UNIVERSIDADE FEDERAL FLUMINENSE

EYDER FRANCO SOUSA RIOS

Exploração de Estratégias de Busca Local em Ambientes CPU/GPU

NITERÓI, RJ

2016

UNIVERSIDADE FEDERAL FLUMINENSE

EYDER FRANCO SOUSA RIOS

Exploração de Estratégias de Busca Local em Ambientes CPU/GPU

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Algoritmos e Otimização.

Orientadores:

Luiz Satoru Ochi
Maria Cristina Silva Boeres

NITERÓI, RJ

2016

Exploração de Estratégias de Busca Local em Ambientes CPU/GPU

Eyder Franco Sousa Rios

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Algoritmos e Otimização.

Aprovada por:

BANCA EXAMINADORA

Prof. Luiz Satoru Ochi, IC/UFF (Orientador)

Prof[a]. Maria Cristina Silva Boeres, IC/UFF (Orientadora)

Prof[a]. Simone de Lima Martins, IC/UFF

Prof. Yuri Abitbol de Menezes Frota, IC/UFF

Prof. Felipe Maia Galvão França, COPPE/UFRJ

Prof. Igor Machado Coelho, IME/UERJ

Prof. Ricardo Cordeiro de Farias, COPPE/UFRJ

Niterói - RJ, 14 de julho de 2016.

À minha esposa Josiane e meus filhos Mateus e Juliana.

Agradecimentos

Apesar de consistir em uma empreitada pessoal, trilhar o longo caminho do curso de doutorado e a consequente conclusão desse trabalho não é, de todo, um mérito individual. Muitas pessoas, de uma forma ou de outra, contribuíram para que essa jornada lograsse em êxito.

Meu primeiro e mais significativo agradecimento vai para minha esposa Josiane, que de forma abnegada assumiu os compromissos de nosso lar e proporcionou as condições imprescindíveis para a viabilização desse projeto. Sua sincera dedicação e amor incodicional à família tornaram mais suportável os longos períodos de ausência. Estendo a mesma deferência aos meus filhos Mateus e Juliana, de cuja presença e ternura fui longamente privado. A vocês declaro meu amor incondicional.

Quando decidi cursar doutorado, o Programa da UFF foi minha primeira opção. Não poderia ser diferente, depois da experiência previamente vivida durante o mestrado. Foi muito gratificante verificar que o ambiente agradável e o clima marcado pela amizade e companheirismo não haviam mudado. Não menos gratificante foi poder reencontrar amigos daquele período, como Jacques, Viterbo, Aline, Alexandre e Haroldo. Dessa época, uma referência especial à minha amiga Renatha Capua, infalível companheira das muitas e desoladas estadas na sala de alunos. Foi muito confortante reencontrá-la, conviver e construir consigo uma sincera amizade.

Não podem ser excluídos os novos amigos forjados no decorrer do curso. O pessoal da República OptHouse: Edcarllos, Pablo, Matheus, Uéverton, Helder e Pequenucho. Os colegas Rafael (gaúcho), Gleiph, Guerine e Petrucci. Os companheiros de república Hugo, Gilberto, Thiago e Teobaldo, bravos representantes da plêiade Paraibana. Os casais amigos Puca e Temis; Juliana e Zamith; Renatha e Thibaut; Igor e Cris; Julliany e Felipe. Não poderia deixar mencionar Teresa Cancela, fiel companheira de conversas e eventos culturais. Foi uma grande satisfação encontrar em vocês, além de colegas e companheiros, verdadeiros amigos.

Finalmente, um agradecimento especial aos meus orientadores Luiz Satoru e Cristina Boeres que com paciência, empenho e dedicação proporcionaram as condições para a efetivação desse trabalho. A experiência propiciada pela convivência com vocês possibilitou a formação, não apenas de um egresso, mas também de um admirador e sincero amigo.

Resumo

Meta-heurísticas paralelas têm sido empregadas em muitas aplicações industriais e científicas para a resolução eficaz de problemas de otimização combinatória. A busca local é um componente essencial para muitos desses métodos e, frequentemente, representa o esforço computacional dominante empenhado pelo algoritmo. Atraídas pelo potencial paralelo das plataformas CPU/GPU, muitas abordagens heurísticas tentam adaptar algoritmos de busca local tradicionais para essa plataforma sem considerar suas características intrínsecas. Esse trabalho apresenta um estudo detalhado sobre aspectos relacionados ao desempenho em sistemas híbridos CPU/GPU, além de introduzir novos métodos para a abordagem de problemas combinatórios nesses sistemas. Para tanto, um algoritmo bastante eficaz baseado na busca local RVND (*Randomized Variable Neighborhood Descent*) foi decomposto, adaptado e avaliado em diferentes configurações nessa plataforma. Adicionalmente, uma nova estratégia de seleção em vizinhança denominada *multi improvement*, capaz de aplicar múltiplos movimentos em uma vizinhança, foi utilizada para acelerar um novo procedimento de busca local chamado DVND (*Distributed Variable Neighborhood Descent*), especialmente concebido para ambientes multi-GPU. Para validar as abordagens propostas, um problema combinatório NP-Difícil, o Problema da Mínima Latência (PML), foi submetido a um *framework* heurístico baseado em GRASP e VNS denominado denominado GVNS. Diversas configurações paralelas do GVNS envolvendo o DVND e RVND foram experimentadas e apresentaram resultados que suplantaram o melhor algoritmo da literatura, com *speedups* de até 13,7 para instâncias do MLP com até 1.000 clientes. Os resultados obtidos indicam a eficácia das técnicas propostas em termos de qualidade da solução, desempenho e escalabilidade.

Palavras-chave: DVND, Multi improvement, Busca Local, GRASP, VNS, VND, GPU, Problema da Mínima Latência.

Abstract

Parallel metaheuristics have been used in many industrial and scientific applications to effectively solve combinatorial optimization problems. The local search is an essential component of many of these methods and, very often, represents the dominant computational effort accomplished by algorithm. Attracted by the massive parallel potential of platform CPU/GPU, many heuristic approaches try to adapt traditional local search algorithms for this plataform without considering its intrinsic characteristics. This work presents a detailed study of aspects related to performance in CPU/GPU hybrid systems, and introduces new methods to tackle combinatorial problems. A very effective algorithm based on the local search RVND (Randomized Variable Neighborhood Descent) was decomposed, adapted and evaluated in different configurations in that platform. Additionally, a new neighborhood search strategy called *multi improvement*, capable to apply multiple moves in a neighborhood, is used to accelerate a new local search procedure called DVND (Distributed Variable Neighborhood Descent), specially designed for multi-GPU environments. To validate the proposed approaches, a NP-Hard combinatorial problem, the Minimum Latency Problem (MLP), was subjected to a heuristic framework based on GRASP and VNS called GVNS. Several parallel configurations of GVNS involving DVND and RVND were tested and showed results that outperformed the best known algorithm of the literature, achieving speedups up to 13.7 for the larger MLP instances with up to 1,000 clients. The results demonstrated the effectiveness of the proposed techniques in terms of solution quality, performance and scalability.

Keywords: DVND, Multi improvement, Local Search, GRASP, VNS, VND, GPU, Minimum Latency Problem.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Muitos desafios da ciência e da indústria impõem a tarefa de encontrar a melhor configuração de um conjunto de parâmetros visando atingir determinado objetivo. Dentre estes desafios encontram-se os Problemas de Otimização Combinatória, que podem ser informalmente descritos como um estudo matemático para encontrar uma configuração ótima de objetos pertencentes a um conjunto discreto finito ou enumerável [Papadimitriou & Steiglitz, 1998]. Frequentemente, aplicações de problemas de otimização combinatória podem atingir escala e complexidade tais que o uso de algoritmos computacionais para sua resolução se torna necessário.

Meta-heurísticas são métodos reconhecidamente eficientes para a abordagem de problemas de otimização de alta complexidade computacional. Na literatura especializada, podem ser encontradas implementações de meta-heurísticas que aplicam diferentes estratégias visando, principalmente, acelerar a obtenção de resultados [Gendreau & Potvin, 2010]. Atualmente, devido à presença ubíqua de paralelismo nos computadores modernos, o desenvolvimento de meta-heurísticas paralelas tornou-se corrente. Com efeito, podem ser encontradas nesse contexto diversas contribuições que buscam explorar o paralelismo presente tanto nos algoritmos quanto nas diversas plataformas de *hardware* existentes [Alba *et al.*, 2013]. Tais algoritmos geralmente conseguem, além de abreviar a execução de aplicações, proporcionar robustez ao método e produzir resultados competitivos [Cung *et al.*, 2002].

Durante a última década, a presença de computadores com processadores *multi-core* se tornou perene em diversas instituições e laboratórios, disponibilizando recursos paralelos de baixo custo que podem ser aproveitados para execução de heurísticas paralelas. No entanto, os constantes avanços tecnológicos na fabricação de *hardware* e a grande variedade de modelos disponíveis, fazem com que a presença de máquinas com

diferentes configurações não seja incomum. Mais recentemente, as *Graphics Processing Units* (GPUs) emergiram como uma alternativa de alto poder computacional provido por sua arquitetura massivamente paralela (*manycore*). Deveras, os recursos paralelos presentes em tais dispositivos já ultrapassaram significativamente aqueles providos pelas CPUs convencionais, apresentando atualmente uma atraente relação custo/benefício dentre os sistemas de alto desempenho. Como consequência, o número de trabalhos envolvendo heurísticas orientadas para ambientes híbridos CPU/GPU tem crescido sensivelmente nos últimos anos [Alba *et al.*, 2013]. Tais abordagens, atraídas pelo potencial facultado por esta plataforma, buscam adaptar métodos heurísticos consagrados na expectativa de alcançar maior desempenho na execução de suas respectivas aplicações.

De fato, não é incomum encontrar trabalhos orientados para CPU/GPU anunciando ganhos de desempenho notáveis quando comparados com os métodos correspondentes projetados para CPU. No entanto, é necessário prudência ao realizar a comparação de desempenho de algoritmos executados em plataformas tão distintas. É preciso considerar que CPUs e GPUs são dispositivos construídos tendo como alvo diferentes tipos de aplicações. CPUs são projetadas para fornecer tempos de resposta rápidos para uma larga variedade de aplicações. Isso resulta em um *hardware* mais complexo, o que estabelece uma limitação tecnológica para a frequência de operação e o número de unidades de processamento em uma única pastilha. Por outro lado, as GPUs são desenvolvidas com foco em aplicações com alto nível de paralelismo de dados, onde cada unidade de processamento realiza a mesma tarefa em diferentes partes dos dados [Lee *et al.*, 2010]. Portanto, para se beneficiar de plataformas heterogêneas compostas por CPUs e GPUs, algoritmos heurísticos devem ser projetados de forma a explorar convenientemente os recursos paralelos presentes nesses sistemas, buscando alcançar um balanceamento adequado entre paralelismo de dados e de tarefas.

Entretanto, poucos trabalhos na literatura estão focados em reavaliar os mecanismos tradicionais de implementação de meta-heurísticas e seus componentes para atender as exigências de desempenho de sistemas CPU/GPU. Dentre estes componentes, a *busca local* se destaca como ingrediente essencial para diversas classes de meta-heurísticas. Esse método consiste na realização de uma sequência de mudanças a partir de uma solução inicial visando melhorar sistematicamente sua qualidade até que um ótimo local seja encontrado [Lenstra, 1997]. Tipicamente, o procedimento de busca local representa o esforço computacional dominante empenhado por uma meta-heurística. Apesar de sua importância para o desempenho do algoritmo, muitas abordagens tentam adaptar modelos tradicionais em plataforma CPU/GPU sem considerar as características intrínsecas

dessa arquitetura. Portanto, meta-heurísticas baseadas em busca local devem levar em conta não só a eficácia do método, mas as características inerentes da arquitetura, especialmente aquelas relacionadas com a distribuição de tarefas de processamento entre CPU e GPU; sincronização de *threads* e transferência de dados; e os fatores relacionados com a capacidade e desempenho de uma rica hierarquia de memória.

Este trabalho procura explorar os aspectos inerentes ao desempenho de meta-heurísticas em ambientes heterogêneos compostos por CPUs e GPUs. Para atingir tal objetivo, foi realizado um estudo detalhado da arquitetura GPU e seu relacionamento com o tradicional modelo de paralelismo orientado para CPU, buscando destacar os requisitos relevantes ao desempenho dos componentes que representam o maior impacto na performance de algoritmos heurísticos. Neste contexto, para melhor explorar os recursos da plataforma, foram investigados vários aspectos que envolvem a execução eficiente de procedimentos de busca local e em vizinhança. Como resultado é apresentada uma nova estratégia de busca local paralela inspirada em VND (*Variable Neighborhood Descent*) [Mladenović & Hansen, 1997], denominada DVND (do inglês, *Distributed* VND), especialmente concebida para sistemas *multicore* e multi-GPU. Além disso, foi introduzida uma nova estratégia de busca em vizinhança rotulada como *multi improvement* (melhoria múltipla), que tira proveito do paralelismo massivo da arquitetura GPU visando acelerar a convergência da busca local. Adicionalmente, foram conduzidos experimentos que destacam requisitos de projeto que visam orientar a exploração adequada dos recursos paralelos das plataformas CPU/GPU.

Para validar essa abordagem, foi selecionado um problema de otimização combinatória NP-Difícil, o Problema da Mínima Latência (PML) [Blum *et al.*, 1994], que possui muitas aplicações importantes na ciência e na indústria. Experimentos computacionais foram conduzidos embutindo os mecanismos mencionados em uma meta-heurística híbrida inspirada em GRASP (*Greedy Randomized Adaptive Search Procedure*) e VNS (*Variable Neighborhood Search*) [Gendreau & Potvin, 2010] objetivando demonstrar a eficácia dos métodos propostos em termos de qualidade de solução, desempenho e escalabilidade.

O restante desse documento está organizado como se segue. O Capítulo 2 apresenta características dos ambientes computacionais paralelos e ressalta aspectos sobre o projeto e desempenho de meta-heurísticas nesses sistemas. Adicionalmente, é realizada uma explanação um pouco mais detalhada sobre a plataforma e modelo de execução das GPUs, oportunidade em que se destacam características intrínsecas dessa arquitetura relacionados com desempenho. O capítulo seguinte apresenta um estudo de *performance* de

seis estratégias de implementação de busca local no ambiente híbrido CPU/GPU onde, oportunamente, são destacados os aspectos inerentes ao desempenho. O Capítulo 4 introduz duas novas metodologias relacionadas com a busca local e busca em vizinhança, onde resultados envolvendo esses métodos são apresentados e analisados. Finalmente, no último capítulo, são expressas as considerações finais e propostas de trabalhos futuros.

Capítulo 2

Meta-heurísticas e Ambientes Computacionais Paralelos

A cada ano, os componentes de *hardware* tornam-se mais compactos e mais rápidos, possibilitando a criação de equipamentos cada vez mais poderosos. Apesar deste desenvolvimento, ainda existem aplicações nas diversas áreas do conhecimento que demandam um poder computacional ainda maior. Para atender a tal demanda, uma alternativa viável de agregar poder computacional a estes equipamentos é a utilização de múltiplos processadores interconectados, de forma que juntos possam aumentar a capacidade de processamento do sistema.

Neste contexto, os recentes avanços na fabricação de computadores tornaram corriqueira a disponibilidade de novas tecnologias e recursos paralelos. Contudo, para que aplicações possam ser executadas de maneira adequada nestes ambientes, é necessário que o paralelismo existente tanto nos algoritmos quanto na arquitetura seja convenientemente explorado. A disponibilidade de novos recursos de computação, aliados ao desenvolvimento de novos modelos de meta-heurísticas paralelas, tornaram possível a resolução de problemas de alta complexidade até então inviáveis. Mais ainda, permitiram o surgimento de novos cenários que desafiam e motivam a investigação nesta área.

Este capítulo procura destacar as características dos ambientes computacionais paralelos relacionados com este trabalho, bem como abordar as principais estratégias de paralelização e classificação de meta-heurísticas. Por sua peculiaridade, um maior destaque é proporcionado à arquitetura GPU, onde são descritos e realçados os elementos relacionados com o desempenho de aplicações e que fundamentam algumas das técnicas utilizadas neste trabalho.

2.1 Ambientes Computacionais Paralelos

Na última década, grandes avanços tecnológicos introduziram recursos paralelos em diversas plataformas alçando-as para um novo patamar em termos de poder computacional. O projeto de meta-heurísticas paralelas procurou acompanhar a evolução do *hardware* buscando explorar os novos recursos proporcionados pelas modernas arquiteturas.

Para melhor entender como estes recursos influenciam e podem ser explorados pela implementação de heurísticas paralelas, esta seção procura descrever brevemente as características fundamentais das plataformas de processamento paralelo relevantes para esta tese. Esta descrição é orientada em função de como os recursos de memória e processamento estão organizados, além de analisar como estas características podem influenciar o projeto de meta-heurísticas paralelas.

A implementação bem sucedida de algoritmos paralelos está diretamente relacionada com a consideração das características disponíveis na plataforma para a qual são projetados. Isto porque a arquitetura de *hardware* tem um importante impacto no tempo e na forma como uma aplicação paralela pode realizar suas tarefas de computação, comunicação, sincronização e compartilhamento de dados. Neste contexto, as principais arquiteturas utilizadas em computação paralela podem ser classificadas como de *memória compartilhada* ou *memória distribuída* [Wilkinson & Allen, 2005].

Plataformas paralelas de memória compartilhada são compostas por múltiplas unidades de processamento integradas em um único dispositivo físico. Os diferentes processadores compartilham uma mesma unidade de memória que é utilizada adicionalmente para sincronização e troca de informações entre as tarefas da aplicação paralela. Embora apresentem um meio conveniente de interação entre tarefas, essas plataformas possuem uma desvantagem imposta pela limitação de escalabilidade, que determina o número de processadores que podem ser integrados em um único equipamento.

Ambientes de memória distribuída caracterizam-se por agregarem processadores interconectados, cada um com seus próprios recursos de memória e processamento. Nestes ambientes, a comunicação entre tarefas paralelas é implementada por meio de troca de mensagens, que alcançam os processadores (ou nós) através do barramento ou de uma rede de computadores. Plataformas distribuídas podem ser encontradas em sistemas formados por pequenos grupos de computadores interligados, compostos por equipamentos geograficamente dispersos ou em dispositivos integrados numa mesma unidade como os sistemas CPU/GPU.

2.2 Arquitetura GPU

Na última década, a presença de computadores compostos por processadores *multicore* tornou-se corrente, disponibilizando recursos com maior performance teórica que seus predecessores compostos por um único núcleo de processamento. Mais recentemente, dispositivos massivamente paralelos (*manycore*) como as *Graphics Processing Units* (GPUs) destacam-se como uma atraente alternativa de alto poder computacional. As GPUs evoluíram a partir de aceleradores gráficos dedicados e deram origem às modernas GPUs de propósito geral (GPGPU, *General Purpose GPU*), que disponibilizam recursos paralelos para computação de alto desempenho.

A arquitetura da GPU é organizada como uma matriz de multiprocessadores de *streaming* (SM, *Streaming Multiprocessors*) altamente encadeados, cada um contendo diversas unidades de processamento ou *cores* [Kirk & Wen-mei, 2012], conforme esboçado pela Figura 2.1 e descrita ao longo desta seção.

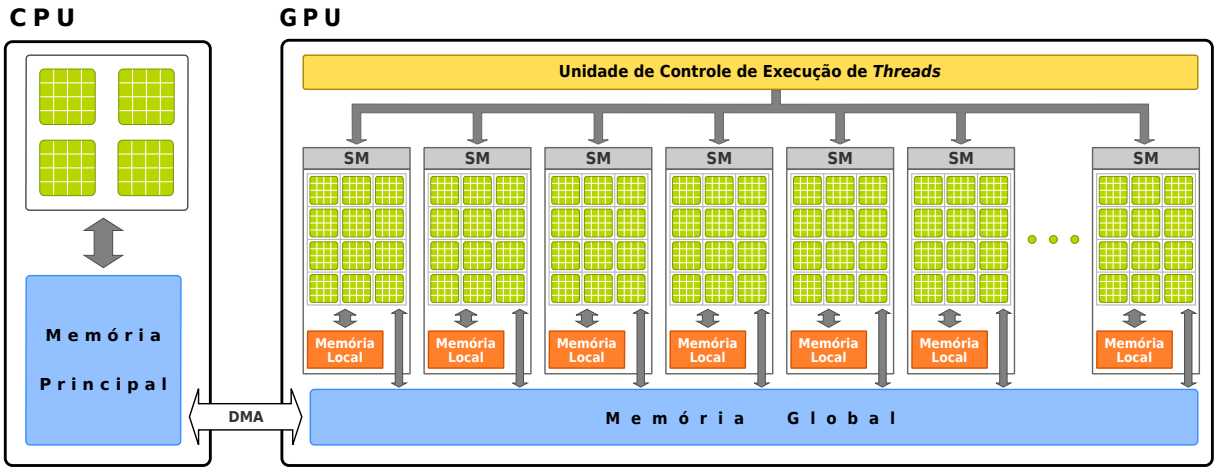


Figura 2.1: Arquitetura de uma GPU.

Apesar das denominações *multicore* e *manycore* serem largamente empregadas para rotular, respectivamente, as arquiteturas CPU e GPU, é preciso ponderar que os núcleos de processamento de cada plataforma possuem características distintas. Um núcleo de CPU é tipicamente mais robusto e projetado para tarefas lógicas e de controle mais complexas, sendo focado na execução de tarefas sequenciais. Um *core* GPU possui, relativamente, menor capacidade computacional sendo otimizado para o paralelismo de dados. Desta forma, apresenta um esquema de controle lógico mais elementar, cuja finalidade é favorecer o *throughput* da aplicação paralela [Cheng *et al.*, 2014].

A configuração de *hardware* da GPU — assim como o número de *cores* por SM

ou o volume de memória — depende da microarquitetura utilizada em sua fabricação e pode variar dependendo do modelo do produto. A capacidade de computação, ou *compute capability*, é um parâmetro que especifica as principais características de *hardware* que estarão disponíveis para o desenvolvedor. Como ilustração, a Tabela 2.1 apresenta algumas das microarquiteturas da linha de produtos da NVIDIA, o mais eminente fabricante de GPUs, as quais foram utilizadas nos experimentos realizados neste trabalho.

Microarquitetura	***Compute capability***
Tesla	1.0 a 1.3
Fermi	2.0 a 2.1
Kepler	3.0 a 3.7
Maxwell	5.0 a 5.3
Pascal	6.0 a 6.1

Tabela 2.1: Microarquiteturas das GPUs fabricadas pela NVIDIA.

2.2.1 Ferramentas de Desenvolvimento e Modelo de Execução

Originalmente, as GPUs foram projetadas para atender às demandas crescentes da indústria de jogos de computador, atuando como um co-processador gráfico com o objetivo de aliviar esta sobrecarga da CPU. Desta forma, qualquer aplicação não gráfica focada neste dispositivo deveria ser mapeada em termos de chamadas para alguma biblioteca gráfica como OpenCL[1], por exemplo. Atualmente, as modernas GPUs oferecem ferramentas de desenvolvimento mais amigáveis e poderosas. As principais ferramentas de programação para GPU são OpenCL e CUDA, sendo a primeira um padrão aberto e a última uma extensão proprietária da linguagem C++ provida pela NVIDIA. Neste trabalho, utilizou-se CUDA (*Compute Unified Device Architecture*) para as implementações orientadas para GPU por se tratar de uma ferramenta mais madura e sofisticada e, principalmente, porque oferece maior controle sobre o dispositivo.

Um programa CUDA consiste em uma ou mais fases que são executadas tanto na CPU quando na GPU. O código GPU é implementado na forma de funções C++ denominadas *kernels*. Toda aplicação CUDA inicia sua execução a partir da CPU, que lança *kernels* na forma de uma grade de computação (*compute grid*) tipicamente composta por um grande número de *threads* visando explorar o paralelismo de dados.

Observando a Figura 2.2 é possível verificar que uma grade de computação, ou

[1]OpenCL: acrônimo para *Open Computing Language*, um *framework* para escrita de programas para execução em plataformas heterogêneas compostas por CPUs, GPUs e outros tipos de processadores.

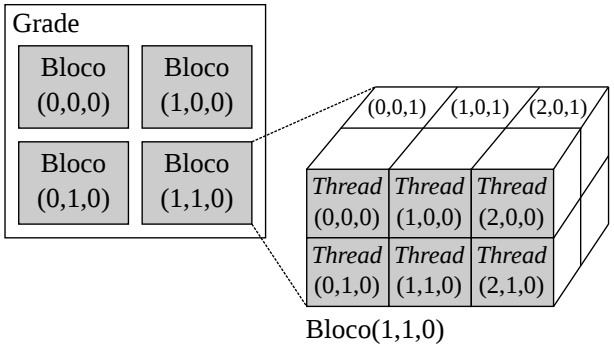


Figura 2.2: Estrutura de uma grade de computação para lançamento de *kernel*.

simplesmente grade, é organizada como uma estrutura de dois níveis. O primeiro nível é composto por uma matriz tridimensional de blocos, cada um podendo conter até 1.024 *threads*[2]. No segundo nível, cada bloco também possui um arranjo tridimensional cujas dimensões máximas variam de acordo com a *compute capability* do dispositivo. Apesar das restrições de dimensionamento, o número máximo de *threads* em uma grade pode atingir valores teóricos da ordem de 2^{36} para dispositivos de microarquitetura Fermi ou superior[3]. No diagrama apresentado, a grade possui dimensões $(x, y, z) = (2, 2, 1)$ e cada bloco $(3, 2, 2)$. Durante a execução de um *kernel*, cada *thread* de uma grade possui uma identificação única provida pelos índices do bloco ao qual pertence e pelos índices dentro de cada bloco. É por meio dessa identificação que cada *thread* é conduzida para a execução de sua tarefa específica, a qual usualmente está relacionada ao segmento das estruturas de dados que irá processar. A geometria de uma grade é definida pelo desenvolvedor no momento do lançamento do *kernel* e seu arranjo é um fator determinante para o desempenho da aplicação.

