nos componentes representados pelos números 1, 2 e 3 na figura 5.

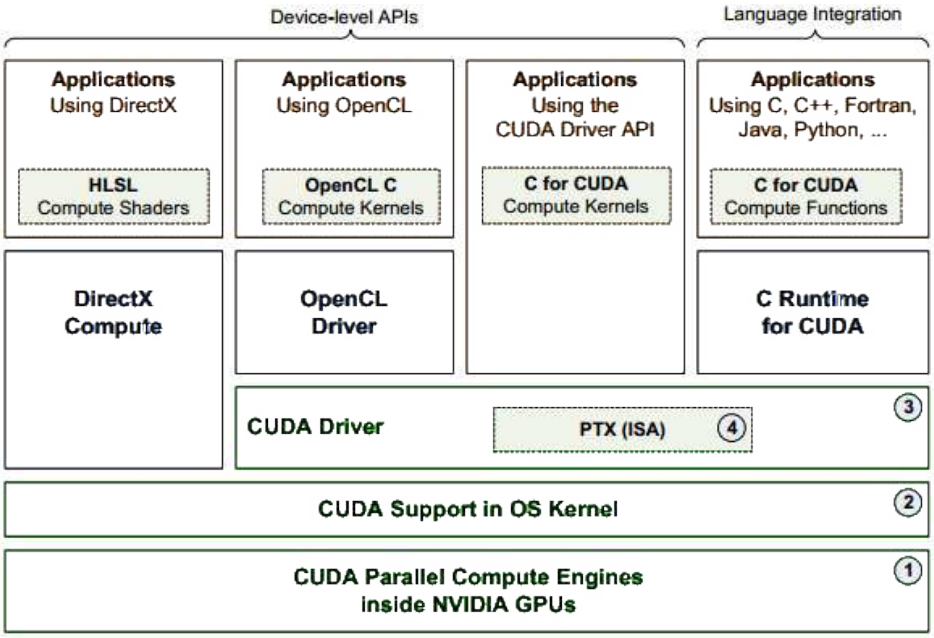

Figura 5: Arquitetura CUDA, adaptado de NVIDIA (2009).

O item 1 corresponde às GPUs, placas aceleradoras gráficas detentoras de tecnologia para processamento paralelo de informações. A camada de número 2 representa o suporte que o CUDA oferece a nível de núcleo (*kernel*) na CPU. o item de valor 3 é o driver do CUDA, com o qual é possível utilizar uma API de desenvolvimento para programação paralela utilizando GPUs. O item 4 é o *PTX instruction set architecture (ISA)* para definição de regras e paralelismo em núcleos e funções.

Possui suporte à computação heterogênea, nas quais as partes seriais do programa são executadas na CPU e as partes em paralelo na GPU (GAZOLLA, 2010). Acrescenta extensões da linguagem C, como qualificadores de tipo de variável, qualificadores de tipo de função, variá-

veis internas de acesso a índices e dimensões de grid, blocos e *threads*, além de nova sintaxe de chamada de função permitindo configuração dos blocos e grid. Como a figura 6 apresenta, um grid possui diversos blocos, que podem ser organizados em ate três dimensões. Cada bloco possui um conjunto de *threads* que podem ser organizadas tambem em três dimensões, sendo ao máximo 512 *threads* por bloco na maioria das placas de vídeo pertencentes à NVidia.

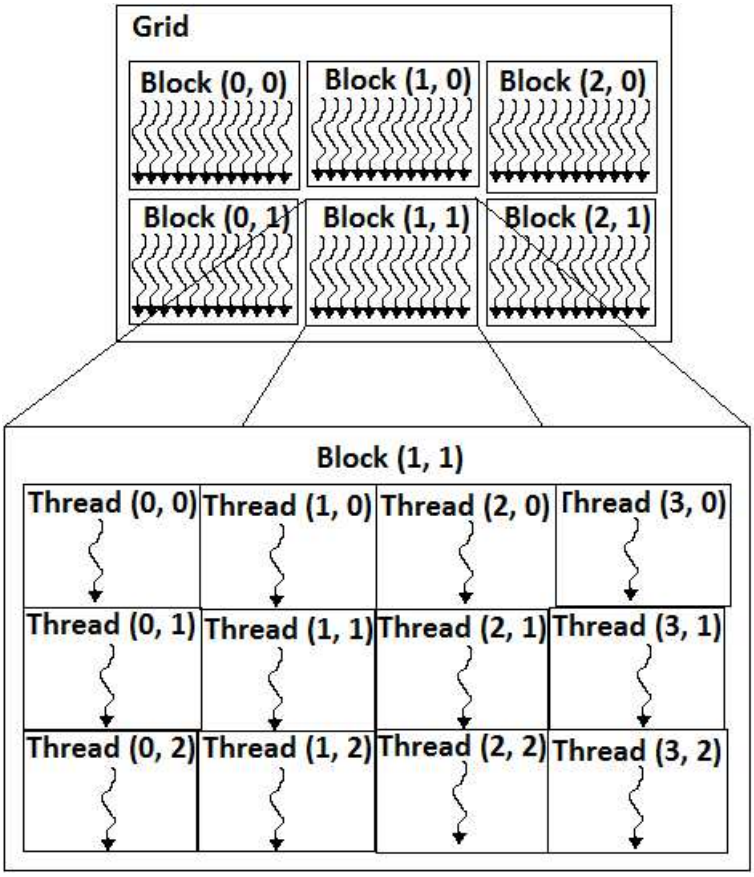

Figura 6: Esquema de organização de um *grid*, adaptado de NVIDIA (2012).

Dentre os qualificadores de tipo de variável, existe: o qualificador __device__ que define uma variável oriunda da memória global da placa de vídeo; o qualificador __constant__ que declara uma variável na memória constante da GPU; o qualificador __shared__ que representa uma variável na memória compartilhada da GPU, sendo acessadas apenas por *threads* do mesmo bloco.

Os qualificadores de tipo de função são três: o __device__ representa uma função que apenas é iniciada e executada pela GPU; o __global__ é conhecido também como kernel e é iniciado pela CPU e executado pela GPU; o __host__ indica que uma função apenas é iniciada e executada na CPU.

Com tais recursos, o CUDA torna-se uma ferramenta em potencial para otimização de códigos extremamente paralelizáveis, tornando eficiente códigos paralelos em grandes instâncias.

