UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

UMA ABORDAGEM HEURÍSTICA E PARALELA EM GPUS PARA O PROBLEMA DO CAIXEIRO VIAJANTE

DISSERTAÇÃO DE MESTRADO

João Gabriel Felipe Machado Gazolla

Niterói, RJ, Brasil

2010

UMA ABORDAGEM HEURÍSTICA E PARALELA EM GPUS PARA O PROBLEMA DO CAIXEIRO VIAJANTE

por

João Gabriel Felipe Machado Gazolla

Dissertação de Mestrado submetida ao "Programa de Pós-Graduação em Computação" da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de Concentração: Computação Visual e Interfaces / Algoritmos e Otimização.

Programa de Pós-Graduação em Computação da
Universidade Federal Fluminense (UFF, RJ),
como requisito parcial para a obtenção do grau de
Mestre em Computação

Orientador: Prof. Dr. Esteban Walter Gonzalez Clua (UFF)

Niterói, RJ, Brasil

2010

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

G291 Gazolla, João Gabriel Machado.
Uma abordagem heurística e paralela em GPUS para o problema do caixeiro viajante / João Gabriel Machado Gazolla. – Niterói, RJ : [s.n.], 2010.
69 f.

Dissertação (Mestrado em Computação) - Universidade Federal Fluminense, 2010.
Orientadores: Esteban Walter Gonzalez Clua.

1. Heurística. 2. Unidade de processamento gráfico. 3. Caixeiro viajante. 4. Busca local iterativa. I. Título.

CDD 005.136

Universidade Federal Fluminense
Instituto de Computação
Programa de Pós-Graduação em Computação

A Comissão Examinadora, abaixo assinada, aprova a Dissertação de Mestrado

UMA ABORDAGEM HEURÍSTICA E PARALELA EM GPUS PARA O PROBLEMA DO CAIXEIRO VIAJANTE

elaborada por
João Gabriel Felipe Machado Gazolla

como requisito parcial para obtenção do grau de
Mestre em Computação

Aprovada por:

D. Sc. Esteban Walter Gonzalez Clua (Presidente) / IC-UFF
Universidade Federal Fluminense

PhD. João Luiz Dihl Comba / INF-UFRGS
Universidade Federal do Rio Grande do Sul

D. Sc. Luiz Satoru Ochi / IC-UFF
Universidade Federal Fluminense

D. Sc. Anselmo Montenegro / IC-UFF
Universidade Federal Fluminense

Niterói, 10 de Agosto de 2010

AGRADECIMENTOS

Agradeço primeiramente a Deus, que em todos os momentos guiou meus passos e iluminou minha vida, me dando a força e a sabedoria necessária para vencer cada novo desafio.

Aos meus pais, Afrânio e Marisa, por todo amor e ensinamentos que sempre me deram. Sem o apoio de vocês esta caminhada não teria sido possível. Obrigado por fazerem dos meus sonhos os seus sonhos, e da minha vida suas vidas.

À minha irmã, Patrícia, que mesmo distante, trouxe incentivo, apoio e colaboração.

Ao meu irmão, Pedro, que durante todos estes anos tive o prazer de sua companhia, principalmente por estarmos distantes da família. Obrigado pelo companheirismo e por sempre me ajudar de todas as formas possíveis.

Ao meu grande amor, Roberta, que me faz a cada dia a pessoa mais feliz do mundo. O seu carinho, paciência, cumplicidade e compreensão foram fundamentais.

Aos meus padrinhos, Orlando e Cristina, porque sem eles esta tarefa teria sido impossível. Obrigado pelo acolhimento.

Aos meus tios, Marilene e Bianchi pelo apoio incondicional durante minha graduação.

À Universidade Federal Fluminense, especialmente ao Instituto de Computação, pela oportunidade de realizar este trabalho.

Em especial, ao meu orientador, professor Esteban Walter Gonzalez Clua, por ter acreditado em mim, pelos ensinamentos, paciência, atenção e disponibilidade em sempre ajudar. O seu apoio e calma foram essenciais para meu processo de amadurecimento.

Aos professores do Instituto de Computação, que contribuíram para minha formação acadêmica e amadurecimento pessoal.

Aos colegas de mestrado, pelos agradáveis momentos de convivência, apoio, incentivo e troca de idéias. Em especial, ao Anand Subramanian pela paciência, amizade e orientação.

A CAPES, pelo apoio financeiro.

A todos que, de algum modo, contribuíram para a conclusão desta etapa da minha vida. Tenho consciência de que, se cheguei até aqui, é porque tive a oportunidade de crescer e aprender ao lado de grandes mestres.

BIOGRAFIA

JOÃO GABRIEL FELIPE MACHADO GAZOLLA, filho de Afrânio Gonçalves Gazolla e Marisa Felipe Machado Gazolla, nasceu em 22 de Abril de 1986, em São Luís, Maranhão, Brasil.

Em 2003, concluiu o 2º Grau no Colégio Dom Bosco Renascença, em São Luís - MA.

Em 2004, iniciou o curso de graduação em Ciência da Computação pela Universidade Federal de Viçosa - MG, onde permaneceu até Julho/2008. Durante a graduação, teve a oportunidade de ser bolsista e monitor do Departamento de Matemática.

Em Agosto de 2008, ingressou no Curso de Mestrado do Programa de Pós-Graduação em Ciência da Computação pela Universidade Federal Fluminense, atuando na linha de pesquisa Computação Visual e Interfaces. Neste tempo, atuou por dois anos como bolsista CAPES, membro do CyberBridges e do MediaLab. Em Agosto de 2010, submeteu-se à defesa desta dissertação.

“É preciso sempre acreditar que o sonho é possível, o céu é o limite e você é imbatível.” –

Autor desconhecido

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Computação
Universidade Federal Fluminense

UMA ABORDAGEM HEURÍSTICA E PARALELA EM GPUS PARA O PROBLEMA DO CAIXEIRO VIAJANTE

Autor: João Gabriel Felipe Machado Gazolla.
Orientador: Prof. Dr. Esteban Walter Gonzalez Clua (UFF).
Local e Data da Defesa: Niterói, 10 de Agosto de 2010.

O problema do caixeiro viajante (PCV) é um problema *NP-Completo* bem estudado e que conta com métodos eficientes, exatos e heurísticos para sua solução. Porém, mesmo através da utilização de heurísticas, o problema continua a ser de difícil solução, pois apresenta um alto custo computacional, inclusive para instâncias relativamente pequenas.

Indo além das pesquisas existentes, as GPUs, com poder de processamento numérico dezenas de vezes maiores que as CPUs possuem um potencial para obtenção de resultados melhores e mais rápidos em diversas áreas da computação. Assim sendo, é natural que surjam trabalhos que utilizem esta arquitetura para abordar o clássico PCV. Este é um campo de prova de difíceis melhorias, que caso sejam alcançadas, demonstraria que as GPUs podem de fato ser úteis na resolução de problemas reais de otimização de alta complexidade computacional.

O objetivo deste trabalho é apresentar e discutir uma nova abordagem heurística e paralela implementada em GPU através da linguagem CUDA, capaz de validar a resolução de meta-heurísticas em GPUs.

Mesmo trazendo resultados eficientes, este trabalho não pretende superar os algoritmos existentes na literatura, mas sim expandir horizontes e estabelecer ligações entre GPUs e problemas de otimização, possibilitando a geração de trabalhos futuros.

Palavras-chave: GPUs, Unidade de Processamento Gráfico, *GPU Computing*, CUDA, Problema do Caixeiro Viajante, Busca Local Iterada, heurísticas.

ABSTRACT

Master's Dissertation
Programa de Pós-Graduação em Computação
Universidade Federal Fluminense

A GPU HEURISTIC PARALLEL IMPLEMENTATION OF THE TRAVELLING SALESMAN PROBLEM

Author: João Gabriel Felipe Machado Gazolla
Advisor: Prof. Dr. Esteban Walter Gonzalez Clua
Place and Date of Presentation: Niterói, August 10, 2010

The travelling salesman problem (TSP) is a widely studied *NP-Complete* problem which has efficient exact and heuristic methods for its solution. However, in spite of the use of heuristics, the TSP remains difficult to solve, and has a high computational cost, even for relatively small datasets.

Our approach takes beyond existing research by employing GPUs, with a computing power many times larger than CPUs. GPUs can be used in several areas of computing to obtain faster results; therefore, it's predictable that a GPU approach should be expected to speed up the classic TSP. This is a challenging research, but if we are able to improve performance, we will be showing that GPUs can actually be useful in solving real optimization problems of high computational complexity.

The aim of this work is to present and discuss a new parallel and heuristic approach implemented in GPU using the CUDA language, which is able to validate the solution of meta-heuristics in GPUs.

Although presenting effective results, this paper does not seek to overcome the existing algorithms in the literature. We rather aim at creating new perspectives, and establish connections between GPUs and optimization problems, enabling and motivating future works.

Keywords: GPUs, *Graphics Processing Units*, *GPU Computing*, CUDA , *Travelling Salesman Problem*, *Iterated Local Search* , *Heuristics*.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

Sigla	Abreviatura
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
DRAM	*Dynamic Random Access Memory*
FLOPS	*Floating-Point Operations Per Second*
GPGPU	*General-Purpose Computing on Graphics Processing Units*
GPU	*Graphics Processing Unit*
GRASP	*Greedy Randomized Adaptive Search Procedure*
ILS	*Iterated Local Search*
ISA	*Instruction Set Architecture*
PCV	Problema do Caixeiro Viajante
RCL	*Restricted Candidate List*
RVND	*Random Variable Neighborhood Descent*
SM	*Streaming Multiprocessor*
SMP	*Symmetric Multi-Processing*
TSP	*Travelling Salesman Problem*
TSPLIB	Biblioteca de instâncias do Problema do Caixeiro Viajante
VND	*Variable Neighborhood Descent*

SUMÁRIO

1 INTRODUÇÃO

Ao longo das últimas décadas o crescimento do poder computacional dos processadores tem obedecido a *Lei de Moore*, que afirma que a capacidade dos processadores dobra a cada 18 meses.

A abordagem atual das empresas fabricantes de processadores como *Intel* e *Advanced Micro Devices* (AMD*)* já não visa aumentar a velocidade dos *clocks*, mas aumentar o número de núcleos em cada processador [1].

É inegável que a abordagem de aumentar o número de núcleos em cada processador seja um caminho traçado pioneiramente pelas empresas produtoras de *GPUs* há alguns anos. Enquanto um processador *Intel® Core™ i7 Processor* – 920, um dos últimos modelos lançados pela *Intel* consegue atingir apenas 42.56 *GFlops* [2] (G*iga* operações de ponto flutuante por segundo) de pico de peformance, uma *GPU GTX480* atinge a incrível marca de 1.350 *TFlops* [3] (Tera operações de ponto flutuante por segundo), o que representa um poder de processamento 32 vezes maior, tornando o uso de GPUs para fins científicos uma realidade acessível e real.

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.

Um problema ideal para testes em GPUs é o clássico Problema do Caixeiro Viajante (PCV), um problema NP-Completo vastamente estudado que conta com métodos heurísticos para sua solução.

A escolha do PCV não se dá sem motivos, pelo contrário, pois mesmo através de heurísticas o problema continua a ser de difícil solução e um campo de difíceis melhorias. Muito já foi pesquisado, porém pouco foi relacionado ao uso de GPUs.

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, ainda mais através da utilização de heurísticas. Como será mostrado neste trabalho, as tarefas computacionais necessárias para esta classe de problemas são possíveis de serem implementadas em arquitetura de GPUs, podendo as mesmas servirem como plataforma computacional de grande paralelismo. Desta forma, um algoritmo paralelizável que demorasse algumas horas para ser executado de forma

seqüencial teria seu tempo reduzido em uma ordem de grandeza, podendo ser resolvido em minutos.

1.1 Objetivos do Trabalho

O objetivo geral deste trabalho é apresentar uma solução heurística paralela para o PCV através do uso de GPUs, de forma a realizar uma prova de conceito para demonstrar que GPUs podem contribuir para acelerar a solução de problemas de otimização em geral.

Especificamente, os objetivos deste trabalho foram:

1. Investigar algoritmos pré-existentes que possam ser mapeados em GPUs;
2. Definir uma estratégia genérica para mapear algoritmos de busca local em GPUs;
3. Apresentar uma implementação para o PCV que ganhe desempenho para processamento em GPUs;
4. Realizar testes e validações que apontem a eficiência destas técnicas.

1.2 Contribuições Alcançadas

Dentre as contribuições alcançadas merece destaque a modelagem da paralelização em GPU de cinco algoritmos de busca local: *2-opt*, *swap*, *OrOpt-1*, *OrOpt-2* e *OrOpt-3*.

Foi desenhado em linhas gerais um modelo que demonstra como tirar proveito das GPUs para problemas de otimização. Técnicas para melhoria de desempenho e otimização também foram descritas, sendo algumas apresentadas no trabalho e outras detalhadas nos trabalhos futuros.

1.3 Estrutura da Dissertação

Na Seção 2, é apresentado o referencial teórico resultado da pesquisa bibliográfica sobre as áreas e tecnologias relacionadas ao trabalho. Esta seção é subdividida em:

- Problema do Caixeiro Viajante (PCV);
- *Graphical Processing Units* (GPU);
- *Compute Unified Device Architecture* (CUDA);

- *Travelling Salesman Problem Library* (TSPLIB).
- *Iterated Local Search* (ILS).
- Discussão sobre os principais trabalhos que influenciaram na realização desta pesquisa.

Na Seção 3, é apresentado e detalhado o modelo desenvolvido para resolver o PCV de forma heurística e paralela através da utilização de GPUs, bem como a estrutura de cada algoritmo e da implementação como um todo.

Na Seção 4, são apresentados os testes e resultados da avaliação de desempenho, realizada através de diversos testes no qual foi colocado a prova o modelo desenvolvido.

Na Seção 5, são descritas as conclusões desta dissertação, assim como as principais constribuições oferecidas por este trabalho, possíveis melhorias e possibilidades de trabalhos futuros que poderão contribuir com esta linha de pesquisa.

2 REFERENCIAL TEÓRICO

Esta seção tem por objetivo introduzir os principais temas relacionados a este trabalho. O PCV é o primeiro tema a ser apresentado por ser o problema central. Em seguida é feita uma breve discussão sobre GPUs, sua linguagem de programação denominada CUDA e finalmente descrevem-se conceitos relacionados à meta-heurísticas e aos algoritmos utilizados.

2.1 O Problema do Caixeiro Viajante (PCV)

O PCV é conhecido na língua inglesa por "*Travelling Salesman Problem*" (TSP), sendo um problema amplamente estudado e pesquisado na literatura [4]. O problema em si é simples de se compreender mas difícil de se resolver pela sua elevada complexidade computacional (*NP-Hard*).

Segundo Applegate *et al*. (2006) [5], dado um conjunto de cidades e os custos para se deslocar entre cada par, o PCV consiste em encontrar a maneira mais barata de visitar todas elas precisamente uma vez antes de retornar à cidade de partida (ciclo hamiltoniano de custo mínimo).

A solução do PCV é representada pela ordem em que as cidades são visitadas, sendo tal ordem denominada *tour*.

O PCV pertence à categoria de problemas *NP-Hard* [6], que implica que não podem ser resolvidos em ordem de tempo polinomial em função do tamanho da entrada, possuindo uma ordem de complexidade exponencial, o que significa que a complexidade do problema cresce exponencialmente à medida que aumenta o tamanho da entrada. Aplicar métodos exatos isoladamente para sua solução é inviável para instâncias grandes, sendo, portanto utilizados métodos heurísticos ou aproximados.

Ainda segundo Applegate *et al*. (2006) [5], a simplicidade do PCV em conjunto com sua aparente intratabilidade, faz dele uma plataforma ideal para desenvolver novas idéias e técnicas para atacar problemas computacionais em geral, sendo um ótimo problema para prova de conceito e *benchmark* de novas técnicas, estratégias e algoritmos.

A Figura 1 representa as cidades e os custos simétricos de deslocamento entre elas. Matemáticamente o PCV pode ser definido da seguinte forma: dada a matriz simétrica de custos C representada pela Figura 2, onde C_{ij} (elemento [i] [j]) representa o custo de ir da

cidade *i* para a cidade *j*, encontre uma permutação de inteiros de 1 até *n* que minimize o *custo* $C_{i1i2} + C_{i2i3} + C_{i3i4} + \ ... + \ C_{ini1}$.

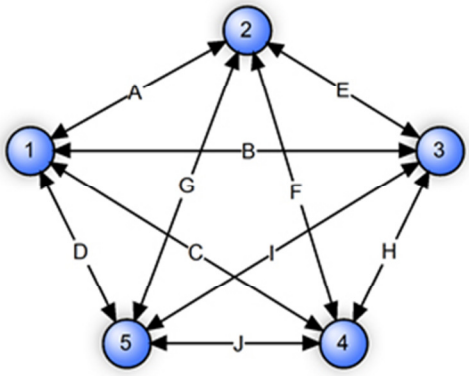

Figura 01: Exemplo de um PCV com N = 5 cidades, grafo completo e simétrico.

	01	02	03	04	05
01	0	A	B	C	D
02	A	0	E	F	G
03	B	E	0	H	I
04	C	F	H	0	J
05	D	G	I	J	0

Figura 02: Matriz simétrica que representa os custos entre cidades.

Segundo Helsgaun (1999) [7], 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*, tal como apresentado na tabela 01:

Tabela 1: Comparativo entre o número de cidades e possibilidades de *tours*.