No decorrer da execução de um *kernel*, cada bloco da grade é atribuído a um único multiprocessador[4], ao qual permanece associado até que sua execução seja concluída. Dentro de cada SM, as *threads* de um bloco são escalonadas em grupos de 32 unidades conhecidos como *warps*, que são executados segundo o modelo SIMD (*Single Instruction, Multiple Data*) [Flynn, 1972], indicando que todas as *threads* executam uma mesma instrução a cada instante. O modelo de execução adotado pela GPU traz associado um problema que pode influenciar negativamente no desempenho de um *kernel*: a divergência de código. Tal distúrbio ocorre durante a execução de código condicional e pode fazer com que diferentes *threads* de um mesmo *warp* processem instruções diferentes. O modelo

[2]Para *hardware* com *compute capability* ≥ 2.0

[3]Resultado obtido pelo produto das valores máximos para cada dimensão da grade (dx, dy e dz) e o número máximo de *threads* por bloco (t). Mais precisamente, $dx \cdot dy \cdot dz \cdot t = 2^{10} \times 2^{10} \times 2^6 \times 2^{10} = 2^{36}$.

[4]Os termos SM e multiprocessador são utilizados como sinônimos de *streaming multiprocessor*.

de execução SIMD faz com que a divergência de código seja tratada pela GPU por meio do mascaramento e serialização da execução das *threads* de um *warp*.

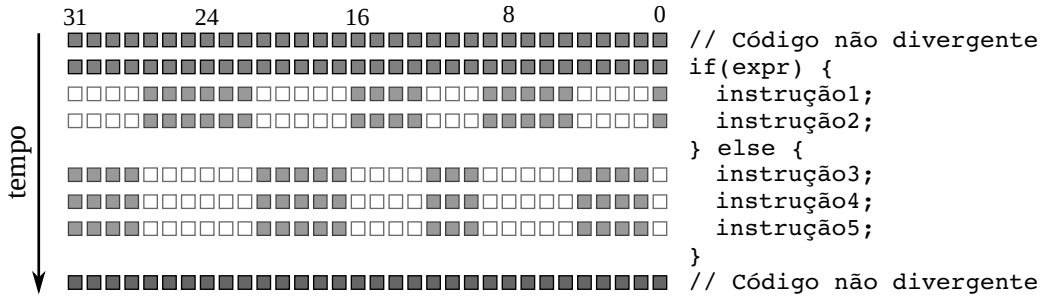


Figura 2.3: Mascaramento e serialização na execução de código condicional. Ilustração inspirada no trabalho de [Brodtkorb *et al.*, 2013].

A Figura 2.3 procura ilustrar este comportamento, onde cada linha de 32 quadros representa um *warp* e a presença de hachuras denota uma *thread* em atividade. No lado direito da figura, um trecho de pseudocódigo condicional é apresentado onde, associado a cada instrução, está retratado o estado correspondente do *warp*. Observando a ilustração, é possível verificar que diferentes *threads* executam as instruções associadas à estrutura *if*, enquanto as demais permanecem ociosas. Quando estas são concluídas, é a vez das *threads* associadas ao bloco *else* iniciarem sua execução, enquanto as demais aguardam sua conclusão. A divergência pode também ocorrer em outras estruturas de controle de fluxo, como *while* ou *for*, ou em instruções, como o operador ternário de C. Apesar do *hardware* da GPU estar preparado para resolver códigos divergentes sem a interferência do desenvolvedor, em muitos casos um ganho de desempenho significativo pode ser obtido evitando a divergência em códigos condicionais. O desenvolvedor pode realizar tal tarefa em tempo de codificação, por meio da combinação adequada de operações aritméticas e binárias. Uma compilação de soluções para contornar esse tipo de problema pode ser encontrada em [Anderson, 2005].

Operações aritméticas ou de acesso à memória global da GPU também podem afetar o desempenho forçando que todas as *threads* de um *warp* fiquem ociosas até que a operação seja completada. Para ocultar a latência associada a estas operações, a GPU escalona outro *warp* para manter o SM ocupado. Isto é possível porque a GPU é capaz de manipular mais *threads* por SM que o número de *cores* disponíveis. Em dispositivos de microarquitetura Kepler, por exemplo, cada SM é capaz é manipular até 2.048 *threads* (64 *warps*) simultaneamente, o dobro do número máximo de *threads* que podem ser lançadas por bloco em uma grade. Para manter o processador ocupado, sempre que um *warp* se torna ocioso, um novo pode ser alocado, pertencente ou não ao mesmo bloco. Desta forma,

uma métrica essencial para o bom desempenho do código GPU é a taxa de ocupação do multiprocessador (*occupancy*), que é dado pela razão entre o número de *warps* ativos em um multiprocessador e o número máximo de *warps* que este pode manipular.

2.2.2 Hierarquia de Memória

Associada à matriz de multiprocessadores, a GPU possui um espaço de memória próprio e hierarquicamente organizado. O dispositivo admite alguns tipos de memória que, quando convenientemente explorados, podem ser um elemento chave para bom desempenho da aplicação. A Figura 2.4 procura projetar como a grade de computação se relaciona com os diversos tipos de memória da GPU.

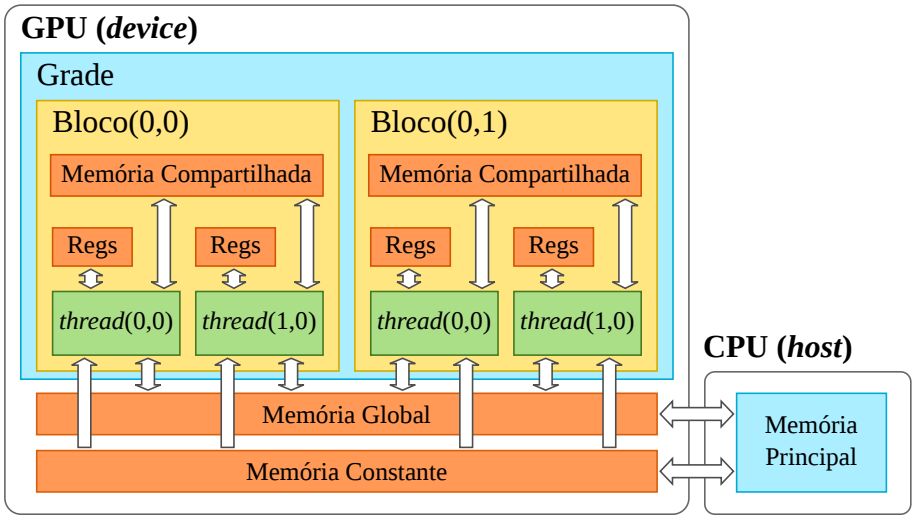


Figura 2.4: Hierarquia de memória da GPU em função da grade de computação.

Nesta ilustração é utilizada uma grade com dimensões $(x, y, z) = (1, 2, 1)$, onde cada bloco possui duas *threads* distribuídas conforme o arranjo $(x, y, z) = (2, 1, 1)$. Na parte inferior do diagrama pode-se observar a memória global e a memória constante, que podem ser acessadas a partir da CPU por meio de chamadas da API CUDA. Na realidade, este é o mecanismo utilizado para a troca de dados entre CPU e GPU, também denominadas, respectivamente, como *host* e *device* no jargão CUDA. No *device*, a memória constante pode ser acessada apenas para leitura e possui baixa capacidade de armazenamento, na ordem de 64KBytes. Por sua vez, a memória global pode ser lida e escrita pelo dispositivo e apresenta capacidade de armazenamento similar a da memória principal dos modernos sistemas CPU[5]. Os dados armazenados na memória global podem

[5]No modelo NVIDIA GTX-780, por exemplo, a capacidade da memória global atinge 3GBytes, enquanto no modelo GTX TITAN X do mesmo fabricante, a capacidade é de 12GBytes.

ser acessados por todas as *threads* e ficam disponíveis durante o tempo de execução da aplicação CUDA.

Apesar de sua ampla capacidade de armazenamento, a memória global apresenta alta latência de acesso, em torno de centenas de ciclos de *clock*. Isto é devido, principalmente, à sua implementação como memória de acesso aleatório dinâmico (DRAM) e à limitação da largura de banda das vias de acesso [Kirk & Wen-mei, 2012]. Embora o lançamento massivo de *threads* possa teoricamente contribuir para ocultação da latência de acesso à memória global, este mesmo recurso pode facilmente congestionar as vias de transferência de dados conduzindo, eventualmente, alguns multiprocessadores à ociosidade. A transferência de/para a memória global é, portanto, um componente crítico a ser considerado no projeto de aplicações baseadas em GPU.

Cada multiprocessador possui registradores e uma pequena porção de memória que podem ser acessados em alta velocidade e de forma altamente paralela. Os dados armazenados em registradores são de acesso restrito a *threads* individuais, sendo utilizados para armazenar dados de uso frequente, como variáveis locais por exemplo. Parte da memória do SM é usada como memória compartilhada e pode ser acedida por todas as *threads* de um bloco, configurando-se num mecanismo altamente eficiente para o compartilhamento de dados neste escopo. A Tabela 2.2 apresenta a relação entre escopo e tempo de vida[6] dos dados armazenados nos diferentes tipos de memória de uma GPU.

Tipo de memória	**Escopo**	**Tempo de vida**
Registrador	*Thread*	*Kernel*
Compartilhada	Bloco	*Kernel*
Global	Grade	Aplicação
Constante	Grade	Aplicação

Tabela 2.2: Escopo e tempo de vida de dados alocados nos diferentes tipos de memória de uma GPU.

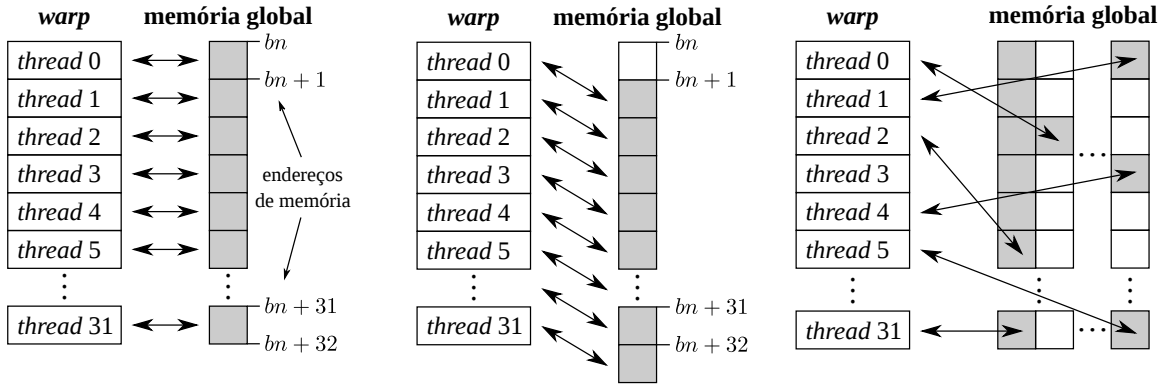
As características intrínsecas dos tipos de memória da GPU estabelecem um importante impasse para o desenvolvedor: a memória global é ampla, porém lenta; enquanto a memória compartilhada é escassa, contudo extremamente rápida. De fato, este *trade-off* quando explorado adequadamente é um dos aspectos marcantes para o desempenho do *kernel*. Neste contexto, uma estratégia de implementação tradicional é carregar para a memória compartilhada a porção de dados que as *threads* de um bloco necessitam para

[6]Neste contexto, o *tempo de vida* indica que os dados estarão alocados e disponíveis para uso durante o tempo de execução da porção de código indicada.

realizar sua tarefa para, posteriormente, operar sobre estes dados. Quando estas concluem sua incumbência, e os dados processados estão prontos para armazenamento, estes podem ser transferidos para a memória global.

Como forma de acelerar o acesso à memória global, a GPU utiliza parte da memória do SM como *cache* de dados. Na microarquitetura Kepler por exemplo, os 64KBytes de memória de cada SM podem ser configurados pelo desenvolvedor para que 48KBytes sejam usados como memória compartilhada e 16KBytes para a *cache* ou vice-versa. É evidente que, em relação à CPU, a capacidade da memória *cache* da GPU é bastante reduzida e, frequentemente, não é suficiente para atender à grande demanda das *threads* executadas em cada multiprocessador. Contudo, o desenvolvedor pode utilizar características intrínsecas da arquitetura da GPU para minimizar a latência de acesso à memória global como, por exemplo, fazer acesso coalescente à memória, uma técnica que consiste em combinar múltiplos acessos à memória em um número reduzido de transações.

Em outras palavras, o acesso coalescente consiste em projetar o código de forma que *threads* adjacentes acessem dados que residam em regiões contíguas da memória global. Por exemplo, em dispositivos Kepler, cada grupo de 128 bytes pode ser acessado por um *warp* em uma única transação. Contudo, para garantir a eficiência do acesso coalescente, o bloco[7] de dados deve residir contiguamente na memória em um endereço alinhado, isto é, em um endereço múltiplo de um tamanho de palavra b determinado pela arquitetura do dispositivo ($b = 128$, no caso da microarquitetura Kepler).



(a) sequencial alinhado (b) sequencial desalinhado (c) aleatório

Figura 2.5: Exemplos de cenários de acesso à memória global.

A Figura 2.5 procura ilustrar diferentes cenários de acesso à memória global a partir das *threads* de um *warp*. O cenário (a) representa a situação de acesso ideal, onde

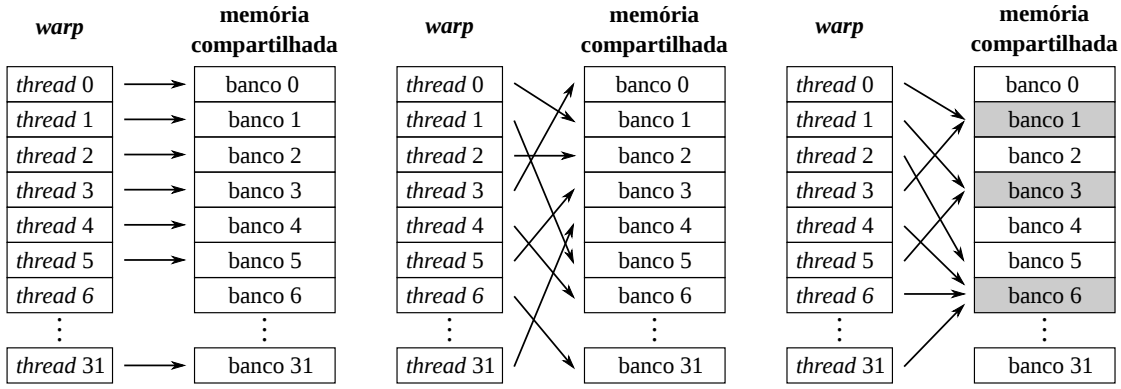
[7]O termo *bloco* aqui se refere a uma sequência contígua de bytes residentes na memória, não tendo relação com o componente bloco da grade de computação.

threads adjacentes acessam um bloco de dados na memória global que está alocado em um endereço alinhado (bn é múltiplo de b). Isto porque, sempre que a GPU realiza uma transação de acesso à memória global, ela o faz em blocos de b bytes, onde b depende da microarquitetura alvo. Desta forma, todos os dados necessários para execução da instrução que está sendo processada pelo *warp* estarão disponíveis com apenas uma operação de leitura na memória global. O mesmo argumento pode ser estendido para as operações de gravação. No cenário (b), o acesso é coalescente, porém o endereço do bloco não está alinhado. Isto faz com que o *hardware* precise realizar duas operações de acesso à memória: uma para carregar o bloco delimitado pelos endereços bn e $bn+31$, e outra para o bloco que inicia no endereço $bn + 32$. Como a maior parte dos dados do segundo bloco não será utilizada pelo *warp*, ocorre um desperdício de uso da banda de acesso à memória global, além do *overhead* associado à carga do bloco extra. Finalmente, em (c), observa-se um exemplo de acesso aleatório, onde diversas transações precisam ser executadas ocasionando o transporte de dados irrelevantes de/para a memória *cache* associada ao *warp* requisitante configurando, portanto, uma situação que deve ser evitada.

Num primeiro momento pode-se presumir que a consideração de tais detalhes de implementação possa parecer preciosismo. Porém, é preciso ponderar que essas operações são realizadas concorrentemente por milhares de núcleos de processamento e, de fato, influenciam de forma contundente no desempenho do *kernel*. Na realidade, um processo similar também ocorre entre acessos feitos às memórias *cache* e principal da CPU. Contudo, nesta arquitetura a concorrência pelo acesso à memória possui uma magnitude bastante reduzida quando comparada com a da GPU.

Como já mencionado, a memória compartilhada (*shared memory*) da GPU foi projetada para permitir acessos em grande velocidade e de forma altamente paralela. Para tanto, os circuitos de memória são organizados na forma de bancos de mesma dimensão que podem ser acessados simultaneamente em uma única transação. Cada banco é organizado em grupos de palavras de precisão simples (32 bits) de forma que cada palavra consecutiva reside em um banco diferente. Assim, quando um *warp* realiza uma transação na memória compartilhada que acesse dados residentes em bancos distintos, esta transação pode ser realizada paralelamente. Em contrapartida, quando *threads* de um mesmo *warp* acessam o mesmo banco, a operação é serializada produzindo um gargalho de desempenho conhecido como conflito de banco.

A Figura 2.6 apresenta alguns padrões de acesso à memória compartilhada. Nos padrões (a) e (b), cada *thread* acessa dados que residem em bancos distintos, de forma



(a) Acesso contíguo (b) Acesso aleatório (c) Acesso concorrente

Figura 2.6: Padrões de acesso aos bancos da memória compartilhada.

que todas as operações são realizadas paralelamente, sem conflitos de banco. Em (c), verifica-se que algumas *threads* concorrem pelo acesso a determinados bancos, gerando conflitos e, consequentemente, a serialização das operações. Convém mencionar que os conflitos de banco ocorrem apenas em operações de escrita na memória compartilhada.

2.2.3 Coordenação entre CPU e GPU

Plataformas compostas por CPU e GPU possuem espaços de memória fisicamente distintos classificando-se, portanto, como sistemas de memória distribuída. Como ilustrado pela Figura 2.4, a memória da GPU (global e constante) pode ser manipulada por um processo executado na CPU. É possível alocar, liberar e transferir dados entre a memória do *host* e do *device* por meio de chamadas específicas da API CUDA. Assim como nos sistemas de memória distribuída tradicionais, a transferência de dados e a sincronização entre os nós envolvidos — analogamente CPU e GPU — geralmente representam componentes chave para o desempenho e devem, portanto, ser criteriosamente considerados no projeto de aplicações.

Até o momento, foram abordados detalhes acerca do modelo de execução paralela adotado internamente pela GPU, onde um *kernel* é executado por meio de GPU *threads* concorrentes. No entanto, existem ainda outros mecanismos que permitem intermediar o paralelismo entre CPU e GPU que, frequentemente, conduzem a um incremento no desempenho global da aplicação. Tais mecanismos estão relacionados com a coordenação entre operações executadas pela CPU e GPU e consistem no lançamento assíncrono de operações e sua execução concorrente na GPU. A compreensão desse processo está relacionada com o conceito de *stream*, que constitui o mecanismo pelo qual grupos de operações sequenciais podem ser disparados de forma assíncrona pela CPU e executados concorren-

temente na GPU. Em outras palavras, as instruções associadas a uma mesma *stream* são executadas na ordem em que são encadeadas, enquanto *streams* distintas podem ter suas respectivas operações executadas paralelamente pela GPU.

Tipicamente, a execução de um *kernel* envolve três etapas distintas, sempre realizadas a partir da CPU: (1) cópia de dados de entrada da memória da principal da CPU para a memória global (ou constante) da GPU; (2) lançamento (execução) do *kernel* para atuar sobre esses dados e (3) cópia de dados processados pelo *kernel* da memória da GPU para a CPU. Cada uma dessas operações pode ser disparada síncrona ou assincronamente em relação à CPU, a critério do desenvolvedor.

A Figura 2.7 apresenta dois cenários envolvendo o lançamento de três *kernels* distintos. No diagrama, as operações rotuladas com *h2d* (*host to device*) representam operações de cópia de dados da memória da CPU para a memória da GPU enquanto, analogamente, os rótulos *d2h* (*device to host*) indicam a mesma operação na direção oposta. Os retângulos hachurados representam, como o rótulo sugere, a execução de um *kernel*.

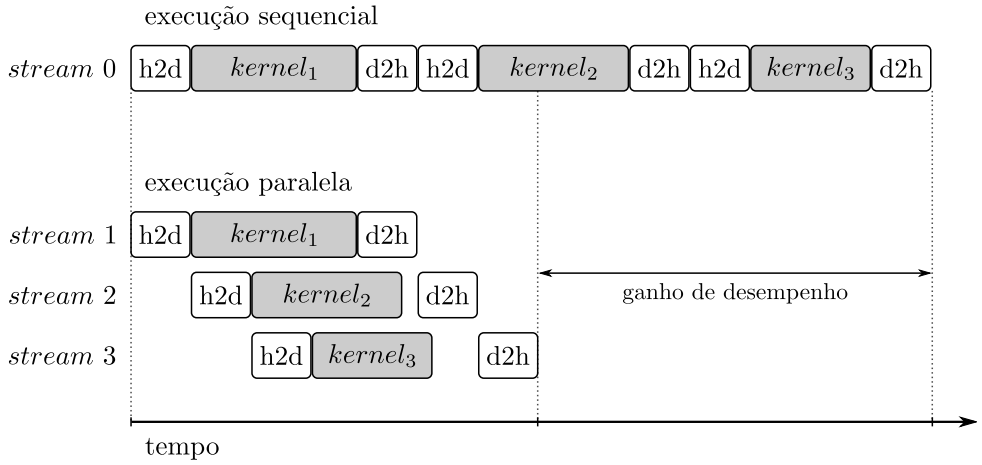


Figura 2.7: Operações síncronas e assíncronas envolvendo três operações de chamada de *kernel*.

Na parte superior da figura, na faixa horizontal associada à *stream* 0, verifica-se a execução sequencial de nove operações. Destacando-se as três primeiras operações, observa-se que houve inicialmente uma cópia de dados da CPU para a GPU. Em seguida, estes dados foram utilizados como entrada pelo $kernel_1$ que, após a conclusão de sua execução, tornou disponível na memória do *device* novos dados que foram transferidos para a memória principal do *host*. Situação similar ocorre com as demais operações associadas à *stream* em questão. Uma vez que essas operações foram encadeadas numa mesma *stream*, o *firmware* da GPU garante sua execução nessa mesma sequência.

Frequentemente, a maior parte do tempo gasto nas três operações em evidência é consumida pela execução do *kernel*. Nessas situações, é possível ocultar a latência da comunicação entre CPU e GPU despachando-se a execução dos *kernels* em *streams* separadas, como ilustrado na porção inferior da Figura 2.7. Neste caso foram utilizadas três *streams* para o lançamento das operações relacionadas com a execução de cada um dos três *kernels*, onde é possível observar sua execução concorrente. Apesar das operações de cada *stream* terem sido lançadas assincronamente em relação à CPU, uma análise mais atenta evidencia que as operações de cópia de dados não foram realizadas concorrentemente. Tal contenção ocorre porque estas utilizam um recurso comum para a condução dos dados: o barramento. Ainda assim, a comparação entre os dois cenários mostra que um ganho de desempenho significativo foi obtido.

2.3 Meta-heurísticas

Duas das abordagens tradicionalmente empregadas na implementação de algoritmos para o tratamento de problemas de otimização combinatória são os métodos exatos e heurísticos. Métodos exatos permitem a obtenção da melhor solução possível para o problema (solução ótima). Porém, para muitas aplicações teóricas e práticas de maior magnitude a demanda computacional necessária para a execução desses métodos é frequentemente impraticável. Por sua vez, métodos heurísticos possibilitam a produção de resultados sub-ótimos — eventualmente ótimos — consumindo tempo e recursos computacionais que normalmente atendem às limitações impostas pela aplicação.

Dentre os métodos heurísticos, as meta-heurísticas são largamente reconhecidas como ferramentas essenciais para o tratamento de problemas complexos em diversas áreas e, frequentemente, consistem na única abordagem viável capaz de atender às demandas de determinadas aplicações [Blum *et al.*, 2005]. Originalmente cunhado por Fred Glover em [Glover, 1986], o termo meta-heurística é utilizado para descrever algoritmos que empregam metodologias gerais de alto nível que são utilizadas como estratégias orientadoras de heurísticas subjacentes para a resolução de problemas de otimização combinatória [Talbi, 2009]. Em outras palavras, este conceito estabelece que as meta-heurísticas executam um processo de busca capaz de escapar de ótimos locais ao tempo em que realizam a varredura "inteligente" do espaço de soluções por meio de heurísticas especializadas.