3 *Trabalho Relacionado*

A seguir, será apresentado um trabalho que já foi realizado e serviu de contribuição para esta monografia. Tal trabalho buscou otimizar o problema do caixeiro viajante com um algoritmo genético paralelizado.

3.1 A Abordagem Heuristica e Paralela em GPUs para o Problema do Caixeiro Viajante de Gazolla

Gazolla (2010) propôs um modelo de implementação de algoritmo paralelo e metalinguístico em GPU, análogo ao proposto por Subramanian et al (2010). Foi utilizado, como entrada de dados, um arquivo pertencente à biblioteca TSPLIB que contém instâncias do Problema do Caixeiro Viajante. Após o arquivo ser carregado em memória, é calculada a distância euclidiana entre todas as cidades resultando em uma matriz de tamanho n^2 (GAZOLLA, 2010).

As soluções iniciais para o problema foram geradas através do uso de três funções heurísticas construtivas: Aleatória, GRASP e Gulosa.

A construção de soluções de forma aleatória é representada pela figura 7.

procedimento $ConstrucaoAleatoria(g(.), s)$;
1 $s \leftarrow \emptyset$;
2 Inicialize o conjunto C de elementos candidatos;
3 enquanto $(C \neq \emptyset)$ faça
4 Escolha aleatoriamente $t_{escolhido} \in C$;
5 $s \leftarrow s \cup \{t_{escolhido}\}$;
6 Atualize o conjunto C de elementos candidatos;
7 fim-enquanto;
8 Retorne s;
fim $ConstrucaoAleatoria$;

Figura 7: Pseudocódigo da heurística de construção aleatória. **Fonte:** Gazolla, 2010

A heurística de construção gulosa é detalhada na figura 8.

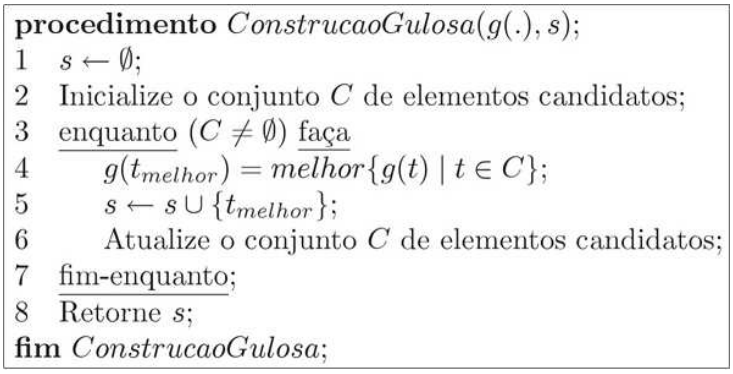

Figura 8: Pseudocódigo da heurística de construção gulosa. **Fonte:** Gazolla, 2010

A figura 9 refere-se à heurística construtiva do *Greedy Randomized Adaptive Search Procedure* (GRASP).

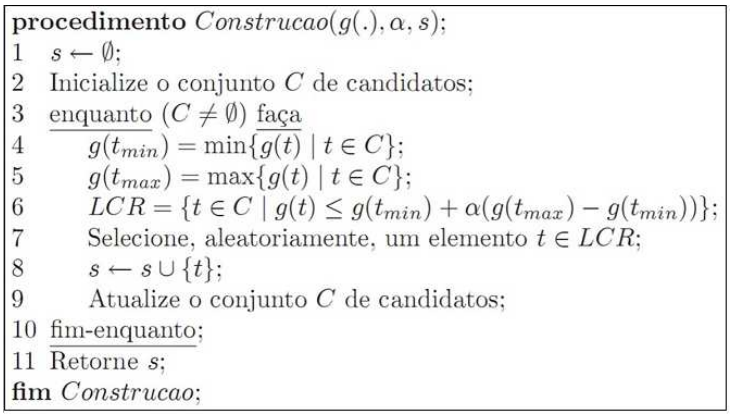

Figura 9: Pseudocódigo da heurística de construção GRASP. **Fonte:** Gazolla, J. G. F. M.

Em seguida, utiliza-se da perturbação *double-bridge* para desconectar quatro arestas de uma solução do PCV e as reconecta de forma diferente, segundo Gazolla (2010). Buscas locais foram implementadas, como os algoritmos 2-Opt, Swap, Or-Opt-1 , OrOpt-2 e OrOpt-3, associados ao uso de GPU para obtenção de resultados mais eficazes.

4 *Solução do T.S.P. em Duas Abordagens de Algoritmos Genéticos*

Segundo Gazolla (2010), a utilização de um grande poder computacional não diminui a complexidade de um problema NP-Completo, seja de qual ordem for: quadrática, cúbica ou exponencial, mas viabiliza o uso destes algoritmos para instâncias maiores, quando o tamanho do problema é alto. O Problema do Caixeiro Viajante pode ser solucionado através da implementação de um algoritmo genético, culminando em soluções aproximadas (em relação à melhor solução possível) que satisfazem a problemática.

Quando o número de cidades envolvidas e o de populações tornam-se elevados, a demanda por processamento torna-se maior. Chega-se em um momento no qual aumentar o poder de processamento da CPU não é proporcional à diminuição do tempo de execução do algoritmo. Um dos jeitos de eliminar este gargalo está na forma como estes dados são trabalhados. Com o uso de técnicas de programação paralela e hardware especifico, é possivel otimizar o algoritmo genético consideravelmente.

Neste cenário, as GPUs tornam-se candidatas para área de pesquisa operacional, análise combinatória e otimização, ajudando na obtenção de soluções de melhor qualidade em um menor tempo. Ou seja, acelerando a solução de problemas complexos de otimização do mundo real que muitas vezes não podem ser resolvidos em tempo polinomial (GAZOLLA, 2010).

Este capítulo apresenta detalhadamente a solução proposta para paralelizar o algoritmo genético.

4.1 Códigos para Análise

O código fonte do algoritmo genético, da função paralela e rotinas CUDA implementadas encontram-se na seção de anexos da monografia.