N° de Cidades	Possibilidades	N° de Possíveis Tours
10	10!	3.628.800 / 2
25	25!	1.551121×10^{25} / 2
50	50!	$3.04140932 \times 10^{64}$ / 2
75	75!	$2.48091408 \times 10^{109}$ / 2
100	100!	$9.33262154 \times 10^{157}$ / 2
125	125!	$1.88267718 \times 10^{209}$ / 2
150	150!	$5.71338396 \times 10^{262}$ / 2

Para a versão simétrica do PCV com *n* cidades existem *(n-1)!/2* possibilidades de *tours* diferentes. Apenas para fins de comparações, em uma instância *n = 100*, existem mais possibilidades de *tours*, do que partículas no universo (10^{87}) [8].

Para calcular uma solução exata do PCV, seria necessário testar todas as possíveis soluções uma a uma. Em um problema pequeno com apenas algumas cidades a serem

visitadas, o número de possibilidades é simples de ser calculado: suponha *20* cidades. No primeiro plano existem *19* cidades a serem visitadas a partir da primeira, após a visita da segunda restarão *18* cidades, depois apenas *17*, e assim por diante.

O processamento explícito de todas as soluções se torna inviável, pois $19! = 1.216451 \times 10^{17}/2$ é um número muito grande, que tornaria o problema difícil de ser solucionado para vários dos mais rápidos computadores pessoais.

2.2 TSPLIB – *Travelling Salesman Problem Library*

Para a verificação do modelo proposto foi escolhida uma biblioteca de instâncias do PCV muito utilizada na literatura, a TSPLIB [9], que fornece aos pesquisadores um conjunto de instâncias do PCV, como também de outros problemas: PCV assimétrico, problema do ciclo hamiltoniano e roteamento de veículos com capacidade. A TSPLIB ainda fornece informações sobre o tamanho do *tour* ótimo, limites inferiores e superiores. A TSPLIB está disponível publicamente [10] através de dezenas de instâncias do problema que variam de 50 até 15.000 pontos.

Para a utilização da TSPLIB no projeto, foi implementado neste trabalho um *parser* em C++ que realiza a leitura do arquivo com as instâncias do problema, e o converte para uma matriz de custo (distâncias) onde o elemento [X,Y] da matriz M, M[X][Y], possui a distância entre as cidades X e Y. Esta matriz é simétrica, portanto a distância M[Y,X] também é igual a M[X,Y], sendo a diagonal principal constituída por zeros.

2.3 Iterated Local Search – ILS

O algoritmo meta-heurístico ILS, proposto originalmente por Lourenço *et al.* (2002) [11], possui por natureza um *design* genérico e de fácil implementação. O algoritmo é portador de uma vasta flexibilidade devido à modularidade de seus componentes, que pode ser a explicação para seu alto desempenho. Em alguns casos pode-se considerar o algoritmo como estado da arte para alguns problemas.

Apesar da meta-heurística utilizada neste trabalho ser parecida com o ILS, ele não foi o modelo utilizado propriamento dito, mas nos serve como inspiração para entender o funcionamento do modelo proposto.

A estratégia do método baseia-se em não ficar "preso" nos mínimos locais de baixa qualidade, gerando para tanto algumas perturbações, geração de novas soluções e também através do uso de buscas locais para o melhoramento de suas soluções.

Segundo Lourenço *et al.* (2002) [11], é melhor utilizar uma busca local do que simplesmente escolher um ponto aleatório do espaço de soluções para tentar melhorá-la.

A perturbação não deve ser muito "fraca", o que poderia fazer com que a solução não saia do atual ótimo local, mas tampouco não pode ser muito "forte", pois pode tornar o processo aleatório, não tirando proveito da qualidade da solução corrente. Veja no Quadro 1 o pseudo-código do algoritmo ILS:

Quadro 1: Pseudo-Código da meta-heurística ILS.

```
procedimento ILS(critérioDeParada,d,t)
1. s¹←Solução Inicial;
2. s ← s¹;
3. enquanto não critérioDeParada faça;
4. s²←Pertubação(s¹,d);
5. s²←BuscaLocal(s²);
6. CritérioAceitação(s,s¹,s²,T);
7. fim-enquanto;
8. retorne s;
fim ILS;
```

O ILS possui quatro componentes fundamentais: Geração de Solução Inicial, Busca Local, Perturbação e Critério de Aceitação:

Geração da Solução Inicial – Nesta etapa é criada a solução inicial do problema, que pode ser obtida de diversas maneiras. Normalmente as escolhas para a criação de uma solução se concentram em soluções aleatórias ou heurísticas gulosas. Quando uma solução gulosa é combinada com busca local, em geral o resultado é uma solução de melhor qualidade. Segundo Lourenço *et al.* (2002) [11], também é perceptível que soluções aleatórias necessitam de mais iterações para melhorar a qualidade da solução encumbente, portanto precisam de um maior tempo de processamento. Portanto, uma boa heurística de construção de solução inicial, pode facilitar a execução do programa, gastando menos tempo e obtendo uma solução de melhor qualidade. Em geral, não existe uma resposta genérica sobre qual algoritmo utilizar, mas a heurística gulosa é recomendável quando se necessita de soluções de baixo custo computacional. Estes autores também afirmam que para execuções em instâncias de grande porte a solução inicial pode perder relevância.

Busca Local – Esta é a parte do algoritmo que se preocupa em tentar melhorar a solução. A cada iteração são executados movimentos que atuam na vizinhança da solução corrente e tem o objetivo de gerar uma nova solução válida que será reavaliada na esperança de uma melhoria. Um exemplo seria a troca de duas posições em um *tour* qualquer do PCV. Esta parte do algoritmo é finalizada quando toda vizinhança é verificada e nenhuma solução de menor custo é encontrada, o que em outras palavras significa que um ótimo local foi atingido. É importante deixar claro que a busca local não deve desfazer a perturbação, implicando possivelmente na estagnação da solução.

Perturbação - O ILS utiliza-se de perturbações para tentar "escapar" de ótimos locais de baixa qualidade através da utilização de perturbações ao mínimo local (Figura 3). A força de uma perturbação é definida como o número de componentes que são modificados na solução. No caso do PCV, é o número arestas que são trocadas no *tour*. É muito importante que a busca local não consiga desfazer a perturbação executada, caso contrário a aplicação voltaria para o ótimo local anteriormente visitado.

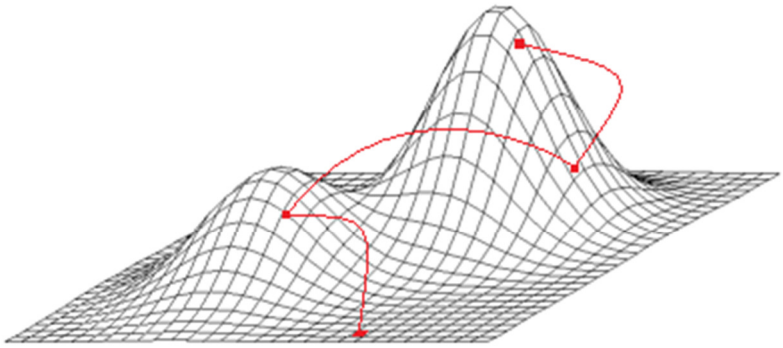

Figura 3: Exemplo visual do funcionamento da perturbação em um espaço de soluções.

Critério de Aceitação – O critério de aceitação consiste em aceitar ou não uma solução atingida anteriormente. A princípio o natural seria apenas aceitar as soluções caso elas sejam melhores que a solução anterior, porém, isto pode gerar problemas na diversificação das soluções, deixando o algoritmo "preso" em um ótimo local. Uma possível abordagem é reiniciar todo o processo quando ocorrer melhoras por algum determinado número de iterações.

2.4 GPU Computing

Definição de GPU - *Graphics Processing Unit* ou Unidade de Processamento Gráfico (GPU) é o nome dado a um tipo de processador especializado em computações gráficas. Hoje em dia, GPUs não apenas manipulam elementos gráficos computadorizados com eficiência, mas também são capazes de serem programáveis e efetuar milhares de cálculos em paralelo a cada instante.

2.4.1 CUDA – *Compute Unified Device Architecture*

CUDA [14] é uma arquitetura de computação paralela de propósito geral desenvolvida pela NVidia para tirar proveito da GPU e ser capaz de resolver uma gama de problemas computacionais complexos e não gráficos [13].

Para programar para a arquitetura CUDA, os desenvolvedores podem utilizar C, uma das mais difundidas linguagens de alto nível que permite uma implementação direta de algoritmos em paralelo. CUDA permite ao programador focar-se mais na paralelização do algoritmo do que na implementação em sí. Possui suporte à computação heterogênea, nas quais as partes seriais do programa são executadas na CPU e as partes em paralelo na GPU.

A linguagem também pode ser utilizada para aplicações já existentes, bastando reprogramar a função ou parte do código em que se deseja obter um melhor desempenho.

A CPU denominada na literatura de *host* e a GPU, chamada *device,* são tratadas como dispositivos distintos que possuem seu próprio espaço de memória, um computador pode possuir uma ou várias *devices*. Esta configuração também permite computação simultânea na CPU, evitando uma competição por recursos de memória.

As GPUs que possuem suporte a CUDA possuem centenas de núcleos que em conjunto podem executar milhares de *threads.* Cada núcleo possui recursos compartilhados, incluindo registradores e memória, que permitem a comunicação do núcleo sem fazer o envio para o barramento de memória do sistema. Informações detalhadas sobre esta arquitetura podem ser obtidas pela NVidia e GPGPU [15,16].

Na Figura 4, verifica-se em escala um comparativo de alto nível entre CPUs e GPUs. 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.

Figura 4: Comparativo entre arquiteturas da CPU e GPU NVidia.
Fonte: NVidia [12].

Diferença de Poder de Processamento entre CPU e GPU – Em função da tecnologia de fabricação atual para processadores convencionais ter atingido um limite físico referente à velocidade do *clock*, os fabricantes vêm investindo na elaboração de arquiteturas maciçamente paralelas. Entretanto, devido à natureza da arquitetura básica das GPUs, estas possuem uma ou até duas ordens de grandeza a mais do que o paralelismo atingido pelas CPUs.

No gráfico da Figura 5, observa-se a diferença de pico de desempenho entre CPUs e GPUs. Um processador *Intel* atinge apenas alguns *GFlops/s* enquanto que uma *GPU GTX480*, pode atingir mais que 1.25 *TeraFlop/s*.

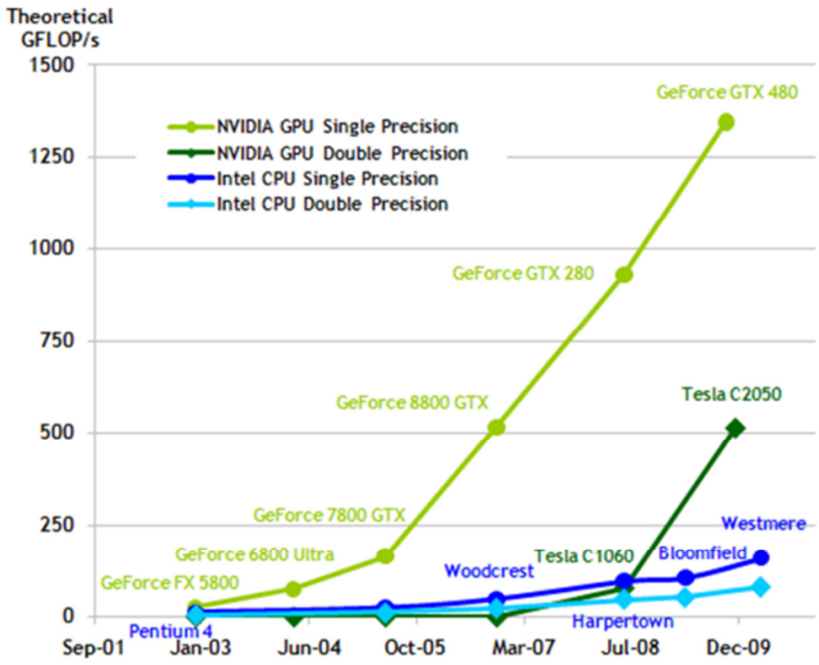

Figura 5: Comparativo entre CPUs e GPUs em número de operações de ponto flutuante.
Fonte: NVidia [12].

Arquitetura da Memória – Os dispositivos com suporte a CUDA possuem diferentes espaços de memória (Figura 6) que detém características particulares e refletem diretamente no desempenho da aplicação. Os espaços de memória estão divididos em: global, local, compartilhada, textura e registradores (Quadro 2).

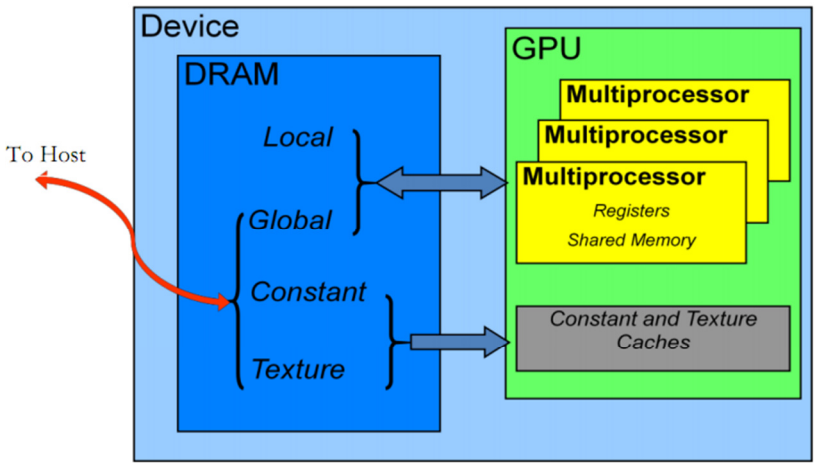

Figura 6: Diferentes espaços de memória em um dispositivo com suporte a CUDA.
Fonte: NVidia CUDA Best Pratices Guide [12].

Quadro 02: Detalhamento em alto nível das memórias da GPU.

Memória	Descrição
Global	Possui um tempo de acesso grande se comparado com os demais tipos de memória, pode ser utilizada para escrita e leitura sendo acessível globalmente por todos os núcleos. Não é *on-chip*.
Local	Possui o mesmo tamanho e tempos de acesso equivalentes ao da memória global, não possui *cache*, pode ser utilizada para escrita e leitura, sendo utilizada para salvar conteúdo de registradores quando não há espaço suficiente neles.
Compartilhada	Possui um tamanho pequeno, apenas alguns *KBytes* por *Streaming Processors* (SM), porém possui um tempo de acesso muito baixo quando não há acesso simultâneo. É tão rápido quanto um registrador, possui *cache*, está alocada *on-chip,* e *threads* do mesmo bloco podem acessá-la para cooperação.
Textura	Possui tamanho igual à memória Global, não está localizada *on-chip*, possui *cache*, a memória é otimizada para localidade espacial, só é permitido leitura e está disponível para acesso por todas as *threads* e o *host*.
Registradores	É o tipo de memória com o menor tamanho e está construída *on-chip,* não possui *cache*, e é a que possui o acesso mais rápido de todas. Acessar um registrador não consome sequer um ciclo de *clock* por instrução.

2.5 TRABALHOS RELACIONADOS

Nesta seção, são apresentados alguns trabalhos que possuem idéias afins e serviram de constribuição para esta dissertação. Devido à originalidade do trabalho, ainda não é possível encontrar uma grande quantidade de trabalhos relacionados. Nas próximas subseções cada um destes trabalhos é discutido sob o ponto de vista de sua contribuição para o desenvolvimento da dissertação.

2.5.1 Buscas Locais Paralelas em GPU por E.G.Talbi (2009)

Talbi (2009) [17] descreve que os algoritmos de busca local são uma classe de algoritmos para resolver problemas complexos na ciência e na indústria, e mesmo que estes consigam reduzir o tempo computacional para resolvê-los quando comparado a métodos exatos de otimização, o processo iterativo ainda é custoso para instâncias grandes. Como solução, Talbi propõe o uso de GPUs, como uma alternativa eficiente para realizar cálculos mais rapidamente se comparado a CPUs, propondo uma nova metodologia para projetar e implementar algoritmos clássicos de buscas locais como: subida da encosta, ILS e busca tabu em GPU.

Segundo Talbi (2009) [17], adaptar métodos de busca local em GPU não é algo direto, devido à forma como é feito o gerenciamento de memória da GPU. As transferências de memória de CPUs para GPUs são lentas. Estas cópias precisam ser minimizadas e gerenciadas manualmente.

A idéia de Talbi pode ser resumida em utilizar-se do poder da GPU para analisar cada vizinhança (Figura 7), e salvar o resultado do *fitness* em apenas um vetor que será copiado de volta para a CPU visando descobrir a vizinhança mais adequada, atualizar a solução, e repetir o processo até que um critério de parada seja atingido.

Mapear as possibilidades de cada algoritmo para serem resolvidos por *threads* não é algo trivial, e precisa ser analisado em cada caso específico.