Esta classe de algoritmos foi originalmente posta em evidência com a introdução dos Algoritmos Genéticos [Holland, 1975]. Porém, foi somente a partir da década

de 1980 que as meta-heurísticas experimentaram um desenvolvimento mais acentuado, se tornando atualmente um método consagrado para abordagem de problemas de alta complexidade. Alguns dos membros mais representativos desta classe de algoritmos são: Algoritmos Evolutivos (*Evolutionary Algorithms*) — que incluem os Algoritmos Genéticos —, *Simulated Annealing*, Busca Tabu (*Tabu Search*), Colônia de Formigas (*Ant Colony Optimization*), Enxame de Partículas (*Particle Swarm Optimization*), GRASP (*Greedy Randomized Adaptive Search Procedures*), VNS (*Variable Neighborhood Search*), ILS (*Iterated Local Search*), Busca por Dispersão (*Scatter Search*), entre outras [Glover & Kochenberger, 2003, Gendreau & Potvin, 2010].

2.3.1 Classificação de Meta-heurísticas

Um elemento chave no projeto de meta-heurísticas é o balanceamento dinâmico entre intensificação e diversificação da busca. O primeiro termo está relacionado com a exploração de soluções promissoras de forma mais intensiva, enquanto o último orienta o algoritmo na sondagem eficiente do espaço de soluções. Isso significa que tal balanceamento deve permitir que o algoritmo seja capaz de identificar rapidamente regiões com soluções de alta qualidade, ao tempo em que deve descartar outras já exploradas ou que não possam prover soluções promissoras [Blum *et al.*, 2005].

No entanto, estabelecer adequadamente o equilíbrio entre intensificação e diversificação não é uma tarefa elementar. As estratégias empregadas por diferentes meta-heurísticas para balancear esses componentes são altamente dependentes das filosofias que orientam suas respectivas concepções. Consequentemente, algumas meta-heurísticas podem mostrar maior tendência para intensificação e outras para diversificação [Blum & Roli, 2003].

Neste contexto, a classificação de meta-heurísticas pode auxiliar a melhor identificar como estes mecanismos são implementados. Dependendo do critério adotado, várias classificações podem ser encontradas na literatura. Alguns desses critérios estão relacionados a seguir [Talbi, 2009]:

(a) Inspiração na Natureza: a estratégia de orientação da busca é baseada em fenômenos naturais;

(b) Uso de memória: existe a coleta e o armazenamento dinâmico de informações durante a busca para a construção de conhecimento acerca do espaço de soluções;

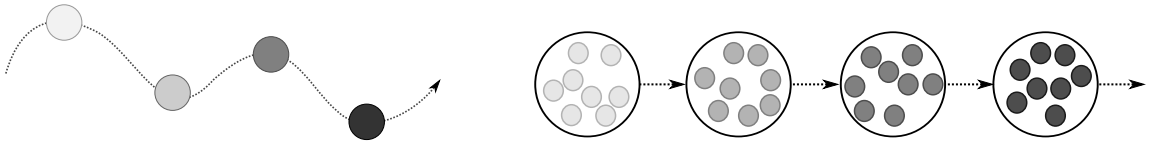
(c) Estocástico × determinístico : durante a busca o algoritmo utiliza ou não elementos aleatórios em sua estrutura (estocástico e determinístico, respectivamente);

(d) Iterativa × gulosa: algoritmos iterativos iniciam com uma solução completa (ou um grupo de soluções) e a transforma a cada iteração. Métodos gulosos tipicamente iniciam com uma solução vazia e a cada passo adicionam um novo componente à solução até que esta esteja completa;

(e) Busca baseada em trajetória ou população: classificação em função do número de soluções usadas pelo algoritmo em cada iteração.

Neste trabalho será adotado o critério de classificação em função do número de soluções utilizadas pelo algoritmo (trajetória ou população) [Blum *et al.*, 2005]. Esta escolha é motivada pelo fato de que esta categorização permite uma descrição mais clara dos diversos modelos de meta-heurísticas a serem apresentados posteriormente.

Uma meta-heurística baseada em trajetória[8] (MHBT) parte de uma única solução e, a cada etapa da busca, procura substituir a solução corrente por outra mais promissora obtida em sua vizinhança. Por vizinhança entenda-se o conjunto de soluções que possuem características similares à solução corrente segundo um determinado critério. Neste processo, as soluções que vão sendo obtidas a cada iteração estabelecem a trajetória traçada pelo algoritmo dentro do espaço de busca, como ilustrado na Figura 2.8(a). Essa metodologia usualmente permite a rápida detecção de ótimos locais, o que leva o algoritmo a adotar uma orientação voltada para a intensificação da busca em determinadas regiões do espaço de soluções [Blum & Roli, 2003, Talbi, 2009].



(a) trajetória (b) população

Figura 2.8: Ilustração das estratégias globais de busca adotadas por meta-heurísticas de acordo com o número de soluções utilizadas a cada iteração do algoritmo.

Por sua vez, as meta-heurísticas baseadas em população (MHBP) trabalham com várias soluções durante cada etapa da busca. Inicialmente, uma população de soluções é gerada segundo um critério preestabelecido e então aprimorada iterativamente. A cada

[8]Outros termos encontrados na literatura para esta classe de algoritmos são: meta-heurística baseada em solução única (*single solution*) ou de caminho único (*single walk*).

iteração, a população inteira — ou parte dela — é modificada por meio de operações entre seus componentes e então substituída por uma nova geração de indivíduos até que um critério de parada seja atingido, como esboçado pela Figura 2.8(b). Estes algoritmos possuem um comportamento orientado para a diversificação do processo de busca, uma vez que tendem a explorar diferentes áreas do espaço de soluções [Blum & Roli, 2003].

2.4 Meta-heurísticas Paralelas

Embora o uso de meta-heurísticas proporcione aumento da eficiência e redução da complexidade temporal do processo de busca, a demanda por tempo e recursos computacionais para resolução de determinadas aplicações ainda constitui um desafio. As necessidades crescentes de setores da ciência e indústria limitam o número de problemas que ainda podem ser resolvidos em tempo aceitável por algoritmos sequenciais. Além disso, uma vez que o desempenho das heurísticas frequentemente depende de características específicas da configuração do problema, o projeto de meta-heurísticas não apenas deve torná-las mais rápidas e eficientes, mas também mais robustas, no sentido de apresentarem um desempenho estável perante uma larga variedade de aplicações.

Neste contexto, as meta-heurísticas paralelas (MHP) surgem com uma alternativa para alcançar estes objetivos. O paralelismo emerge não apenas como recurso para reduzir o tempo de computação, mas também como meio para incrementar a qualidade das soluções obtidas e proporcionar robustez ao algoritmo [Cung *et al.*, 2002].

A computação paralela ou distribuída de aplicações para resolução de problemas de otimização combinatória implica em vários processadores trabalhando simultaneamente para resolver determinada instância[9] de um problema. O paralelismo, portanto, está relacionado com a decomposição e distribuição da carga computacional da aplicação entre tarefas que são executadas pelos diversos processadores do sistema [Talbi, 2009]. Segundo [Crainic & Gendreau, 2010], a principal fonte de paralelismo nas meta-heurísticas encontra-se na execução concorrente dos laços mais internos do algoritmo que, dependendo do paradigma, são responsáveis pela busca em vizinhanças (MHBT) ou avaliação de indivíduos (MHBP). Assim, estes são geralmente os únicos trechos do algoritmo onde o paralelismo é evidente e pode ser mais facilmente implementado. A maior parte dos

[9] O termo *instância* é largamente utilizado na área de otimização para se referir a uma representação concreta de um problema como, por exemplo, um arquivo de entrada. Contudo, este termo não possui tal acepção na língua portuguesa configurando, portanto, um caso de estrangeirismo devido à sua similaridade com a palavra inglesa *instance*. Mesmo assim, esse termo será adotado com este significado uma vez que é parte do jargão do campo de otimização.

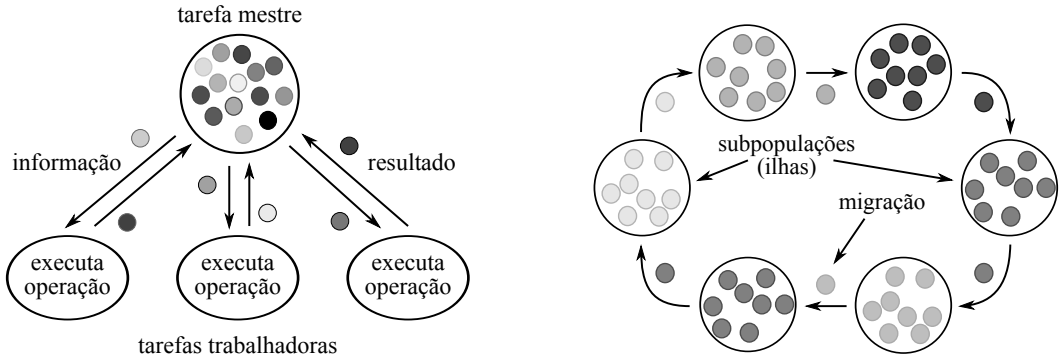
demais componentes possui algum tipo de dependência relacionada com tempo ou com a execução de passos anteriores para que possam ser completados. Mesmo quando outras fontes de paralelismo podem ser identificadas neste contexto, o esforço empenhado na sincronização ou comunicação das tarefas pode gerar atrasos significativos, o que pode tornar irrelevante o ganho alcançado pelo paralelismo.

Uma outra fonte de paralelismo pode ser encontrada no domínio do problema ou em seu espaço de busca correspondente. Quando não há dependência de dados entre funções de cálculo de custo ou de avaliação de diferentes soluções, estes procedimentos podem ser computados em paralelo. Existem, porém, algumas restrições na exploração eficiente do paralelismo neste contexto. Uma limitação óbvia é que não é possível atribuir a avaliação de cada solução a um processador diferente devido ao elevado número de operações necessárias. Isto significa que o espaço de soluções deve ser particionado e a avaliação de cada partição executada serialmente por um mesmo processador. Um dificuldade decorrente é que, além do particionamento em si não ser uma tarefa trivial, o resultado geralmente consiste em componentes de alta cardinalidade, que ainda demandam grande esforço computacional para serem processados [Crainic & Gendreau, 2010].

2.4.1 Modelos de Meta-heurísticas Paralelas Baseadas em População

As meta-heurísticas baseadas em população são algoritmos iterativos de natureza estocástica que operam sobre um conjunto de indivíduos (população). Cada indivíduo codifica uma solução que é avaliada por uma função que quantifica sua adequação ao problema abordado. A cada iteração, um grupo de indivíduos da população é selecionado e operadores são probabilisticamente aplicados na tentativa de produzir soluções de melhor qualidade. Como exemplos de meta-heurísticas pertencentes a esta família, podem ser citados os Algoritmos Evolutivos, Colônia de Formigas, Enxame de Partículas, Busca por Dispersão, entre outros [Gendreau & Potvin, 2010].

A paralelização de meta-heurísticas baseadas em população é relativamente natural, especialmente quando cada indivíduo é um componente independente do problema [Cung *et al.*, 2002]. De fato, quando implementados paralelamente, o desempenho de algoritmos sequenciais desta família geralmente é incrementado [Alba *et al.*, 2013]. Duas estratégias de paralelização dominam os modelos de MHBP encontrados na literatura [Alba & Dorronsoro, 2005]: (a) paralelização da computação e (b) paralelização da população.



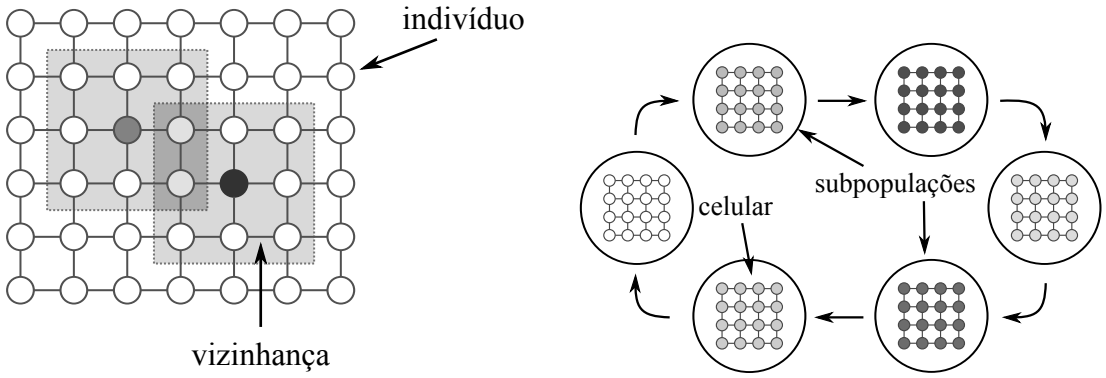
(a) paralelização da computação (b) paralelização da população

Figura 2.9: Modelos clássicos de meta-heurísticas paralelas baseadas em população.

Implementações do modelo de paralelização da computação usualmente são baseados no paradigma mestre-trabalhador, como ilustrado na Figura 2.9(a). O uso deste modelo remonta às primeiras implementações paralelas de MHBP, onde uma tarefa mestre concentra a população, além de selecionar e distribuir as operações entre tarefas trabalhadoras associadas, que atuam sobre os indivíduos da população. Outra abordagem implementada a partir deste modelo consiste em execuções independentes de um algoritmo sequencial em processadores distintos sem interação entre as tarefas paralelas.

O modelo de paralelização da população consiste num rearranjo espacial dos indivíduos para, em seguida, efetuar o processamento paralelo das subpopulações obtidas. Neste esquema, as topologias paralelas distribuída e celular estão entre as mais utilizadas [Alba & Tomassini, 2002]. No caso da topologia distribuída, apresentada na Figura 2.9(b), a população é particionada em um pequeno número de subpopulações (ilhas) sobre as quais algoritmos sequenciais são executados por diferentes processadores. Eventuais trocas de indivíduos e/ou informações podem ser realizadas entre as tarefas paralelas com o objetivo de estabelecer um mecanismo de diversificação para o algoritmo. Nos métodos estruturados com topologia celular, o conceito de vizinhança estabelece que um determinado indivíduo só deve interagir com outros em sua proximidade. Como pode ser visto na Figura 2.10(a), pode ocorrer a sobreposição de vizinhanças de indivíduos próximos. Este fato é interessante, pois permite a exploração de características de um mesmo indivíduo por tarefas paralelas distintas, o que induz um mecanismo implícito de diversificação no algoritmo.

Métodos fundamentados na combinação das topologias distribuída e celular também podem ser implementados, como ilustrado na Figura 2.10(b). Esses modelos híbridos geralmente utilizam uma abordagem de paralelização em dois níveis. Tipicamente, apli-



(a) celular (b) híbrido

Figura 2.10: Modelos de meta-heurísticas paralelas por rearranjo da população segundo os paradigmas distribuído e celular.

cam um esquema de granularidade[10] grossa no nível mais alto, que aloca tarefas aos nós; e granularidade fina no nível inferior, que explora o paralelismo intrínseco da arquitetura da máquina alvo.

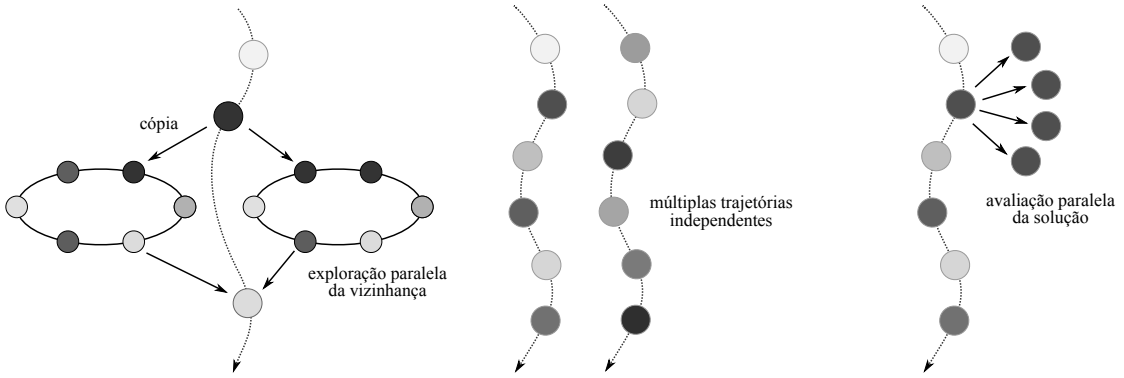
2.4.2 Modelos de Meta-heurísticas Paralelas Baseadas em Trajetória

Como abordado previamente, meta-heurísticas baseadas em trajetória buscam aprimorar a solução corrente selecionando uma outra mais vantajosa a partir de sua vizinhança. Segundo [Crainic & Toulouse, 2003], este tipo de algoritmo procura resolver um problema "percorrendo vizinhanças", demarcando assim trajetórias de busca no domínio de soluções do problema. *Simulated Annealing*, Busca Tabu, ILS, VNS e GRASP são exemplos representativos de meta-heurísticas que adotam esta estratégia.

Esses algoritmos tendem a concentrar a busca em torno de soluções promissoras podendo, eventualmente, negligenciar outras regiões do espaço de soluções. Uma forma de conferir maior eficiência a esses métodos é a utilização de paralelismo. Segundo Alba [Alba, 2005], os diferentes modelos paralelos para MHBT encontrados na literatura podem ser classificados em três grupos distintos: (a) modelo de movimentos paralelos; (b) modelo de multipartida paralela; e (c) modelo de aceleração de movimento.

O modelo de movimentos paralelos (*parallel moves model*) é baseado no paradigma

[10]granularidade: indica a relação entre os níveis de computação e comunicação das tarefas que compõem uma aplicação paralela. Granularidade grossa (*coarse-grained*) designa uma decomposição em um pequeno número de tarefas pesadas (maior computação, menor comunicação). Granularidade fina (*fine-grained*) indica um grande número de pequenas tarefas leves (menor computação, maior comunicação).



(a) movimentos paralelos (b) multipartida paralela (c) aceleração de movimento

Figura 2.11: Modelos clássicos de meta-heurísticas paralelas baseadas em trajetória.

mestre-trabalhador[11] clássico e quase sempre não altera o comportamento da heurística de busca original. Como ilustrado na Figura 2.11(a), no início de cada iteração, a tarefa mestre replica a solução corrente entre os processadores do ambiente paralelo. Cada qual, por meio de sua respectiva tarefa trabalhadora, gerencia e executa separadamente o processamento da solução recebida enviando o resultado para a tarefa mestre.

No modelo de multipartida paralela (*parallel multistart model*) várias heurísticas baseadas em trajetória — ou cópias de uma mesma — são disparadas em cada processador da máquina paralela. As tarefas paralelas podem ser independentes ou cooperativas, iniciar com a mesma ou com diferentes soluções e podem, ainda, ser configuradas com parâmetros idênticos ou distintos. A Figura 2.11(b) procura esboçar esta ideia.

A Figura 2.11(c) mostra uma representação do modelo de aceleração de movimento (*move acceleration model*). Neste paradigma, a qualidade de cada movimento é computada paralelamente, com o objetivo de acelerar sua avaliação. Este modelo é particularmente interessante quando a função de avaliação requer um maior esforço computacional ou realiza intensivamente operações de entrada/saída.

2.5 Motivação e Trabalhos Relacionados

Conforme discutido previamente, a principal fonte de paralelismo na implementação de meta-heurísticas encontra-se na execução concorrente dos laços mais internos do algoritmo. No modelo de meta-heurística baseada em população, a principal fonte é tipicamente determinada pela avaliação de soluções. Já para o paradigma orientado em trajetó-

[11] modelo paralelo mestre-trabalhador: uma tarefa mestre é responsável por dividir e distribuir a carga de processamento entre tarefas trabalhadoras, que enviam o resultado de sua execução de volta para a tarefa mestre.

ria, as buscas local e em vizinhança representam os componentes com maior potencial de paralelização. Este trabalho é focado no estudo de metodologias e análise de desempenho da implementação de buscas local e em vizinhança em ambientes híbridos compostos por CPU e GPU. A principal motivação para tal estudo é que esses componentes tipicamente concentram o esforço computacional de muitas meta-heurísticas orientadas em trajetória, bem de alguns métodos evolutivos, como no caso dos Algoritmos Meméticos [Moscato *et al.*, 1989].

Embora muitas meta-heurísticas orientadas para CPU tornem possível a obtenção de soluções de qualidade sub-ótima a um tempo computacional aceitável, elas usualmente falham quando a abordagem envolve problemas de maior cardinalidade [Van Luong *et al.*, 2013]. Neste contexto, plataformas híbridas envolvendo CPUs e GPUs podem oferecer os recursos adicionais necessários para ampliação da capacidade computacional de sistemas paralelos a um custo/benefício conveniente. Há quase uma década, projetos de algoritmos baseados em GPU tem sido utilizados na computação científica. Podem ser encontradas na literatura diversas aplicações em álgebra linear, visualização, geometria e simulações [Brodtkorb *et al.*, 2010]. Um dos trabalhos pioneiros foi proposto em [Janiak *et al.*, 2008], que implementou algoritmos de Busca Tabu para dois problemas de otimização por meio de seus respectivos mapeamentos em aplicações gráficas (ver Subseção 2.2.1). Com o desenvolvimento de CUDA, a implementação de aplicações de propósito geral tornou-se menos penosa incentivando o surgimento de novas aplicações. Em [Vidal & Alba, 2010] e [Melab *et al.*, 2010], os autores desenvolveram e avaliaram o desempenho de Algoritmos Evolutivos nesta plataforma, demonstrando que implementações orientadas para GPU podem apresentar desempenho superior quando comparadas ao equivalente em CPU. Aplicações para as duas fases de um algoritmo baseado em Colônia de Formigas foram examinadas em [Cecilia *et al.*, 2011] e uma proposta baseada nessa mesma meta-heurística para o Problema de Roteamento de Veículos (PRV) foi discutida em [Diego *et al.*, 2012]. Referências sobre outros trabalhos envolvendo heurísticas para problemas de otimização combinatória orientados para GPU podem ser encontradas na seção correspondente do *survey* publicado por [Alba *et al.*, 2013].

No entanto, a pesquisa envolvendo implementações de problemas de otimização combinatória para plataformas híbridas envolvendo CPU e GPU ainda não é abundante. Um algoritmo mestre-escravo inspirado em Colônia de Formigas para um problema de roteamento (*Orienteering Problem*) foi proposto em [Catala *et al.*, 2007]. Em [Zhu & Curry, 2009], os autores buscam explorar como esta meta-heurística pode ser potencialmente acelerada numa plataforma baseada em GPU. Em um trabalho mais recente, [Coelho *et al.*,

2016] desenvolveram um método baseado em VND que explora mecanismos de busca em vizinhança orientados para GPU.

Um fator comum aos métodos citados previamente é que estes procuram aplicar diretamente na plataforma GPU metodologias tradicionais usadas no projeto de meta-heurísticas para sistemas CPU. Na verdade, a literatura acerca de estudos que busquem reavaliar tais metodologias de implementação para a plataforma híbrida CPU/GPU ainda é escassa. Um trabalho seminal nesta direção foi realizado por [Brodtkorb *et al.*, 2013], onde os autores apresentam considerações sobre a arquitetura GPU e traçam orientações para o desenvolvimento eficiente de aplicações nesta plataforma. Em um trabalho mais recente, [Schulz, 2013] realiza um estudo para investigar a aplicação efetiva de alguns procedimentos tradicionais de busca em vizinhança para o PRV. Em seu trabalho, uma implementação básica para o PRV é analisada e sistematicamente ajustada com o objetivo de alcançar maior desempenho. Em outro trabalho, [Van Luong *et al.*, 2013] apresenta diretrizes para o desenho e implementação de meta-heurísticas de busca local, investigando a cooperação eficaz entre CPU e GPU e o controle de execução de tarefas para atender às restrições de acesso à memória dessa plataforma. Fica evidente que estes dois últimos trabalhos concentram parte de seus esforços no projeto de um componente relevante para o desempenho de seus respectivos algoritmos: a busca local.

Como discutido previamente, a pesquisa em torno da reavaliação de metodologias tradicionais para implementação em ambientes híbridos compostos por CPU/GPU ainda é escassa sendo, portanto, um fator motivador para a investigação neste campo. No próximo capítulo, serão apresentadas e sistematicamente avaliadas algumas configurações de busca local envolvendo ambiente CPU/GPU, ocasião em que diversos aspectos relacionados ao desempenho nesses sistemas serão destacados, aferidos e ponderados.

Capítulo 3

Estratégias de Busca Local para Arquiteturas CPU/GPU

A busca local é um componente fundamental para muitas meta-heurísticas. De fato, diversos trabalhos têm sido conduzidos no sentido de estudar esse importante componente para os métodos de optimização combinatória [Johnson *et al.*, 1988, Stuart & Podolny, 1996, Lenstra, 1997, Hoos & Stützle, 2004]. Com o advento de novas tecnologias de *hardware*, modernos dispositivos com recursos para potencializar esses métodos se tornaram disponíveis, como as plataformas híbridas CPU/GPU. Essa plataforma ganhou evidência na última década impulsionada, tanto pelos constantes incrementos tecnológicos agregados, quanto pela relação custo/benefício proporcionada pela expectativa de maior poder computacional.