4.2 Linhas Gerais

Baseado no que foi previamente abordado, o objeto da pesquisa é validar o uso da tecnologia CUDA para otimização de problemas combinatórios. Em específico, usou-se o problema do caixeiro viajante como campo de testes e algoritmos genéticos para solucionar tal problema. Buscou-se otimizar o processo de cálculo do *fitness* no Algoritmo Genético através do uso de programação paralelizada para grandes instâncias do problema. A função paralela foi executada por um acelerador gráfico NVidia GeForce GT 220 que possui 48 núcleos CUDA.

O desenvolvimento do projeto se baseou em implementar um Algoritmo Genético que solucionasse o Problema do Caixeiro Viajante utilizando as linguagens de programação C/C++. O algoritmo programado consta de implementação sequencial e uma função paralelizada via CUDA, que corresponde ao cálculo do *fitness* de cada cromossoma, o qual representa o maior consumo de recursos da CPU no algoritmo genético.

O algoritmo genético foi construído respeitando os conceitos e o fluxo natural de execução. Inicialmente constatou-se que algumas de suas rotinas não estavam otimizadas. Houve dificuldade de encontrar trabalhos correlatos e que possuíssem código-fonte completo, aberto à consulta e estudos. Não existe uma gama variada de trabalhos envolvendo programação paralela e utilização do CUDA para otimização do problema do caixeiro viajante.

Um caminho investigado nesta pesquisa que não é recomendado seria o uso de estruturas de dados ou orientação a objetos no tentativa de facilitar a programação. As estruturas de dados tornam mais complexos os mapeamentos de memória da CPU na GPU, assim como a orientação a objetos. Os testes não foram feitos pois diversos erros em tempo de compilação ocorreram devido às regras do CUDA.

Muitos erros de referência às áreas de memória aconteceram em tempo de execução, forçando à uma reestruturação do código fonte. Considerando-se o trabalho na área de otimização, não deve-se focar na facilidade de implementação: mas sim nos resultados obtidos, o que revelaram os resultados obtidos através da programação procedural (sem utilização de orientação a objetos) empregada neste projeto.

Outra dificuldade encontrada no projeto foi testar se o kernel estava obtendo os resultados esperados. Para tal, foi utilizado o "cuPrintf", função semelhante ao "printf"nativo da linguagem C, porém com foco no *output* das GPUs. Este código fonte não está incluso na API do CUDA, sendo necessário buscar na internet maiores informações sobre tal código e como utilizá-lo. Com posse do código fonte em mãos, diversos erros, desconhecidos até então, aconteciam ao compilar o projeto.

4.3 Barreiras Encontradas

Ao contrário da linguagem C que possui uma biblioteca padrão com uma série de estrutura de dados, a linguagem CUDA não dispõe de nenhuma estrutura de dados ou biblioteca nativa disponível, sendo tudo manipulado pelo programador (GAZOLLA, 2010). Recomenda-se a implementação da forma mais simples possível, com primitivas e tipos simples.

Gazolla (2010) ainda afirma que "apesar da CPU e GPU estarem localizadas no mesmo barramento, a comunicação é algo necessário para o trabalho cooperativo entre elas, porém, é um gargalo, e toda comunicação deve ser minimizada para um ganho de desempenho nos algoritmos". Um recurso que foi utilizado neste trabalho foi o uso de *pinned memory*, que permite uma transferência de dados da CPU para GPU de forma mais rápida, com uso planejado para não implicar na perda de desempenho.

Mais uma dificuldade que foi encontrada refere-se à alocação dinâmica de memória, não sendo possível alocar ponteiros para a GPU. É necessário que, previamente, todo o espaço de memória seja alocado para posteriores utilizações.

4.4 Visão Geral e Modelo de Desenvolvimento

A estruturação do projeto é composta por quatro arquivos: Attributes.h, FunctionsCPU.h, FunctionsGPU.cpp e Main.cu, ou seja, dois *headers* e um arquivo de extensão .cpp que são compilados pelo compilador C/C++. Os arquivos com extensão .cu são compilados pelo compilador nvcc distribuído pela NVIDIA segundo Yano (2010). A placa gráfica utilizada neste projeto foi a geForce GT 220, de 48 núcleos CUDA.

Em Attributes.h, estão os valores referentes à distância máxima de uma cidade até a outra (98 km neste caso), o percentual de cruzamento dos genes de cada cromossoma (definido como 50%), o número de cidades que serão visitadas pelo caixeiro (10.000 cidades), o tamanho da população de cromossomas (variando entre 200 e 500), número de gerações (definido inicialmente em 1), o máximo de *threads* por bloco (de acordo com as especificações da GeForce GT 220, são 512 *threads*), além da estrutura de um cromossoma, composta pelo *fitness* e por um array de genes de tamanho correspondente ao número de cidades.

No arquivo FunctionsCPU.h estão as declarações das funções que compõem o algoritmo genético e das funções que auxiliam no funcionamento e entendimento do problema. Tais funções permitem: preencher a matriz de cidades; criar cada cromossoma com genes aleatórios ou uma população com cromossomas randômicos; verificar se um cromossoma já possui um

determinado gene; conferir se uma população já possui um determinado cromossoma ou se dois cromossomas são iguais (portanto se eles possuem a mesma sequência de genes); imprimir cada gene de um cromossoma no console ou cada cromossoma no console; calcular o *fitness* de cada cromossoma via CPU; selecionar os dois cromossomas de maior aptidão e cruzar os cromossomas restantes.

O FunctionsGPU.cpp corresponde às implementações das funções acima descritas.

Em Main.cu está a função que calcula o *fitness* de forma paralela utilizando os recursos da placa de vídeo, além da função main onde é executado o algoritmo genético.

4.5 Organização das *threads*

No algoritmo genético foi feita uma análise com objetivo de paralelizar a função fitness de forma eficiente. Com isso, cada *thread* ficou responsável por um par de cidades (representando a cidade de partida e a de chegada) de cada cromossoma presente na população, totalizando uma quantidade de *threads* correspondentes ao tamanho da população multiplicado pela quantidade de cidades, como apresenta a figura 10.