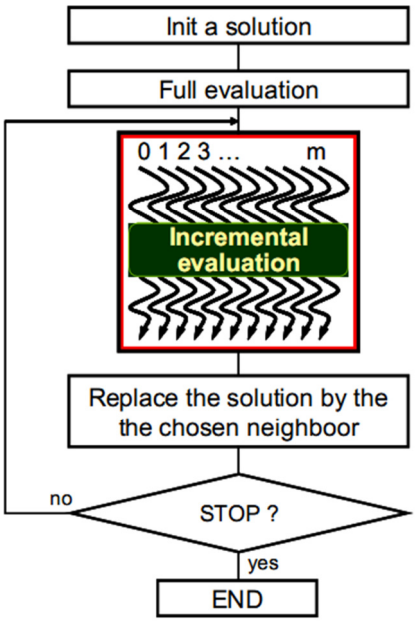

Figura 7: Visão geral do modelo proposto por Talbi para a implementação de buscas locais em GPU. Nota-se que os movimentos são realizados simultaneamente e a solução é atualizada com o melhor movimento até obedecer um critério de parada. **Fonte:** Talbi (2009, pp.10) [17].

2.5.2 Uma heurística paralela para o problema de roteamento de veículos com coleta e entrega simultânea por Subramanian *et al.* (2010) [18]

Subramanian *et al.* (2010) [18] propôs um algoritmo misto através da junção do ILS e do *Random Variable Neighborhood Descent* (RVND) através de uma heurística *multistart* para o problema de roteamento de veículos com coleta e entrega simultânea, que por sua vez é um problema que também resolve o PCV. O autor realizou testes para instâncias de tamanho entre 50 e 400 clientes em arquiteturas *multi-cores* utilizando 32 nodos *Symmetric Multi-Processing* (SMP), onde cada nodo portava dois processadores Xeon 2.66 GHz *quad-core*.

O problema pesquisado por Subramanian *et al.* (2010) [18] também é um problema NP-Difícil, e sua abordagem tira proveito do alto grau de paralelismo das arquiteturas *multi-core,* o que seria uma excelente prova de conceito para GPUs que atualmente podem chegar a ter 480 núcleos, gastam uma fração da energia de todos estes nodos e possuem um alto poder computacional.

Os resultados atingidos por Subramanian *et al.* (2010) [18] melhoraram a solução de vários problemas propostos pela literatura, e provaram ser escaláveis, conseguindo tirar vantagens do grande número de processadores para resolução de instâncias grandes.

Dentre as idéias propostas por estes autores está seu algoritmo que foi utilizado como *template* para a construção do modelo deste trabalho, conforme apresentado no Quadro 03. Os autores utilizam o ILS atrelado ao RVND, onde este último possui uma lista de vizinhanças

que são escolhidas ao acaso. Em caso de melhoria da solução a lista com as vizinhanças é repovoada, e o processo torna a tentar melhorar a solução, caso essa vizinhança não melhore a solução, ela é removida, e uma nova vizinhança aleatoriamente escolhida é utilizada, até que não existam mais vizinhanças disponíveis. Após isto, a execução da função RVND (linha 9 do Quadro 3) é finalizada e devolve o contexto para o procedimento ILS-RVND.

Quadro 3: Visão alto nível do algoritmo ILS-RVND proposto por Subramanian *et al.* (2010) [18].

```
1:  Procedure ILS − RVND(MaxIter, MaxIterILS, γ, v)
2:  LoadData( );
3:  f* := ∞;
4:  for i := 1,...,MaxIter do
5:    s := GenerateInitialSolution(γ, v, seed);
6:    s' := s;
7:    iterILS := 0;
8:    while iterILS ≤ MaxIterILS do
9:      s := RVND(N(.), f(.), r, s) {r = of neighborhoods}
10:     if f(s) < f(s') then
11:       s' := s;
12:       f(s') := f(s);
13:       iterILS := 0;
14:     end if;
15:     s := Perturb(s');
16:     iterILS := iterILS + 1;
17:   end while;
18:   if f(s') < f* then
19:     s* := s';
20:     f* := f(s');
21:   end if;
22: end for;
23: return s*;
24: end ILS-RVND.
```

```
1:  Procedure RVND(N(.), f(.), r, s)
2:  Initialize the Neighborhood List (NL);
3:  while NL ≠ 0 do
4:    Choose a neighborhood N(η) ∈ NL at random;
5:    Find the best neighbor s' of s ∈ N(η);
6:    if f(s') < f(s) then
7:      s := s';
8:      s := IntraRouteSearch(s);
9:      Update NL;
10:   else
11:     Remove N(η) from the NL;
12:   end if;
13: end while;
14: return s;
15: end RVND.
```

Em sua versão paralela, Subramanian *et al.* (2010) [18] também afirmam que o balanceamento de carga é essencial para o desempenho do modelo, visto que o tempo necessário para executar uma iteração não é conhecido *a priori* e pode variar bastante.

Para finalizar, a seção de trabalhos futuros dos autores motivou a utilização deste algoritmo no presente trabalho, pois os autores tinham como objetivo tirar maior proveito das arquiteturas *multi-cores*. Na Figura 8 é possível visualizar claramente a diferença de ordem de grandeza em número de processadores entre CPU e GPU.

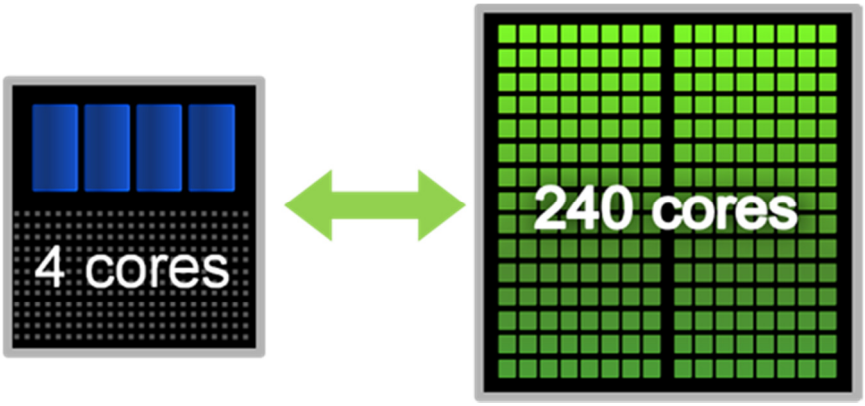

Figura 8: Comparativo entre CPU e GPU em número de núcleos. **Fonte:** NVidia [34].

2.5.3 O problema de roteamento de veículos: extensões e métodos de resolução estado da arte por Armin *Lüer et al. (2009).*

Em seu artigo, *Lüer et al.*(2009) [19] realizam uma revisão bibliográfica do PCV, fazendo uma agregação dos métodos de resolução existentes e os avanços tecnológicos da área que tem possibilitado a obtenção de melhores resultados. Segundo estes autores os métodos que tem atingido melhores sucessos são as meta-heurísticas híbridas.

Vale à pena ressaltar a seção que comenta sobre a influência dos avanços tecnológicos em que recentemente tem se observado o crescimento da potência dos computadores, bem como a diminuição do custo de seus componentes. Isto tem levado a uma nova linha de investigação: o uso de *hardware* especializado em tarefas de computação gráfica tridimensional, que se aproveita de arquiteturas altamente paralelas das placas de vídeo utilizadas originalmente para desenho 3D e jogos.

3 MODELO PARALELO IMPLEMENTADO EM GPU

Este capítulo apresenta os detalhes do modelo proposto, aspectos particulares e teóricos. Ao longo do texto também é explicado como os algoritmos foram paralelizados para tirar maior proveito da arquitetura de GPUs.

3.1 Disponibilidade do Código para Análise

O código fonte desta aplicação bem como os arquivos para testes encontram-se disponíveis para *download* [20] sob a licença *Creative Commons Attribution-Share Alike 3.0 Brazil License* [21].

3.2 Linhas gerais sobre a abordagem utilizada

Uma parte considerável do tempo de pesquisa desta dissertação foi gasto implementando o esquema mostrado na Figura 9, primeiramente em CPU, de forma a entender exatamente o que deveria ser transposto para as GPUs. A implementação direta em GPU é um passo não recomendado, pois esta estratégia pode propiciar a ocorrência de erros que consequentemente causarão um aumento do tempo de implementação [22].

Um caminho investigado nesta pesquisa que não é recomendado seria utilizar os algoritmos seqüenciais de CPU, e simplesmente executá-los separadamente em cada processador da GPU para se obter a melhor solução.

Além de não ser uma alternativa viável, a arquitetura da GPU sendo mais simples não permite a execução de cada *kernel* por mais de alguns segundos, obrigando a abortar caso a execução se prolongue por muito tempo.

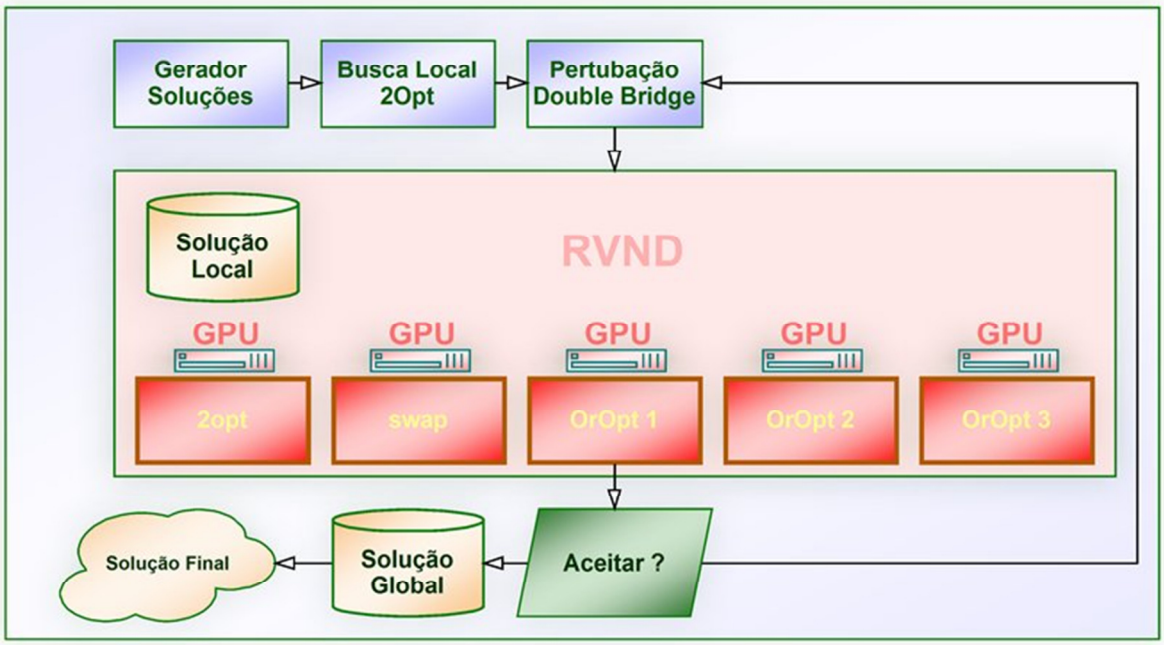

Figura 9: Visão geral do modelo proposto.

3.3 Visão Geral do Modelo

O modelo proposto neste trabalho implementa um algoritmo paralelo e metaheurístico em GPU, análogo ao proposto por Subramanian *et al.* (2010) [18] (Quadro 3). Em linhas gerais, o modelo substitui a busca local do ILS pelo RVND, que é constituído por um conjunto de movimentos (vizinhanças) em paralelo que tentam melhorar a solução conforme descrito na seção 2.6.2. Estes movimentos foram paralelizados e implementados em GPU.

3.4 Visão Detalhada de Cada Componente

A seguir detalharemos cada módulo:

3.4.1 Entrada de Dados

A entrada de dados é feita a partir de um arquivo obtido da biblioteca TSPLIB que possui instâncias do PCV. Na Figura 10 é possível visualizar o arquivo que representa a instância “d493.tsp”, que possui 493 coordenadas (X,Y) no plano.

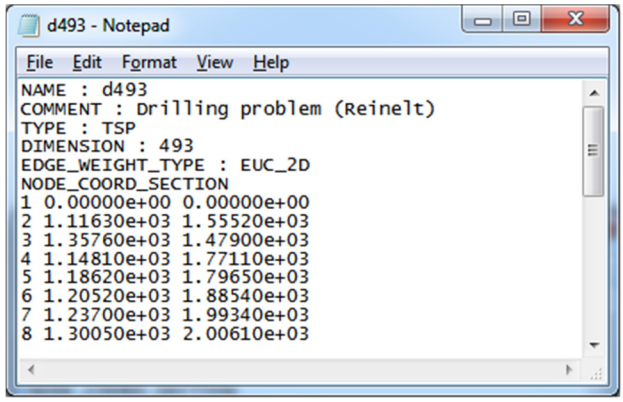

Figura 10: Exemplo de arquivo de entrada.

Após o arquivo ser carregado em memória, é calculada a distância euclidiana entre todas as cidades resultando em uma matriz de tamanho N².

3.4.2 Construção de Soluções

Este módulo possui três funções (Figura 11) e tem por objetivo gerar as soluções iniciais para o problema. Uma explicação mais detalhada de cada um destes módulos será realizada a seguir:

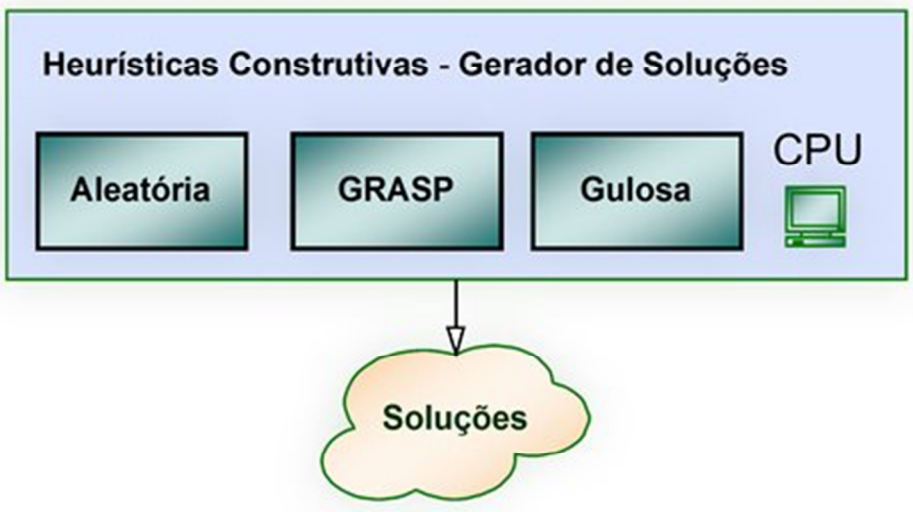

Figura 11: Heurísticas utilizadas para geração de soluções.

a) Aleatória

A construção de soluções de forma aleatória para o PCV é simples, porém as soluções tendem a possuir um *tour* com alto custo (*score*) e deixam portanto a desejar no tocante a sua qualidade.

Para construir soluções aleatórias para o PCV, basta criar uma lista com *n* cidades e removê-las ao acaso, inserindo as cidades já sorteadas em outra lista ou vetor. O pseudocódigo do algoritmo pode ser verificado no Quadro 04.

Quadro 04: Pseudocódigo da heurística de construção aleatória.
Fonte: Souza, M.J.F [23].

```
procedimento ConstrucaoAleatoria(g(.), s);
1  s ← ∅;
2  Inicialize o conjunto C de elementos candidatos;
3  enquanto (C ≠ ∅) faça
4      Escolha aleatoriamente t_escolhido ∈ C;
5      s ← s ∪ {t_escolhido};
6      Atualize o conjunto C de elementos candidatos;
7  fim-enquanto;
8  Retorne s;
fim ConstrucaoAleatoria;
```

A Figura 12 demonstra duas possíveis soluções geradas aleatoriamente e nelas existem vários cruzamentos e interseções de arestas, que significam que as soluções aleatórias geradas passam duas vezes pelo mesmo lugar de forma desnecessária, dispendiosa e não inteligente.

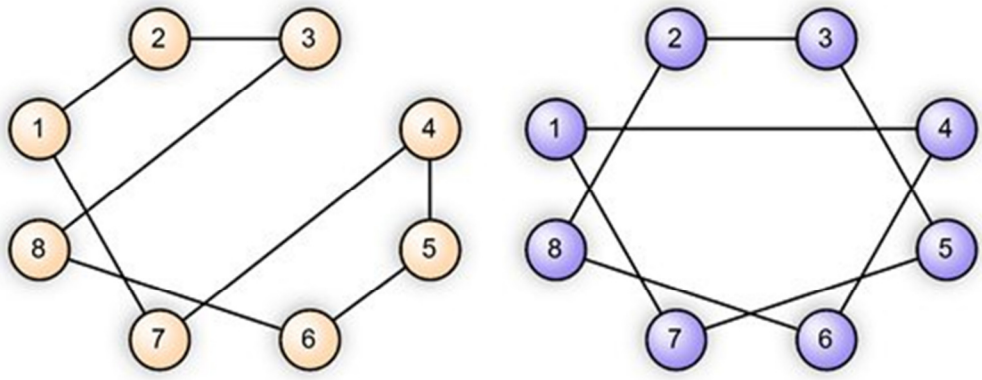

Figura 12: Exemplos de construção de possíveis soluções aleatórias.

b) Gulosa

Outra maneira de se gerar uma solução para o PCV consiste em escolher a cada iteração a cidade que estiver mais próxima da cidade atual e que ainda não tenha sido incluída na solução parcial. Apesar de parecer interessante, esta heurística produz apenas uma solução inicial e não possuir visão em longo prazo, o que a torna o processo míope, porém ainda assim é usualmente melhor que a geração aleatória. O pseudocódigo do algoritmo pode ser verificado no Quadro 05.