Não é raro encontrar trabalhos para esta plataforma que anunciam ganhos de performance notáveis. Contudo, alguns estudos ressaltam que a comparação entre algoritmos baseados em sistemas CPU/GPU precisa ser conduzida com cautela [Gregg & Hazelwood, 2011, Lee *et al.*, 2010], ou que muitas abordagens ainda não exploram ou consideram adequadamente os recursos disponíveis [Brodtkorb *et al.*, 2013, Schulz, 2013, Van Luong *et al.*, 2013]. Este capítulo procura abordar esses dois aspectos inerentes ao projeto de algoritmos de busca local para plataformas CPU/GPU. Neste contexto, são resgatados conceitos elementares de busca local e os aspectos relevantes para sua paralelização nesta plataforma. Na sequência, uma metodologia de aceleração de busca bastante eficiente é posta em evidência, e aspectos inerentes à sua implementação em sistemas CPU/GPU são analisados. Finalmente, experimentos computacionais envolvendo diversos cenários são conduzidos, com a finalidade aferir seus respectivos desempenhos e eleger uma metodologia de implementação adequada.

3.1 Busca Local e Busca em Vizinhança

A busca local é provavelmente um dos métodos heurísticos mais antigos e elementares [Lenstra, 1997]. Informalmente, consiste na realização de mudanças a partir de uma solução inicial objetivando incrementar sistematicamente sua qualidade por meio da seleção de soluções candidatas de uma vizinhança. A vizinhança de uma solução s, denotada por $N(s)$, é dada pelo conjunto de soluções obtidas mediante a aplicação de um operador de *movimento* sobre s. Cada movimento produz uma solução vizinha efetuando uma pequena alteração na *solução base* s, de acordo com um critério específico [Talbi, 2009].

O Algoritmo 1 descreve o procedimento de uma busca local. Dada uma solução inicial s e uma estrutura de vizinhança N, o método inicia executando um procedimento de pré-processamento específico para inicializar eventuais estruturas de dados locais. Em seguida, uma função de seleção é executada resultando em uma operação de movimento que é posteriormente aplicada sobre a solução base s gerando uma nova solução s'. Na sequência, uma função objetivo f avalia a qualidade da nova solução s' em relação à solução corrente s. Caso uma melhora de qualidade tenha sido obtida, a solução corrente é substituída e eventuais atualizações nas estruturas de dados locais são realizadas. Finalmente, quando a qualidade da solução corrente não mais puder ser incrementada, o método encerra retornando o melhor resultado encontrado.

Algoritmo 1: Pseudocódigo do procedimento de busca local.

```
1  algoritmo busca_local(s, N)
2      Inicializa estruturas de dados locais;
3      repita
4          move ← seleciona_movimento(s, N);
5          s' ← aplica_movimento(s, move);
6          se f(s') < f(s) então
7              s ← s';
8              Atualiza estruturas de dados locais;
9          fim
10     até f(s') ≥ f(s);
11     retorna s;
12 fim
```

A avaliação de soluções realizada pelo Algoritmo 1 (linhas 6 e 10) considera que o problema de otimização em questão é de minimização, já que busca por soluções que produzam valores decrescentes da função objetivo. Contudo, sem perda de generalidade, o mesmo algoritmo pode ser adaptado para problemas de maximização utilizando-se os operadores relacionais adequados.

Vizinhanças são dependentes do problema de otimização alvo, e encontrar operadores de movimento que conduzam a ótimos locais de alta qualidade pode ser considerado um dos desafios da busca local [Lenstra, 1997]. Além disso, a geração e avaliação de soluções vizinhas pode facilmente dominar a complexidade computacional no contexto do método. Isto porque, dependendo do problema e do operador de movimento utilizado, não é raro que a geração de tais estruturas atinjam cardinalidades expressivas ou que a avaliação das operações de movimento revele-se igualmente complexa. Por esses motivos, tais procedimentos são potenciais candidatos à paralelização quando o objetivo é acelerar a busca local. No contexto deste trabalho, as operações de geração e avaliação de movimentos são referenciadas como *busca em vizinhança*.

A busca paralela em vizinhança envolve uma metodologia mestre-trabalhador baseada em tarefas independentes que atuam sobre diferentes segmentos de dados da aplicação. A Figura 3.1 procura ilustrar esse processo, onde tarefas paralelas são encarregadas pela geração e/ou avaliação de subconjuntos de movimentos a partir de uma mesma solução base. O resultado da busca em vizinhança depende da consolidação dos resultados das tarefas trabalhadoras, onde uma solução vizinha é obtida de acordo com uma estratégia de seleção específica.

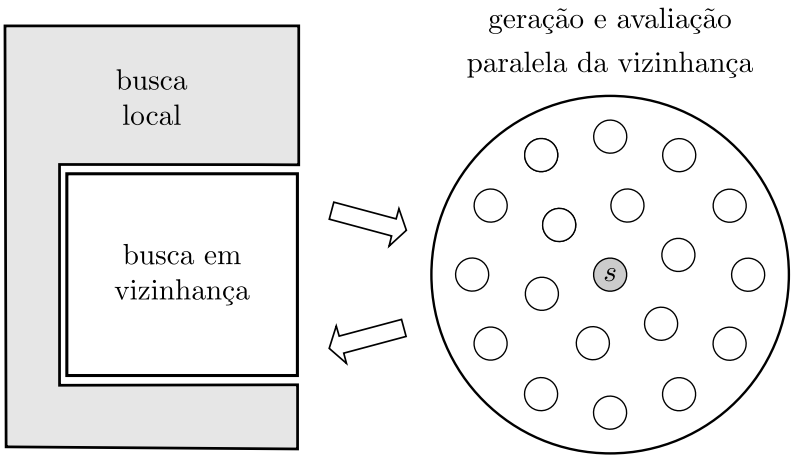


Figura 3.1: Geração e avaliação paralela da vizinhança.

Tal metodologia adapta-se muito bem à arquitetura GPU, que é altamente eficiente na execução de tarefas sincronizadas e que envolvem computação e transferências regulares de dados. Contudo, convém ressaltar que em sistemas de memória distribuída, a performance global das meta-heurísticas geralmente é limitada pela latência envolvendo transferências de dados [Crainic & Gendreau, 2010]. Mesmo com as altas taxas de transfe-

rência[1] suportadas por esses sistemas híbridos, a comunicação de dados entre CPU e GPU ainda representa um ponto crítico para o desempenho da aplicação e deve ser planejada com critério.

Durante a execução paralela da busca em vizinhança em plataformas CPU/GPU, as transferências de dados envolvem, tipicamente, soluções a serem geradas e/ou avaliadas. Neste contexto, duas abordagens principais podem ser encontradas na literatura [Van Luong *et al.*, 2013]:

- *Geração na CPU e avaliação na GPU*: a cada iteração do processo de busca, a vizinhança é gerada na CPU e posteriormente transferida para a GPU para avaliação. Nesse caso, as estruturas de dados associadas à vizinhança são transferidas da memória da CPU para a GPU e a avaliação da vizinhança é atribuída a uma grade de *threads* GPU;

- *Geração e avaliação na GPU*: este processo envolve apenas a transferência de dados necessários para a geração e avaliação da vizinhança na GPU. Nesse cenário, cada *thread* GPU fica responsável pela produção e avaliação de um ou mais elementos da vizinhança.

Fica evidente que a primeira abordagem tende a envolver um maior volume de transferência de dados, uma vez que as estruturas de vizinhança geralmente apresentam alta cardinalidade. Considerando que a comunicação é um requisito crítico para os métodos distribuídos, isso pode comprometer sensivelmente o desempenho da aplicação. Já a segunda técnica, realiza um menor volume de cópia de dados, atribuindo para a GPU a tarefa de geração e avaliação da vizinhança tirando proveito de sua arquitetura massivamente paralela.

Uma técnica recente para reduzir o esforço computacional da busca local envolve o pré-processamento de estruturas de dados locais antes do início efetivo da busca em vizinhança. A ideia por trás deste método é construir estruturas de dados de permitam a redução da complexidade computacional da busca, acelerando o processo de avaliação de soluções vizinhas. Para a classe de heurísticas da área de roteamento de veículos, foco dos estudos de caso dessa tese, alguns dos métodos mais bem sucedidos utilizam ideias similares [Kindervater & Savelsbergh, 1997, Ngueveu *et al.*, 2010, Vidal *et al.*, 2011, Silva *et al.*, 2012, Subramanian *et al.*, 2013, Penna *et al.*, 2013]. No entanto, tal pré-processamento traz

[1] A título de referência, uma GPU NVIDIA GTX-780, por exemplo, suporta uma taxa de transferência nominal de 288,4 Gigabytes/s.

embutido um custo computacional adicional que influencia diretamente no desempenho global da busca. Uma questão que emerge nesse contexto é se a arquitetura massivamente paralela das GPUs pode se beneficiar ao empregar essa abordagem. Para responder a essa questão, este capítulo apresenta um estudo de desempenho envolvendo seis diferentes cenários para um problema de roteamento de veículos.

3.2 O Problema da Mínima Latência

O Problema de Roteamento de Veículos (PRV) é uma família de problemas de otimização combinatória que tem atraído um grande interesse de pesquisadores nas últimas décadas, principalmente pela importância econômica de suas aplicações em áreas como transporte, telecomunicação a planejamento de produção [Toth & Vigo, 2001]. Muitas variantes do PRV surgem de acordo com o conjunto de restrições aplicadas para o problema. Um dos problemas mais estudados desta família é o Problema do Caixeiro Viajante (PCV), que consiste em encontrar uma rota que minimize o tempo de viagem entre um conjunto de clientes [Applegate *et al.*, 2007].

O Problema da Mínima Latência (PML) é uma variante do PCV clássico cujo objetivo é minimizar o tempo de chegada aos clientes, em vez do tempo de viagem como ocorre no PCV. O PML pode ser definido como um grafo direcionado $G = (V, E)$, onde $V = \{0, 1, \ldots, v\}$ é um conjunto de $v + 1$ vértices representando os clientes e $E = \{(i, j) : i, j \in V, i \neq j\}$ um conjunto de arcos que conectam esses clientes. Cada arco (i, j) possui um tempo de viagem ou distância associado dado por $d(i, j)$. O vértice 0 representa o ponto de partida (ou depósito) e os vértices remanescentes, os demais clientes a serem visitados. Uma solução para o PML é uma permutação dos vértices de V, que inicia e termina no depósito consistindo, portanto, em um circuito com $n = v + 1$ vértices. Para uma dada solução s, a expressão $d_s(i, j) = d(s_i, s_j)$ denota o tempo de viagem do clientes s_i para o cliente s_j. A latência de um vértice s_i é definida como o tempo de viagem acumulado entre o depósito e o cliente s_i, e pode ser obtida pela expressão $l_s(i) = \sum_{k=0}^{i-1} d_s(k, k + 1)$. O objetivo do PML é, iniciando do depósito, determinar o circuito Hamiltoniano (*tour*) que minimiza a latência total, expressa por $f(s) = \sum_{i=0}^{n-1} l_s(i)$ e que representa o custo da solução. Alguns estudos não consideram o retorno ao depósito como uma restrição do problema. Neste caso, a solução consiste em um caminho (*path*), indicando que $l_s(n - 1) = 0$.

Apesar de sua formulação simples, o PML é um problema complexo. De fato, foi

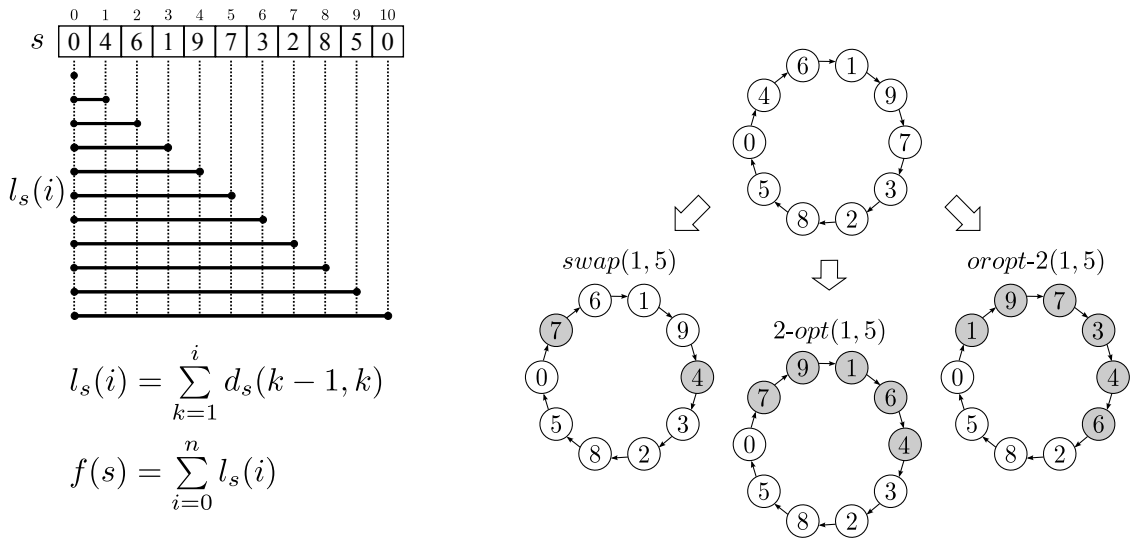
provado que este problema pertence à classe NP-Difícil quando os vértices residem em espaços métricos gerais ou em planos Euclidianos [Sahni & Gonzalez, 1976] e também quando o espaço considerado é induzido por uma árvore ponderada [Sitters, 2006]. Além disso, a despeito de sua similaridade com PCV, este problema possui a reputação de ter resolução computacionalmente mais difícil que o primeiro. Como ressaltado por alguns pesquisadores, do ponto de vista computacional o PML apresenta características que o distinguem do PCV. Uma delas é que pequenas alterações locais na configuração dos dados de entrada podem conduzir a mudanças globais significativas na estrutura da solução final [Minieka, 1989]. Outra característica marcante é a natureza não local da função objetivo, onde a simples inserção ou alteração de posição de um arco da rota afeta a latência dos vértices subsequentes, como ressaltado por [Arora & Karakostas, 2003].

Aplicações do PML podem ser encontradas em diversos contextos. Em sistemas de distribuição por exemplo, o problema pode ser aplicado para priorizar a satisfação do usuário em detrimento do custo de deslocamento, já que avalia o sistema do ponto de vista do cliente. Em outras palavras, ao invés de considerar o tempo de deslocamento como o PCV, o PML busca minimizar o tempo de espera do cliente [Archer *et al.*, 2008]. No contexto de problemas de escalonamento de tarefas em máquinas de um ambiente de produção, o PML está relacionado com o problema de minimizar o tempo de finalização das tarefas alocadas em uma única máquina, como abordado por [Angel-Bello *et al.*, 2013]. Além dessas, outras aplicações do problema podem ser encontradas na recuperação de dados em dispositivos de armazenamento em disco [Ezzine *et al.*, 2010], no roteamento de veículos automaticamente guiados [Simchi-Levi & Berman, 1991] e no processamento de busca de informações em redes de computadores [Ausiello *et al.*, 2000].

O PML (do inglês, *Minimum Latency Problem*) também é referenciado na literatura como *Travelling Repairman Problem* [Tsitsiklis, 1992], *Delivery Man Problem* [Fischetti *et al.*, 1993, Mladenović *et al.*, 2013], *Cumulative Traveling Salesman Problem* [Bianco *et al.*, 1993] e *School Bus Driver Problem* [Chaudhuri *et al.*, 2003].

3.3 Solução e Vizinhanças para o PML

Uma solução para o PML pode ser implementada como um vetor de valores discretos representando uma rota para os clientes a serem visitados. A Figura 3.2(a) ilustra uma solução s, com $n = 11$ elementos, e as respectivas latências $l_s(i)$ associadas às visitas a cada cliente s_i $(0 \leq i < n)$.



(a) Uma solução para o PML (b) Operações de movimento

Figura 3.2: Exemplos de solução para o PML e dos operadores de movimento *swap*, 2*opt* e *oropt-k* ($k = 2$).

Neste trabalho foram consideradas cinco estruturas de vizinhança clássicas para o PCV que podem ser facilmente adaptadas para o PML. A reprodução desas vizinhanças depende da aplicação dos seguintes operadores de movimento:

- *swap*(i, j): os clientes nas posições i e j da solução são permutados;
- 2*opt*(i, j): a rota delimitada pelos clientes de índices i e j é invertida;
- *oropt-k*(i, j): A partir do índice i, k clientes adjacentes são realocados para a posição j. Foram implementadas três variações desse operador, para valores de $k \in \{ 1, 2, 3 \}$.

A Figura 3.2(b) apresenta exemplos de uma única operação de movimento para cada um dos operadores de vizinhança descritos. A partir de uma solução base s, cada operação resulta numa solução vizinha correspondente, onde os elementos destacados indicam os clientes da rota que foram diretamente afetados. Todos os operadores de movimento são descritos em função de dois operandos i e j, indicando a posição (índice) dos clientes na solução de base s. Para o operador *oropt-k*, apenas o caso quando $k = 2$ é descrito, porém as outras duas variantes, para $k = 1$ e $k = 3$, podem ser facilmente deduzidas.

3.4 Estruturas de Dados Locais

No contexto desse trabalho, as Estruturas de Dados Locais (EDL) fazem referência a um conjunto de dados associado a uma solução para o PML. Cada EDL é vinculada a uma solução específica, sendo gerada com o objetivo de acelerar a busca em vizinhança associada a essa solução.

A busca em vizinhanças de soluções para o PML possui um fator complicador introduzido pela função de avaliação, que é bastante sensível mesmo a pequenas alterações na rota. Observando a Figura 3.2, é possível verificar que cada operação de movimento afeta um número variável de clientes da solução, além de influenciar a latência relativa aos clientes subsequentes. Por exemplo, tomando a operação $2opt(1, 5)$ ilustrada na figura, verifica-se que cinco clientes foram afetados $(4, 6, 1, 9, 7)$, tornando necessário o cálculo de suas respectivas latências. Contudo, as latências relativas aos clientes posteriores $(3, 2, 8, 5, 0)$ também são afetadas, incrementando o número de operações necessárias para calcular do custo da solução. Portanto, a avaliação de um único movimento de uma vizinhança do PML tipicamente envolve a iteração sobre uma larga proporção de clientes da rota, conduzindo a uma quantidade de operações da ordem de $O(n)$, onde n representa o número de elementos da solução. Considerando que a quantidade de movimentos de cada vizinhança é dada pelo arranjo de n elementos em função dos índices i e j do operador correspondente, obtém-se uma grandeza da ordem de $O(n^2)$. Consequentemente, a busca em cada uma das cinco estruturas de vizinhanças em questão demanda um número de avaliações da ordem de $O(n^3)$ [Vidal *et al.*, 2013], corroborando a tendência de complexidade e crescimento exponenciais das vizinhanças.

Uma vez que a avaliação de movimentos é uma operação crucial para o desempenho da busca em vizinhança, uma metodologia que contribua para sua aceleração deve ser considerada, principalmente se a execução do método de resolução do problema envolve instâncias de médio e grande porte. Neste sentido, uma metodologia proposta por [Silva *et al.*, 2012] permite que cada operação de movimento possa ser avaliada em tempo constante. Essa abordagem considera que qualquer movimento pode ser expresso como uma sequência de segmentos (sub-rotas) da solução que podem ser reorganizados e concatenados formando a solução vizinha correspondente. A Figura 3.3 procura ilustrar essa ideia, mostrando como as operações *swap*, 2*opt* e *oropt*-2 podem ser desmembradas em segmentos e reorganizadas formando as respectivas soluções resultantes.

O método introduzido por [Silva *et al.*, 2012] depende do cômputo de uma EDL

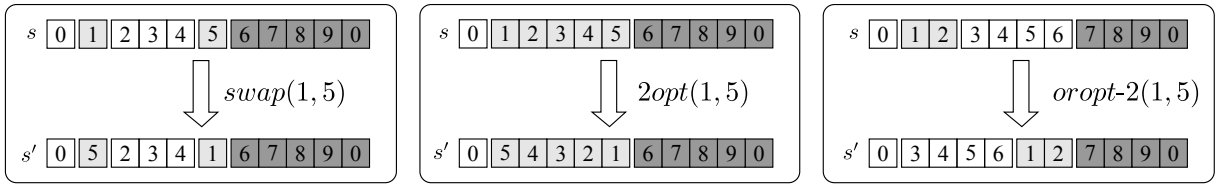


Figura 3.3: Exemplos de execução das operações de movimento por meio da concatenação de segmentos.

constituída por quatro componentes principais, aqui denominados de D, W, T e C. O objetivo é tornar disponível em tempo de execução toda a informação necessária para avaliar o custo de qualquer solução resultante de um movimento para a vizinhança de uma determinada solução s. Esses componentes são definidos pelas seguintes expressões:

$$D(i,j) = d(s_i, s_j) \tag{3.1}$$

$$W(i,j) = \begin{cases} 0 & ,i = 0 \text{ e } j = 0 \\ 1 & ,i = j \\ |i-j|+1 & ,\text{c.c.} \end{cases} \tag{3.2}$$

$$T(i,j) = \begin{cases} 0 & ,i = j \\ t_{i,j-1} + d_s(i,j) & ,i < j \\ t_{i,j+1} + d_s(j,i) & ,i > j \end{cases} \tag{3.3}$$

$$C(i,j) = \begin{cases} 0 & ,i = j \\ c_{i,j-1} + d_s(i,j) & ,i < j \\ c_{i,j+1} + d_s(j,i) & ,i > j \end{cases} \tag{3.4}$$

onde,

- D: tempo de viagem entre os clientes de índices i e j da solução. Estes dados são obtidos a partir da instância de entrada do PML;
- W: número de partidas oriundas do depósito necessárias para visitar todos os clientes pertencentes a uma sub-rota[2];
- T: tempo de viagem acumulado entre dois clientes da solução;
- C: custo da viagem entre dois clientes da solução.

Uma característica interessante desta metodologia, é que o custo de cada movimento pode ser calculado sem a necessidade de construir efetivamente a solução vizinha

[2]Por exemplo, considerando a Figura 3.2(a), para visitar a sub-rota entre os índices 3 e 6, inclusive, 4 partidas do depósito são necessárias.

correspondente. Isto porque, a partir da EDL, o custo de um movimento pode ser obtido pela indução da concatenação de segmentos, definida a seguir. Sejam $\sigma_1 = (i_1, j_1)$ e $\sigma_2 = (i_2, j_2)$, dois segmentos (sub-rotas) de uma solução s para o PML. O custo da concatenação de σ_1 e σ_2, definida como $\sigma_1 \oplus \sigma_2$, é dado por [Silva *et al.*, 2012]:

$$C(\sigma_1 \oplus \sigma_2) = C(i_1, j_1) + C(i_2, j_2) + W(i_2, j_2) \cdot [T(i_1, j_1) + D(j_1, i_2)] \quad (3.5)$$

$$T(\sigma_1 \oplus \sigma_2) = T(i_1, j_1) + T(i_2, j_2) + D(j_1, i_2) \quad (3.6)$$

$$W(\sigma_1 \oplus \sigma_2) = W(i_1, j_1) + W(i_2, j_2) \quad (3.7)$$

Uma abordagem típica para implementação dessas estruturas seria a utilização de quatro matrizes $n \times n$. Esse método pode ser bastante eficiente para a arquitetura CPU, onde a memória *cache* usualmente possui capacidade suficiente para acomodar e acelerar o grande número de acessos aleatórios realizado a essas estruturas durante a busca. Na plataforma GPU, a pequena quantidade de memória de acesso rápido disponível (memória compartilhada) é um obstáculo para o uso eficiente dessas estruturas, especialmente quando envolvem instâncias de médio e grande porte. Além disso, sempre que a solução base da vizinhança muda, a EDL correspondente precisa ser atualizada. Considerando que tal estrutura pode ocupar um volume considerável de memória ou envolver computação intensiva, a questão de sua atualização ser executada na CPU ou GPU precisa ser avaliada.

Num primeiro cenário, a atualização da EDL na CPU parece promissora, já que a larga memória *cache* típica das CPUs modernas se comporta bem com estruturas onde acesso aleatório intenso é um requisito. No entanto, caso a avaliação da vizinhança seja realizada na GPU, a EDL correspondente precisa ser transferida para a memória desse dispositivo. Isso pode ocasionar um *overhead* que se concentra no ponto crítico dos métodos de memória distribuída: a transferência de dados. Num segundo cenário, onde a atualização da EDL seria realizada na GPU, apenas os dados referentes à solução base precisam ser transferidos. Porém, é preciso ponderar que o cômputo dos elementos de alguns componentes da EDL (D, T e C) dependem de resultados anteriores, o que compromete o nível de paralelismo que pode ser alcançado numa arquitetura SIMD como a da GPU.