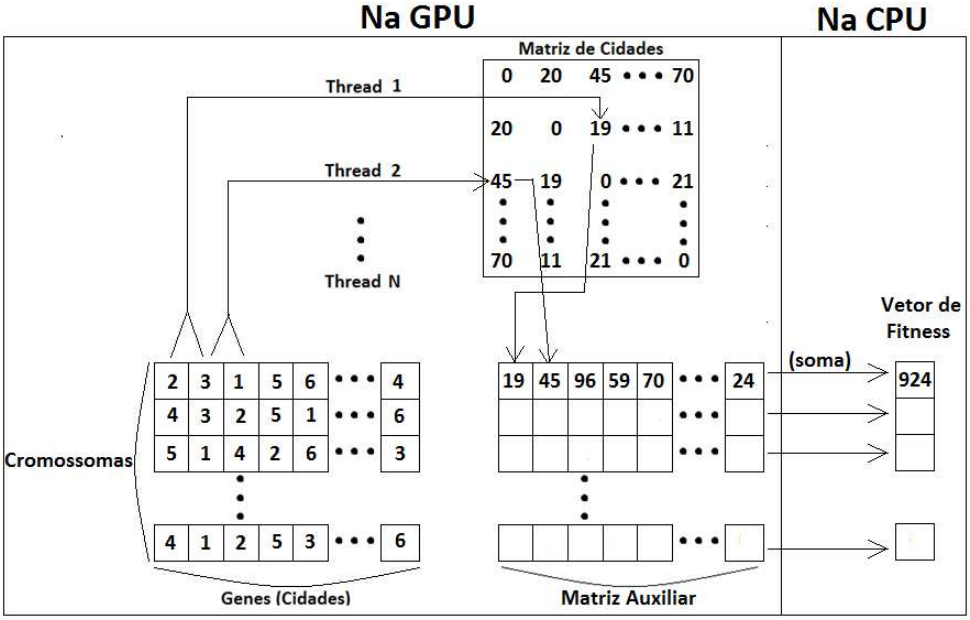

Figura 10: Esquema de organização das *threads*.

Paralelamente, cada *thread* utilizada fica responsável por buscar na matriz de cidades o valor referente à associação dos pares de cidades envolvidas. A linha na matriz de cidades é encontrada com o valor da cidade de saída. A coluna é buscada de acordo com o valor da cidade

de chegada. Desta forma, o valor encontrado através deste índice representa a distância entre o par de cidades. Em seguida, esta distância é armazenada na matriz auxiliar na posição referente à mesma que a cidade de origem apresenta no vetor de genes do cromossoma. O último passo é acessar esta matriz auxiliar na CPU, fazer o somatório de todos os valores presentes em uma mesma linha da matriz (cada linha corresponde à um cromossoma) e atribuir o valor total de cada linha ao *fitness* do cromossoma correspondente. Sendo uma população de 500 cromossomas e 10.000 cidades, haverá então cinco milhões de *threads* e a rotina de obtenção do *fitness* na GPU será partilhada e executada simultaneamente para cada par de cidades pertencente à cada cromossoma da população.

4.6 Implementação das funções *fitness*

Como dito anteriormente, a diferença entre o algoritmo sequencial e o paralelizado está na implementação da função *fitness*, que sofre alterações que referem-se às regras do CUDA. Na figura 11, observa-se a estrutura da implementação da função generateFitnessCPU, que calcula o esforço que será feito pelo caixeiro viajante em cada uma das soluções (cromossomas) existentes no *array* de população.

```
void generateFitnessCPU(Chromosome *population, int *hCityArray){
    int start = 0;
    int end = 0;
    for(int i = 0; i < numPopulation; i++){
        population[i].fitness = 0;
        for(int j = 0; j < numCitys - 1; j++){
            start = population[i].genes[j] - 1;
            end = population[i].genes[j + 1] - 1;
            population[i].fitness += hCityArray[start*numCitys + end];
        }
        population[i].fitness += hCityArray[end*numCitys + (population[i].genes[0]-1)];
    }
}
```

Quadro 1: Função de calcular *fitness* sequencialmente na CPU.

De acordo com a figura 11, observa-se que, para cada elemento da população, percorre-se o *array* de cidades considerando a cidade de partida (representada pela variável *start*) e a cidade de chegada (representada pela variável *end*). Para cada par de cidades, o valor da sua distância é adicionado à variável *Fitness* correspondente, totalizando no final a distância que será percorrida pelo caixeiro viajante ao passar por todas as cidades.

A figura 12 refere-se à implementação paralela com o uso das regras do CUDA para permitir a manipulação de *threads* e execução na placa de vídeo. O procedimento, neste caso, é obter o *index* de cada *thread* (através da dimensão dos blocos e a posição "x"dos blocos em um grid e das *threads* em um bloco). O *index* substitui os dois loops antes existentes (o ato de percorrer cada cromossoma da população e cada gene de cada cromossoma), configurando assim a função para poder ser executada de forma paralela e independente.

```
__global__ void generateFitnessGPU(Chromosome *population, int *matrizAuxiliar, int *cityArray){
    int index = (blockIdx.x * blockDim.x) + threadIdx.x;
    int i = index / numCitys;
    int j = index % numCitys;

    int start = population[i].genes[j] - 1;
    int end = population[i].genes[j + 1] - 1;

    if(j == (numCitys-1)){
        matrizAuxiliar[numCitys*i+(numCitys-1)]=cityArray[end*numCitys+(population[i].genes[0]-1)];
    } else {
        matrizAuxiliar[numCitys * i + j] = cityArray[start*numCitys + end];
    }
}
```

Quadro 2: Função de calcular *fitness* paralelamente na GPU.

O termo __global__ é um qualificador de tipo de função, pertence às regras predefinidas pelo CUDA e significa que esta função é iniciada na CPU e executada na GPU com a quantidade de blocos e *threads* por bloco configuradas pelo programador. Em outras palavras, esta função é um Kernel. Um kernel executa um programa sequencial num conjunto de threads paralelas. Segundo YANO (2010), as funções __global__ retornam void, obrigatoriamente; qualquer chamada a uma função __global__ deve especificar a dimensão do grid e dos blocos; chamadas as funções __global__ são assíncronas, ou seja, a execução continua na CPU mesmo que não tenha terminado na GPU.

```
cutStartTimer(timer1);
generateFitnessGPU<<<numCitys, numPopulation>>>(dPopulation, dAuxMatrix, dCityArray);
cutStopTimer(timer1);

cutStartTimer(timer2);
applyFitness(hPopulation, dAuxMatrix);
cutStopTimer(timer2);
```

Quadro 3: Chamada do kernel passando o número de blocos, *threads* e os parâmetros.