Quadro 05: Pseudocódigo da heurística de construção gulosa.
Fonte: Souza, M.J.F [23].

```
procedimento ConstrucaoGulosa(g(.), s);
1  s ← ∅;
2  Inicialize o conjunto C de elementos candidatos;
3  enquanto (C ≠ ∅) faça
4      g(t_melhor) = melhor{g(t) | t ∈ C};
5      s ← s ∪ {t_melhor};
6      Atualize o conjunto C de elementos candidatos;
7  fim-enquanto;
8  Retorne s;
fim ConstrucaoGulosa;
```

A Figura 13 representa uma possível solução gerada por uma heurística gulosa. O *tour* calculado evita as arestas pontilhadas que possuem um alto custo.

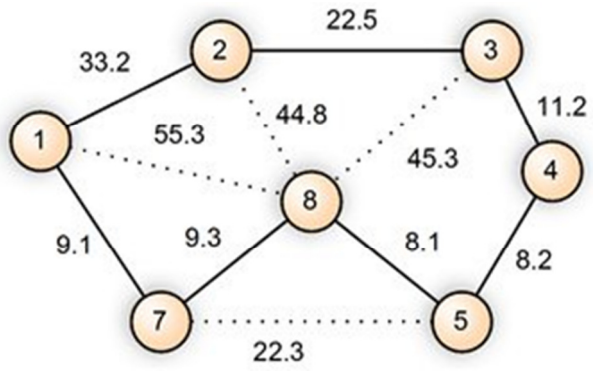

Figura 13: Exemplo de construção de um possível *tour* para a heurística gulosa.

c) GRASP

A heurística construtiva do *Greedy Randomized Adaptive Search Procedure* (GRASP) [24] é a principal forma utilizada neste trabalho para a construção de soluções iniciais eficientes para o PCV, pois possui características das duas heurísticas citadas anteriormente. Não é totalmente aleatória, pois não seleciona as cidades ao acaso, e também não é completamente gulosa, pois não seleciona necessariamente o melhor candidato a cada passo. A heurística construtiva GRASP une o melhor das duas propostas, as vantagens da construção aleatória e gulosa, o que tende a gerar soluções de boa qualidade (Figura 14).

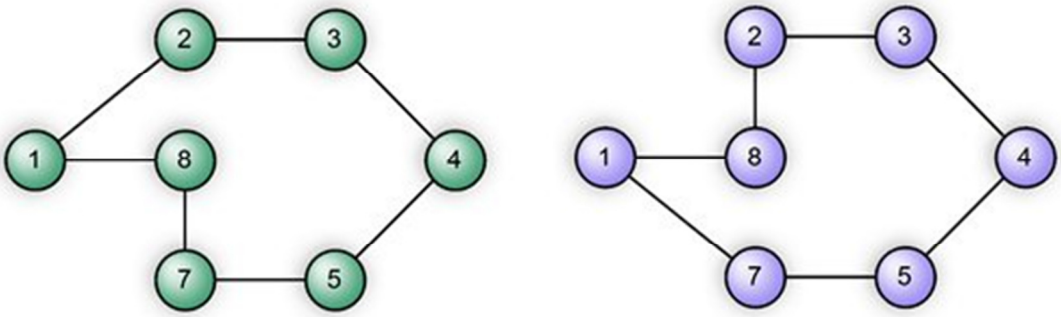

Figura 14: Exemplos de duas possíveis construções GRASP.

Na prática, a heurística GRASP funciona da seguinte forma: primeiro na fase de construção, sorteia-se um elemento para compor a solução e a partir disto tenta-se adicionar mais uma cidade à solução. Neste momento é criada uma *Restricted Candidate List* (RCL), onde ficam armazenadas as melhores inserções e o custo extra que cada uma irá trazer à solução. Esta lista possui um tamanho X, determinado por um valor *alfa* baseado em *n*, tamanho do problema. Neste caso, um problema de tamanho $n=100$ e $\alpha = 0.05$ possuiria tamanho $|X| = 5$.

Dado que esta lista de tamanho |X| está construída e que a RCL contém apenas os melhores candidatos que acarretam num menor aumento de custo (pois queremos minimizar o custo), neste instante é feito um sorteio de forma aleatória que define qual candidato de X irá fazer parte da solução, e o processo se repete até que a RCL esteja vazia.

O tamanho da RCL é essencial para o sucesso da heurística, pois se o *alfa* é muito grande ($\alpha = 1$), a heurística semi-gulosa se torna puramente aleatória, pois a lista representa todas as possibilidades. Caso *alfa* seja muito pequeno ($\alpha = 0$), a lista pode se restringir a

apenas um candidato, o que a torna puramente gulosa. *Alfa* deve ser determinado de forma cautelosa. O pseudocódigo do algoritmo pode ser verificado no Quadro 06.

Quadro 06: Pseudocódigo da heurística de construção GRASP.
Fonte: Souza, M.J.F [23].

procedimento $Construcao(g(.), \alpha, s)$;
1 $s \leftarrow \emptyset$;
2 Inicialize o conjunto C de candidatos;
3 enquanto $(C \neq \emptyset)$ faça
4 $g(t_{min}) = \min\{g(t) \mid t \in C\}$;
5 $g(t_{max}) = \max\{g(t) \mid t \in C\}$;
6 $LCR = \{t \in C \mid g(t) \leq g(t_{min}) + \alpha(g(t_{max}) - g(t_{min}))\}$;
7 Selecione, aleatoriamente, um elemento $t \in LCR$;
8 $s \leftarrow s \cup \{t\}$;
9 Atualize o conjunto C de candidatos;
10 fim-enquanto;
11 Retorne s;
fim $Construcao$;

3.4.3 Perturbação *double-bridge*

A perturbação d*ouble-bridge* é um exemplo de perturbação simples, porém de grande eficácia que desconecta quatro arestas de uma solução do PCV e as reconecta de forma diferente conforme pode ser visualizado na Figura 15. Segundo Lourenço *et al*. (2002) [11], quase todos os trabalhos de ILS e PCV têm utilizado este tipo de perturbação que tem surtido efeito para instâncias de todos os tamanhos. Este tipo de perturbação funciona porque atua mesmo em cidades distantes, o que favorece a busca local.

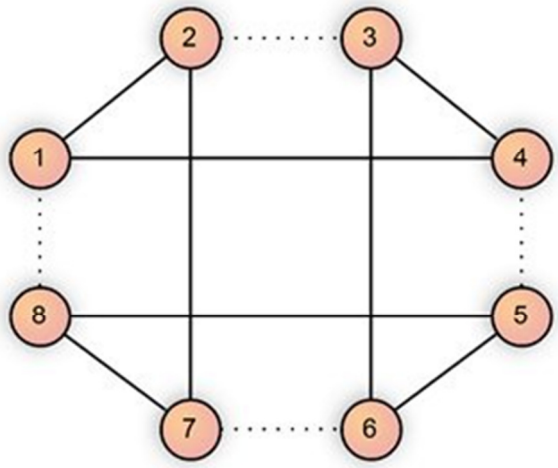

Figura 15: Exemplo de uma perturbação *double-bridge*.

Para Lourenço *et al*. (2002) [11], um fato a ser considerado é que esta perturbação não pode ser facilmente restaurada, nem mesmo por algoritmos como: *2-Opt*, *3-Opt* e *Lin-Kernighan* [25] que tem se mostrado o melhor algoritmo de busca local para o PCV.

Outra característica desta perturbação é que ela altera o *tour,* mas não piora a solução de forma abrupta, o que faz com que a partir de uma solução com custo X, seja possível conseguir uma solução com custo X + C ou X - C, onde C não é muito grande.

Ainda Lourenço *et.al* (2002) afirmam que o uso de pequenas perturbações podem levar a grandes *speed-ups. A* explicação é que para o mesmo tempo de computação um número muito maior de buscas locais pode ser aplicado.

3.4.4 Buscas Locais

As buscas locais utilizadas nesta dissertação bem como todo o processo de paralelização e implementação em GPU serão detalhadas nesta seção.

1. **2-Opt** – Croes [27]

O algoritmo *2-Opt* foi inicialmente proposto por Croes (1958) [27]. Segundo Hasegawa *et al.* (1997) [26] o *2-Opt* pode ser considerado um algoritmo básico entre outros métodos heurísticos, pois é possível encontrar uma solução minimamente boa com um algoritmo simples, que produz bons resultados em instâncias Euclidianas a respeito de tempo e qualidade de solução [28]. O movimento *2-opt* remove duas arestas não adjacentes de uma solução do PCV, quebrando o *tour* em duas partes, e reconecta os caminhos de outra maneira (Figura 16).

O algoritmo pode ser descrito em alto nível da seguinte forma: dado um *tour* qualquer o *2-opt* tenta melhorar a solução de forma incremental, trocando as arestas do *tour* duas a duas por vez. O nome deriva do fato de que à partir de uma solução inicial a melhor solução alterando Z arestas não adjacentes é obtida ao final.

O processo pode ser visualizado de forma mais clara na Figura 16, onde a cada iteração o algoritmo seleciona duas arestas não adjacentes, no exemplo {a,b} e {c,d} e as troca por {a,c} e {b,d}. Antes de efetuar a troca, uma estimativa do novo custo é realizada, e caso diminua, a troca é efetivada, e uma nova iteração é feita até que não existam mais melhoras que reduzam o custo daquele *tour*. Segundo Bentley (1990) [29] o movimento *2-opt*

possui complexidade O(n²) e calcula soluções que ficam apenas alguns pontos percentuais do ótimo global [30].

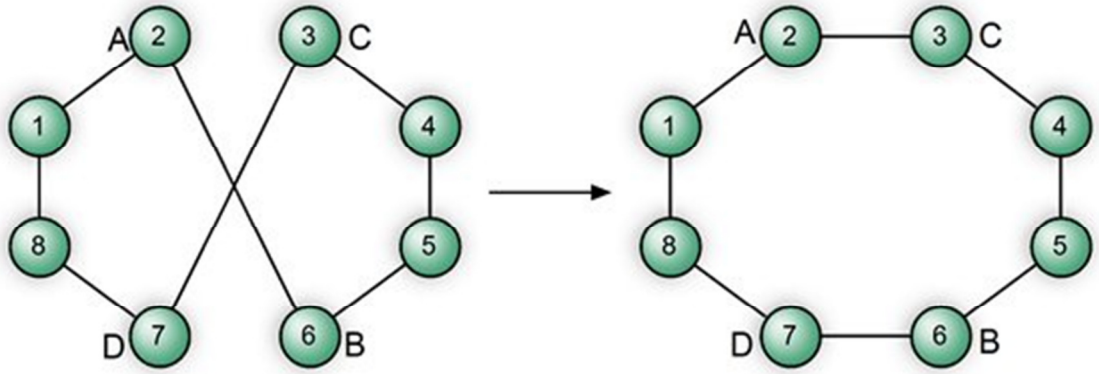

Figura 16: Exemplo de funcionamento da busca local *2-opt*.

Naturalmente, para executar o algoritmo *2-opt*, todas as permutações possíveis seriam feitas, ou seja, supondo n=5, seriam 25 possibilidades para serem analisadas, i e j variando de [0,5], dois *fors* aninhados. Porém, para viabilizar e facilitar a paralelização, estes dois laços foram linearizados e substituídos por apenas um (linha 05 do Quadro 7), onde os índices *a* e *b* foram calculados pelas linhas 06 e 08 a partir de *i*.

Quadro 7: Visão em alto nível sequencial do algoritmo *2-opt e*m CPU.

```
procedimento 2optCPU(vetor soluçãoAntiga)
01. melhorTrocaA ← -1;
02. melhorTrocaB ← -1;
03. melhorTrocaCusto ← - ∞;
04. maximoIterações = ((dim % 2 + dim / 2) * dim) – dim;
05. para i=1 até maximoIterações faça
06.     a ← (i-1) % dim;
07.     k ← ((i-a)/dim) + 1;
08.     b ← (a+k) % dim ;
09.     se (custo(ant(a),b) + custo(a,prox(b))) < custo(ant(a),a) + custo(b,prox(b)) então
10.         se (custo(ant(a),b) + custo(a,prox(b))) > melhorTrocaCusto) então
11.             melhorTrocaA ← a;
12.             melhorTrocaB ← b;
13.             melhorTrocaCusto ← custo(ant(a),b) + custo(a,prox(b))
14.         fim se
15.     fim se
16. fim para
17. insira arco(ant(melhorTrocaA),melhorTrocaB) e
18. insira arco(melhorTrocaA,prox(melhorTrocaB));
19. remova arco(anterior(melhorTrocaA),melhorTrocaA);
20. remova arco(melhorTrocaB,próximo(melhorTrocaB));
21. retorne vetor soluçãoNova;
fim 2optCPU;
```

Para *n=5,* o numero máximo de permutações seriam 10, obtido pela linha 4 do algoritmo, e não 25, como seria necessário para o caso de tentar listar todas as possibilidades. Isto é reforçado pelo fato de que várias das permutações não fazem sentido, como por exemplo, [3,3] que seria trocar a posição por ela mesma, ou [1,3] que é equivalente a [3,1].

Na tabela 02, é possível observar todos os valores gerados para a e b a partir de *i=1* até o número máximo de iterações (permutações), sendo que cada par de valores *a* e *b* representam uma tentativa de permutação que deve ser executada:

Tabela 02: Indíces para efetuar as trocas.

i	a	b
1	0	1
2	1	2
3	2	3
4	3	4
5	4	0
6	0	2
7	1	3
8	2	4
9	3	0
10	4	1

Após este mapeamento ter sido concluído, basta realizar as trocas de a e b desde i=1 até i = maximoIterações, conforme descrito entre as linhas 09 e 13 do algoritmo e identificar a troca que acarretará em maior benefício.

Para obter a versão em paralelo deste algoritmo é preciso dividir o trabalho entre as linhas 05 e 16 com todas as *threads* executadas em GPU. Neste ponto esta tarefa se torna bem menos complexa, pois já foi criado intencionalmente um mapeamento entre o índice *i* e qual troca (tarefa) será feita por cada índice através da obtenção de *a* e *b.*

Para facilitar a compreensão do processo, apresentaremos primeiro um exemplo para posteriormente apresentar a formalização do algoritmo: Suponha-se que o problema possui *n=10*, isto implica que o número máximo de possibilidades ou iterações (linha 04 do Quadro 08) possui valor 40.

Suponha-se também que na GPU foi definido que seriam utilizados dois blocos com 4 *threads* por bloco, totalizando 8 *threads* que dividiriam as tarefas de forma igualitária de acordo com a tabela 03:

Tabela 03: Divisão de tarefas entre as *threads*.

Thread 01	Thread 02	Thread 03	Thread 04	Thread ...	Thread 08
01,02,03,04,05	06,07,08,09,10	11,12,13,14,15	16,17,18,19,20	...	36,37,38,39,40

Visto que todas as tarefas estão divididas, cada *thread* será alocada para um processador específico na GPU de forma transparente, e a execução realizada em paralelo.

Na versão em CPU, existe apenas uma posição de memória para indicar qual será a melhor troca a ser feita, já na versão em GPU serão necessárias 8 posições de memória para armazenar os melhores candidatos a troca de cada *thread.*

Após a execução em paralelo este vetor é copiado para a CPU que é responsável por encontrar a melhor troca dentre as possibilidades encontradas pelas *threads,* fazendo com que o algoritmo retome sua execução normalmente. Note que a GPU foi utilizada como auxiliar na parte mais custosa da execução. O código do algoritmo executado em GPU pode ser definido conforme mostrado no Quadro 8 (o trecho em paralelo está em negrito).

Quadro 8: Visão em alto nível paralela do algoritmo *2-opt e*m GPU.

```
procedimento 2optGPU(vetor oldSolution, int dim)
01. melhorTrocaA ← -1;
02. melhorTrocaB ← -1;
03. melhorTrocaCusto ← - ∞;
04. maximoIterações = ((dim % 2 + dim / 2) * dim) – dim;
05. numTarefasPorThread ← máximoIterações / totalThreads;
06. threadStart ← ThreadId * numTarefasPorThread;
07. threadEnd  ← threadStart + numTarefasPorThread;
08. para i=threadStart até ThreadEnd faça
09.     a ← (i-1) % dim;
10.     k ← ((i-a)/dim) + 1;
11.     b ← (a+k) % dim ;
12.  se (custo(ant(a),b)+custo(a,prox(b))) < custo(ant(a),a) + custo(b,prox(b)) então
13.         se (custo(ant(a),b) + custo(a,prox(b))) > melhorTrocaCusto) então
14.               melhorTrocaAGPU[threadId] ← a;
15.               melhorTrocaBGPU[threadId] ← b;
16.           melhorTrocaCustoGPU[threadId] ← custo(ant(a),b) + custo(a,prox(b))
17.         fim se
18.     fim se
19. fim para
20. Copiar melhorTrocaAGPU,melhorTrocaBGPU,melhorTrocaCustoGPU para CPU
21. Encontrar melhorTrocaCusto e atribuir a MelhorTrocaA e MelhorTrocaB na CPU
17. insira arco(ant(melhorTrocaA),melhorTrocaB);
18. insira arco(melhorTrocaA,prox(melhorTrocaB));
19. remova arco(ant(melhorTrocaA),melhorTrocaA);
20. remova arco(melhorTrocaB,próx(melhorTrocaB));
21. retorne vetor newSolution;
fim 2opt;
```

Nos Quadros 7 e 8 estão representados apenas os códigos do algoritmo *2-opt*, que são chamados *n* vezes até que a função não consiga mais melhorar a solução. O *loop* não foi representado para simplificar a representação. No quadro 9 é possível visualizar parte do kernel do algoritmo *2-opt*.