Na implementação do método proposto por [Silva *et al.*, 2012] realizada para este estudo, a estrutura da EDL foi reavaliada considerando três aspectos principais: o volume de memória consumido; a adequação aos níveis de memória mais rápidos da GPU; e a exploração racional de recursos otimizados desse dispositivo, como a computação

de operações matemáticas [Kirk & Wen-mei, 2012]. Esse processo resultou na seguinte reestruturação da EDL para uma solução s com n clientes:

$$R(i) = \begin{cases} 0 & , i = 0 \\ d(s_{i-1}, s_i) & , \text{c.c.} \end{cases} \tag{3.8}$$

$$X(i) = (x_i, y_i) \tag{3.9}$$

$$L(i) = T(0, i) \tag{3.10}$$

Os três primeiros componentes da nova estrutura da EDL são definidos pelas Expressões (3.8) a (3.10). O componente R representa o tempo de viagem entre clientes adjacentes na solução base da vizinhança a ser avaliada. A estrutura X contém as coordenadas Euclidianas (x_i, y_i) de cada cliente i, obtidas da instância de entrada. Essas coordenadas são utilizadas para calcular, em tempo de execução, as distâncias necessárias durante as operações de concatenação de segmentos. O componente L é equivalente à primeira linha da matriz T, dada pela Expressão (3.3), representando o tempo acumulado de viagem do depósito até cada cliente da solução. Usando apenas esta informação, é possível obter qualquer elemento de T, computando $T(i, j) = |\, L(j) - L(i)\,|$. O quarto componente, C, foi mantido como na abordagem original, dado pela Expressão (3.4). Por sua vez, o componente W, definido pela Expressão (3.2), foi excluído da nova EDL. Isto foi possível porque o valor da cada elemento desse componente pode ser calculado em tempo de execução pela expressão $W(i, j) = |i - j| + (i > 0)$. Essa notação considera que o resultado da operação de desigualdade estrita $(i > 0)$ é avaliado como 0 ou 1 representando, respectivamente, os valores lógicos *falso* e *verdadeiro*.

Com este novo arranjo, a cardinalidade da EDL foi reduzida para $n^2 + 3n$ elementos, representando um decréscimo de aproximadamente 75% em relação aos $4n^2$ elementos da estrutura anterior. Além disso, a representação vetorial torna essas estruturas mais adequadas para armazenamento na memória local dos multiprocessadores da GPU, significando menor latência durante a carga a partir da memória global e acesso mais rápido por meio das *threads* GPU.

A seguir, são considerados diferentes cenários envolvendo o uso de EDL para a aceleração da busca em vizinhança.

3.5 Estratégias de Busca Local CPU/GPU

Nesta seção, são apresentadas e avaliadas seis diferentes estratégias de busca local orientadas para plataforma CPU/GPU, todas fundamentadas no Algoritmo 1. Duas estratégias utilizam apenas os recursos da CPU, replicando os melhores algoritmos da literatura para o Problema da Mínima Latência. Outras duas estratégias processam os dados inteiramente na GPU e duas abordagens finais consideram o uso de ambos os recursos da plataforma híbrida.

Com o objetivo de considerar os possíveis cenários de implementação, a busca local foi desmembrada em três diferentes componentes: *controle de fluxo*, *atualização da* EDL e *avaliação da vizinhança*, abreviados como F, A e V, respectivamente. Os diferentes cenários de alocação envolvendo esses componentes estão esboçados na Figura 3.4.

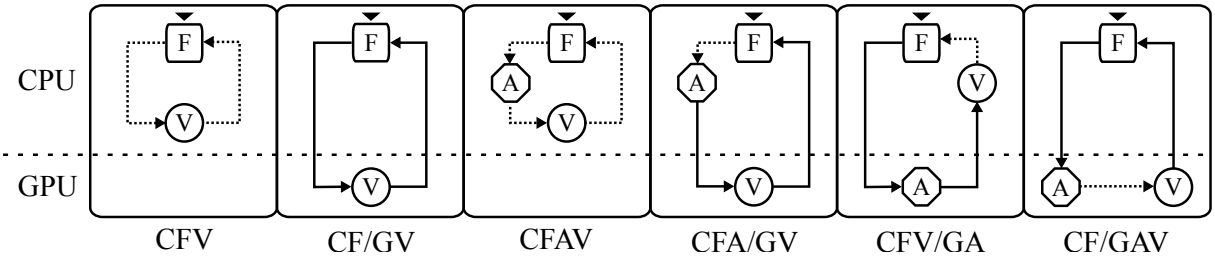


Figura 3.4: Estratégias para a implementação de busca em vizinhança em plataforma CPU/GPU.

Os diagramas da figura ilustram cada uma das seis estratégias experimentadas, indicando o fluxo de execução e transferência de dados entre CPU e GPU. As linhas sólidas indicam a ocorrência de uma operação de transferência de dados (cópia) entre as memórias da CPU e GPU. As linhas tracejadas indicam que nenhuma movimentação de dados entre as plataformas foi necessária. Os acrônimos associados a cada diagrama empregam a seguinte simbologia: o primeiro caractere indica a arquitetura onde os componentes foram alocados, adotando C e G para indicar CPU e GPU, respectivamente. Os demais caracteres indicam quais componentes (F, A e V) foram executados na arquitetura correspondente ao primeiro caractere da palavra. A presença de uma barra (/) indica que ambas as arquiteturas foram utilizadas, quando a convenção aqui definida pode ser aplicada recursivamente.

A execução do algoritmo sempre inicia a partir da CPU por meio do componente de controle de fluxo (F), que determina qual estratégia de busca em vizinhança será aplicada. Se uma estratégia que envolva pré-processamento é adotada, o próximo passo a

ser realizado é a atualização da EDL (A). Em seguida, a busca em vizinhança (V) realiza a avaliação do custo de cada operação de movimento, retornando ao componente F as informações necessárias para a seleção do melhor vizinho. Embora o componente F seja sempre atribuído para a CPU, os componentes A e V podem ser alocados em ambas as plataformas.

Como ressaltado na seção anterior, a EDL consiste em uma estrutura de dados cuja atualização envolve um custo computacional da ordem de $O(n^2)$ mas que, contudo, reduz a complexidade da busca em vizinhança para esta mesma magnitude. Apesar do *overhead* adicional, a atualização da EDL ocorre apenas quando a melhor solução do método é alterada. Portanto, esta operação é executada cada vez menos à medida que a qualidade da solução melhora ao longo da busca, o que promove a aceleração do algoritmo. Desta forma, as estratégias CFAV e CFV representam, respectivamente, a busca local realizada na CPU com e sem o uso de EDL e são utilizadas como referência na comparação de desempenho nos experimentos realizados.

As demais estratégias testadas — CF/GV, CFA/GV, CFV/GA e CF/GAV — consistem na computação heterogênea envolvendo CPU e GPU. Cada algoritmo inicia sua execução pelo controle de fluxo (F) na CPU, mas os componentes A e V são sistematicamente alocados na CPU e/ou GPU, incluindo as eventuais transferências de dados interarquitetura. A estratégia CF/GV não adota EDL no algoritmo, mas simplesmente copia a solução corrente para a GPU, onde a busca em vizinhança é executada e de onde retorna um conjunto de melhores movimentos para seleção do melhor vizinho. Isto ocorre porque uma redução parcial dos movimentos é realizada na GPU, conforme será detalhado adiante. O esquema CFA/GV é uma extensão de CF/GV, onde o componente A realiza a atualização a EDL na CPU e a transfere para a memória da GPU, onde a avaliação da vizinhança é executada. Usando esta estratégia, o montante de trabalho de cada *thread* GPU é reduzido, porém é introduzido um *overhead* associado à transferência de memória. A estratégia CF/GAV atualiza a EDL usando a GPU, o que evita a transferência de memória extra associada. No entanto, a forma de cálculo da EDL limita o nível de paralelismo que pode ser obtido na GPU, pois apresenta muitas dependências de dados durante sua atualização. Finalmente, o esquema CFV/GA realiza a busca em vizinhança na CPU depois de atualizar a EDL na GPU e receber os movimentos resultantes para seleção do melhor vizinho.

Na próxima seção é apresentada uma série de experimentos envolvendo as seis estratégias previamente descritas. Cada uma das buscas em vizinhança para o PML

descritas são analisadas e executadas no contexto de cada estratégia. Adicionalmente, é realizada uma análise de desempenho do algoritmo de atualização da EDL na GPU com o objetivo de determinar o limite teórico de sua aceleração neste dispositivo. Na oportunidade, detalhes relacionados ao desempenho na arquitetura GPU são destacados e ponderados.

3.6 Experimentos Computacionais

A principal finalidade dos experimentos apresentados a seguir é procurar determinar quais das seis estratégias apresentadas são mais adequadas para a implementação de *kernels* para as buscas em vizinhança consideradas. Outro propósito é determinar se o uso de EDL — com seu efetivo sucesso em métodos sequenciais — poderá contribuir para acelerar a busca paralela ou será ofuscado pelos *overheads* associados à sua atualização e transferências entre CPU e GPU.

Os algoritmos utilizados nos experimentos desta seção foram implementados em C++ utilizando a biblioteca NVIDIA® CUDA™ versão 7.0 e compilados com o NVCC (NVIDIA *C++ Compiler*) para uma plataforma de 64-bits executando Linux Ubuntu 14.04. Para uma mensuração mais precisa, os tempos de execução na CPU foram coletados por meio da biblioteca PAPI 5.4 [Mucci *et al.*, 1999]. Os experimentos foram conduzidos em dois computadores com processador Intel® i7-4820K 3.70GHz, com 4 CPU *cores*, 10MB de *cache* L3 e 16GB memória RAM. As duas máquinas possuem arquitetura CPU idênticas, porém cada uma está equipada com diferentes GPUs, modelos NVIDIA® GeForce™ GTX-550 e GTX-780, cujas especificações estão descritas na Tabela 3.1.

Modelo	**Micro-arquitetura**	**Frequência operação**	**Total de SMs**	***Cores* por SM**	**Total de *Cores***	**Memória**	
						Compart.	**Global**
GTX-550	Fermi	1,80GHz	4	48	192	48KB	1GB
GTX-780	Kepler	0,99GHz	12	192	2.304	48KB	3GB

Tabela 3.1: Especificações das GPUs NVIDIA GeForce GTX-580 e GTX-780.

A execução dos algoritmos foi monitorada com auxílio do *software* NDIVIA® Visual Profiler, uma ferramenta de análise que fornece diversas métricas de desempenho relacionadas com a execução de *kernels* e transferências de dados. Os dados fornecidos por esta ferramenta foram utilizados para analisar e ajustar os algoritmos visando maximizar seus respectivos desempenhos.

As instâncias do PML utilizadas nos ensaios foram extraídas de diversas fontes da

literatura[3] e estão compiladas no trabalho de [Silva *et al.*, 2012]. Desse montante, foram selecionadas sete instâncias usando como critério o número de clientes de cada uma. Essa amostra foi realizada de forma a intercalar o tamanho da instância no intervalo de 100 e 1.000 clientes. Convém ressaltar que os experimentos realizados nesta seção não consideram a qualidade da solução como critério de avaliação, uma vez que estão focados no desempenho computacional da busca na plataforma alvo, o que torna a identificação das instâncias irrelevante.

Dos três componentes desmembrados da busca local — *controle de fluxo*, *atualização da* EDL e *avaliação da vizinhança* — os dois últimos tiveram sua implementação realizada para ambas as plataformas CPU e GPU. O controle de fluxo, por acomodar o núcleo da busca local (Algoritmo 1) e ser o responsável pelo lançamento de *kernels* e transferências de dados, foi codificado apenas para CPU. A Tabela 3.2 sintetiza os componentes da busca indicando suas respectivas plataformas suportadas, nomes dos *kernels* associados e finalidades no contexto do algoritmo.

Componente	**Arquitetura suportada**	**Nome do *Kernel***	**Finalidade**
(F) Controle de fluxo	CPU	—	Busca local
(A) Atualização da EDL	CPU/GPU	*kernelEDL*	Atualização da EDL
(V) Avaliação da vizinhança	CPU/GPU	*kernelSwap*	Vizinhança do operador *swap*
"	"	*kernel2Opt*	Vizinhança do operador 2*opt*
"	"	*kernelOrOpt*	Vizinhança do operador *oropt-k*

Tabela 3.2: Sumários dos componentes da busca local implementados e suas respectivas arquiteturas e *kernels*.

3.6.1 Atualização da EDL

O procedimento *kernelEDL* é o *kernel* GPU responsável pela computação e atualização da EDL associada à solução corrente da busca local, estando representado pelo componente A nos esquemas da Figura 3.4. Quando esse componente é executado na CPU, a solução corrente da busca já reside na memória principal, o que permite a atualização da EDL sem a necessidade de transferência de dados (estratégias CFAV e CFA/GV). Nessas duas estratégias, caso o componente V esteja alocado na GPU (CFA/GV), a EDL deve transferida para a memória daquele dispositivo para avaliação da vizinhança. Já quando a avaliação da vizinhança é realizada na própria CPU (CFAV), o componente V já pode

[3]Estas fontes e instâncias serão apresentadas com mais detalhes no próximo capítulo, quando experimentos focados em desempenho e qualidade de solução serão conduzidos.

dispor da EDL na memória principal. Outros dois cenários surgem quando o componente A está alocado para execução na GPU (estratégias CFV/GA e CF/GAV). Nesse caso, quando o componente V também está alocado na GPU, o *kernel* responsável pela avaliação da vizinhança recupera a EDL da memória global do dispositivo (CF/GAV). Caso contrário, após a execução da avaliação, os dados resultantes são transferidos para CPU para seleção do melhor movimento (CFV/GA). Desta forma, é possível concluir que a escolha da melhor estratégia de alocação dos componentes depende do tempo de computação da EDL e dos custos de transferência envolvidos.

A geração da EDL consiste na atualização de quatro estruturas de dados identificadas por R, X, L e C, que estão descritas pelas Expressões 3.8, 3.9, 3.10 e 3.4, respectivamente. Considerando que n é a dimensão da solução, as três primeiras estruturas são representadas por vetores com n elementos, enquanto a última por uma matriz $n \times n$. É fácil deduzir que a atualização da EDL tem um custo computacional da ordem de $O(n^2)$ quando realizada sequencialmente pela CPU, sendo esta complexidade determinada pelo cômputo da matriz C. O modelo de execução da GPU acomoda bem as estruturas vetoriais da EDL, onde cada elemento pode ser processado paralelamente por uma *thread* da grade. No entanto, os elementos de cada linha da matriz C possuem dependência entre si e, portanto, precisam ser atualizados sequencialmente. As linhas da matriz, entretanto, são independentes e podem ser processadas paralelamente pela GPU, o que reduz a complexidade teórica do cômputo da EDL para $O(n)$.

O *kernelEDL* foi projetado para ser lançado em uma grade de computação GPU de dimensões $n \times n$ (ver Seção 2.2.1), como ilustrado pela Figura 3.5.

Inicialmente, os dados da instância do problema e da solução base são carregadas paralelamente pelas *threads* de cada bloco da memória global para a memória compartilhada (1). Em seguida, uma *thread* de cada bloco (2) processa um elemento de R, X e L e, sequencialmente, atualiza os elementos de uma linha de C (3). Finalmente, em (4), os dados da EDL são copiados paralelamente da memória compartilhada para a memória global. Quando todos os blocos finalizam suas respectivas tarefas, a EDL torna-se disponível na memória global para os *kernels* de avaliação de vizinhança.

Para computar os dados da EDL, muitas operações de leitura e escrita na memória são necessárias. Por este motivo, visando minimizar a latência de acesso, os dados de entrada são carregados para a memória compartilhada do multiprocessador associado a cada bloco. Lá os dados podem ser manipulados em alta velocidade e, ao final da execução do *kernelEDL*, são copiados para a memória global. Durante a carga, os dados

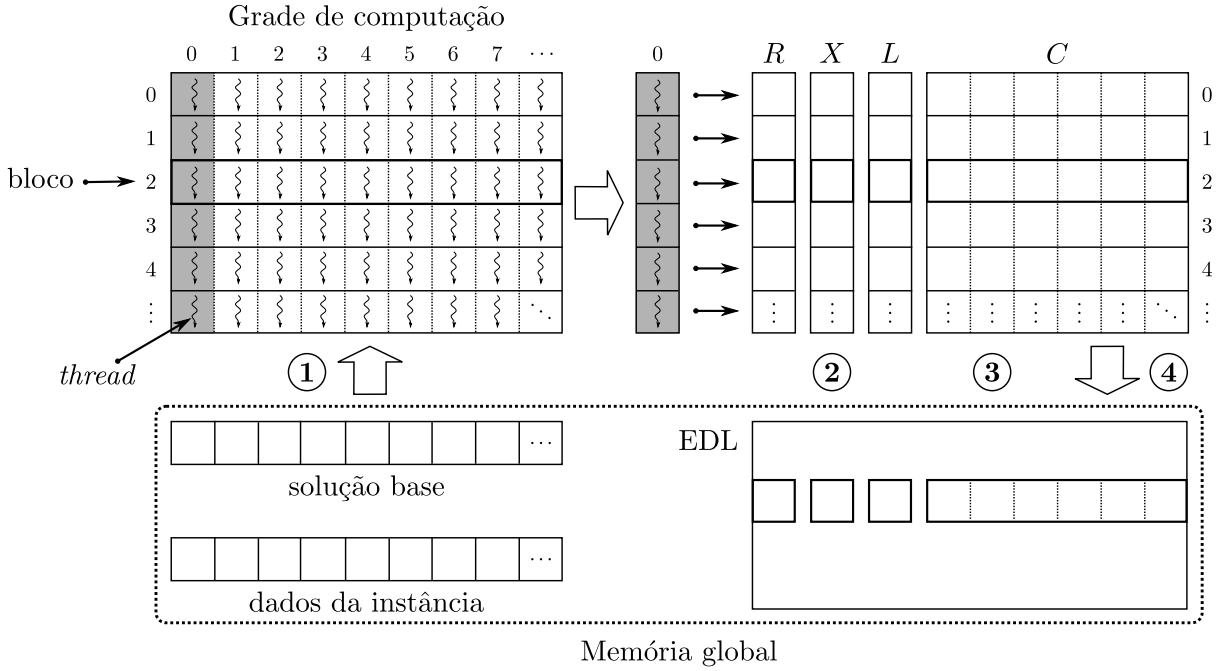


Figura 3.5: Esquema de execução do *kernelEDL*.

são organizados de tal forma que cada *thread* de um bloco acessa um banco de memória compartilhada distinto, evitando o problema do conflito de bancos. Um outro aspecto explorado por esta abordagem é que os acessos de leitura e escrita na memória global são feitos de modo coalescente, o que reduz ainda mais a latência de acesso a esse dispositivo. Isso é possível porque os dados residentes na memória global são organizados de tal forma que *threads* vizinhas em um mesmo *warp* acessam dados que residem em endereços de memória contíguos e alinhados. A combinação dessas técnicas, conduz ao uso mais eficiente dos dispositivos de memória mais rápidos da GPU e das vias de acesso à memória global (ver Seção 2.2.2).

Adicionalmente, foi realizado um ensaio para estimar a aceleração teórica máxima que pode ser obtida pelo *kernelEDL*. Essa estimativa foi realizada de acordo com a Lei de Amdahl [Amdahl, 1967], que determina que o ganho de desempenho obtido por um algoritmo paralelo é limitado pela porção de tempo consumida por sua porção serial. Assumindo que f representa o percentual correspondente à porção de código paralelo de um algoritmo, a aceleração teórica máxima que pode ser obtida pela utilização de p processadores é dada por:

$$S_T(p) = \frac{1}{(1-f) + \frac{f}{p}} \tag{3.11}$$

Nesse estudo, as comparações de desempenho entre algoritmos seriais e paralelos

utilizam os conceitos de *speedup absoluto* e *relativo* [McCool *et al.*, 2012], respectivamente definidos pelas Equações (3.12) e (3.13):

$$S_A(p) = \frac{\text{tempo de execução do melhor algoritmo sequencial}}{\text{tempo de execução do algoritmo paralelo usando } p \text{ processadores}} \quad (3.12)$$

$$S_R(p) = \frac{\text{tempo de execução do algoritmo paralelo usando 1 processador}}{\text{tempo de execução do algoritmo paralelo usando } p \text{ processadores}} \quad (3.13)$$

Em relação ao *speedup* teórico máximo, Expressão (3.11), foi utilizada a seguinte metodologia para determinar a porção percentual de código paralelo do *kernelEDL*. Inicialmente, o código do *kernel* foi decomposto em quatro partes, descritas a seguir na mesma sequência em que são executadas:

- K: código correspondente ao *overhead* de lançamento do *kernel*. Esse tempo foi obtido pelo lançamento do *kernel* com a mesma configuração de grade, mas com o corpo vazio (sem instruções).
- P_r: sequência paralela responsável pela leitura dos dados da solução e da instância a partir da memória principal;
- S: laço serial que realiza a atualização de uma linha da matriz C;
- P_w: trecho paralelo que armazena os resultados do processamento da EDL na memória global.

Tal artifício foi necessário porque não é possível realizar medidas de tempo de execução precisas dentro do *kernel*, mas apenas entre chamadas. Porém, a metodologia empregada mostrou-se bastante precisa, apresentando resultados coerentes com as medidas coletadas. Desta forma, tomando $K + P_r + S + P_w$ como o tempo total de execução do *kernel*, foram determinados os tempos de execução individuais de cada parcela e calculadas as proporções correspondentes. As medidas de tempo de execução do *kernel* na GPU foram coletadas por meio da API CUDA, que possibilita uma aferição bastante acurada, com precisão em torno de $0,5\mu$s ($10^{-6}s$) [NVIDIA, 2016]. O *kernel* foi executado 11 vezes para cada instância avaliada, sendo a primeira execução descartada por acumular o custo de cópia do código do *kernel* para a GPU[4]. Os tempos coletados para as 10 execuções

[4]Em CUDA, o processo de carga e execução do *kernel* é totalmente transparente para o desenvolvedor sendo similar à chamada de uma função. Contudo, na primeira execução envolvendo um *kernel*, seu código precisa ser copiado pelo *driver* para a memória da GPU, o que implica numa operação implícita de transferência de dados. Esse comportamento pode variar dependendo da microarquitetura utilizada. Nos experimentos realizados, a GPU GTX-550 mostrou-se mais sensível a esse *overhead* que a GTX-780, mais moderna.

de cada instância foram bastante estáveis, com desvio padrão sempre muito próximo de zero, o que justifica o número reduzido de amostras.

A Tabela 3.3 apresenta os resultados de aceleração alcançados pelo *kernelEDL* quando executado nas GPUs GTX-550 e GTX-780. Na tabela, a primeira coluna indica o número de clientes da instância do problema, seguida pela aferição do percentual correspondente à porção de código que foi paralelizada. As colunas $S_T(p)$ e $S_R(p)$ indicam, respectivamente, o *speedup* teórico máximo e relativo verificados nos testes considerando p processadores. A última coluna, *gap*, indica a diferença percentual entre $S_T(p)$ e $S_R(p)$.

GTX-550 ($p = 192$ *cores*)

n	f (%)	$S_T(p)$	$S_R(p)$	*gap* (%)
100	60,12	2,49	2,31	7,23
200	60,62	2,52	2,31	8,33
318	61,09	2,55	2,31	9,41
500	61,18	2,56	2,37	7,42
657	61,17	2,55	2,38	6,67
783	61,15	2,55	2,37	7,06
1.000	61,22	2,56	2,38	7,03
Média	**60,94**	**2,54**	**2,35**	**7,59**

GTX-780 ($p = 2.304$ *cores*)

n	f (%)	$S_T(p)$	$S_R(p)$	*gap* (%)
100	64,15	2,79	2,59	7,17
200	66,82	3,01	2,84	5,65
318	68,44	3,17	2,92	7,89
500	68,97	3,22	2,98	7,45
657	69,47	3,27	3,03	7,34
783	68,90	3,21	3,02	5,92
1.000	69,45	3,27	3,06	6,42
Média	**68,03**	**3,13**	**2,92**	**6,83**

Tabela 3.3: Comparação entre *speedup* teórico máximo e *speedup* relativo obtidos para o *kernelEDL*.

Para a GPU GTX-550, as operações de leitura/escrita paralela equivalem, em média, a 60,94% do tempo de execução do *kernel*. Portanto, uma aceleração média máxima de 2,54 seria esperada se todos os blocos da grade tivessem sido executados paralelamente nos 192 *cores* do dispositivo. Na prática, devido à latência de acesso à memória e a eventual serialização no escalonamento de blocos, um *speedup* 7,59% inferior foi obtido, o que ainda representa um resultado bastante satisfatório. Um comportamento similar foi observado para a GTX-780, que com seus 2.304 *cores* obteve uma aceleração relativa apenas 6,83% menor que o limite teórico de 3,13. A maior porção de paralelismo obtido

na GTX-780 pode ser explicado pelo fato de que os processadores operam mais rapidamente do que na GTX-550, de forma que o tempo de execução da porção estritamente serial tende a ser reduzido. Mesmo com parte do código serializado, um notável nível de paralelismo foi obtido em *kernelEDL* em ambas as GPUs, bastante próximo do limite teórico determinado pela Lei de Amdahl.

Um segundo experimento foi concebido para verificar qual a melhor configuração de execução para o *kernelEDL*. Esse experimento analisa a eficiência de execução do *kernel* em função da taxa de ocupação da GPU. Sempre que um bloco é atribuído a um multiprocessador (SM) todos os recursos necessários para a execução de suas *threads* são reservados. Alguns desses recursos, como registradores e variáveis locais utilizadas pelo *kernel*, são alocados na escassa memória compartilhada. Isso significa que um maior ou menor consumo de recursos por parte do *kernel* influencia na quantidade de blocos que podem ser executados paralelamente em um mesmo multiprocessador. Os recursos utilizados por um bloco são alocados em função do número de *warps* (grupos de 32 *threads*) necessário para executá-lo. Porém, existe um limite para a quantidade de *warps* que pode ser alocada em um multiprocessador, o que influencia na taxa de ocupação. Nos dispositivos GPU, a taxa de ocupação (*occupancy*) é dada pela razão entre o número de *warps* ativos (alocados) em um multiprocessador e o número máximo de *warps* que este pode gerenciar (determinado pela microarquitetura). Como regra geral para um melhor desempenho, o ideal é manter a taxa de ocupação alta, próxima ou igual ao seu limite de 100%. Isso indica que muitos *warps* estarão com seus recursos alocados e disponíveis para execução sempre que uma preempção ocorrer. Por outro lado, a alocação desnecessária de recursos do SM reduz a quantidade de blocos que poderiam ser executados paralelamente, o que pode deteriorar o desempenho do *kernel*.