Na figura 13 está a chamada do kernel, onde consta um *array* bidimensional delimitado pelos três caracteres "maior que"e "menor que", em que o primeiro parâmetro (numCitys, ou 10.000) é a dimensão do grid e o segundo parâmetro (numPopulation, ou 500) é a dimensão do

bloco. Deve-se considerar que o máximo de *threads* por blocos definido para diversas placas de vídeo é de 512.

Para o uso da função paralela na GPU, foi preciso utilizar o conceito de *Pinned Memory*, resultando em um ponteiro de memória na GPU para a CPU, evitando perda de desempenho com constantes cópias de memória entre CPU e GPU. O acesso às informações da matriz auxiliar são feitos sem gargalos e o algoritmo genético não tem o seu desempenho desgastado.

4.7 Resultados e Testes

Os resultados foram obtidos respeitando os termos referentes às cidades e populações. Dois cenários foram feitos: o primeiro, levando em consideração valores pequenos (até 500 cidades e população de 100) e o segundo, com valores consideráveis (10.000 cidades e população de 500). Ambas as abordagens (sequencial e paralela) foram avaliadas nestes cenários.

- Cenário 1:

Dadas 500 cidades, suas distâncias entre elas e uma população de 100 cromossomas, foram feitos 5 testes, onde foram anotados os valores dos tempos de duração que cada processo levou. A tabela 2 apresenta os tempos aproximados constatados através da abordagem sequencial.

Tabela 2: Tempos aproximados das execuções da abordagem sequencial no cenário 1, em milisegundos.

Nº do Teste	generateFitnessCPU
1	3,32 x 10^{-2}
2	3,13 x 10^{-2}
3	3,79 x 10^{-2}
4	3,05 x 10^{-2}
5	3,22 x 10^{-2}

A abordagem paralela tem os tempos de execução de suas tentativas relatadas na tabela 3.

Tabela 3: Tempos aproximados das execuções da abordagem paralela no cenário 1, em milisegundos.

Nº do Teste	generateFitnessGPU	ApplyFitness	Total
1	3,29 x 10^{-2}	2,11	2,15
2	3,15 x 10^{-2}	2,00	2,03
3	2,71 x 10^{-2}	1,90	1,92
4	2,98 x 10^{-2}	1,97	2,00
5	2,94 x 10^{-2}	1,96	1,99

Nota-se que para estes valores das variáveis envolvidas, a abordagem sequencial apresenta menores tempos de execução, mesmo com o foco da abordagem paralela em otimizar o algoritmo genético, em particular a função *fitness*. Nesta última abordagem, o fato de percorrer a matriz auxiliar e aplicar os valores das distâncias aos seus respectivos cromossomas torna o processo mais lento que a abordagem sequencial.

- Cenário 2:

Dadas 10.000 cidades, suas distâncias entre elas e uma população de 500 cromossomas, foram feitos 5 testes semelhantes aos testes do cenário 1. Os valores dos tempos de duração que cada processo levou foram anotados. A tabela 4 apresenta os tempos aproximados constatados através da abordagem sequencial.

Tabela 4: Tempos aproximados das execuções da abordagem sequencial no cenário 2, em milisegundos.

Nº do Teste	generateFitnessCPU
1	114,09
2	123,75
3	114,03
4	114,34
5	111,69

Os tempos de execução das tentativas da abordagem paralela estão presentes na tabela 5.

Tabela 5: Tempos aproximados das execuções da abordagem paralela no cenário 2, em milisegundos.

Nº do Teste	generateFitnessGPU	ApplyFitness	Total
1	5,31 x 10^{-2}	11,25	11,30
2	5,00 x 10^{-2}	12,06	12,11
3	4,79 x 10^{-2}	12,04	12,09
4	4,93 x 10^{-2}	11,93	11,98
5	5,91 x 10^{-2}	11,92	11,97

.

Para o cenário 2, a abordagem sequencial, nas 5 tentativas de execução, apresenta uma média maior que 110 milisegundos, enquanto que na abordagem paralela este valor cai para uma média de 12 milisegundos, sendo 0,05 milisegundos de execução da função *generateFitnessGPU* e 12 milisegundos para a função *applyFitness*. O preenchimento da matriz auxiliar de forma paralela automatiza o processo de cálculo da aptidão, restando apenas percorrer esta matriz via CPU e atribuir o valor correspondente à cada membro da população.

Os resultados obtidos levaram em consideração o tamanho da população de cromossomas e a quantidade de cidades presentes no Algoritmo Genético e no Problema do Caixeiro Viajante: para até 500 cidades e população de 100 cromossomas, abordagem sequencial apresenta melhor desempenho, com tempo de execução de 0,034 milisegundos em média. Para valores consideravelmente maiores, como 10.000 cidades e 500 cromossomas, a abordagem paralela é a mais indicada, apresentando o tempo de 12 milisegundos em média. Foram utilizadas *threads* na abordagem paralela, sendo correspondentes ao número de cidades multiplicado pelo número de cromossomas na população, ou seja: 50.000 *threads* no cenário 1 e cinco milhões de *threads* no cenário 2. O *speedup* do algoritmo paralelizado em relação à CPU chega a ser 9,8 vezes menor, em média, considerando o cenário 2.

5 *Conclusão*

Foram apresentados os conceitos e uma implementação do problema do caixeiro viajante, assim como a sua solução sequencial e também paralelizada via CUDA, baseada em tornar o cálculo do *fitness* mais eficaz e eficiente. O intenção desta monografia foi fornecer à comunidade acadêmica e de pesquisa uma alternativa na otimização de forma paralela do problema do caixeiro viajante, envolvendo algoritmos genéticos e utilizando a tecnologia CUDA da NVidia para grandes instâncias do problema.