Quadro 9: Visão parcial do kernel *2-opt* (GPU).

```
__global__ void localSearch2OptKernel(...){

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    flipPointAD[idx] = -1;
    flipPointBD[idx] = -1;
    float melhorInsercaoCusto = 999999999.0,
    float custoInicial = *score;
    int threadStart = idx * threadWorkLoad +1;
    int threadEnd   = threadStart + threadWorkLoad;
    for( int i = threadStart ; i <= threadEnd ; i++){

                ia = (i-1) % dim;
                k  = ((i - ia) / dim) + 1;
                ib = ( ia+k ) % dim;

                a  = ia;
                b  = ib;

                pa = previousD(ia,dim);
                nb = nextD(ib,dim);

                v1 = sol[pa];
                v2 = sol[nb];
                v3 = sol[b];
                v4 = sol[a];

    insercao =  tex2D(texMatrix,v1,v3) + tex2D(texMatrix,v4,v2);
    remocao  =  tex2D(texMatrix,v1,v4) + tex2D(texMatrix,v3,v2);

                if( insercao < remocao ){
                      if( insercao < melhorInsercaoCusto){
                                  melhorInsercaoCusto = insercao;
                                  estimativeD[idx] = insercao;
                                  flipPointAD[idx] = a;
                                  flipPointBD[idx] = b;
                      }
                }
    }

    a = flipPointAD[idx] ; b = flipPointBD[idx];
    pa = previousD(a,dim); nb = nextD(b,dim);

    if( a != -1 && b != -1 ){
          v1 = sol[pa];v2 = sol[nb];v3 = sol[b];v4 = sol[a];
          insercao = tex2D(texMatrix,v1,v3) + tex2D(texMatrix,v4,v2);
          remocao  = tex2D(texMatrix,v1,v4) + tex2D(texMatrix,v3,v2);
          estimativeD[idx] = insercao;}
    else{ estimativeD[idx] = custoInicial;}
    }
```

Na linha 03 é identificado o id da *thread* que está sendo executada, e os vetores *flipPointAD* e *flipPointBD* nas linhas 04 e 05 representam os pontos onde a CPU deve efetuar o movimento. No vetor *estimativeD* na linha 33 está armazenado a estimativa de melhora. Já nas linhas 8 e 9 são determinados os limites de trabalho de cada *thread*. A alocação de variáveis e outras linhas não importantes para as explicações foram removidas para facilitar a compreensão, o código na integra pode ser obtido em [20].

A partir do momento em que o laço da linha 10 é iniciado, são calculados os índices *ia* e *ib*, que serão atribuídos à *a* e *b*. Entre as linhas 19 e 37, é testado se o movimento vai de fato reduzir o custo da solução, perceba que nas linhas 27 e 28 os valores são acessados através de uma textura bidimensional para reduzir o tempo de acesso.

O fato de as linhas 16 e 17 fornecerem índices adjacentes como $a = 0$ e $b = 1$ não é um problema, pois os índices *a* e *b* servem apenas de referencial, pois trabalhamos com o anterior de *a* e o próximo de *b*, obtendo assim estimativas corretas.

Na linha 43 do quadro 09, é verificado por segurança se *a* e *b* não possuem valores inválidos. A partir do momento em que todas as *threads* foram finalizadas, este vetor é copiado para a CPU, onde o vetor é escaneado e a melhor opção que implica em uma maior redução de custo é efetivada.

A fórmula relativa às linhas 12 a 14 que são chaves para a paralelização do problema foram obtidas através de dedução matemática. Para a coluna relativa ao índice *a* (tabela 02), precisamos de uma função circular que gere índices na seguinte sequência (0,1,2,3,4,0,1,2,3,4,...), esta sequência é obtida através da linha 12, já para obter o índice b, precisamos dos cálculos obtidos pelas linhas 13 e 14 que fornecem os índices maiores que o *a*, pois estamos interessados na matriz triangular superior, onde todos os índices de *b* são maiores que *a*, tabela 4.

Tabela 4: Indíces das trocas e índices gerados pela fórmula (linhas 12 a 13 do quadro 09).

	0	**1**	**2**	**3**	**4**
0	-	1	6	9	
1	-	-	2	7	10
2	-	-	-	3	8
3	-	-	-	-	4
4	5	-	-	-	-

2. *Swap*

O s*wap* também foi um dos métodos baseados em vizinhanças implementados neste trabalho, e consiste em executar todas as possíveis permutações de vértices dois a dois. O funcionamento do operador pode ser visualizado na Figura 17.

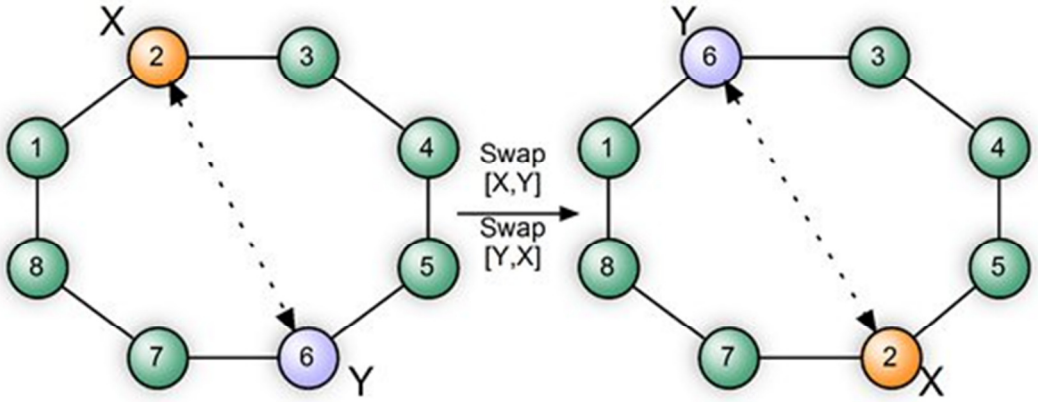

Figura 17: Possível movimento do operador *swap*.

A complexidade deste algoritmo é bastante intuitiva, pois troca-se todos os elementos com todos os outros restantes, o que implica em uma complexidade O(n²). O algoritmo em alto nível de forma seqüencial e linearizado para CPU pode ser visualizado no Quadro 10.

Quadro 10: Visão em alto nível sequencial do algoritmo *swap e*m CPU.

```
procedimento swapCPU(vetor oldSolution, int dim)
01. maximoIterações ← (dim * (dim-1)) / 2;
02. indiceSwapA ← -1;
03. indiceSwapB ← -1;
04. melhorTrocaCusto ← ∞;
05. para i←0 até maximoIterações faça
06.     b ← int (sqrt(2 * (i+1)) + 0.5);
07.     a ← b -1 – (b * (b+1) / 2 – (i+1) );
08.     estimativa ← obtemEstimativaDeTroca(oldSolution,a,b)
09.     se (estimativa < melhorCusto) então
10.             indiceSwapA ← a;
11.             indiceSwapB ← b;
12.             melhorTrocaCusto ← estimativa
13.     fim se
14. fim para
15. newSolution ← efetuaTroca(oldSolution,indiceSwapA,indiceSwapB);
16. retorne vetor newSolution;
fim swapCPU;
```

Assim como o algoritmo *2-opt*, o algoritmo *swap* também teve seu *loop* principal linearizado (linha 05 do Quadro 10) com o intuito de facilitar a paralelização em GPU. Se fossemos construir o algoritmo da maneira tradicional, seriam utilizadas duas iterações

aninhadas, variando *i* e *j* no intervalo [0,n], porém, a divisão de tarefas entre os processadores de forma igualitária utilizando dois índices *i* e *j* não é uma tarefa trivial.

O processo de linearização é um desafio matemático, que consiste em analisar os índices, no caso *i*, e achar uma fórmula que mapeie em outros dois índices *a* e *b.* Uma vez que este problema é resolvido, o processo de paralelização se torna menos complicado.

Nas linhas 06 e 07 do Quadro 10 é possível visualizar as fórmulas que convertem o valor de *i* nos índices *a* e *b.* Mesmo que existam outras fórmulas mais simples que realizem este mapeamento, esta foi a fórmula passível de ser deduzida.

A paralelização do algoritmo *swap* é análoga à paralelização realizada pelo algoritmo *2-opt*, mesmo assim, o processo será exemplificado para facilitar a compreensão.

Suponha que a instância a ser resolvida possua tamanho n=10. Neste caso, o número de permutações possíveis é de 45 (linha 01 do Quadro 10).

Quadro 11: Visão em alto nível paralela do algoritmo *swap e*m GPU.

```
procedimento swapGPU(vetor oldSolution, int dim)
01. maximoIterações ← (dim * (dim-1)) / 2;
02. indiceSwapA ← -1;
03. indiceSwapB ← -1;
04. melhorTrocaCusto ← ∞;
05. numTarefasPorThread ← máximoIterações / totalThreads;
06. threadStart ← ThreadId * numTarefasPorThread;
07. threadEnd  ← threadStart + numTarefasPorThread-1;
08. para i=threadStart até ThreadEnd faça
06.     b ← int (sqrt(2 * (i+1)) + 0.5);
07.     a ← b -1 – (b * (b+1) / 2 – (i+1) );
08.     estimativa ← obtemEstimativaDeTroca(oldSolution,a,b)
09.     se (estimativa < melhorTrocaCustoGPU[threadId]) então
10.             indiceSwapAGPU[ThreadId] ← a;
11.             indiceSwapBGPU[ThreadId] ← b;
12.             melhorTrocaCustoGPU[ThreadId] ← estimativa
13.     fim se
14. fim para
20. Copiar indiceSwapAGPU,indiceSwapBGPU,melhorTrocaCustoGPU para CPU
21. Encontrar melhorTrocaCusto e atribuir a indiceSwapA e indiceSwapB
22. newSolution ← efetuaTroca(oldSolution,indiceSwapA,indiceSwapB);
16. retorne vetor newSolution;
fim swapGPU;
```

Considerando também que esta tarefa esteja sendo executada por 2 blocos com 2 *threads* por bloco, totalizando 4 *threads,* tem-se que será necessário dividir 45 tarefas por 4 *threads,* que teria como resultado mais balançeado a divisão inteira que consta na Tabela 05.

Tabela 05: Divisão de tarefas entre as *threads*.

Thread 01	Thread 02	Thread 03	Thread 04
11 tarefas	11 tarefas	11 tarefas	12 tarefas
0,1,2,3,...,10	11,12,13,...,21	22,23,24,...,32	33,34,35,...,44

Quando a divisão do número de tarefas por *thread* não é exata, é preciso sempre fazer um ajuste nos intervalos do *for* (linha 08 do Quadro 11) da última *thread* para que ele não ultrapasse seu limite ou deixe de executar algum movimento.

Devido ao processamento estar ocorrendo paralelamente com 4 *threads,* faz-se necessário um vetor de 4 posições para indicar os melhores índices de troca *a, b e* o melhor custo, pois agora são 4 possíveis candidatos a serem efetuados.

Após o término da execução das linhas 08 a 14 os vetores são copiados para a CPU, que se encarregará de encontrar a melhor troca a ser realizada, fazendo com que o algoritmo volte a ser executado sequencialmente. Esta função é chamada continuamente até que não haja mais melhorias a serem realizadas.

No quadro 12 é possível ter uma visão parcial do *kernel*, na linha 03 é identificado o id da *thread* que está sendo executada em paralelo, e os vetores *flipPointAD* e *flipPointBD* nas linhas 5 e 6 armazenam os possíveis pontos onde a CPU deve efetuar o movimento. Já nas linhas 8 e 9 são determinados os limites de trabalho de cada *thread*. A alocação de variáveis e outras linhas não importantes para as explicações foram removidas para facilitar a compreensão e o código na integra pode ser obtido em [20].

Através das linhas 16 e 17 do quadro 12, é possível efetuar todas as trocas 2 a 2 necessárias para a heurística *swap*, entre as linhas 19 a 41 é testado se o movimento vai de fato reduzir o custo da solução, perceba que os valores são acessados por uma textura bidimensional e existem três casos que precisam ser testados.

Ao fim da execução os vetores são copiados para a CPU e o melhor movimento é executado com o intuito de melhorar a qualidade da solução.

A fórmula relativa às linhas 16 e 17 é essencial para a paralelização e foi deduzida matematicamente após algum esforço, o objetivo é encontrar todas as permutações 2 a 2 sem repetições, e a sequência a ser visitada é análoga a da tabela 02, só que em uma ordem diferente de visitação. Esse esforço matemático evita repetições desnecessárias.

Quadro 12: Visão parcial do kernel *swap* (GPU).

```
01 __global__ void localSearchSwapKernel(...){
02
03    int idx = blockIdx.x * blockDim.x + threadIdx.x;
04    estimativeD[idx] = 0.0;
05    flipPointAD[idx] = -1;
06    flipPointBD[idx] = -1;
07
08    int threadStart = idx * threadWorkLoad;
09
10    if (threadStart>=maxIteractions){return;}
11    int threadEnd   = threadStart + threadWorkLoad -1;
12    if(threadEnd>maxIteractions) threadEnd = maxIteractions-1;
13
14    for( int i = threadStart ; i <= threadEnd ; i++){
15
16          b = int(sqrtf(2 * (i+1)) + 0.5);
17          a = b - 1 - (b * (b+1) / 2 - (i+1) );
18
19          pa = previousD(a,dim);
20          na = nextD(a,dim);
21          pb = previousD(b,dim);
22          nb = nextD(b,dim);
23
24          if( b-a == 1 ){
25
26          sub = tex2D(t,[pa],sol[a])+tex2D(t,sol[b],sol[nb]);
27          add = tex2D(t,sol[pa],sol[b])+tex2D(t,sol[a],sol[nb]);
28
29          }else if(b-a == dim-1){
30
31          sub = tex2D(t,sol[pb],sol[b]) + tex2D(t,sol[b],sol[a])
32          +tex2D(t,sol[a ],sol[na])  ;
33
34          add = tex2D(t,sol[pb],sol[a]) + tex2D(t,sol[a],sol[b] )+
35          tex2D(t,sol[b ],sol[na]);
36          }else{
37          sub = tex2D(t,sol[pa],sol[a]) + ex2D(t,sol[a],sol[na])
38          +tex2D(t,sol[pb],sol[b]) + tex2D(t,sol[b],sol[nb]);
39
40          add = tex2D(t,sol[pa],sol[b]) +tex2D(t,sol[b],sol[na]) +
41          tex2D(t,sol[pb],sol[a]) + tex2D(t,sol[a],sol[nb]);
42
43}
44                result = sub - add;
45                if(result > estimativeD[idx]){
46                      estimativeD[idx] = result;
47                      flipPointAD[idx] = a;
48                      flipPointBD[idx] = b;
49                }
50          }
51}
```

3. Or-Opt-1

Este movimento de busca local foi proposto por Or em 1976 [31] o que explica o nome *Or-Opt*. O movimento consiste em remover *k* clientes consecutivos de um *tour,* e inseri-los em todas as posições possíveis deste mesmo *tour*, onde k ∈ {1,2,3}. Um exemplo deste movimento para *k=1* pode ser observado na Figura 18.

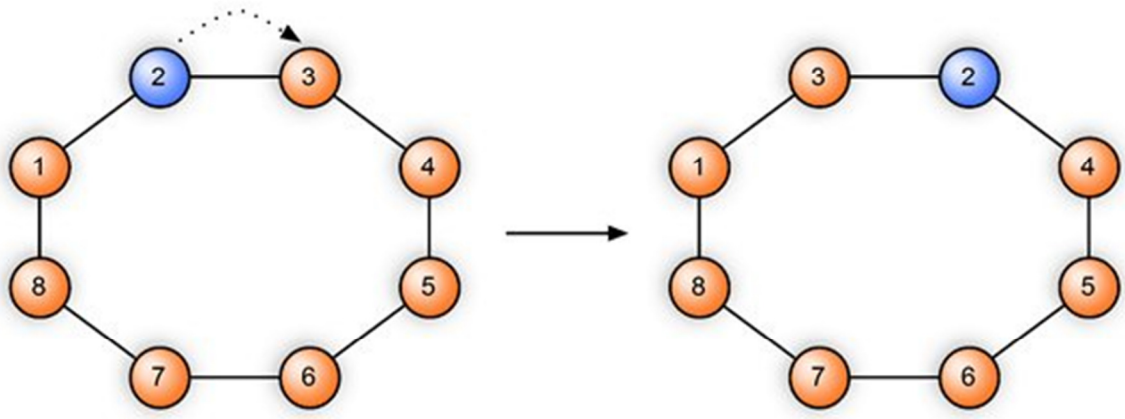

Figura 18: Exemplo de como um *tour* pode ser modificado pelo movimento *OrOpt-1*.

Como pode ser verificado em Babin *et al.* (2007) [32] a complexidade desta busca local é O(n²) [33], pois todos os conjuntos de *tours,* precisam ser deslocados em todas as posições possíveis dentro do mesmo *tour*.

O custo do movimento pode ser calculado em O(1), pois invariavelmente para um *tour* de uma ou *n* cidades, apenas duas arestas são removidas e 2 arestas são inseridas, resultando em um significativo ganho de performance na avaliação dos candidatos se comparado com o desempenho O(n) de avaliação da maneira tradicional. Apenas o movimento que tiver a maior redução de custo é escolhido, o que corresponde ao custo mais negativo dentre todos os movimentos.