Desta forma, para alcançar uma taxa de ocupação adequada é necessário efetuar na configuração do bloco um balanceamento entre paralelismo e a computação executada por cada *thread*. Neste experimento, o *kernel* foi parametrizado para executar esse balanceamento de forma que à medida que o paralelismo aumenta, a carga de trabalho de cada *thread* diminui. No contexto do *kernelEDL*, isso equivale a variar a dimensão de um bloco à medida que cada *thread* processa um número inversamente proporcional de elementos da EDL.

Neste ensaio, o número de blocos da grade foi fixado como igual ao número de linhas na matriz C. A dimensão do bloco varia com valores discretos no intervalo de 1 a 1.024, de acordo com as seguintes configurações: B1, B128, B256, B384, B512, B672,

B768 e B1024. O número após o prefixo B indica a quantidade de *threads* presentes no bloco. Esses valores foram selecionados empiricamente por meio da ferramenta NVIDIA Profiler, que coleta e analisa métricas de desempenho durante a execução do *kernel*. A configuração B1, que utiliza uma única *thread* para realizar todo o trabalho, é equivalente à versão serial do *kernelEDL* utilizada na análise dos dados da Tabela 3.3.

A Figuras 3.6 e 3.7 apresentam os resultados do experimento para os dispositivos GTX-780 e GTX-550, respectivamente. Para a GTX-780 a configuração serial B1 apresenta, conforme a expectativa, a pior performance, induzida pela baixa taxa de ocupação. O melhor balanceamento para esta GPU foi obtido pela configuração B384. Por questão de clareza, o gráfico da Figura 3.6 não apresenta os resultados para B128 e B1024, uma vez que foram apenas 1% pior que B256. O mesmo critério foi adotado para B512 e B768 em relação a B672.

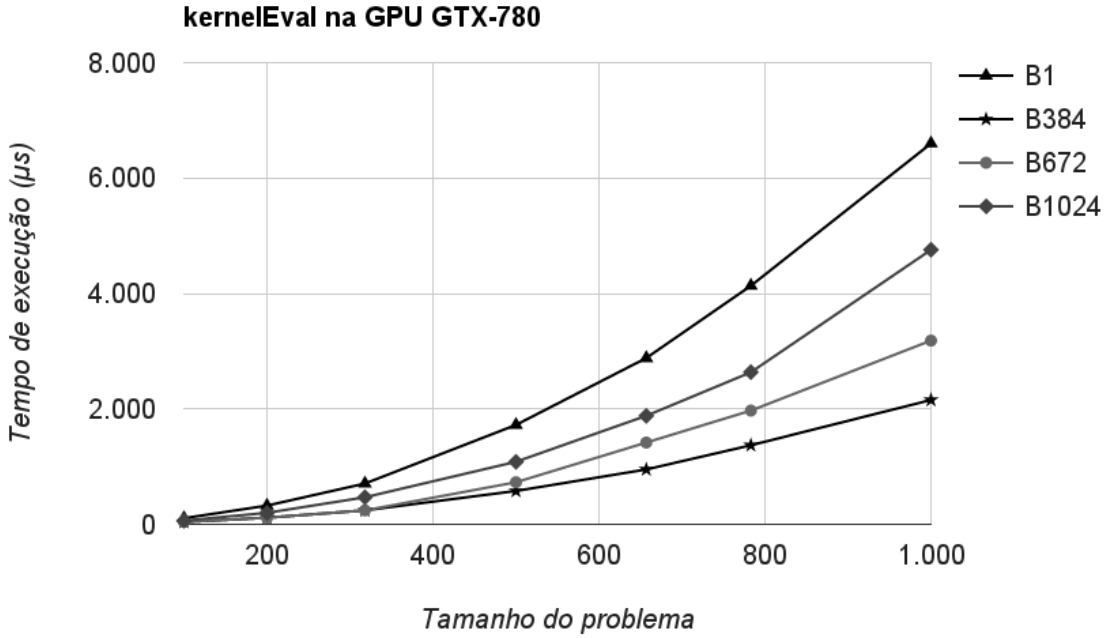


Figura 3.6: Análise da taxa de ocupação do kernelEDL na GPU GTX-780.

Para a GPU GTX-550, a configuração B1024 teve o pior desempenho, mesmo quando comparada com a configuração serial B1. Isto é explicado pela grande quantidade de sincronizações realizadas após a carga dos dados para a memória compartilhada, em relação com a pequena quantidade de trabalho realizado por cada *thread*. Finalmente, o melhor balanceamento entre paralelismo e processamento na GTX-550 foi obtido por B256, com uma vantagem não maior que 1% em relação a B128 e B384.

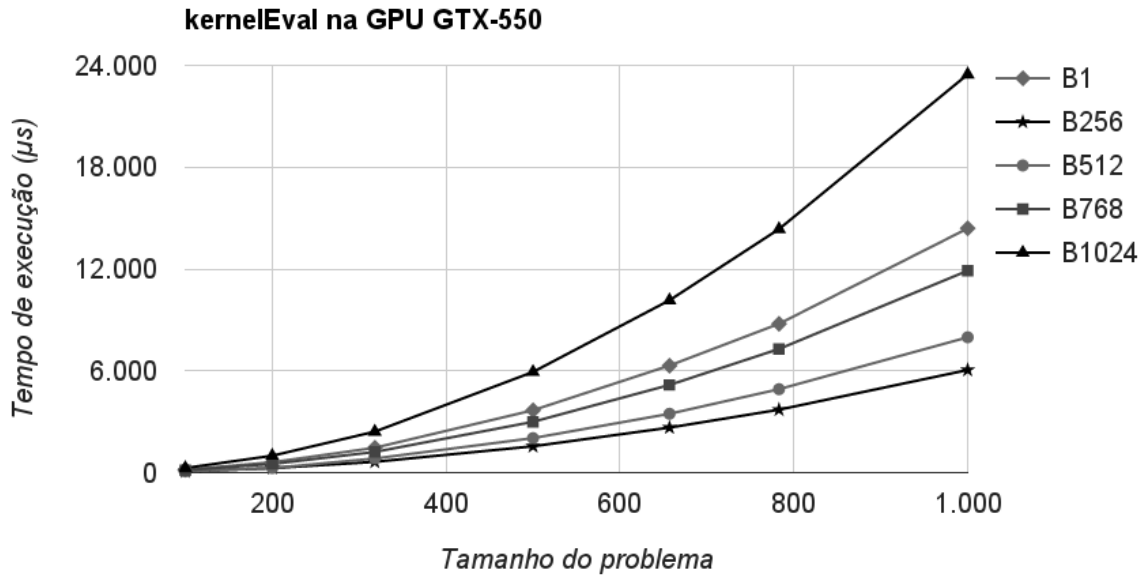


Figura 3.7: Análise da taxa de ocupação do kernelEDL na GPU GTX-550.

3.6.2 Transferência da EDL

Como já ressaltado, o ponto crítico relacionado ao desempenho das meta-heurísticas paralelas usualmente está associado à comunicação entre tarefas. Nos ambientes CPU/GPU tal aspecto corresponde, não apenas à transferência de dados entre as memórias da CPU e GPU, mas também ao acesso concorrente à memória global dessa última arquitetura. Nesse contexto, alguns experimentos foram conduzidos para mensurar o custo de atualização e movimentação da EDL nos seis cenários estudados.

As Tabelas 3.4 e 3.5 apresentam os tempos computacionais em microssegundos (μs) obtidos para as estratégias envolvendo o componente A e as transferências de dados relacionadas utilizando as GPUs GTX-550 e GTX-780, respectivamente. A primeira coluna, n, indica o tamanho do problema considerado (número de clientes). A coluna seguinte indica o tempo médio de transferência da EDL entre as memórias da CPU e da GPU. As colunas restantes formam duas seções com duas colunas cada. Essas seções representam, respectivamente, os cenários em que a estratégia utilizada necessita que a EDL resida na memória da CPU ou da GPU para que a avaliação da vizinhança possa ser realizada. Na primeira seção, a coluna CA apresenta o tempo de atualizar a EDL na CPU e a coluna GA+TGC indica o tempo de computar a EDL na GPU a transferi-la para a CPU. Finalmente, as duas últimas colunas apresentam, respectivamente, o tempo de atualizar a EDL na GPU (sem transferência) seguida do tempo de calcular a EDL na

CPU somado ao tempo de sua cópia para a GPU.

	GPU GTX-550				
n	**Tempo**	**EDL na CPU**		**EDL na GPU**	
	Transf.EDL	**CA**	**GA+TGC**	**GA**	**CA+TCG**
100	22	**34**	101	79	**56**
200	59	**139**	326	267	**198**
318	131	**347**	765	634	**478**
500	317	**952**	1.871	1.554	**1.269**
657	543	**1.616**	3.199	2.656	**2.159**
783	754	**2.372**	4.464	3.710	**3.126**
1.000	1.230	**3.706**	7.279	6.049	**4.936**

Tabela 3.4: Tempos de execução e transferências de dados (em μs) envolvendo a EDL para a GPU GTX-550.

Quando a avaliação da vizinhança é feita na CPU, para a GTX-550 é mais vantajoso atualizar e utilizar a EDL sem o uso da GPU, como indicam os números destacados na coluna CA. Já quando a EDL é necessária na GPU é mais proveitoso atualizá-la na CPU e efetuar sua transferência para a GPU, como indicado pela coluna CA+TCG. Já para a GTX-780, é mais interessante atualizar a EDL na CPU quando o volume de dados é menor (instâncias com 100, 200 e 318 clientes). Já para as instâncias maiores que 318, é mais vantajoso calcular os dados na GPU e copiar a estrutura já calculada para a memória da CPU. Nos cenários onde é necessário o uso da EDL na GPU, para a GTX-780 é sempre mais vantajoso calcular esses dados diretamente no dispositivo através do *kernelEDL*. De forma geral, os dados confirmam que a GTX-780 realiza não somente transferências de dados mais rapidamente que a GTX-550, mas também processa de forma mais eficiente os dados armazenados na memória global.

	GPU GTX-780				
n	**Tempo**	**EDL na CPU**		**EDL na GPU**	
	Transf.EDL	**CA**	**GA+TGC**	**GA**	**CA+TCG**
100	18	**28**	59	**41**	46
200	41	**129**	155	**114**	170
318	78	**308**	318	**240**	386
500	176	839	**754**	**578**	1.015
657	296	1.491	**1.247**	**951**	1.787
783	415	2.099	**1.785**	**1.370**	2.514
1.000	676	3.686	**2.831**	**2.155**	4.362

Tabela 3.5: Tempos de execução e transferências de dados (em μs) envolvendo a EDL para a GPU GTX-780.

3.6.3 Avaliação da Vizinhança

Os experimentos apresentados a seguir estão relacionados com a avaliação de vizinhança realizada pelo componente V presente nos diagramas da Figura 3.4. O objetivo foi avaliar a contribuição da EDL na busca local e verificar a influência dos custos de processamento e transferências de dados relacionados com essa estrutura. Para tanto, foram desenvolvidos *kernels* para avaliar a vizinhança dos operadores de movimento *swap*, 2*opt* e *oropt-k*, considerando dois cenários: com e sem o uso de EDL. A intenção foi verificar se o uso dessa estrutura pode tirar proveito do paralelismo massivo da GPU, ou se o *overhead* de cálculo e transferências associados pode ofuscar seu benefício.

Conforme já abordado, o uso da EDL permite que cada operação de movimento possa ser avaliada em tempo constante. Caso contrário, cada avaliação precisa ser computada de acordo com o respectivo operador de movimento, que pode envolver um número constante ou variável de operações. As subseções seguintes descrevem como cada *kernel* foi projetado apresentando resultados e ressaltando os requisitos de desempenho relevantes.

kernel2Opt

O *kernel2Opt* equivale à implementação para GPU do procedimento de avaliação da vizinhança gerada pelo operador 2*opt*. Conceitualmente, esse operador atua sobre uma solução base removendo duas arestas da rota e reconectando-as numa configuração diferente. Na implementação realizada neste trabalho, isso equivale a inverter a ordem dos clientes em uma sub-rota, conforme ilustrado pela Figura 3.2(b). A metodologia de avaliação de vizinhança baseada na EDL dispensa a geração da solução vizinha, uma vez que o custo de um movimento $2opt(i, j)$ pode ser obtido acessando o elemento $C(i, j)$ correspondente. Já a avaliação iterativa (sem EDL) desse movimento conduz a complexidade da operação para um custo da ordem de $O(n)$, considerando que n representa o número de clientes na solução.

Independente do uso ou não de EDL, a avaliação desta vizinhança envolve a computação de uma quantidade de movimentos da ordem de $O(n^2)$. Sendo assim, o mapeamento de cada movimento $2opt(i, j)$ pode ser realizado de forma que os operandos i e j sejam associados, respectivamente, às linhas e colunas de uma matriz. Esse mapeamento é bastante conveniente para o modelo de execução da GPU, pois pode ser projetado sobre uma grade de computação bidimensional, associando cada linha da matriz a um bloco, de forma que cada *thread* fique encarregada por avaliar um movimento. Respeitadas as

restrições de movimentação de alguns clientes da rota (depósito), cada bloco necessita de $n-3$ *threads* para levar a cabo sua tarefa. No entanto, considerando que a operação 2*opt* implementada é comutativa, somente as *threads* correspondentes aos movimentos da porção triangular superior (ou inferior) da grade ficariam ocupadas. A Figura 3.8(a) procura ilustrar esse mapeamento para uma solução com $n = 10$ clientes.

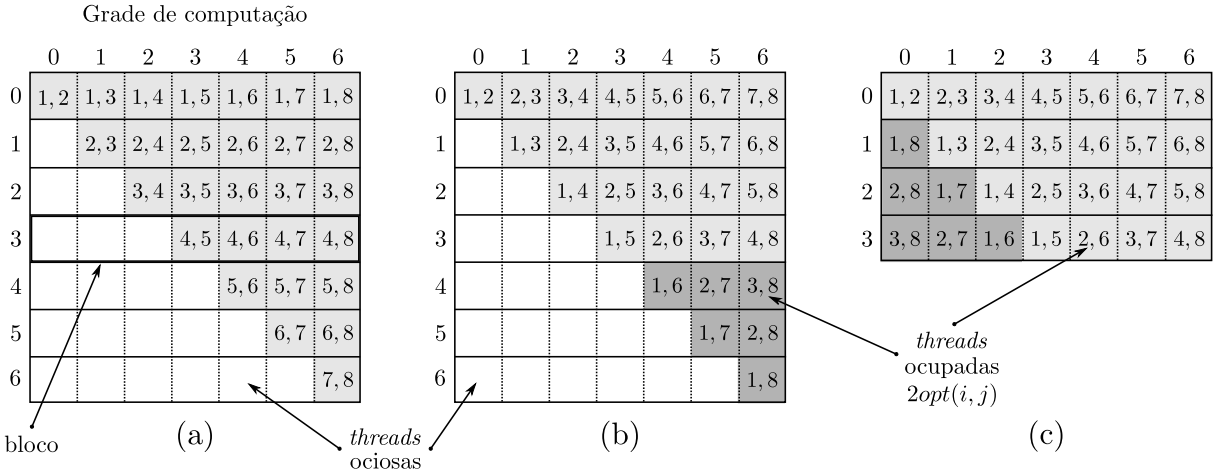


Figura 3.8: Configurações para a grade de computação do *kernel2Opt* sem uso de EDL.

Na Figura 3.8, os quadros hachurados representam as *threads* responsáveis por um movimento $2opt(i, j)$, enquanto os demais indicam tarefas ociosas. Os números i, j dentro de cada quadro representam os operandos do 2*opt*, que delimitam a sub-rota afetada. De fato, o arranjo disposto em (a) desperdiça muitos recursos de multiprocessador, pois quase metade dos blocos estão com a maioria de suas *threads* desocupadas, porém com seus respectivos recursos alocados. Além disso, essa disposição introduz divergência no código do *kernel*, pois o cômputo dos movimentos de um mesmo bloco envolve a inversão de sub-rotas de comprimentos distintos[5]. Esse problema pode ser resolvido com a redistribuição da função de cada *thread* de modo que as operações que envolvam inversões de mesmo comprimento residam em um mesmo bloco, como ilustrado em (b). Finalmente, para ocupar as *threads* ociosas, uma nova disposição pode ser feita de forma que as operações associadas aos blocos de menor ocupação sejam realocadas, como indicado por (c). Esse último arranjo traz dois benefícios bastante relevantes: elimina as *threads* ociosas e reduz o número de blocos praticamente pela metade. No entanto, essa mesma configuração volta a introduzir divergência nos blocos. Porém, com uma análise mais cuidadosa, verifica-se que existem apenas dois comprimentos de sub-rotas distintos em um mesmo bloco.

[5]A avaliação do custo de inversão de uma sub-rota envolve a execução de um laço com $|i - j| + 1$ iterações. A divergência provém, portanto, do diferente número de iterações executado por cada *thread* em seus respectivos laços.

Sabendo que a divergência afeta apenas as *threads* de um mesmo *warp* e que este é a unidade de escalonamento adotada pela GPU, constata-se que no máximo um *warp* por bloco será afetado, o que torna a questão da divergência irrelevante.

O mapeamento de movimentos apresentado procura otimizar diversos aspectos de desempenho inerentes à arquitetura GPU. Porém, ainda há espaço para mais um ajuste, agora relacionado com a taxa de ocupação dos multiprocessadores. Somente a ocupação de todas as *threads* de um bloco não garante uma taxa de ocupação apropriada no multiprocessador. Como foi discutido, isso é alcançado por meio do balanceamento entre paralelismo e carga de trabalho das *threads*. Tal balanceamento é um exercício empírico, já que depende de características da microarquitetura e dos recursos utilizados pelo *kernel* que, normalmente, só podem ser aferidos com precisão pela análise do *kernel* em tempo de execução[6]. Neste contexto, foram realizados experimentos similares aos que resultaram nas Figuras 3.6 e 3.7, de onde foi concluído que a melhor configuração foi de 256 *threads*/bloco para a GTX-550 e 384 *threads*/bloco para GTX-780.

É importante observar que a otimização de configuração apresentada foi idealizada para o cenário onde o *kernel2opt não utiliza* EDL. Quando o uso de EDL é considerado, o *kernel* possui uma orientação distinta e, portanto, diferente implementação. Como se sabe, quando o uso de EDL é empregado, a avaliação de um movimento $2opt(i, j)$ consiste em acessar o elemento $C(i, j)$, que reside na memória global da GPU. Para projetar a grade de computação para este cenário é preciso ponderar que, além da alta latência, esse recurso será acessado concorrentemente por um número significativo de *threads*, de ordem teórica $O(n^2)$. Desta forma, o arranjo das atividades executadas por cada *thread* precisa ser efetuado de forma a minimizar a latência envolvida na recuperação de dados da memória global e tirar proveito da escassa memória *cache* do multiprocessador. Isso pode ser obtido por meio do acesso coalescente, onde *threads* adjacentes acedem dados que residem em endereços alinhados e contíguos da memória global (ver Seção 2.2.2). Neste cenário, a melhor configuração de grade para o *kernel2Opt* é a apresentada pela Figura 3.9(b), já com a adequada reacomodação das *threads* residentes em blocos com baixa ocupação observada em (a). Esse arranjo é apropriado porque as *threads* residentes em um mesmo bloco acessam, no máximo, duas diferentes linhas da matriz C e *threads* adjacentes acessam elementos contíguos.

[6]As versões mais recentes da biblioteca CUDA incluem a *Occupancy* API, um conjunto de funções que retornam informações relativas à taxa de ocupação. Essas informações podem ser utilizadas para promover o balanceamento dinâmico do bloco em tempo de execução. No entanto, esse recurso não substitui a análise experimental com a ferramenta NVIDIA Visual Profiler, que expõe outros aspectos relativos ao desempenho do *kernel*.

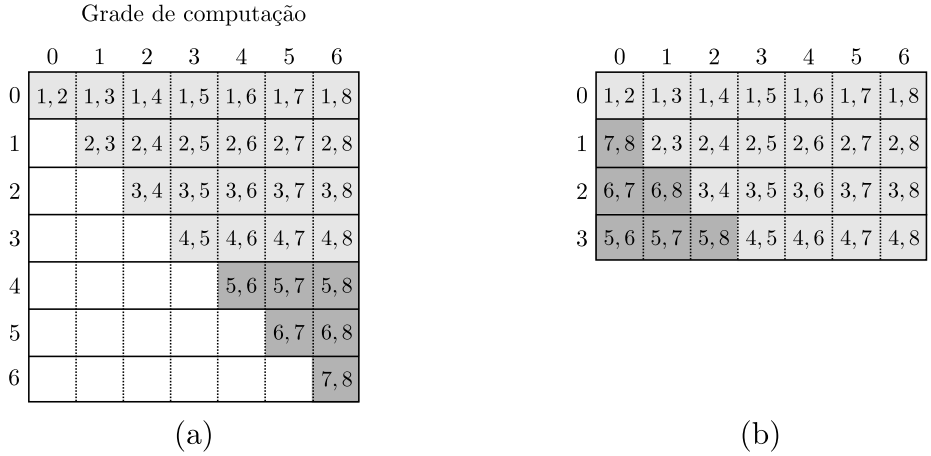


Figura 3.9: Configurações para a grade de computação do *kernel2Opt* com o uso de EDL.

Finalmente, para concluir a descrição do *kernel2Opt*, é preciso mencionar o destino dos dados processados. Independente do uso ou não de EDL, uma consolidação parcial dos resultados de cada movimento é realizada dentro de cada bloco. Isso significa que, após carregarem ou computarem os resultados de movimentos, as *threads* de um bloco passam por uma barreira de sincronização e uma redução de minimização dos ganhos de custo é realizada. O *ganho de custo* de um movimento m, expresso por $g(m)$, é definido como o valor que, adicionado ao custo da solução base s, resulta no custo da solução vizinha s' correspondente, isto é, $f(s') = f(s) + g(m)$.

Após a redução, cada bloco detém os dados do movimento (ganho de custo e operandos) que conduz ao menor custo de solução que, finalmente, são transferidos para a memória global numa única operação paralela. Ao final do processo um vetor com a consolidação dos movimentos de cada bloco fica disponível na memória global. Esses dados são posteriormente transferidos para memória da CPU, onde uma nova redução é executada e a solução vizinha resultante pode ser construída. A decisão de não fazer a redução total na GPU teve como objetivo não apenas evitar novas transferências de dados entre memória global e compartilhada, mas também está relacionada com uma nova metodologia de avaliação da vizinhança que será apresentada no capítulo seguinte.

Esta redução parcial é importante porque diminui significativamente a quantidade de resultados de avaliação de movimentos de $O(n^2)$ para o valor equivalente ao número de blocos da grade, da ordem de $O(n)$, o que reduz radicalmente o volume de dados a serem transferidos pela congestionada via de acesso à memória global[7]. A redução é realizada na memória compartilhada sendo executada em paralelo pelas *threads* de um mesmo bloco

[7]Mais precisamente, o número de elementos cai de $(n-3)(\frac{n}{2}-1)$ para $\frac{n}{2}-1$.

empregando o método ilustrado pela Figura 3.10. Esse método é mais eficiente que uma redução linear tradicional, pois pode ser realizado em tempo $O(\log n)$. Uma descrição mais detalhada desse processo pode ser encontrada em [Harris, 2007].

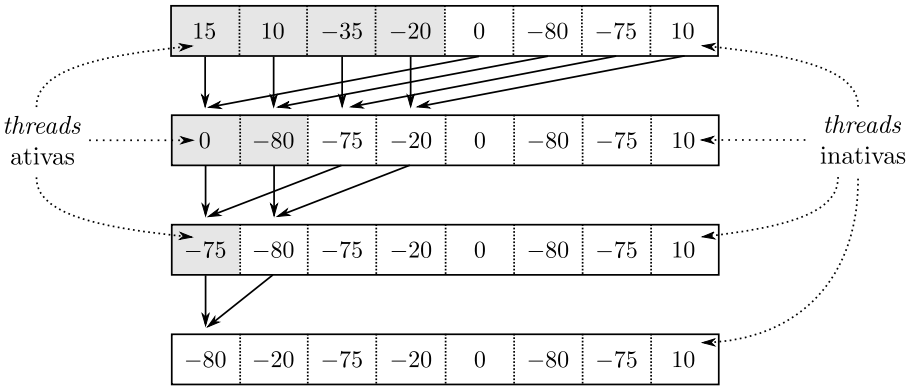


Figura 3.10: Exemplo de execução de uma redução de minimização paralela para um vetor com 8 elementos.