Dois cenários foram montados levando em consideração o tamanho da população de cromossomas e a quantidade de cidades presentes no Algoritmo Genético e no Problema do Caixeiro Viajante. O primeiro cenário foi definido com um conjunto de até 500 cidades e população de 100 cromossomas. O segundo cenário possuiu 10.000 cidades e população de 500 cromossomas. Duas abordagens do procedimento de obtenção do *fitness* foram realizadas com o objetivo de otimizar o tempo de execução do que apresenta-se como maior gargalo de um algoritmo genético. A sequencial constou de dois "fors aninhados"para percorrer cada gene pertencente a cada cromossoma da população e, assim, calcular a distância entre os pares de cidades percorridos e gerar o total em quilômetros percorridos. A abordagem paralela utiliza até cinco milhões de *threads* (no cenário 2, onde as instâncias são maiores: 10.000 genes vezes 500 cromossomas) para calcular o custo em distância entre os pares de cidades a serem percorridas e preencher a matriz auxiliar na GPU, sendo que cada linha da matriz representa um cromossoma e cada coluna representa a distância entre duas cidades adjacentes. Em seguida, percorreu-se esta matriz, somou-se todos os valores de cada linha e atribuiu-se o número gerado à variável *fitness* de casa cromossoma.

Para o cenário 1, abordagem sequencial apresentou melhor desempenho, com tempo de execução de 0,034 milisegundos, enquanto que, para o cenário 2, a abordagem paralela foi a mais indicada, apresentando o tempo de 12 milisegundos em média. Desta forma, foi demonstrado o potencial da tecnologia CUDA da NVidia para otimização de algoritmos genéticos em grandes instâncias do problema do caixeiro viajante.

Durante o desenvolvimento, foi notada uma certa dificuldade em encontrar trabalhos na área de programação paralela que utilizem o CUDA, principalmente que possuíssem código-fonte completo e aberto para pesquisa. O projeto, incluindo também o código-fonte construído através desta monografia, se torna relevante neste contexto, servindo como base livre para consulta, posteriores análises e estudos neste nincho de pesquisa.

Mesmo com o *speedup* apresentado e considerado satisfatório, esta monografia não teve intenção de superar os melhores algoritmos de solução do problema do caixeiro viajante, nem provar que a programação paralela é mais eficaz que a programação sequencial, até porque isto já foi comprovado em outras obras. Este trabalho visou validar a utilização da arquitetura CUDA em GPUs para processamento paralelo. Apesar da pequena quantidade de publicações existentes sobre o assunto, deve-se considerar que, mesmo com os resultados apresentados, muito ainda deve ser melhorado para que consiga-se usufruir do máximo da paralelização e utilização de placas gráficas com o recurso do CUDA.

Dentre os trabalhos futuros que podem ser sugeridos, está um *framework* para o desenvolvimento de algoritmos genéticos com a função *fitness* paralelizada via CUDA.

Referências

ASIA-PACIFIC SCIENCE AND TECHNOLOGY CENTER (APSTC) SUN MICROSYSTEMS (China). *Survey on Parallel Programming Model*. Disponível em: http://apstc.sun.com.sg/activities/events/past/download/SurveyOnPPM.pdf.
Acesso em: 12 jul 2012.

DA CUNHA, Cláudio Barbieri; BONASSER, Ulisses de Oliveira; ABRAHÃO, Fernando Teixeira Mendes. *Experimentos Computacionais com Heuristicas de Melhorias para o Problema do Caixeiro Viajante*. São Paulo, 2002.

FATAHALIAN, K. *Beyond Programmable Shading*. Stanford University, SIGGRAPH, 2009. Disponível em: http://s09.idav.ucdavis.edu/talks/02_kayvonf_gpuArchTalk09.pdf
Acesso em: 19 jul 2012.

GAZOLLA, João Gabriel Felipe Machado. *Uma Abordagem Heurística e Paralela em GPUS para o Problema do Caixeiro Viajante*. Niterói, RJ, 2010.

GRAMMELSBACHER, Alessandro Vieira; MEDRADO, João Carlos Campoi. *Comparação de Desempenho entre GPGPU e Sistemas Paralelos*. São Paulo, 2009.

GOLDBERG, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

HOFFMAN, A. J.; WOLFE, P. *History, in The Traveling Salesman Problem* LAWLER, E. L (Ed.); LENSTRA, J. K. (Ed.);

LACERDA, E. G. M.; CARVALHO, A. C. *Sistemas inteligentes: aplicações a recursos hídricos e ciências ambientais*. Porto Alegre, RS: Universidade/UFRGS, 1999. cap. *Introdução aos Algoritmos Genéticos*, p. 99-150. Disponível em: http://www.dca.ufrn.br/ estefane/metaheuristicas/ag.pdf.

LINDEN, R. *Algoritmos Genéticos*. 2 ed. Rio de Janeiro: Brasport, 2008. 428 p.

MELAMED, I. I.; SERGEEV, S. I.; SIGAL, I. Kh. *The Traveling Salesman Problem*. Surveys. [S.l.]: Plenum Publishing Corporation, 1990. p. 1147-1173.

NVIDIA. *NVIDIA CUDA Architecture*. Version 1.1 April 2009. Disponível em http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf
Acesso em: 23 mai 2012.

NVIDIA. *NVIDIA CUDA C Programming Guide*. Version 4.2 April 2012. Disponível em http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Acesso em: 22 mai 2012.

PAPADIMITRIOU, C. H.; STEIGLITZ K. *Combinatorial Optimization: Algorithms and Complexity*. 1 ed. Dover Science, 1998. 512 p.

SUBRAMANIAN, A.; DRUMMOND, L.M.A.; BENTES, C.; OCHI, L.S.; FARIAS, R. *A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery*. Computers & Operations Research, v.37. n.11, p.1899-1911, 2010.

YANO, Luis Gustavo Abe. *Avaliação e Comparação de desempenho utilizando tecnologia CUDA*. São José do Rio Preto, 2010.