Segundo Babin *et al.* (2007) [32], apesar da superioridade de algumas implementações *Lin-Kernighan*, heurísticas mais simples como o *2-opt* e *Or-Opt* ainda se mantém populares devido a sua facilidade de implementação e desempenho bastante razoável.

O pseudocódigo do algoritmo em CPU está apresentado no Quadro 11 e possui duas funções auxiliares fundamentais: a função que obtém a estimativa do movimento, cuja complexidade é de O(1) (linha 08), e a função que efetiva o movimento, movendo um elemento da posição A para a posição B, esta função possui custo O(n), custo equivalente de

uma inserção e remoção em um *array*. No caso da primeira função, não é necessário realizar o movimento para depois chamar a função que obtém o custo, o que seria muito dispendioso, pois além de ter que gerar uma cópia da solução, todo vetor teria que ser visitado e cada custo consultado na matriz que representa a distância entre cada par de cidades.

Quadro 13: Visão em alto nível sequencial do algoritmo *OrOpt-1 e*m CPU.

```
procedimento OrOpt1CPU(vetor oldSolution)
01. maximoIterações ← (dim * dim);
02. pontoTrocaA ← -1;
03. pontoTrocaB ← -1;
04. melhorTrocaCusto ← ∞;
05. para i←0 até maximoIterações faça
07.     a ← i/dim;
06.     b ← i – a * dim;
08.     estimativa ← obtemEstimativaDeMovimentoOrOpt1(oldSolution,a,b)
09.     se (estimativa < melhorCusto) então
10.             pontoTrocaA ← a;
11.             pontoTrocaB ← b;
12.             melhorTrocaCusto ← estimativa
13.     fim se
14. fim para
15. newSolution ← Move1elemAtoB(oldSolution,pontoTrocaA,pontoTrocaB);
16. retorne vetor newSolution;
fim OrOpt1CPU;
```

A versão paralelizada para GPU é totalmente análoga às versões criadas para o *2-opt* e *swap* o que dispensa exemplos. O algoritmo pode ser visualizado no Quadro 13 e o *kernel* do algoritmo pode ser obtido em [20].

Quadro 14: Visão em alto nível paralela do algoritmo *OrOpt-1 e*m GPU.

```
procedimento OrOpt1GPU(vetor oldSolution)
01. maximoIterações ← (dim * dim);
02. pontoTrocaA ← -1;
03. pontoTrocaB ← -1;
04. melhorTrocaCusto[ThreadId] ← ∞;
05. numTarefasPorThread ← máximoIterações / totalThreads;
06. threadStart ← ThreadId * numTarefasPorThread;
07. threadEnd  ← threadStart + numTarefasPorThread-1;
08. para i=threadStart até ThreadEnd faça
07.     a ← i/dim;
06.     b ← i  - a * dim;
08.     estimativa ← obtemEstimativaDeMovimentoOrOpt1(oldSolution,a,b)
09.     se (estimativa < melhorTrocaCustoGPU[threadId]) então
10.             pontoTrocaAGPU[ThreadId] ← a;
11.             pontoTrocaBGPU[ThreadId] ← b;
12.             melhorTrocaCustoGPU[ThreadId] ← estimativa
13.     fim se
14. fim para
20. Copiar pontoTrocaAGPU,pontoTrocaBGPU,melhorTrocaCustoGPU para CPU
21. Encontrar melhorTrocaCusto e atribuir a pontoTrocaA e pontoTrocaB
22. newSolution ← Move1elemAtoB(oldSolution,pontoTrocaA,pontoTrocaB);
16. retorne vetor newSolution;
fim OrOpt1GPU;
```

4. OrOpt-2

O movimento *OrOpt-2* pode ser entendido analogamente como o *OrOpt-1*, só que com *k*=2. Uma ilustração do movimento para *k*=2 pode ser visualizada na Figura 19:

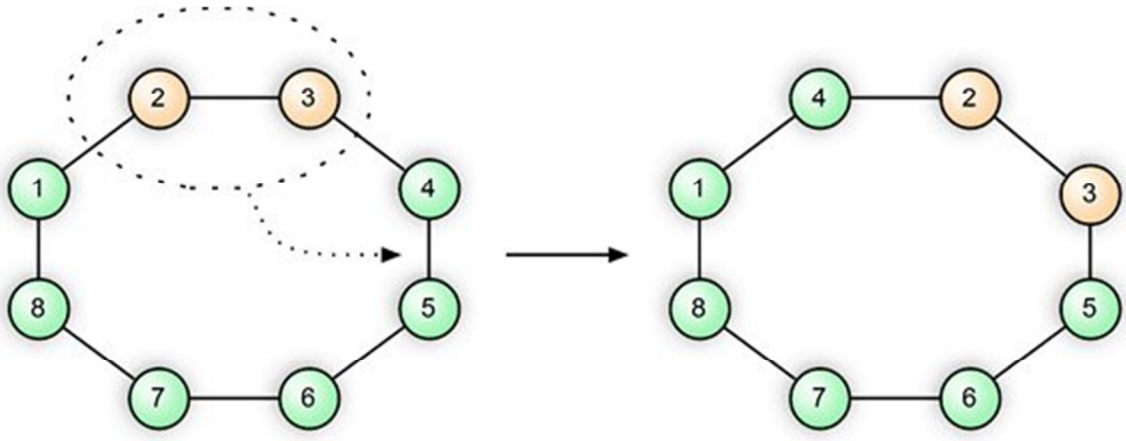

Figura 19: Exemplo de como um *tour* pode ser modificado pelo movimento *OrOpt-2*.

Os algoritmos para o *OrOpt-2* são semelhantes aos descritos nos Quadros 13 e 14, exceto pelas funções auxiliares, pois passa ser necessária uma função que obtém estimativas

para o deslocamento de duas cidades no *tour*. Esta função precisa ser projetada com bastante cautela pois há uma série de movimentos inválidos como por exemplo a troca de vizinhos.

Além desta função é necessário criar uma outra para realizar a seguinte tarefa: dada uma posição de origem, uma posição de destino e um vetor, executar este movimento. O restante é completamente análogo ao método anterior, o *kernel* do algoritmo pode ser obtido em [20].

5. OrOpt-3

O movimento *OrOpt-3* foi calculado analogamente como o *OrOpt-2*. Uma ilustração do movimento para *k=3* pode ser visualizada na Figura 20:

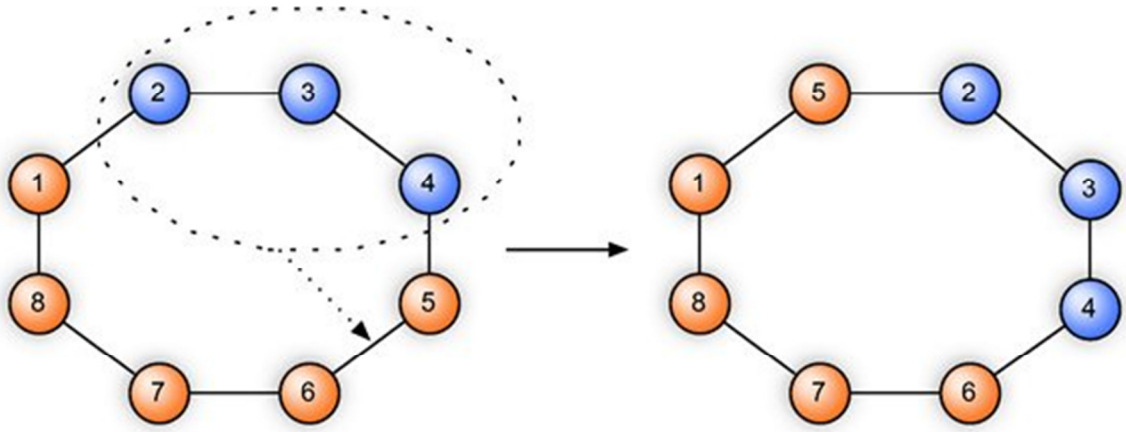

Figura 20: Exemplo de como um *tour* pode ser modificado pelo movimento *OrOpt-3*.

Os algoritmos para o movimento *OrOpt-3* são semelhantes aos descritos na seção sobre o *OrOpt-2,* exceto pelos métodos que realizam a estimativa de custo e o método que executa o movimento. Uma explicação detalhada não é realizada nesta seção para evitar repetições desnecessárias, o *kernel* do algoritmo pode ser obtido em [20].

3.5 Desafios Encontrados

Alguns desafios relevantes foram enfrentados durante a realização deste estudo:

Precisão Dupla em GPU – Ao contrário dos novos modelos de GPU [34], a GPU utilizada para o desenvolvimento da solução não possui implementação em *hardware* para a precisão dupla (*double*), o que gerou uma grande dificuldade, pois erros se acumulavam ao serem feitas estimativas para o custo da nova solução após as alterações em cada iteração do

programa. O problema foi superado utilizando a GPU apenas para estimativas através de *floats* e a CPU para efetivar as alterações utilizando *doubles.* Desta forma a CPU proveria uma nova estimativa correta dos custos a cada iteração ao se comunicar novamente com a GPU, eliminando os erros.

Suporte a Ponteiros – Todos os dados inseridos e tratados na GPU devem estar na forma de vetores e matrizes, não sendo possível alocar ponteiros para a GPU, o que dificulta o trabalho, pois todo o espaço de memória necessário deve ser conhecido *a priori*, já que não existe alocação dinâmica.

Ausência de Estrutura de Dados e Bibliotecas Nativas – 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, sem que o usuário possa abstrair dessas preocupações para se concentrar no problema em si. Isto torna o trabalho de programar qualquer algoritmo mais complexo e passível de erros.

Poucos Trabalhos Relacionados – A arquitetura CUDA é recente, tornando quaisquer trabalhos de otimização combinatória em GPUs trabalhos pioneiros. Não existem muitos trabalhos a que se possa recorrer para comparações e até mesmo para melhorias de algoritmos, o que cria um obstáculo a ser enfrentado.

Comunicação entre CPU e GPU – 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.

3.6 Otimizações

Recálculo e Estimativas de Custo em O(1) - Em todos os movimentos realizados são necessárias funções que estimam os custos das alterações que se deseja realizar. Caso fossemos analisar o custo da maneira tradicional, seria necessário percorrer todo o *tour,* consultar cada par de cidades e totalizar seus custos. Esta tarefa possui custo O(N).

Uma estratégia vantajosa pode ser utilizada para reduzir os custos do algoritmo para O(1), simplesmente aproveitando-se de informações do custo da solução original já previamente calculada.

Na Figura 21 é possível visualizar o *tour* original que possui custo (A+B+C+D+E+F).

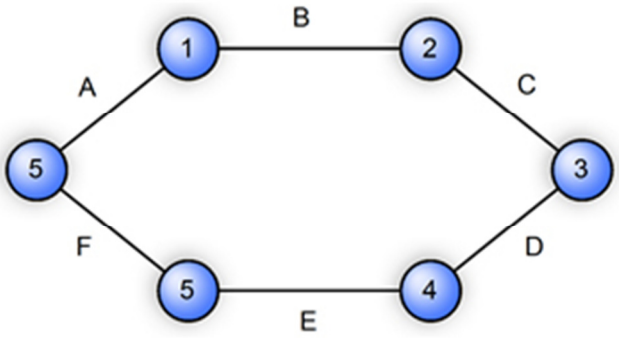

Figura 21: *Tour* original.

Para obter o custo do *tour* da Figura 22 basta remover os custos das arestas B e E, e adicionar os custos das arestas G e H, o que implica em um novo custo (A+C+D+F+G+H). Desta forma, o custo para estimar um movimento será constante, independente do tamanho de *n*.

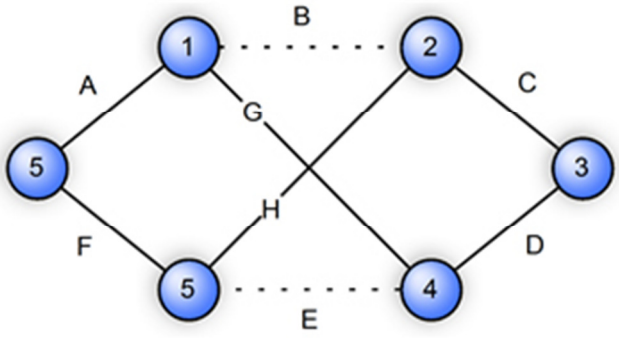

Figura 22: Novo *tour* obtido a partir do original com modificações.

Utilização de Memória de Textura - Na implementação do algoritmo ILS-RVND, foi criada uma textura que é enviada para a GPU apenas uma vez antes de iniciar o algoritmo. Esta textura quadrangular de tamanho n^2 utiliza *floats* e representa a matriz de distâncias entre as cidades, sendo cada *pixel,* na posição (i,j) a distância entre as cidades (i,j). Este tipo de memória é mais rápida que a memória global, pois é otimizada apenas para leitura e possui localidade espacial.

Criação de Variáveis Globais – Após o término da implementação, diversas variáveis e vetores que eram frequentemente utilizados foram declarados como globais para ganho de desempenho em alocação de memória, evitando que cada um tivesse que ser alocado a cada nova iteração de cada algoritmo.

4 TESTES E RESULTADOS

Nesta seção são expostos os ambientes em que os testes foram realizados, é testada e discutida a análise individual de cada algoritmo paralelizado, os testes do modelo proposto e seus resultados.

4.1 Ambiente de testes

A aplicação foi desenvolvida em linguagem C++ por exigências de compatibilidade com a plataforma CUDA e os testes foram realizados em dois ambientes distintos:

Ambiente de testes 01: *Linux Ubuntu* 9.04, processador *Core 2 Quad* Q6600 – 2.4 Ghz com 4 GB de memoria RAM. A GPU utilizada para os testes foi o TESLA C1060 (Figura 23) da NVidia [35].

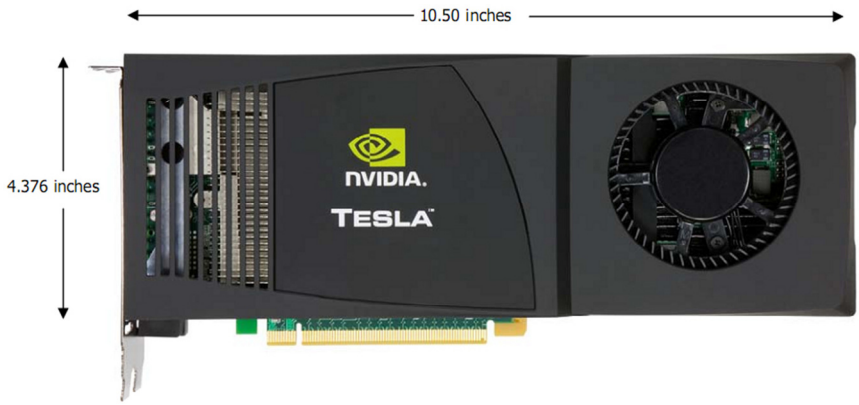

Figura 23: Placa gráfica NVidia TESLA C1060.
Fonte: NVidia.

A placa de vídeo Tesla C1060 possui 240 núcleos, sendo que cada processador possui *clock* de 1.296 Ghz, utiliza barramento PCI *Express* 2.0 e possui um consumo menor que 200W. Consegue atingir até 933 *GFLOPs/s* e possui 4 GB de memória GDDR3 com largura de banda de até 102 *GB/s*.

Ambiente de testes 02: *Linux Ubuntu* 9.04, processador *Core 2 Duo E7200* – 2.53 Ghz, com 4 GB de memoria RAM. A GPU utilizada para os testes foi o a GTS250 (Figura 24) da NVidia [35].

Figura 24: Placa gráfica NVidia GTS250. Fonte: **NVidia.**

A placa de vídeo GTS250 possui 128 núcleos, sendo que cada processador possui *clock* de 1.836 Ghz, utiliza barramento PCI *Express* 2.0 e possui um consumo menor que 150W, consegue atingir até 470 *GFLOPs/s* e possui 512 MB de memória GDDR3 com uma largura de banda de até 70.4 *GB/s*.

4.2 Calibração de parâmetros

Muitos parâmetros neste trabalho são variáveis como: a força da perturbação, o valor de *alfa* para a heurística de construção das soluções GRASP, bem como o número de iterações e reinicializações.

A calibração de parâmetros para meta-heurísticas pode ser uma tarefa dispendiosa, por consumir bastante tempo devido ao infinito número de possibilidades de combinações, chegando até mesmo a ser um assunto de pesquisa dentro da área de meta-heurísticas.

Este trabalho apesar de apresentar bons resultados em termos de qualidade, tem um foco principal em um novo paradigma de modelagem do problema, em ambiente altamente paralelo das GPUs, não focando na calibração propriamente dita.

4.3 Testes

Nesta seção são apresentados os resultados computacionais obtidos pelos algoritmos e pelo modelo proposto. Para validar as implementações, os testes foram divididos em dois conjuntos: testes individuais da paralelização de cada algoritmo e testes do modelo proposto. A medida GAP foi utilizada para definir o quão próxima estão às soluções obtidas das soluções ótimas, a melhor solução conhecida para cada instância:

$$\mathrm{GAP} = \ 100\ \mathrm{X}\ \frac{\mathrm{Valor} - \mathrm{MelhorValor}}{\mathrm{Melhorvalor}}$$

4.3.1 Teste Individual de Velocidade para o Algoritmo *2-opt*

Cenário: O teste foi executado no ambiente de testes 01, consistindo em 10 execuções com geração da solução inicial GRASP com α = 0.5%. Em cada execução, foram realizadas 100 iterações do algoritmo *2-opt* seguido de uma perturbação *double-bridge* de tamanho variável entre 1 e 15, em caso de não melhoria.