A seguir são apresentados os resultados computacionais obtidos pelos *kernels* implementados. As Tabelas 3.6 e 3.7 apresentam valores médios obtidos por 11 execuções do *kernel2Opt* nas GPUs GTX-550 e GTX-780, respectivamente. Assim como no experimento anterior, a primeira execução foi descartada visando eliminar o *overhead* de carga do *kernel* e, novamente, os tempos de cada execução foram bastante estáveis. Nestas tabelas, a coluna n indica o tamanho da instância do problema e as demais colunas os resultados de tempo ou de *speedup absoluto*, $S_A(p)$, para cada uma das seis estratégias ilustradas na Figura 3.4. São apresentadas duas comparações de *speedup*, uma entre CFV e CF/GV, e outra entre CFAV e as estratégias CFA/GV, CFV/GA e CF/GAV. Quando a avaliação da vizinhança é realizada sem o uso de EDL (colunas 2 e 3), *speedups* máximos de 82,74 e 410,83 são alcançados por CF/GV sobre CFV nas GPUs GTX-550 e GTX-780, respectivamente. Esse resultado foi possível porque a avaliação de uma grande quantidade de computação, $O(n^3)$, tirou proveito do paralelismo massivo empenhado pela estratégia CF/GV na GPU. Conforme esperado, quando a CPU utiliza a EDL o tempo de execução do algoritmo cai drasticamente, de forma que a estratégia CFAV, puramente baseada em CPU, é mais rápida que qualquer outra estratégia orientada para GPU na GTX-550, como pode ser observado nas três últimas colunas da Tabela 3.6.

Na GPU GTX-780 foi possível obter uma aceleração máxima de 2,04 com a estratégia CF/GAV. Nas demais estratégias, os *speedups* médios não foram melhores apresentando valores modestos frente ao potencial paralelo oferecido pelo dispositivo. Este resultado sugere que o montante de trabalho paralelo executado na GPU não foi sufici-

GTX-550 ($p = 192$)						
	sem EDL		**com EDL**			
	tempo(μs)	$S_A(p)$	**tempo(μs)**	$S_A(p)$		
n	**CFV**	**CF/GV**	**CFAV**	**CFA/GV**	**CFV/GA**	**CF/GAV**
100	634	19,81	63	0,47	0,35	0,40
200	4.333	38,35	240	0,65	0,48	0,54
318	13.436	46,65	545	0,65	0,48	0,55
500	49.432	56,49	1.390	0,71	0,54	0,62
657	121.482	64,86	2.433	0,75	0,57	0,65
783	215.960	71,44	3.449	0,75	0,58	0,66
1.000	483.175	82,74	5.481	0,73	0,56	0,64
Média		**54,33**		**0,67**	**0,51**	**0,58**
Máximo		**82,74**		**0,75**	**0,58**	**0,66**

Tabela 3.6: Resultados da avaliação da vizinhança do operador 2*opt* na GPU GTX-550.

ente para compensar os custos de transferências de dados e permitir que o dispositivo pudesse alcançar acelerações mais significativas. Apesar disso, uma comparação entre as duas tabelas mostra que o *speedup* é incrementado quando o número de *cores* GPU aumenta.

GTX-780 ($p = 2.304$)						
	sem EDL		**com EDL**			
	tempo(μs)	$S_A(p)$	**tempo(μs)**	$S_A(p)$		
n	**CFV**	**CF/GV**	**CFAV**	**CFA/GV**	**CFV/GA**	**CF/GAV**
100	531	31,24	52	0,63	0,55	0,68
200	4.258	118,28	231	0,98	1,05	1,29
318	13.102	187,17	506	1,02	1,18	1,44
500	52.426	274,48	1.300	1,09	1,39	1,71
657	130.607	363,81	2.243	1,10	1,49	1,85
783	220.227	340,38	3.162	1,11	1,48	1,84
1.000	489.296	410,83	5.424	1,12	1,63	2,04
Média		**246,60**		**1,01**	**1,25**	**1,55**
Máximo		**410,83**		**1,12**	**1,63**	**2,04**

Tabela 3.7: Resultados da avaliação da vizinhança do operador 2*opt* na GPU GTX-780.

kernelOrOpt

O *kernelOrOpt* é o procedimento responsável pela avaliação das vizinhanças geradas pelos operadores *oropt*-1, *oropt*-2 e *oropt*-3. Esse *kernel* recebe um parâmetro $k \in \{ 1, 2, 3 \}$ que indica qual operação *oropt* será executada. Por definição, uma operação *oropt*-k realiza a relocação de k clientes adjacentes de uma posição para outra na rota, como pode ser observado na Figura 3.2(b). Quando a avaliação de um movimento *oropt*-$k(i, j)$ é auxiliado pela EDL a operação, novamente, se resume em acessar o elemento $C(i, j)$ correspondente. Sem o uso da estrutura auxiliar, o cômputo do custo desse movi-

mento implica na relocação de $|i-j|+k$ clientes da solução, o que leva a um número de operações da ordem de $O(n)$. Como a operação *oropt-k*(i,j) é binária, a quantidade de movimentos relativos à vizinhança gerada é da ordem de $O(n^2)$. Portanto, uma projeção similar à realizada para o operador 2*opt* pode ser feita sobre uma grade computacional com duas dimensões, onde o tamanho de cada bloco é igual a $n-k-2$ *threads*. A Figura 3.11 esboça esse mapeamento para a vizinhança do operador *oropt*-2 para uma solução com $n=11$ elementos.

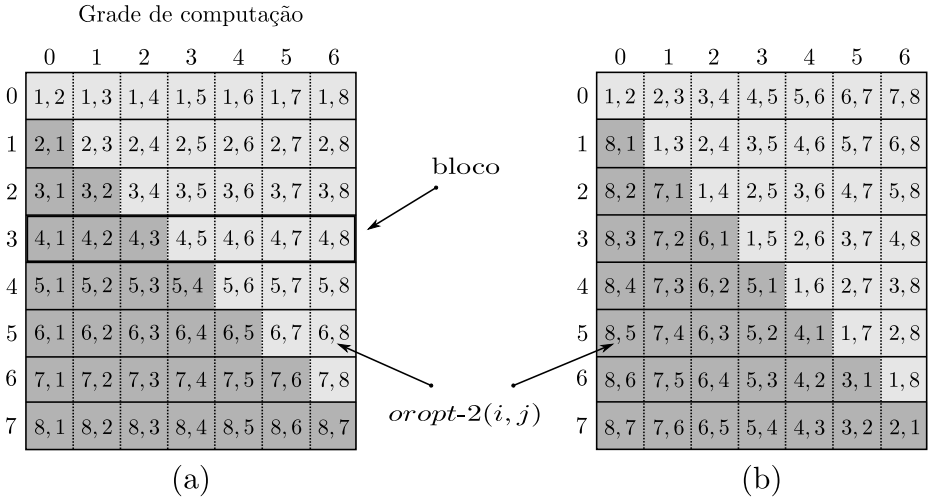


Figura 3.11: Configurações para a grade de computação do *kernelOrOpt* para o operador *oropt*-2.

Uma vez que a operação *oropt-k* não é comutativa, não existe o problema relacionado com *threads* ociosas, como pode ser constatado na Figura 3.11(a). Porém, como no caso do 2*opt*, existe a questão da divergência no número de operações realizado por *threads* de um mesmo bloco. Fazendo um novo arranjo, de forma que relocações de mesmo comprimento residam em um mesmo bloco, chega-se a uma configuração mais adequada, disposta pela Figura 3.11(b). A questão da dualidade de comprimentos em um mesmo bloco surge novamente mas, pelas mesmas razões ressaltadas previamente, também é insignificante neste contexto.

Quando o uso de EDL é empregado, a configuração de grade mais adequada é a ilustrada pela Figura 3.11(a), que permite o acesso coalescente à memória global. Neste caso, porém, a multiplicidade de comprimentos de relocação não influencia na execução do *kernel*, já que todas as *threads* de um mesmo bloco carregam o resultado a partir de uma mesma linha da matriz C. Em ambos os cenários — com e sem o uso de EDL — uma redução parcial é executada, antes que os resultados da avaliação de movimentos sejam armazenados na memória global.

Os resultados apresentados a seguir seguem a mesma metodologia dos experimentos realizados para o *kernel2Opt*. As Tabelas 3.8 e 3.9 exibem os resultados para execução do *kernelOrOpt* para o operador *oropt*-1 ($k = 1$). Como pode ser visto nas tabelas, *speedups* absolutos máximos de 50,24 e 150,68 foram obtidos, respectivamente, pelas GPUs GTX-550 e GTX-780 para a instância de maior porte. Relativamente, os métodos baseados em EDL desempenham melhor na GTX-780 do que na GTX-550 a qual, em média, sequer alcança os resultados obtidos por CFAV, exclusivamente baseado em CPU. A explicação para esse fato está relacionada, novamente, com os custos de movimentação de dados, tanto dentro da GPU, quanto entre CPU e GPU. Neste contexto, pode-se concluir que a larga memória *cache* da CPU contribui positivamente para o desempenho do algoritmo serial, enquanto os custos de movimentação de dados parecem determinar o gargalo dos métodos baseados em EDL.

GTX-550 ($p = 192$)

	sem EDL		**com EDL**			
	tempo(μs)	$S_A(p)$	**tempo(μs)**	$S_A(p)$		
n	**CFV**	**CF/GV**	**CFAV**	**CFA/GV**	**CFV/GA**	**CF/GAV**
100	531	11,06	91	0,44	0,36	0,40
200	2.657	17,25	299	0,57	0,46	0,50
318	8.772	19,58	735	0,69	0,54	0,60
500	42.466	31,00	1.882	0,79	0,63	0,70
657	110.254	29,16	3.378	0,84	0,67	0,75
783	202.895	37,18	4.854	0,86	0,69	0,78
1.000	469.559	50,24	11.440	1,29	1,02	1,15
Médio		**27,92**		**0,78**	**0,62**	**0,70**
Máximo		**50,24**		**1,29**	**1,02**	**1,15**

Tabela 3.8: Resultados da avaliação da vizinhança do operador *oropt*-1 na GPU GTX-550.

GTX-780 ($p = 2.304$)

	sem EDL		**com EDL**			
	tempo(μs)	$S_A(p)$	**tempo(μs)**	$S_A(p)$		
n	**CFV**	**CF/GV**	**CFAV**	**CFA/GV**	**CFV/GA**	**CF/GAV**
100	470	24,74	78	0,69	0,62	0,72
200	2.990	57,50	311	1,04	1,10	1,28
318	9.319	66,56	735	1,21	1,37	1,60
500	44.750	133,58	1.840	1,36	1,69	2,01
657	111.171	145,89	3.210	1,42	1,86	2,24
783	205.228	103,34	4.518	1,44	1,87	2,26
1.000	473.436	150,68	10.462	1,99	2,81	3,43
Médio		**97,47**		**1,31**	**1,62**	**1,94**
Máximo		**150,68**		**1,99**	**2,81**	**3,43**

Tabela 3.9: Resultados da avaliação da vizinhança do operador *oropt*-1 na GPU GTX-780.

Uma questão interessante surge quando são comparados os resultados obtidos por

kernelOrOpt e *kernel2Opt* para a estratégia CF/GV: porque esse último *kernel* teve um desempenho médio tão superior ao primeiro se ambos realizam, assintoticamente, a mesma quantidade de relocação de clientes? A resposta está relacionada com a remodelagem da EDL realizada neste trabalho. Resgatando a estrutura da EDL, verifica-se que o componente R, Expressão (3.8), armazena as distâncias referentes a clientes adjacentes da solução. Já o componente X guarda as coordenadas utilizadas para o cálculo das distâncias de clientes não adjacentes. Como as instâncias do problema são simétricas e o operador 2*opt* realiza menos concatenações de segmentos que o *oropt*, ele se beneficia mais com a disponibilidade desses dados (ver Figura 3.3).

Os resultados referentes aos operadores *oropt*-2 e *oropt*-3 não serão discutidos porque refletem o desempenho apresentado nas Tabelas 3.8 e 3.9. Na realidade, já era esperado que os resultados de *speedup* entre os operadores *oropt* fossem similares, já que a variação no valor de k afeta apenas a cardinalidade das vizinhanças, que será a mesma encontrada por cada uma das estratégias *oropt*-k testadas.

kernelSwap

A operação *swap* realiza a permutação de dois clientes da rota sendo executada pelo procedimento *kernelSwap*. Diferentemente dos demais operadores, uma operação $swap(i, j)$ afeta sempre dois clientes da solução sendo, portanto, executada em tempo $O(1)$, independente dos valores dos operandos. A avaliação da vizinhança, no entanto, segue o mesmo padrão dos demais operadores e apresenta cardinalidade $O(n^2)$. Para este operador, uma mesma configuração de grade para o lançamento do *kernel* satisfaz os requisitos de desempenho para os algoritmos que adotam ou não a EDL. Como a operação *swap* é comutativa, a disposição de grade mais adequada é similar à adotada pelo *kernel2Opt*, Figura 3.9(b), repeitadas as restrições de variação e combinação dos operandos de *swap*.

Neste contexto, não é esperado que o uso de EDL contribua para a aceleração do método, tanto na arquitetura CPU quanto na GPU. Isto porque o custo computacional da avaliação da vizinhança em ambos os casos é o mesmo: $O(n^2)$. Os resultados apresentados nas Tabelas 3.10 e 3.11 corroboram essa expectativa. Por conta da menor complexidade, a estratégia básica CFV é capaz avaliar toda a vizinhança mais rapidamente que em qualquer outro operador. Tomando a instância de 1.000 clientes como exemplo, pode-se verificar que toda a vizinhança é avaliada em torno de 3 ms pela CPU. Embora a estratégia CF/GV alcance um *speedup* máximo de 19,63 na GTX-780, essa aceleração é menor que

as obtidas por *kernel2Opt* e *kernelOrOpt*. Como esperado, as estratégias baseadas em EDL apresentam os piores resultados. Esse comportamento foi induzido pelos custos de computação e tráfego na memória global envolvendo essa estrutura, que foram mais lentos que a computação dos mesmos dados quando necessário.

GTX-550 ($p = 192$)

	sem EDL		**com EDL**			
	tempo(μs)	$S_A(p)$	**tempo(μs)**	$S_A(p)$		
n	**CFV**	**CF/GV**	**CFAV**	**CFA/GV**	**CFV/GA**	**CF/GAV**
100	32	1,78	79	0,88	0,58	0,69
200	138	2,65	354	1,18	0,83	0,96
318	358	3,51	868	1,13	0,82	0,94
500	932	5,24	2.062	1,02	0,79	0,89
657	1.375	4,63	3.244	0,78	0,63	0,71
783	1.775	4,37	4.343	0,76	0,62	0,69
1.000	2.813	4,20	10.479	1,18	0,93	1,05
Médio		**3,77**		**0,99**	**0,74**	**0,85**
Máximo		**5,24**		**1,18**	**0,93**	**1,05**

Tabela 3.10: Resultados da avaliação da vizinhança do operador *swap* na GPU GTX-550.

GTX-780 ($p = 2.304$)

	sem EDL		**com EDL**			
	tempo(μs)	$S_A(p)$	**tempo(μs)**	$S_A(p)$		
n	**CFV**	**CF/GV**	**CFAV**	**CFA/GV**	**CFV/GA**	**CF/GAV**
100	28	2,00	73	1,09	0,94	1,24
200	127	6,35	324	1,60	1,72	2,19
318	354	12,21	849	1,75	2,18	2,72
500	979	16,88	2.068	1,76	2,28	2,84
657	1.590	19,63	3.407	1,56	2,24	2,78
783	2.118	16,81	4.333	1,32	1,78	2,15
1.000	3.177	16,05	10.197	1,98	2,82	3,47
Médio		**12,85**		**1,58**	**1,99**	**2,48**
Máximo		**19,63**		**1,98**	**2,82**	**3,47**

Tabela 3.11: Resultados da avaliação da vizinhança do operador *swap* na GPU GTX-780.

3.6.4 Desempenho das Estratégias

A seguir são analisados os dados referentes às melhores estratégias que usam EDL para realizar a avaliação da vizinhança, que são CFAV, CFA/GV, CFV/GA e CF/GAV. A Tabela 3.12 apresenta os dados relativos à execução dos componentes V e A na GPU GTX-550, enquanto os resultados referentes à GTX-780 são mostrados pela Tabela 3.13. Nas tabelas, a coluna n indica o tamanho da instância analisada, seguida por duas seções com quatro colunas cada e terminando com o ganho referente à alocação do componente V

na CPU em relação à sua atribuição na GPU. Em cada uma das duas seções, a primeira coluna indica o tempo de execução do componente V alocado na arquitetura indicada pelo rótulo da seção, usando o prefixo C para indicar CPU e G para GPU. A segunda coluna (X ou Y) representa uma máscara que indica em qual arquitetura a atualização da EDL (A) foi mais bem sucedida, seguida pelo tempo de execução correspondente (XA ou YA), que já inclui o custo referente a eventuais transferências de dados. A última coluna mostra o tempo total de execução da busca em vizinhança na configuração considerada, representando a soma dos tempos relacionados conforme indicado pelo rótulo. Os valores relativos à última coluna, ganho relativo, foi obtido pela razão entre as colunas CV+XA e GV+YA. Portanto, um valor de ganho menor que 1 indica que a estratégia considerada tem melhor resultado quando V está alocado na CPU em vez de na GPU. Todos os tempos de execução são dados em microssegundos (μs).

Analisando os resultados para a GTX-780, quando o componente V é fixado na CPU a execução da atualização da EDL (A) também na CPU somente é vantajosa para as três instâncias de menor porte. Nos demais casos, a execução de A na GPU é mais promissora. Quando a avaliação da vizinhança (V) é fixada na GPU é sempre mais vantajoso realizar a atualização da EDL nesta mesma arquitetura. A vizinhança 2*opt* se mostra com os menores ganhos relativos em relação à CPU quando o componente V é alocado na GPU, não ultrapassando o limite de 1,72. No entanto, as buscas associadas aos demais operadores conseguem obter ganhos de pouco mais de 3 vezes para a maior instância. Para a GPU GTX-550, a atualização da EDL (A) é sempre mais eficiente se for realizada na CPU, independente de onde a avaliação da vizinhança (V) esteja alocada. Neste dispositivo, mais uma vez, os menores ganhos relativos também foram obtidos pela vizinhança do 2*opt*.

O fato da plataforma equipada com a GPU GTX-550 não conseguir tirar proveito do paralelismo da GPU durante a atualização da EDL está relacionada com dois fatores. O primeiro tem a ver com largura de banda das vias de acesso à memória global desse dispositivo, que é quase três vezes menor do que na GTX-780[8]. Isso significa, que na GTX-780 os dados da EDL chegam mais rapidamente à memória compartilhada dos multiprocessadores, para onde são carregados durante o processamento. O outro fator é que a GTX-550 possui uma quantidade menor de multiprocessadores e de núcleos de processamento (*cores*) do que a GTX-780 (ver Tabela 3.1), justificando porque este último dispositivo desempenha melhor nos cenários que envolvem processamento na GPU.

[8]Segundo as especificações da NVIDIA, a largura de banda de acesso à memória global da GPU GTX-780 é de 288,4 GBytes/s, enquanto que na GTX-550 é de 98,4 GBytes/s.

GTX-550 - Vizinhança 2*opt*

n	Componente V na CPU				Componente V na GPU				Ganho
	CV	X	XA	CV+XA	GV	Y	YA	GV+YA	Relativo
100	29	C	34	63	78	C	56	134	0,47
200	101	C	139	240	174	C	198	372	0,65
318	198	C	347	545	365	C	478	843	0,65
500	438	C	952	1.390	686	C	1.269	1.955	0,71
657	817	C	1.616	2.433	1.095	C	2.159	3.254	0,75
783	1.077	C	2.372	3.449	1.488	C	3.126	4.614	0,75
1.000	1.775	C	3.706	5.481	2.524	C	4.936	7.460	0,73

GTX-550 - Vizinhança *oropt*1

n	Componente V na CPU				Componente V na GPU				Ganho
	CV	X	XA	CV+XA	GV	Y	YA	GV+YA	Relativo
100	57	C	34	91	149	C	56	205	0,44
200	160	C	139	299	326	C	198	524	0,57
318	388	C	347	735	593	C	478	1.071	0,69
500	930	C	952	1.882	1.127	C	1.269	2.396	0,79
657	1.762	C	1.616	3.378	1.874	C	2.159	4.033	0,84
783	2.482	C	2.372	4.854	2.550	C	3.126	5.676	0,86
1.000	7.734	C	3.706	11.440	3.926	C	4.936	8.862	1,29

GTX-550 - Vizinhança *swap*

n	Componente V na CPU				Componente V na GPU				Ganho
	CV	X	XA	CV+XA	GV	Y	YA	GV+YA	Relativo
100	46	C	33	79	35	C	55	90	0,88
200	215	C	139	354	102	C	199	301	1,18
318	521	C	347	868	289	C	478	767	1,13
500	1.109	C	953	2.062	752	C	1.270	2.022	1,02
657	1.573	C	1.671	3.244	1.929	C	2.213	4.142	0,78
783	1.975	C	2.368	4.343	2.578	C	3.120	5.698	0,76
1.000	6.740	C	3.739	10.479	3.940	C	4.968	8.908	1,18

Tabela 3.12: Comparação entre as melhores estratégias baseadas em CPU e GPU que envolvem o uso de EDL executadas na GPU GTX-550.

Frente a esses resultados pode-se concluir que, quanto maior a carga de trabalho, maior é o ganho obtido com o uso de GPUs. Apesar do cômputo da EDL reduzir a complexidade da avaliação da vizinhança para $O(n^2)$, os custos relacionadas com sua atualização e movimentação não trazem benefícios mais significativos do que as estratégias paralelas que não a utilizam. Isto pode ser deduzido pelos *speedups* de até 410 vezes obtidos pelas estratégias que não usam EDL, enquanto que nos métodos baseados nessa estrutura auxiliar os ganhos são limitados, nas instâncias testadas, a aproximadamente 3 vezes.

Uma vez que as abordagens se mostraram mais eficientes quando executadas na GPU GTX-780, a seguir são mostrados os resultados dos tempos de execução nesse dispo-

GTX-780 - Vizinhança 2*opt*

n	Componente V na CPU				Componente V na GPU				Ganho
	CV	X	XA	CV+XA	GV	Y	YA	GV+YA	Relativo
100	24	C	28	52	36	G	41	77	0,68
200	102	C	129	231	65	G	114	179	1,29
318	198	C	308	506	111	G	240	351	1,44
500	461	G	754	1.215	182	G	578	760	1,60
657	752	G	1.247	1.999	259	G	951	1.210	1,65
783	1.063	G	1.785	2.848	345	G	1.370	1.715	1,66
1.000	1.738	G	2.831	4.569	501	G	2.155	2.656	1,72

GTX-780 - Vizinhança *oropt*-1

n	Componente V na CPU				Componente V na GPU				Ganho
	CV	X	XA	CV+XA	GV	Y	YA	GV+YA	Relativo
100	50	C	28	78	67	G	41	108	0,72
200	182	C	129	311	129	G	114	243	1,28
318	427	C	308	735	220	G	240	460	1,60
500	1.001	G	754	1.755	336	G	578	914	1,92
657	1.719	G	1.247	2.966	480	G	951	1.431	2,07
783	2.419	G	1.785	4.204	626	G	1.370	1.996	2,11
1.000	6.776	G	2.831	9.607	897	G	2.155	3.052	3,15

GTX-780 - Vizinhança *swap*

n	Componente V na CPU				Componente V na GPU				Ganho
	CV	X	XA	CV+XA	GV	Y	YA	GV+YA	Relativo
100	42	C	31	73	17	G	42	59	1,24
200	195	C	129	324	34	G	114	148	2,19
318	514	G	319	833	71	G	241	312	2,67
500	1.224	G	755	1.979	151	G	578	729	2,71
657	1.793	G	1.247	3.040	273	G	951	1.224	2,48
783	2.113	G	1.784	3.897	647	G	1.370	2.017	1,93
1.000	6.514	G	2.830	9.344	784	G	2.154	2.938	3,18

Tabela 3.13: Comparação entre as melhores estratégias baseadas em CPU e GPU que envolvem o uso de EDL executadas na GPU GTX-780.

sitivo de cada uma das estratégias. As Tabelas 3.15, 3.16 e 3.14 exibem, respectivamente, os dados relativos às vizinhanças 2*opt*, *oropt*-1 e *swap*. Os resultados para as vizinhanças *oropt*-2 e *oropt*-3 não foram apresentados por serem muito similares aos de *oropt*-1, conforme já argumentado. As tabelas são divididas em três seções. Na primeira, a coluna n apresenta o número de clientes de cada instância testada. Na segunda seção, são exibidos os tempos de execução, em microssegundos (μs), de cada uma das seis estratégias considerando a vizinhança em evidência. Na última seção, a primeira coluna CFV $\times$ CF/GV apresenta o *speedup* absoluto entre entre o método CPU sem EDL e o método GPU mais rápido. Por sua vez, a última coluna CFAV $\times$ CF/GV, indica o ganho em tempo de execução do método CPU baseado em EDL em relação a CF/GV.

Dentre as três estruturas de vizinhança apresentadas, pode-se verificar que o operador *swap* é que menos se beneficia com o uso da EDL. Isto porque este operador realiza um número de operações constante em cada movimento, de forma que a complexidade da avaliação de sua vizinhança não é afetada pelo uso da EDL como nos demais operadores. Contudo, quando a EDL é considerada na CPU, seu uso se mostra vantajoso também nos métodos orientados para GPU.

n	**GTX-780 – *swap*** **Estratégias**						$S_A(p)$ **CFV × CF/GV**	**Ganho CFAV × CF/GV**
	CFV	**CF/GV**	**CFAV**	**CFA/GV**	**CFV/GA**	**CF/GAV**		
100	28	14	73	67	78	59	2,00	5,21
200	127	20	324	203	188	148	6,35	16,20
318	354	29	849	484	390	312	12,21	29,28
500	979	58	2.068	1.172	906	729	16,88	35,66
657	1.590	81	3.407	2.183	1.520	1.224	19,63	42,06
783	2.118	126	4.333	3.281	2.431	2.017	16,81	34,39
1.000	3.177	198	10.197	5.143	3.614	2.938	16,05	51,50
						Média	**12,85**	**30,61**

Tabela 3.14: Desempenho da busca em vizinhança *swap* em cada uma das estratégias de implementação na GPU GTX-780.