APÊNDICE A – Especificação dos Atributos e da Estrutura do Cromossomo

```
1  // Valor maximo para distancia de 1 cidade ate a outra
2  static const int maxDistance = 98;
3
4  //Porcentagem de Cruzamento
5  static const int numPercentCross = 50;
6
7  //Numero de Cidades, referente ao numero de genes de cada cromossomo
8  static const int numCitys = 500;
9
10 //Numero de Cromossomas que comportarao a populacao
11 static const int numPopulation = 100;
12
13 //Numero de Geracoes que o algoritmo genetico tera
14 static const int numGenerations = 1;
15
16 //Numero maximo de threads por bloco
17 static const int maxThreadsPerBlock = 512;
18
19 //Numero maximo de threads por bloco
20 static const int maxBlocksPerGrid = 512 * 512 * 64;
21
22 //Estrutura de um Cromossomo
23 typedef struct {
24   int fitness;
25   int genes[numCitys];
26 } Chromosome;
```

APÊNDICE B – Implementação das Funções do Algoritmo Genético e de Impressão

```
1  #include <cstdlib>
2  #include <cstring>
3  #include <cstdio>
4  #include "Attributes.h"
5
6  void printCityArray(int *hCityArray){
7    printf("City Array:\n");
8
9    for(int i = 0; i < numCitys; i++){
10     for(int j = 0; j < numCitys; j++){
11       printf("%i, ", hCityArray[i * numCitys + j]);
12     }
13     printf("\n");
14   }
15 }
16
17 void makeCityArray(int *hCityArray){
18   for(int i = 0; i < numCitys; i++){
19     for(int j = 0; j < numCitys; j++){
20       if(i == j){
21         hCityArray[i * numCitys + j] = 0;
22       } else if(i < j){
23         hCityArray[i * numCitys + j] = rand() % maxDistance + 1;
24       } else{
25         hCityArray[i * numCitys + j] = hCityArray[j * numCitys + i];
26       }
27     }
28   }
```

```
29 }
30
31 void initChromosome(Chromosome &c){
32   c.fitness = 0;
33   for(int i = 0; i < numCitys; i++){
34     c.genes[i] = -1;
35   }
36 }
37
38 void generateGenes(Chromosome &c){
39   int value;
40   initChromosome(c);
41   for(int i = 0; i < numCitys; i++){
42     do{
43       value = rand() % numCitys;
44     }while(c.genes[value] != -1);
45     c.genes[value] = (i + 1);
46   }
47 }
48
49 bool chromosomeContainsGene(Chromosome &c, int gene){
50   for(int i = 0; i < numCitys; i++){
51     if(c.genes[i] == gene){
52       return true;
53     }
54   }
55   return false;
56 }
57
58 void generateRandomGenes(Chromosome &c){
59   int value;
60   for(int i = 0; i < numCitys; i++){
61     if(c.genes[i] < 0){
62       do{
63         value = (rand() % numCitys) + 1;
64       }while(chromosomeContainsGene(c, value));
65       c.genes[i] = value;
66     }
```

```
67    }
68  }
69
70  void printChromosome(Chromosome &c){
71    printf("genes = ");
72    for(int i = 0; i < numCitys; i++){
73      printf("%i, ", c.genes[i]);
74    }
75    printf(" fitness = %d\n", c.fitness);
76  }
77
78  bool equalsChromosome(Chromosome &c1, Chromosome &c2){
79    for(int i = 0; i < numCitys; i++){
80      if(c1.genes[i] != c2.genes[i]){
81        return false;
82      }
83    }
84    return true;
85  }
86
87  void printPopulation(Chromosome *population){
88    printf("Population:\n");
89    for(int i = 0; i < numPopulation; i++){
90      printChromosome(population[i]);
91    }
92    printf("\n");
93  }
94
95  void generateFitnessCPU(Chromosome *population, int *hCityArray){
96    int start = 0;
97    int end = 0;
98    for(int i = 0; i < numPopulation; i++){
99      population[i].fitness = 0;
100     for(int j = 0; j < numCitys - 1; j++){
101       start = population[i].genes[j] - 1;
102       end = population[i].genes[j + 1] - 1;
103       population[i].fitness += hCityArray[start*numCitys + end];
104     }
```

```
105       population[i].fitness += hCityArray[end*numCitys + (population[i
              ].genes[0]-1)];
106     }
107   }
108
109   void applyFitness(Chromosome *population, int *matrizFitness){
110     for(int i = 0; i < numPopulation; i++){
111       population[i].fitness = 0;
112       for(int j = 0; j < numCitys; j++){
113         population[i].fitness += matrizFitness[numCitys * i + j];
114       }
115     }
116   }
117
118   bool populationContainsChromosome(Chromosome *population, Chromosome
        &c){
119     for(int i = 0; i < numPopulation; i++){
120       if(equalsChromosome(population[i], c)){
121         return true;
122       }
123     }
124     return false;
125   }
126
127   void generatePopulation(Chromosome *population){
128     for(int i = 0; i < numPopulation; i++){
129       Chromosome c;
130       generateGenes(c);
131       if(!populationContainsChromosome(population, c)){
132         population[i] = c;
133       } else {
134         i--;
135       }
136     }
137   }
138
139   void printBestGenes(Chromosome *population){
140     printf("The most able: %i\n", population[0].fitness);
```

```
141 }
142
143 void sortByFitness(Chromosome *population, int maxElements){
144   for(int i = 0; i < maxElements; i++){
145     for(int j = i; j < numPopulation; j++){
146       if(population[i].fitness > population[j].fitness){
147         Chromosome aux = population[i];
148         population[i] = population[j];
149         population[j] = aux;
150       }
151     }
152   }
153 }
154
155 void elitism(Chromosome *population){
156   sortByFitness(population, 2);
157 }
158
159 void crossOver(Chromosome *population){
160   const int val = (int)(numCitys * numPercentCross/100);
161   int posA, posB, aleloA[val], aleloB[val], cacheA_Antigo[val],
        cacheB_Antigo[val];
162   int *cacheA_Novo, *cacheB_Novo;
163
164   for(int i = 2; i < numPopulation - 1; i=i+2){
165     posA = 0;
166     posB = 0;
167
168     cacheA_Novo = &population[i+1].genes[val];
169     cacheB_Novo = &population[i].genes[val];
170
171     memcpy(cacheA_Antigo, &population[i].genes[val], sizeof(int)*val)
          ;
172     memcpy(cacheB_Antigo, &population[i+1].genes[val], sizeof(int)*
          val);
173
174     for(int k = 0; k < val; k++){
175       for(int l = 0; l < val; l++){
```

```
176           if(cacheA_Antigo[l] == -1){
177             break;
178           }else if(cacheA_Novo[k] == cacheA_Antigo[l]){
179             aleloA[posA] = cacheA_Novo[k];
180             cacheA_Antigo[l] = -1;
181             posA++;
182           }
183         }
184
185         for(int l = 0; l < val; l++){
186           if(cacheB_Antigo[l] == -1){
187             break;
188           }else if(cacheB_Novo[k] == cacheB_Antigo[l]){
189             aleloB[posB] = cacheB_Novo[k];
190             cacheB_Antigo[l] = -1;
191             posB++;
192           }
193         }
194       }
195
196       for(int k = 0; k < val; k++){
197         if(cacheA_Antigo[k] != -1){
198           aleloA[posA] = cacheA_Antigo[k];
199           posA++;
200         }
201         if(cacheB_Antigo[k] != -1){
202           aleloB[posB] = cacheB_Antigo[k];
203           posB++;
204         }
205       }
206
207       memcpy(&population[i].genes[val], aleloA, sizeof(aleloA));
208       memcpy(&population[i+1].genes[val], aleloB, sizeof(aleloB));
209     }
210 }
```