Tabela 6: Resultados obtidos pela execução do algoritmo *2-Opt* em CPU.

Instancia	N	Menor T. (s)	Média. T. (s)	Maior T. (s)	Melhor	Literatura	GAP
kroB100	100	0.420	0.452	0.489	22178,6	22141	0,17%
ch150	150	0.788	0.884	0.973	6630	6528	1,56%
kroB200	200	1.597	1.970	2.235	29582	29437	0,49%
a280	280	2.516	3.262	3.773	2624	2579	1,74%
pr299	299	3.630	4.331	5.440	49527	48191	2,77%
fl417	417	14.284	17.373	20.259	11964	11861	0,87%
d493	493	15.099	17.465	19.970	35909	35002	2,59%
u574	574	22.034	25.550	29.996	38401	36905	4,05%
d657	657	31.329	35.678	41.285	51940	48912	6,19%
u724	724	38.423	40.344	43.393	44646	41910	6,53%
pr1002	1002	96.731	114.497	129.654	270914	259045	4,58%

Tabela 7: Resultados obtidos pela execução do algoritmo *2-Opt* em GPU.

Instancia	N	Menor T. (s)	Média T. (s)	Maior T. (s)	Melhor	Literatura	GAP
kroB100	100	0.081	0.084	0.088	22261,4	22141	0,54%
ch150	150	0.106	0.110	0.113	6558	6528	0,46%
kroB200	200	0.164	0.170	0.175	29595,1	29437	0,54%
a280	280	0.271	0.276	0.281	2629,4	2579	1,95%
pr299	299	0.311	0.330	0.356	49541,2	48191	2,80%
fl417	417	0.961	0.988	1.018	12005,8	11861	1,22%
d493	493	1.053	1.077	1.109	35732,5	35002	2,09%
u574	574	1.284	1.321	1.360	38387,8	36905	4,02%
d657	657	1.742	1.801	1.855	52079,3	48912	6,48%
u724	724	1.967	2.023	2.084	44476,9	41910	6,12%
pr1002	1002	4.515	4.658	4.836	271599	259045	4,85%

Análise: O *speedup* do algoritmo em relação a CPU chega a ser em média de 5,38 vezes para a menor instância (*n=100*) e de 24,28 vezes em média para a maior instância (*n=1002*). Apesar de ser um teste de velocidade, é importante ressaltar que os resultados variam, pois foram utilizadas entradas diferentes. Além disso, há o componente aleatório da perturbação. Ainda assim verificam-se tempos aproximados. O algoritmo individualmente não produz soluções com um GAP razoável.

4.3.2 Teste Individual de Velocidade para o Algoritmo *swap*

Cenário: O teste foi executado no ambiente de testes 01, consistindo em 10 execuções com geração da solução inicial GRASP com α = 0.5%. Em cada execução, foram realizadas 100 iterações do algoritmo *swap* seguido de uma perturbação *double-bridge* de tamanho variável entre 1 e 15, em caso de não melhoria.

Tabela 8: Resultados obtidos pela execução do algoritmo *swap* em CPU.

Instancia	N	Menor T. (s)	Média T. (s)	Maior T. (s)	Melhor	Literatura	GAP
kroB100	100	0.434	0.446	0.469	24337,3	22141	9,92%
ch150	150	0.940	1.037	1.184	7052,7	6528	8,04%
kroB200	200	2.009	2.205	2.468	33341,3	29437	13,26%
a280	280	3.458	3.760	4.029	2886,5	2579	11,92%
pr299	299	4.707	4.981	5.295	55317,5	48191	14,79%
fl417	417	12.258	13.440	13.947	12398,3	11861	4,53%
d493	493	14.336	15.171	16.336	38968,6	35002	11,33%
u574	574	19.823	20.877	22.690	41786,6	36905	13,23%
d657	657	25.200	27.532	28.930	55351,3	48912	13,17%
u724	724	33.430	35.518	37.474	47870,7	41910	14,22%
pr1002	1002	67.768	73.260	77.573	291218	259045	12,42%

Tabela 9: Resultados obtidos pela execução do algoritmo *swap* em GPU.

Instancia	N	Menor T. (s)	Média T. (s)	Maior T. (s)	Melhor	Literatura	GAP
kroB100	100	0.081	0.082	0.083	23975,9	22141	8,29%
ch150	150	0.124	0.125	0.129	7084,2	6528	8,52%
kroB200	200	0.190	0.195	0.204	32863,9	29437	11,64%
a280	280	0.267	0.290	0.319	2946,5	2579	14,25%
pr299	299	0.360	0.387	0.426	54695,9	48191	13,50%
fl417	417	0.673	0.788	0.969	12432,1	11861	4,81%
d493	493	0.729	0.811	0.928	38739,8	35002	10,68%
u574	574	1.016	1.086	1.109	42019,9	36905	13,86%
d657	657	1.387	1.402	1.584	55040,4	48912	12,53%
u724	724	1.682	1.816	1.933	47731,3	41910	13,89%
pr1002	1002	3.247	3.507	3.813	286957	259045	10,77%

Análise: O *speedup* do algoritmo em relação a CPU chega a ser em média 5,43 vezes para a menor instância (*n=100*) e 20,88 vezes em média para a maior instância (*n=1002*). Apesar de ser um teste de velocidade, é importante ressaltar que os resultados variam, pois foram utilizadas entradas distintas e existem componentes aleatórios da perturbação. Ainda assim verificam-se GAPs médios aproximados de 11,53% para a CPU e 11,16% para a GPU. O algoritmo individualmente não produz soluções com um GAP razoável.

4.3.3 Teste Individual de Velocidade para o Algoritmo *OrOpt-1*

Cenário: O teste foi executado no ambiente de testes 01, consistindo em 10 execuções com geração da solução inicial GRASP com α = 0.5%. Em cada execução, foram realizadas 100 iterações do algoritmo *OrOpt-1,* seguido de uma perturbação *double-bridge* de tamanho variável de 1 a 15, em caso de não melhoria.

Tabela 10: Resultados obtidos pela execução do algoritmo *OrOpt-1* em CPU.

Instancia	N	Menor T. (s)	Média T. (s)	Maior T. (s)	Melhor	Literatura	GAP
kroB100	100	1.140	1.307	1.559	22618,2	22141	2,16%
ch150	150	3.013	3.189	3.435	6667,1	6528	2,13%
kroB200	200	6.426	7.194	7.858	31039,5	29437	5,44%
a280	280	12.588	15.329	19.094	2673,6	2579	3,67%
pr299	299	17.784	20.504	22.301	51268,5	48191	6,39%
fl417	417	41.410	43.073	48.252	12273,9	11861	3,48%
d493	493	68.672	73.750	77.188	37194,7	35002	6,26%
u574	574	96.436	112.720	126.561	39402,4	36905	6,77%
d657	657	135.740	153.322	178.803	53190,1	48912	8,75%
u724	724	211.756	231.711	256.142	45493,6	41910	8,55%
pr1002	1002	404.534	465.176	476.827	280812	259045	8,40%

Tabela 11: Resultados obtidos pela execução do algoritmo *OrOpt-1* em GPU.

Instancia	N	Menor T. (s)	Média T. (s)	Maior T. (s)	Melhor	Literatura	GAP
kroB100	100	0.176	0.182	0.187	22948,3	22141	3,65%
ch150	150	0.305	0.326	0.369	6719,3	6528	2,93%
kroB200	200	0.429	0.494	0.593	31430,2	29437	6,77%
a280	280	0.877	1.054	1.151	2730	2579	5,85%
pr299	299	1.203	1.327	1.439	51364,2	48191	6,58%
fl417	417	1.521	1.745	1.956	12121,4	11861	2,20%
d493	493	2.964	3.246	3.558	37063,1	35002	5,89%
u574	574	5.853	6.287	6.971	39265,1	36905	6,40%
d657	657	8.351	8.704	9.336	52914,7	48912	8,18%
u724	724	9.913	11.120	12.440	45454	41910	8,46%
pr1002	1002	20.324	22.103	24.184	279125	259045	7,75%

Análise: O *speedup* do algoritmo em relação a CPU chega a ser em média 7,18 vezes para a menor instância (*n=100*) e 21,04 vezes em média para a maior instância (*n=1002*). Apesar de ser um teste de velocidade, é importante ressaltar que os resultados variam, pois foram utilizadas entradas distintas e existem componentes aleatórios da perturbação. Ainda assim verificam-se GAPs médios aproximados de 5,64% para a CPU e 5,88% para a GPU. O algoritmo individualmente não trás soluções com um GAP razoável.

4.3.4 Teste Individual de Velocidade para o Algoritmo *OrOpt-2*

Cenário: O teste foi executado no ambiente de testes 01, consistindo em 10 execuções com geração da solução inicial GRASP com α = 0.5%. Em cada execução, foram realizadas 100 iterações do algoritmo *OrOpt-1*, seguido de uma perturbação *double-bridge* de tamanho variável de 1 a 15, em caso de não melhoria.

Tabela 12: Resultados obtidos pela execução do algoritmo *OrOpt-2* em CPU.

Instancia	N	Menor T. (s)	Média T. (s)	Maior T. (s)	Melhor	Literatura	GAP
kroB100	100	0.744	0.810	0.927	22928,9	22141	3,56%
ch150	150	1.926	2.034	2.215	7009,8	6528	7,38%
kroB200	200	3.956	4.392	5.228	32029	29437	8,81%
a280	280	8.173	9.477	10.695	2772,14	2579	7,49%
pr299	299	11.189	12.431	13.127	53661,5	48191	11,35%
fl417	417	34.756	38.424	41.100	12309,4	11861	3,78%
d493	493	36.491	42.834	46.590	38613,2	35002	10,32%
u574	574	63.454	67.239	71.919	41.793	36905	13,24%
d657	657	77.971	90.043	100.282	55629,4	48912	13,73%
u724	724	124.730	133.336	142.035	47640,2	41910	13,67%
pr1002	1002	253.779	286.623	317.567	291.569	259045	12,56%

Tabela 13: Resultados obtidos pela execução do algoritmo *OrOpt-2* em GPU.

Instancia	N	Menor T. (s)	Média T. (s)	Maior T. (s)	Melhor	Literatura	GAP
kroB100	100	0.109	0.115	0.120	23147	22141	4,54%
ch150	150	0.190	0.194	0.198	7037	6528	7,80%
kroB200	200	0.279	0.292	0.308	31470	29437	6,91%
a280	280	0.532	0.585	0.620	2855,1	2579	10,71%
pr299	299	0.645	0.730	0.775	53232	48191	10,46%
fl417	417	1.319	1.556	1.847	12664,6	11861	6,78%
d493	493	1.561	1.689	1.958	38386	35002	9,67%
u574	574	2.893	3.263	3.730	41516	36905	12,49%
d657	657	4.446	4.923	5.356	55292,7	48912	13,05%
u724	724	5.704	6.142	6.816	47695,3	41910	13,80%
pr1002	1002	11.262	12.608	14.373	289613	259045	11,80%

Análise: O *speedup* do algoritmo em relação a CPU chega a ser em média 7,04 vezes para a menor instância (*n=100*) e 22,73 vezes para a maior instância (*n=1002*). Apesar de ser um teste de velocidade, é importante ressaltar que os resultados variam, pois foram utilizadas entradas distintas e existem componentes aleatórios da perturbação. Ainda assim verificam-se GAPs médios aproximados de 9,63% para a CPU e 9,82% para a GPU. O algoritmo individualmente não trás soluções com um GAP razoável.

4.3.5 Teste Individual de Velocidade para o Algoritmo *OrOpt-3*

Cenário: O teste foi executado no ambiente de testes 01, consistindo em 10 execuções com geração da solução inicial GRASP com α = 0.5%. Em cada execução, 100 iterações do algoritmo *OrOpt-3* seguido de uma perturbação *double-bridge* de tamanho variável de 1 a 15.

Tabela 14: Resultados obtidos pela execução do algoritmo *OrOpt-3* em CPU.

Instancia	N	Menor T. (s)	Média T. (s)	Maior T. (s)	Melhor	Literatura	GAP
kroB100	100	0.651	0.727	0.774	23729,8	22141	7,18%
ch150	150	1.567	1.774	1.882	7096,6	6528	8,71%
kroB200	200	3.476	3.789	4.195	32213,3	29437	9,43%
a280	280	7.101	7.851	8.600	2850,9	2579	10,54%
pr299	299	9.569	10.609	12.421	54449,7	48191	12,99%
fl417	417	28.504	32.431	38.040	12664,6	11861	6,78%
d493	493	31.879	35.278	39.215	38791,9	35002	10,83%
u574	574	46.838	50.771	56.079	42322,2	36905	14,68%
d657	657	68.132	74.224	84.543	56070,9	48912	14,64%
u724	724	100.669	108.437	115.788	48093,4	41910	14,75%
pr1002	1002	207.053	217.571	229.115	293400	259045	13,26%

Tabela 15: Resultados obtidos pela execução do algoritmo *OrOpt-3* em GPU.

Instancia	N	Menor T. (s)	Média T. (s)	Maior T. (s)	Melhor	Literatura	GAP
kroB100	100	0.041	0.050	0.084	23605,4	22141	6,61%
ch150	150	0.143	0.146	0.154	7194,1	6528	10,20%
kroB200	200	0.234	0.243	0.265	32418,7	29437	10,13%
a280	280	0.397	0.443	0.484	2782,5	2579	7,89%
pr299	299	0.488	0.524	0.559	54648,9	48191	13,40%
fl417	417	1.087	1.282	1.385	12268,8	11861	3,44%
d493	493	1.101	1.220	1.329	38.723	35002	10,63%
u574	574	2.562	2.701	2.879	42357,2	36905	14,77%
d657	657	3.248	3.305	3.506	56303,4	48912	15,11%
u724	724	3.663	4.030	4.293	47780,4	41910	14,01%
pr1002	1002	8.002	8.623	9.000	293657	259045	13,36%

Análise: O *speedup* do algoritmo em relação a CPU chega a ser em média 4,97 vezes para a menor instância (*n=100*) e 25,22 vezes em média para a maior instância (*n=1002*). Apesar de ser um teste de velocidade, é importante ressaltar que os resultados variam, pois foram utilizadas entradas distintas e existem componentes aleatórios da perturbação. Ainda assim verificam-se GAPs médios aproximados de 11,25% para a CPU e 10,87% para a GPU. O algoritmo individualmente não trás soluções com um GAP razoável.

4.3.6 Teste de Robustez e Comparativo entre CPU e GPU

Cenário: O teste foi executado por uma execução de 1000 iterações para testar o algoritmo ao extremo, utilizando o ambiente de testes 02 por se tratar de uma GPU mais fraca. Além disso, buscou-se verificar se com apenas uma execução, utilizando uma solução inicial básica ([1,2,3,...,n]) o algoritmo traria bons resultados.

Tabela 16: Resultados obtidos pela execução do modelo em CPU e GPU.

Instancia	N	Custo CPU	Custo GPU	Gap CPU	Gap GPU	Tempo CPU (s)	Tempo GPU (s)	Literatura
kroB100	100	22232.80	22255.30	0.41%	0.52%	10.91	9.143	22141
ch150	150	6571.00	6530.90	0.66%	0.04%	33.934	16.411	6528
kroB200	200	29456.00	29569.00	0.06%	0.45%	161.8	17.57	29437
a280	280	2586.77	2599.91	0.30%	0.81%	192.441	40.22	2579
pr299	299	48286.00	48245.00	0.20%	0.11%	276.778	39.646	48191
fl417	417	11913.70	11922.20	0.44%	0.52%	832.4	119.331	11861
d493	493	35289.00	35323.00	0.82%	0.92%	1620	251	35002
u574	574	37240.30	37246.80	0.91%	0.93%	1901	288	36905
d657	657	49080.30	49212.00	0.34%	0.61%	4815	687	48912
u724	724	42051.00	42075.20	0.34%	0.39%	5700	757	41910
pr1002	1002	262048.00	259787.00	1.16%	0.29%	24676	4627	259045

Análise: O *speedup* do algoritmo em relação a CPU chega a ser em média até 5,83 vezes mais rápido, mesmo em uma GPU mais fraca, produzindo *speedups* de até 9,2 vezes. Demonstrando-se ainda sua robustez, ao conseguir resultados com GAP de 0,51% tanto para CPU quanto para GPU, a partir de uma solução trivial.

4.3.7 Teste de GPU com um baixo número de iterações e variando a construção da solução.

Cenário: Os testes foram executados no ambiente de testes 02, consistindo em 10 execuções, sendo o primeiro utilizando construção aleatória e o segundo com geração da solução inicial GRASP com $\alpha = 0.5\%$. Sendo que em cada execução, foram realizadas 50 iterações do modelo proposto, seguido de perturbação *double-bridge* de tamanho variável de 1 a 15.

Tabela 17: Resultados obtidos pela execução do modelo em GPU utilizando construção da solução aleatória.

Instancia	N	Melhor Custo	Média Custo	Pior Custo	Gap Melhor	Gap Med.	Gap Pior	T (s)	Literatura
kroB100	100	22654,7	23068,1	23179,1	2,32%	4,19%	4,69%	2,7	22141
ch150	150	6671,1	6722,5	6841,5	2,19%	2,98%	4,80%	3,2	6528
kroB200	200	29799,3	30590,6	31486,1	1,23%	3,92%	6,96%	7,7	29437
a280	280	2641,3	2731,7	2799,9	2,41%	5,92%	8,57%	13,0	2579
pr299	299	49092,8	49994,3	50690,3	1,87%	3,74%	5,19%	24,9	48191
fl417	417	11933,8	12008,6	12369,7	0,61%	1,24%	4,29%	98,5	11861
d493	493	35620,5	36366,2	37201,5	1,77%	3,90%	6,28%	110,7	35002
u574	574	37623,4	37976,5	38390,5	1,95%	2,90%	4,03%	321,6	36905
d657	657	49731,2	50092,9	50738,3	1,67%	2,41%	3,73%	482,6	48912
u724	724	42695,9	42935,7	43292,7	1,88%	2,45%	3,30%	637,5	41910
pr1002	1002	264786,0	267202,0	270561,0	2,22%	3,15%	4,45%	1575,0	259045

Tabela 18: Resultados obtidos pela execução do modelo em GPU utilizando construção GRASP.

Instancia	N	Melhor Custo	Média Custo	Pior Custo	Gap Melhor	Gap Med.	Gap Pior	T (s)	Literatura
kroB100	100	22492,2	22935,1	23223,3	1,59%	3,59%	4,89%	3,2	22141
ch150	150	6650,8	6743,0	6815,0	1,88%	3,29%	4,40%	4,7	6528
kroB200	200	29982,0	30221,4	30473,2	1,85%	2,66%	3,52%	6,0	29437
a280	280	2621,9	2705,1	2725,8	1,66%	4,89%	5,69%	19,8	2579
pr299	299	48806,4	50215,1	50437,8	1,28%	4,20%	4,66%	30,5	48191
fl417	417	11944,2	12062,9	12254,8	0,70%	1,70%	3,32%	88,0	11861
d493	493	35632,0	36349,5	36943,7	1,80%	3,85%	5,55%	112,7	35002
u574	574	37543,3	37999,0	38506,3	1,73%	2,96%	4,34%	293,0	36905
d657	657	49742,6	50432,5	51031,9	1,70%	3,11%	4,33%	355,7	48912
u724	724	42475,9	42694,5	43117,3	1,35%	1,87%	2,88%	716,9	41910
pr1002	1002	264223,0	265104,0	266258,0	2,00%	2,34%	2,78%	2027,2	259045

Análise: Os testes demonstram médias para o melhor GAP de 1.83% para o teste com construção da solução aleatória e de 1,59% para a construção do GRASP, o que representa uma diferença pouco significativa. Este resultado aponta que a solução inicial possui pouca influência sobre o resultado final ao longo de várias iterações. Pode se perceber também que para a melhoria dos resultados, é importante investir em um maior número de iterações. O modelo executado com poucas iterações não trás soluções com um GAP razoável.

4.3.8 Teste de GPU com um alto número de iterações e variando a construção da solução.

Cenário: Os testes foram executados no ambiente de testes 02, consistindo em 10 execuções sendo o primeiro utilizando construção aleatória e o segundo com geração da solução inicial GRASP com α = 0.5%. Em cada execução foram realizadas 500 iterações do modelo proposto, seguido de perturbação *double-bridge* de tamanho variável de 1 a 15.

Tabela 19: Resultados obtidos pela execução do modelo em GPU utilizando construção da solução aleatória.

Instancia	N	Melhor Custo	Média Custo	Pior Custo	Gap Melhor	Gap Med.	Gap Pior	T (s)	Literatura
kroB100	100	22219,5	22462,3	22924,1	0,35%	1,45%	3,54%	33,2	22141
ch150	150	6530,9	6622,0	6705,4	0,04%	1,44%	2,72%	56,6	6528
kroB200	200	29527,2	30044,3	31386,0	0,31%	2,06%	6,62%	112,2	29437
a280	280	2592,0	2618,1	2652,8	0,50%	1,52%	2,86%	245,2	2579
pr299	299	48211,0	48870,9	49956,1	0,04%	1,41%	3,66%	386,6	48191
fl417	417	11918,6	11927,6	11933,2	0,49%	0,56%	0,61%	709,2	11861
d493	493	35143,4	35359,1	35536,3	0,40%	1,02%	1,53%	1656,5	35002
u574	574	36985,0	37204,4	37488,9	0,22%	0,81%	1,58%	2830,0	36905
d657	657	49154,1	49316,9	49560,4	0,49%	0,83%	1,33%	3552,0	48912
u724	724	42041,3	42196,4	42360,8	0,31%	0,68%	1,08%	5596,0	41910
pr1002	1002	260391,0	261625,0	262647,0	0,52%	1,00%	1,39%	15355,0	259045

Tabela 20: Resultados obtidos pela execução do modelo em GPU utilizando construção GRASP.

Instancia	N	Melhor Custo	Média Custo	Pior Custo	Gap Melhor	Gap Med.	Gap Pior	T (s)	Literatura
kroB100	100	22143,2	22626,9	23890,5	0,01%	2,19%	7,90%	37,2	22141
ch150	150	6530,9	6653,1	6790,1	0,04%	1,92%	4,02%	61,6	6528
kroB200	200	29503,0	30028,1	30483,0	0,22%	2,01%	3,55%	98,7	29437
a280	280	2594,8	2639,7	2712,5	0,61%	2,35%	5,17%	198,0	2579
pr299	299	48297,7	48609,0	49092,8	0,22%	0,87%	1,87%	490,8	48191
fl417	417	11918,6	11924,0	11928,2	0,49%	0,53%	0,57%	840,1	11861
d493	493	35196,4	35406,1	35636,0	0,56%	1,15%	1,81%	1790,3	35002
u574	574	37050,1	37303,1	37528,2	0,39%	1,08%	1,69%	3370,0	36905
d657	657	49122,0	49300,0	49345,0	0,43%	0,79%	0,89%	4915,0	48912
u724	724	42108,0	42238,2	42404,0	0,47%	0,78%	1,18%	5276,0	41910
pr1002	1002	260830,0	261616,0	262256,0	0,69%	0,99%	1,24%	17003,0	259045

Análise: Os testes demonstram médias para o melhor GAP de 0.33% para o teste com construção da solução aleatória e 0.38% para a construção do GRASP. Este resultado também aponta que a solução inicial possui pouca influência sobre o resultado final ao longo de várias iterações. Pode se perceber também que o aumento no número de iterações trouxeram boas melhorias nos resultados. O modelo executado trouxe resultados com GAPs baixos e foi testado no ambiente de testes 02, mais fraco.

4.4 Limitações

Apesar do excelente desempenho computacional em termos de *speedup* este trabalho visa apenas validar a utilização de GPUs para meta-heurísticas. Não foi seu objetivo superar os algoritmos considerados como estado da arte, mas sim remodelá-los para a arquitetura CUDA.

Apesar de este trabalho provar que a utilização de GPUs em meta-heurísticas pode ser eficiente e trazer ótimos resultados, muito ainda deve ser feito para melhoria da qualidade das respostas, bem como para o uso de múltiplas GPUs. Ainda há de se lembrar que não existem muitas publicações sobre o assunto.

5 CONCLUSÃO E TRABALHOS FUTUROS

Nesta seção, é realizada uma avaliação geral de todo o trabalho desenvolvido. Para isto, os objetivos propostos inicialmente foram considerados, bem como os meios utilizados para alcançá-los. Desta forma, torna-se possível apresentar as principais conclusões e contribuições. Em seguida, são enumeradas algumas sugestões para trabalhos futuros com a intenção de contribuir para esta linha de pesquisa.

5.1 Conclusões

Neste trabalho, foi proposta uma abordagem heurística e paralela implementada em GPUs para o PCV. Cinco algoritmos foram paralelizados seguindo o paradigma de arquiteturas de GPUs e todos apresentaram ganhos significativos, variando de 5 até 20 vezes para instâncias grandes, mesmo para apenas uma GPU.

O modelo como um todo também foi testado e se mostrou eficiente em trazer as melhores soluções, com GAP médio de no máximo 0.33% de acordo com o teste 4.3.8 em todas as instâncias testadas. O tempo também se mostrou várias vezes mais rápido que a implementação em CPU mesmo para instâncias pequenas.

Em termos de qualidade de solução, que não era foco deste trabalho, apesar de não ter encontrado resultados ótimos para o problema, o modelo trouxe soluções razoáveis. O modelo

pode ser implementado para heurísticas mais competitivas como *Link-Kernighan,* sendo que os parâmetros devem ser ajustados cautelosamente.

Em termos de escalabilidade e energia, o modelo proposto pode ser expandido para uso em múltiplas GPUs, visto que o *overhead* de comunicação é muito pequeno, tornando o modelo passível de trabalhar com instâncias realmente grandes com algumas dezenas de GPUs e gastando uma fração da energia dos grandes *clusters*.

5.2 Trabalhos Futuros

São apresentadas a seguir algumas recomendações e sugestões para trabalhos futuros:

1. **Executar movimentos simultâneos de troca** – Em cada heurística, todas as possibilidades de movimentos são listadas, e apenas o melhor movimento é utilizado como tentativa para melhorar a solução, o que representa um custo computacional muito alto para tirar proveito apenas do melhor movimento. Nem sempre é viável aplicar todos os movimentos que acarretariam em melhorias, pois muitas vezes eles são dependentes. Tal funcionalidade potencializaria o poder do modelo proposto.

2. **Utilização da mesma abordagem para executar em um *cluster* de GPUs** – Apesar das GPUs colaborarem na melhoria da qualidade da solução e no aumento da velocidade para instâncias de tamanho médio, para valores grandes de *n* apenas uma GPU pode não ser suficiente. A abordagem proposta neste trabalho permite uma extensão do modelo para instâncias maiores.

3. **Paralelização de outras meta-heurísticas** – Paralelização de outras meta-heurísticas para serem adicionadas ao modelo proposto bem como a análise de desempenho.

4. **Otimização do código** – O modelo proposto foi implementado de uma maneira não linear, portanto o código deve ser refatorado e otimizado para um maior ganho de desempenho.

5. **Adicionar probabilidades a escolha das buscas locais –** Ao invés de simplesmente escolher as buscas locais ao acaso, uma boa estratégia poderia ser dar probabilidades as buscas locais que mais conseguem melhorar a solução, reduzindo o tempo de execução em buscas locais que não estão melhorando a solução.

6. **Representação da matriz de custos na *Shared Memory* –** A velocidade de acesso à memória *shared* é duas ordens de grandeza maior que o acesso à memória global,

portanto, se alguma técnica permitir a representação desta matriz na memória compartilhada implicará em um maior *speedup* do modelo.

7. **Adaptação do código para o problema de coleta e entrega mista** – Subramanian *et al.* (2010) [18] apresenta resultados de excelente qualidade sobre o problema de coleta e entrega mista. Porém, os tempos ainda podem ser melhorados, para associar qualidade à velocidade. Boa parte da modelagem para este problema já pode ser encontrada neste trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] OSKIN, M. The revolution inside the box. *Communications of the ACM*, v.51, n.7, p.70-78, 2008.

[2] INTEL. Processors. Disponível em: <http://www.intel.com/support/processors/sb/cs-023143.htm>. Acesso em: 05 abr 2010.

[3] THE TECH REPORT. PC Hardware Explored. Disponível em: <http://techreport.com/articles.x/18682>. Acesso em: 03 abr 2010.

[4] LAWLER, E.L.; LENSTRA, J.K.; RINNOOY KAN, A.H.G.; SHMOYS, B.D. *The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley-Interscience Publication, Chichester, 1985. 476p.

[5] APPLEGATE, D.L.; BIXBY, R.E.; CHVÁTAL, V.; COOK, W.J. *The Traveling Salesman Problem. A Computational Study.* Princeton University Press, Princeton, New Jersey, 2006. 606p.

[6] LAPORTE, G. What you should know about the vehicle routing problem. *Naval Research Logistics,* v.54, n.9, p. 811–819, 2007.

[7] HELSGAUN, K. *An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic*. Manuscript, Roskilde University, Denmark, 1999. 77p.

[8] ALL EXPERTS SEARCH. Disponível em: <http://en.allexperts.com/q/Astronomy-1360/2008/2/Amount-particles-universe.htm> Acesso em: 10 abr 2010.

[9] REINELT, G. TSPLIB - A Traveling Salesman Problem Library. ORSA Journal on Computing, v.3, n.4, p.376-384, 1991. Disponível em:

<http://joc.journal.informs.org/cgi/content/abstract/3/4/376>. Acesso em: 15 out. 2009.

[10] TSPLIB. Disponível em: <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>. Acesso em: 15 jun. 2010.

[11] LOURENÇO, H.R.; MARTIN, O.; STUETZLE, T. Iterated Local Search. In: GLOVER, F.; KOCHENBERGER, G. *Handbook of Metaheuristics*. Kluwer Academic Publishers, Norwell, MA, 2002, p.321-353.

[12] NVIDIA. NVIDIA CUDA™. NVIDIA CUDA. Best Practices Guide. Version 3.0. 2010. Disponível em: <http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf>. Acesso em: 15 mar 2010.

[13] NVIDIA. Getting Started. NVIDIA CUDA C. Installation and Verification on Microssoft Windows XP, Windows Vista and Windows 7. 2009. Disponível em: <http://developer.download.nvidia.com/compute/cuda/3_0/docs/GettingStartedWindows.pdf>. Acesso em: 16 mar 2010.

[14] NVIDIA. CUDA ZONE. Disponível em: <http://www.nvidia.com/object/cuda_home_new.html>. Acesso em: 09 mar 2010.

[15] NVIDIA. Developer Zone. CUDA 3.0 Downloads. Disponível em: <http://developer.nvidia.com/object/cuda_3_0_downloads.html>. Acesso em: 15 out 2009.

[16] GPGPU. General-Purpose Computation on Graphics Hardware. Disponível em: <http://gpgpu.org/>. Acesso em: 15 set 2009.

[17] TALBI, E.G. Parallel Local Search on GPU, 2009. Institut National de Recherche en Informatique et en automatique.

[18] 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.

[19] LÜER, A.; BENAVENTE, M.; BUSTOS, J.; VENEGAS, B. *El problema de rutas de vehículos: Extensiones y métodos de resolución, estado del arte.* In: Workshop Internacional EIG. Departamento de Ingeniería de Sistemas. Universidad de La Frontera, Chile, 2009.

[20] CÓDIGO fonte da dissertação. Disponível em: <http://tinyurl.com/2g4gmnf>. Acesso em: 09 jul 2010.

[21] CREATIVE COMMONS. Disponível em: <http://creativecommons.org/licenses/by-sa/3.0/>. Acesso em: 09 jun 2010.

[22] GAZOLLA, J.G.F.M.; DELGADO, J.; CLUA, E.; SADJADI, S.M. *An incremental approach to porting complex scientific applications to GPU/CUDA*. In Proceedings of the IV Brazilian e-Science Workshop, Minas Gerais, Brazil, July 2010.

[23] SOUZA, M.J.F. *Inteligência Computacional para Otimização*. Notas de aula da disciplina Inteligência Computacional para Otimização 2008/1. DECOM/ICEB/UFOP. 28p.

[24] FEO, T.A.; RESENDE, M.G.C. Greedy randomized adaptive search procedures, *Journal of Global Optimization,* v.6, p.109-134, 1995.

[25] LIN, S.; KERNIGHAN, B.W. An effective heuristic algorithm for the travelling salesman problem. *Operations Research*, v.21, p.498–516, 1973.

[26] HASEGAWA, M.; IKEGUCHI, T.; AIHARA, K. Combination of chaotic neurodynamics with the 2-opt algorithm to solve traveling salesman problems. *Phys. Rev. Lett*. v.79, n.12, p.2344–2347, 1997.

[27] CROES, G.A. A method for solving traveling salesman problems. *Operations. Research,* v.6, n.6, p.791–812, 1958.

[28] ENGLERT, M;. RÖGLIN, H.; VÖCKING, B. *Worst case and probabilistic analysis of the 2-Opt algorithm for the TSP*. In: Proceedings of the eighteenth annual ACM-SIAM. Symposium on Discrete Algorithms. New Orleans, Louisiana, 2007, p.1295-1304.

[29] BENTLEY, J.L. *Experiments on Traveling Salesman Heuristics.* In: Proceedings of the first annual ACM-SIAM. Symposium on Discrete Algorithms. San Francisco, Califórnia, 1990. P.91-99.

[30] JOHSON, D.S.; MCGEOCH, L.A. The travelling salesman problem: A case study in local optimization. In AARTS, E.H.L.; LENSTRA, J.K. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Chichester, England, 1997. p.215 - 310.

[31] Or, I. Traveling Salesman-type Combinatorial Problems and their relation to the Logistics of Blood Banking. Ph.D. thesis, Department of Industrial Engineering and Management Science, Northwestern University, Evanston, 1976.

[32] BABIN, G.; DENEAULT, S.; LAPORTE, G. Improvements to the Or-opt heuristic for the symmetric travelling salesman problem. *Journal of the Operational Research Society*, v.58, n.3, p.402–407, 2007.

[33] LAPORTE, G.; MERCURE, H. Balancing Hydraulic Turbine Runners: A Quadratic Assignment Problem. *European Journal of Operational Research*, v.35, n.3, p.378-381, 1988.

[34] NVIDIA. Next Generation CUDA Architecture. Disponível em: <http://www.nvidia.com/object/fermi_architecture.html>. Acesso em: 10 mai. 2010.

[35] NVIDIA. Disponível em: <http://www.nvidia.com/>. Acesso em: 15 jan. 2010.