APÊNDICE C – Kernel do CUDA

```
1  __global__ void generateFitnessGPU(Chromosome *population, int *
       matrizAuxiliar, int *cityArray){
2
3    int index = (blockIdx.x * blockDim.x) + threadIdx.x;
4    int i = index / numCitys;
5    int j = index % numCitys;
6
7    int start = population[i].genes[j] - 1;
8    int end = population[i].genes[j + 1] - 1;
9
10   if(j == (numCitys-1)){
11     matrizAuxiliar[numCitys*i+(numCitys-1)] = cityArray[end*numCitys
         +(population[i].genes[0]-1)];
12   } else {
13     matrizAuxiliar[numCitys*i+j] = cityArray[start*numCitys+end];
14   }
15 }
```

APÊNDICE D – Função Main

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <cutil_inline.h>
5  #include <cuda.h>
6  #include "FunctionsCPU.h"
7
8  int main(){
9    int threads = numCitys * numPopulation;
10   if(threads > maxThreadsPerBlock){
11     threads = maxThreadsPerBlock;
12   }
13
14   int blocks = numCitys * numPopulation / threads;
15   if((numCitys * numPopulation) % threads > 0){
16     blocks++;
17   }
18
19   cudaSetDeviceFlags(cudaDeviceMapHost);
20
21   //usado para o contador
22   unsigned int timer1 = 0;
23   unsigned int timer2 = 0;
24
25   //sera usado para gerar valores randomicos para criar cromossomas
26   srand(time(NULL));
27
28   //Tamanho da matriz Auxiliar
29   size_t sizeAuxMatrix = numCitys * numPopulation * sizeof(int);
```

```
30
31  //Matriz Auxiliar na CPU(h) e GPU(d)
32  int *hAuxMatrix, int *dAuxMatrix;
33
34  //Alocacao de memoria na CPU para a matriz Auxiliar
35  cudaHostAlloc((void **)&hAuxMatrix, sizeAuxMatrix,
        cudaHostAllocMapped);
36
37  //Alocando memoria na GPU para a matriz Auxiliar
38  cudaHostGetDevicePointer((void **)&dAuxMatrix, (void *)hAuxMatrix,
        0);
39
40  //Tamanho da matriz de Cidades
41  size_t sizeCityArray = numCitys * numCitys * sizeof(int);
42
43  //Matriz de Cidades na CPU(h) e GPU(d)
44  int *hCityArray = (int*)malloc(sizeCityArray);
45  int *dCityArray;
46
47  //Gerando Matriz de Cidades
48  makeCityArray(hCityArray);
49
50  //Alocando memoria na GPU para o matriz de Cidades
51  cudaMalloc((void**)&dCityArray, sizeCityArray);
52
53  //Copiando memoria para GPU da vetor de Cidades
54  cudaMemcpy(dCityArray, hCityArray, sizeCityArray,
        cudaMemcpyHostToDevice);
55
56  //Imprimindo Matriz de Cidades
57  printCityArray(hCityArray);
58
59  //Tamanho do vetor de Populacao de Cromossomos
60  size_t sizePopulation = numPopulation * sizeof(Chromosome);
61
62  //Vetor de Populacao de Cromossomos na CPU(h) e na GPU(d)
63  Chromosome *hPopulation, Chromosome *dPopulation;
64
```

```
65    //Alocacao de memoria na CPU para o vetor de Populacao
66    cudaHostAlloc(&hPopulation, sizePopulation, cudaHostAllocMapped);
67
68    //Alocando memoria na GPU para o vetor de Populacao
69    cudaHostGetDevicePointer((void **)&dPopulation, (void *)hPopulation
          , 0);
70
71    //Iniciando os contadores
72    cutCreateTimer(&timer1);
73    cutCreateTimer(&timer2);
74
75    //Gerando Populacao
76    generatePopulation(hPopulation);
77
78    //Neste ponto e usada uma das rotinas apresentadas nos apendices E
          ou F
79
80      printf("time 1 = %f ms\n", cutGetTimerValue(timer1));
81      printf("time 2 = %f ms\n", cutGetTimerValue(timer2));
82      printf("total = %f ms\n", cutGetTimerValue(timer1) +
            cutGetTimerValue(timer2));
83      printf("best solution = %d\n", hPopulation[0].fitness);
84
85    cutDeleteTimer(timer1);
86    cutDeleteTimer(timer2);
87
88    //Libera a Matriz de Cidades, Vetor de Populacao e Matriz Auxiliar
          da memoria da CPU e da GPU respectivamente
89    free(hCityArray);
90    cudaFreeHost(hPopulation);
91    cudaFreeHost(hAuxMatrix);
92
93    cudaFree(dCityArray);
94    cudaFree(dPopulation);
95    cudaFree(dAuxMatrix);
96    return 0;
97  }
```

APÊNDICE E – Rotina de execução sequencial do cálculo de Fitness

```
1  for(int i = 0; i < numGenerations; i++){
2      cutStartTimer(timer1);
3      generateFitnessCPU(hPopulation, hCityArray);
4      cutStopTimer(timer1);
5
6      elitism(hPopulation);
7
8      crossOver(hPopulation);
9    }
```

APÊNDICE F – Rotina de execução paralela do cálculo de Fitness

```
1  for(int i = 0; i < numGenerations; i++){
2      cutStartTimer(timer1);
3      generateFitnessGPU<<<numCitys, numPopulation>>>(dPopulation,
           dAuxMatrix, dCityArray);
4      cutStopTimer(timer1);
5
6      cutStartTimer(timer2);
7      applyFitness(hPopulation, dAuxMatrix);
8      cutStopTimer(timer2);
9
10     elitism(hPopulation);
11
12     crossOver(hPopulation);
13   }
```