Já o operador 2*opt* é o que melhor tira proveito do paralelismo disponibilizado pela GPU, atingindo as maiores taxas de *speedup* quando a versão sem EDL é comparada com o método correspondente na CPU. Isso ocorre porque a CPU precisa lidar serialmente com a inversão das sub-rotas inerentes à operação 2*opt*, enquanto na GPU tal inversão é realizada paralelamente. Mesmo as sub-rotas tendo comprimentos distintos, a GPU lida bem com esse fato, pois os blocos encarregados dos laços mais curtos vão sendo concluídos à medida que os associados a sub-rotas mais longas vão sendo processados. Quando a EDL é considerada, a CPU tira vantagem da redução de complexidade da avaliação da vizinhança, principalmente, por poder contar com sua larga memória *cache* para armazenar a EDL. Mesmo assim, os métodos GPU com EDL ainda conseguem imprimir algum ganho, principalmente se a atualização da estrutura também é processada no dispositivo.

Com a vizinhança do operador *oropt-k* a CPU enfrenta um problema similar ao ocorrido com 2*opt* quando a EDL não é considerada, mas dessa vez relacionada com a relocação de segmentos. Além disso, a cardinalidade dessa vizinhança é duas vezes maior que de 2*opt* devido à não comutatividade da operação, o que explica o desempenho inferior desse primeiro quando executado paralelamente na GPU. Nos métodos envolvendo EDL o comportamento também é similar, onde a GPU também consegue contribuir com a aceleração do método quando equiparado com a versão para CPU correspondente.

n	GTX-780 – 2*opt* Estratégias						$S_A(p)$	Ganho
	CFV	**CF/GV**	**CFAV**	**CFA/GV**	**CFV/GA**	**CF/GAV**	**CFV × CF/GV**	**CFAV × CF/GV**
100	531	17	52	82	95	77	31,24	3,06
200	4.258	36	231	235	220	179	118,28	6,42
318	13.102	70	506	497	429	351	187,17	7,23
500	52.426	191	1.300	1.197	936	760	274,48	6,81
657	130.607	359	2.243	2.046	1.506	1.210	363,81	6,25
783	220.227	647	3.162	2.859	2.130	1.715	340,38	4,89
1.000	489.296	1.191	5.424	4.863	3.332	2.656	410,83	4,55
						Média	**246,60**	**5,60**

Tabela 3.15: Desempenho da busca em vizinhança 2*opt* em cada uma das estratégias de implementação na GPU GTX-780.

n	GTX-780 – *oropt*-1 Estratégias						$S_A(p)$	Ganho
	CFV	**CF/GV**	**CFAV**	**CFA/GV**	**CFV/GA**	**CF/GAV**	**CFV × CF/GV**	**CFAV × CF/GV**
100	470	19	78	113	126	108	24,74	4,11
200	2.990	52	311	299	284	243	57,50	5,98
318	9.319	140	735	606	538	460	66,56	5,25
500	44.750	335	1.840	1.351	1.090	914	133,58	5,49
657	111.171	762	3.210	2.267	1.727	1.431	145,89	4,21
783	205.228	1.986	4.518	3.140	2.411	1.996	103,34	2,27
1.000	473.436	3.142	10.462	5.259	3.728	3.052	150,68	3,33
						Média	**97,47**	**4,38**

Tabela 3.16: Desempenho da busca em vizinhança *oropt*-1 em cada uma das estratégias de implementação na GPU GTX-780.

Com uma análise global dos resultados anteriores, é fácil concluir que o método CF/GV, sem o uso de EDL, é a estratégia que oferece o maior benefício em termos de *speedup* em todas as vizinhanças. Observando a penúltima coluna das tabelas, também é possível inferir que o desempenho do algoritmo melhora à medida que o tamanho do problema aumenta, sugerindo que o uso de GPU para problemas ainda maiores é indicado. O segundo melhor método, CF/GAV, também envolve GPU demonstrando que esse dispositivo é consistentemente uma ferramenta profícua para a redução do tempo computacional da busca local.

Apesar do método de pré-avaliação de movimentos ser bastante eficiente na plataforma CPU, sua adaptação para a arquitetura GPU não apresentou uma aceleração proporcional ao potencial paralelo disponibilizado. Entretanto, a utilização de técnicas que exploram as características intrínsecas da arquitetura GPU, combinadas com seu potencial paralelo, foram suficientes para promover a aceleração da avaliação de vizinhanças a níveis bastante satisfatórios. No capítulo seguinte, a melhor metodologia de implementação aqui avaliada será posta à prova no contexto de alguns métodos de busca local e de algoritmos heurísticos. Além disso, dois novos métodos para auxiliar heurísticas orien-

tadas para busca local serão introduzidos e seus respectivos desempenhos em termos de qualidade de solução, aceleração e escalabilidade serão avaliados.

Capítulo 4

Algoritmos Propostos

Os ambientes CPU/GPU proporcionam a promessa de uma plataforma de alto poder computacional paralelo a um custo relativamente baixo. Como já demonstrado, a exploração desses ambientes por algoritmos paralelos deve ser planejada criteriosamente levando em consideração diversos aspectos de desempenho inerentes à essa arquitetura híbrida. Nesse contexto, a mera implementação de métodos consagrados para plataforma CPU pode não ser suficiente para justificar o esforço de projeto e desenvolvimento necessários, ou os ganhos de performance obtidos. De fato, é preciso que os métodos explorem adequadamente os numerosos requisitos relacionados com o modelo de execução e as características intrínsecas da plataforma, sob pena de não serem recompensados com um desempenho proporcional ao esforço empenhado.

Este capítulo introduz um novo procedimento de busca local paralela especialmente concebido para ambientes heterogêneos compostos por CPU e múltiplas GPUs. Esse método representa uma abordagem para a exploração concorrente de múltiplas estruturas de vizinhança empregando os recursos conjuntos de CPU e GPUs. Além disso, é apresentada uma nova metodologia de seleção em vizinhança que busca tirar proveito da massiva disponibilidade de resultados proporcionada pelos métodos executados em GPU. Para validar e demonstrar a eficácia dessas propostas, são conduzidos diversos experimentos e análise de resultados em termos de qualidade de solução, desempenho e escalabilidade.

4.1 O Algoritmo DVND

O *Distributed Variable Neighborhood Descent* DVND é inspirado no método VND (*Variable Neighborhood Descent*) concebido por [Mladenović & Hansen, 1997]. A ideia por trás do VND é que quando uma solução representa um ótimo local com relação a uma deter-

minada estrutura de vizinhança, o mesmo pode não necessariamente ocorrer para outras estruturas. Desta forma, o VND efetua a busca utilizando uma sequência predeterminada de vizinhanças, fazendo a mudança para uma nova estrutura sempre que um ótimo local é encontrado. Quando isso ocorre, o processo é reiniciado a partir da primeira vizinhança, iterando até que que nenhuma melhoria na qualidade da solução corrente possa ser obtida. O método VND é apresentado pelo Algoritmo 2.

Algoritmo 2: Pseudocódigo da busca local VND.

```
1  algoritmo VND(s)
2      Inicializa estruturas de dados locais;
3      N ← {N_1, N_2, ..., N_max};
4      k ← 1;
5      enquanto k ≤ max faça
6          m ← seleciona_movimento(s, N_k);
7          s' ← aplica_movimento(s, m);
8          se f(s') < f(s) então
9              s ← s';
10             k ← 1;
11             Atualiza estruturas de dados locais;
12         senão
13             k ← k + 1;
14         fim
15     fim
16     retorna s;
17 fim
```

Em alguns trabalhos, a exploração das vizinhanças é feita em ordem aleatória. Nesses casos, o método é denominado RVND (*Randomized* VND), o qual busca evitar a exploração das vizinhanças em uma ordem específica, o que pode conduzir a melhores resultados [Souza *et al.*, 2010]. De fato, o RVND é o núcleo de muitos algoritmos eficientes de busca da literatura recente [Silva *et al.*, 2012, Penna *et al.*, 2013, Subramanian *et al.*, 2013, Coelho *et al.*, 2016] sendo tipicamente aplicado juntamente com algum método de diversificação como, por exemplo, as meta-heurísticas *Variable Neighborhood Search* (VNS), GRASP ou ILS [Toth & Vigo, 2001].

O Algoritmo 3 apresenta o pseudocódigo do RVND, que recebe como parâmetros de entrada uma solução inicial s e um conjunto de estruturas de vizinhança N. O método começa inicializando eventuais estruturas de dados locais e um conjunto C de vizinhanças candidatas à seleção (linhas 2 e 3). A cada iteração do laço principal (linhas 4 a 15), uma estrutura de vizinhança N_k é selecionada aleatoriamente do conjunto C, de onde um movimento é selecionado segundo um critério específico, para depois ser aplicado sobre

Algoritmo 3: Pseudocódigo da busca local RVND.

```
1  algoritmo RVND(s, N)
2      Inicializa estruturas de dados locais;
3      C ← N;
4      enquanto |C| > 0 faça
5          N_k ← Seleciona aleatoriamente uma vizinhança de C;
6          m ← seleciona_movimento(s, N_k);
7          s' ← aplica_movimento(s, m);
8          se f(s') < f(s) então
9              s ← s';
10             C ← N;
11             Atualiza estruturas de dados locais;
12         senão
13             C ← C − { N_k };
14         fim
15     fim
16     retorna s;
17 fim
```

a solução corrente da busca local (linhas 5 a 7). Caso a solução resultante s' melhore o custo da solução corrente, esta é substituída, o conjunto de vizinhanças candidatas reinicializado, e as estruturas de dados locais atualizadas (linhas 8 a 11). Caso contrário, a vizinhança N_k se encontra em um ótimo local em relação à solução corrente s, sendo removida de C para não mais ser considerada nas próximas iterações (linhas 12 a 14). O método itera até que nenhuma vizinhança possa melhorar a qualidade da solução corrente s quando, finalmente, a melhor solução encontrada é retornada (linha 16).

Uma vez que o procedimento de busca local tipicamente representa o componente computacionalmente mais intensivo da meta-heurística, a sua paralelização tem o potencial de reduzir significativamente o tempo global de execução do algoritmo. Nesse contexto, a paralelização do RVND é uma tarefa desafiadora em dois aspectos principais. O primeiro é que a ordem de processamento da busca em vizinhança pode influenciar diretamente no desempenho global do algoritmo, uma vez que as instâncias do problema alvo podem ser sensíveis à sequência de aplicação dos operadores de movimentos empregados durante a busca (linha 5 do Algoritmo 3). O segundo aspecto está relacionado com o próprio método, onde existe uma dependência de dados previamente processados após cada iteração, já vez que a solução atual é sempre considerada a cada etapa da busca (linhas 6 e 7 do Algoritmo 3).

O método DVND aqui proposto, lida com esses dois aspectos da seguinte forma. Em relação ao primeiro tópico, o escalonamento de *threads* e blocos realizado pela GPU

e as diferenças de tempo de processamento de cada busca em vizinhança desempenham um papel semelhante ao do mecanismo de seleção aleatória de vizinhança empenhado pelo RVND. Neste sentido, a questão relacionada com a ordem de processamento das vizinhanças é naturalmente resolvida no DVND, mesmo que os tempos de processamento de cada busca na vizinhança sejam similares. O segundo aspecto é resolvido por meio da inclusão de uma estrutura de dados denominada *histórico* que armazena a melhor solução corrente de cada vizinhança, juntamente com a melhor solução global encontrada durante a busca, o que implementa um mecanismo de compartilhamento de soluções entre todos os processos. Não há garantia de que cada diferente processo de busca esteja utilizando a melhor solução conhecida num determinado instante. Portanto, após cada busca local, esses processos serão reiniciados com a melhor solução do *histórico*. Vale ressaltar que, a pesquisa realizada para elaboração desse trabalho, sugere que o DVND é a primeira implementação paralela do VND. Os Algoritmos 4 e 5 apresentam o pseudocódigo do DVND.

Algoritmo 4: Pseudocódigo da tarefa mestre do DVND.

```
1  algoritmo dvnd(s)
2      g ←Número de dispositivos GPU no sistema;
3      N ← { N_1, N_2, ..., N_max };
4      P ←Partição de N, tal que |P| = g ;  // ex:  P = { { N_1, N_2, N_3 }, { N_4, N_5 } }
5      H ← { s };
6      para i=1 até g faça
7          Excuta assincronamente dvnd_thread (H, P_i);
8      fim
9      Aguarda a finalização da execução das threads;
10     s ← arg min_{s'∈H} { f(s') };
11     return s;
12 fim
```

O DVND é um método de busca local que foi projetado para ser composto por um conjunto de tarefas executadas em paralelo, empregando um modelo mestre-trabalhador. O Algoritmo 4 mostra o algoritmo da tarefa mestre, que incia detectando o número de GPUs presentes no sistema (linha 2) e inicializando o conjunto N de estruturas de vizinhança a serem exploradas (linha 3). Esse conjunto de vizinhanças é então particionado visando obter um balanceamento da carga de trabalho entre os dispositivos GPU (linha 4). Em seguida a estrutura H, que representa o histórico, é inicializada com a solução de entrada s do DVND (linha 5) e, na sequência, uma *thread* CPU é lançada para gerenciar cada dispositivo executando o procedimento descrito pelo Algoritmo 5 (linhas 6 a 8). Cada procedimento `dvnd_thread` realiza uma busca independente por soluções do

PML em sua respectiva GPU a partir de um subconjunto de vizinhanças $P_i \subseteq N$. Apesar da execução autônoma, cada procedimento `dvnd_thread` compartilha o mesmo *histórico* global H, o que permite que as tarefas troquem informações sobre as soluções encontradas durante suas respectivas buscas implementando, portanto, um mecanismo de cooperação. O algoritmo então aguarda até que todas as tarefas sejam completadas (linha 9), coleta a melhor solução do histórico e a retorna encerrando a busca (linhas 10 e 11).

Algoritmo 5: Pseudocódigo da tarefa trabalhadora do DVND.

```
 1 procedimento dvnd_thread(H, N)
 2     s ← arg min_{s'∈H} { f(s') };
 3     para cada N_k ∈ N faça
 4         s_k ← s;
 5         Lança assincronamente kernel_k(s_k);
 6     fim
 7     Q ← { };
 8     enquanto |Q| < |N| faça
 9         k ← aguarda_kernel();
10         m ← seleciona_movimento(s_k, N_k);
11         s ← aplica_movimento(s_k, m);
12         se f(s) < f(s_k) então
13             H ← H ∪ { s };
14         fim
15         s ← arg min_{s'∈H} { f(s') };
16         se f(s) < f(s_k) então
17             s_k ← s;
18             Lança assincronamente kernel_k(s_k);
19         senão
20             Q ← Q ∪ {(k, s_k)};
21         fim
22         para cada (i, s_i) ∈ Q faça
23             se f(s) < f(s_i) então
24                 s_i ← s;
25                 Lança assincronamente kernel_i(s_i);
26                 Q ← Q − { (i, s_i) };
27             fim
28         fim
29     fim
30 fim
```

O Algoritmo 5 descreve o mecanismo distribuído implementado pelas tarefas trabalhadoras do DVND. O algoritmo inicia recebendo como parâmetros de entrada uma referência para o histórico global H e seu próprio subconjunto de vizinhanças N a ser explorado. Uma solução s é então selecionada de H (linha 2) e utilizada como solução inicial pelos *kernels* de avaliação de vizinhança gerenciados pelo algoritmo. Para cada

vizinhança $N_k \in N$, existe um $kernel_k$ responsável por sua avaliação na GPU, os quais são lançados assincronamente pelo laço implementado nas linhas de 3 a 6. A ideia que motiva o DVND é lançar buscas em vizinhança paralelamente até que todas alcancem seus respectivos ótimos locais. Desta forma, Q representa o conjunto de todas as vizinhanças que estão correntemente em um ótimo local com respeito a uma solução específica (linha 7). A cada iteração do laço principal (linhas 8 a 29), o método aguarda até que algum *kernel* conclua sua execução na GPU, retornando o índice de identificação k correspondente (linha 9). Então, o algoritmo seleciona um movimento a partir dos resultados retornados pelo $\texttt{kernel}_k$ s e o aplica sobre a solução s_k associada ao *kernel* (linhas 10 e 11). Se a solução s melhora a solução corrente s_k, ela é submetida ao histórico (linhas 12 a 14), compartilhando essa solução entre as demais tarefas do DVND. Em seguida, s_k é confrontada com a melhor solução do histórico, agora atribuída a s (linhas 15 e 16). Se s possui melhor custo que a solução corrente do $\texttt{kernel}_k$, esta última é atualizada e ocorre um novo lançamento assíncrono de *kernel* para a vizinhança N_k (linhas 17 a 18). Caso contrário, uma tupla (k, s_k) é armazenada em O indicando que um ótimo local s_k para a vizinhança N_k foi atingido (linhas 19 a 21). O método então itera sobre cada vizinhança N_i que está correntemente em um ótimo local (linhas 22 a 28). A cada iteração, se a solução s apresenta melhor custo que o ótimo local s_i associada à vizinhança N_i, a solução s_i é atualizada, uma busca na vizinhança de s_i é lançada, sendo posteriormente removida da lista de ótimos locais Q (linhas 23 a 27). O processo continua até que todas as vizinhanças atinjam um ótimo local (linha 8) e a melhor solução encontrada estará armazenada no histórico H.

Na busca local, o esforço computacional está tipicamente concentrado na verificação da viabilidade da função objetivo e na avaliação de vizinhanças [Schulz, 2013]. No caso do DVND, o maior empenho computacional é dedicado a essa última tarefa, motivo pelo qual ela é executada em GPU. Como já enfatizado, os métodos orientados para sistemas híbridos CPU/GPU devem considerar a transferência de dados como um fator limitante do desempenho. Para lidar com esse fato e tentar tirar proveito da grande quantidade de dados disponibilizados pela GPU, é proposto um novo método de seleção de movimentos em vizinhanças que busca tanto reduzir a transferência de dados, quanto a computação de *kernels*, como explicado a seguir.

4.2 Seleção *Multi Improvement*

Conforme apresentado no capítulo anterior, depois de construir e transferir os dados relativos à solução corrente para a GPU, as *threads* do *kernel* que processa uma vizinhança são lançadas assincronamente numa grade de computação bidimensional. Nessa grade, cada linha corresponde a um bloco e cada *thread* processa um único movimento da vizinhança, como ilustrado pela Figura 4.1.

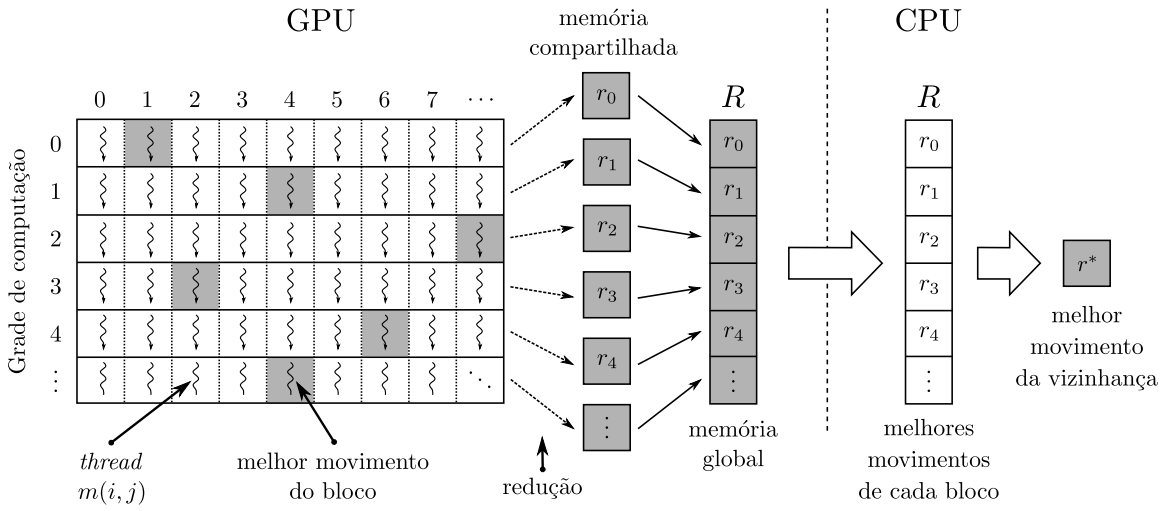


Figura 4.1: Avaliação da vizinhança realizada na GPU.

O esquema ilustrado na figura indica que, após o cômputo das *threads* de um mesmo bloco, o *kernel* realiza uma redução de minimização para encontrar o movimento que traz a maior redução de custo da solução base da vizinhança. Depois que todos os blocos são processados, o resultado da execução do *kernel* consiste em um vetor R, residente na memória global. Esse vetor, que contém o melhor movimento de cada bloco, é em seguida transferido para a memória principal da CPU, onde a seleção de um movimento é realizada. Cada elemento $r_k \in R$ pode ser visto como uma tupla (m, i, j, c), onde $i, j \in \mathbb{N}$ representam os operandos do operador de movimento m; $c \in \mathbb{Z}$ é o valor de ganho de custo correspondente ao movimento $m(i, j)$, onde $m \in \{\, swap, 2opt, oropt1, oropt2, oropt3 \,\}$, um dos cinco operadores de vizinhança para o MLP definidos na Seção 3.3.

Uma análise um pouco mais cuidadosa nesse esquema traz à tona uma questão interessante: porque não realizar outra redução na GPU para encontrar o movimento que obtém o maior ganho para o custo da solução? A resposta para essa questão possui alguns desdobramentos. Após a redução, o melhor movimento de cada bloco reside na memória compartilhada, portanto somente é acessível pelas *threads* pertencentes a esse bloco. Desta forma, para consolidar os dados da avaliação de vizinhança é necessário

realizar a transferência dos dados relativos a cada bloco para a memória global. Uma nova redução envolveria, numa abordagem direta, o lançamento de um novo *kernel* que faria a carga do vetor R a partir da memória global e realizaria a seleção de um movimento. Contudo, as implicações desse método envolvem o *overhead* associado ao lançamento do novo *kernel*; o tráfego adicional na já concorrida via de acesso entre a memória principal e o multiprocessador; e uma barreira de sincronização implícita entre os blocos.

Por outro lado, durante a busca local, os procedimentos que realizam a avaliação da vizinhança na GPU são massivamente executados produzindo grande volume de acessos à memória global, transferências entre CPU e GPU e lançamentos de *kernel*. Portanto, qualquer método que possa reduzir o número dessas operações é passível de contribuir para a aceleração global do algoritmo. Considerando que as operações de movimento são executadas em paralelo e seus resultados parciais (reduções) estão disponíveis na memória da GPU, uma nova estratégia foi desenvolvida para tirar proveito deste fato, em vez de realizar apenas uma seleção de movimento sobre R. Este método, além de reduzir o volume global de dados transferidos, acelera a busca em vizinhança, como explanado a seguir.

A busca em vizinhança é tipicamente realizada de acordo com três critérios principais [Talbi, 2009]:

- *best improvement*: o método examina exaustivamente a vizinhança e elege o movimento que produz a solução de melhor qualidade;
- *first improvement*: consiste em selecionar o primeiro movimento encontrado que incrementa a qualidade da solução;
- *seleção aleatória*: um movimento de melhora é selecionado aleatoriamente da vizinhança.

Este trabalho introduz um novo critério de seleção, o *multi improvement*, que consiste em selecionar não apenas um, mas múltiplos movimentos simultaneamente visando acelerar a convergência na direção de um ótimo local. Esta estratégia depende do conceito de *movimentos independentes* que são aqueles que podem ser aplicados simultaneamente sobre uma mesma solução base de forma que cada movimento afete um conjunto diferente de arestas da rota. Quando mais de um movimento é aplicado durante uma mesma busca em vizinhança, a qualidade da solução base pode sofrer um incremento mais significativo do que se apenas uma operação fosse empregada. Quando isso ocorre, a busca converge

mais rapidamente para o ótimo local, o que contribui para a redução do número de chamadas para os procedimentos de busca em vizinhança, o que contribui para a aceleração do método.

No entanto, encontrar o conjunto de movimentos independentes que maximize o ganho de custo sobre uma solução é uma tarefa complexa. De fato, trata-se de um outro problema de otimização combinatória conhecido como Problema do Conjunto Independente de Peso Máximo (PCIPM), do inglês *Maximum Weight Independent Set Problem* [Pardalos & Xue, 1994]. O PCIPM pode ser formulado como um grafo $W = (M, F)$ onde $M = \{ m_1, m_2, \ldots, m_q \}$ representa um conjunto de movimentos de uma vizinhança e $F = \{ (m_i, m_j) : m_i, m_j \in M, m_i \neq m_j \}$ indica o conjunto de arestas conectando os *movimentos conflitantes*, ou seja, os movimentos que não podem ser aplicados simultaneamente sobre uma mesma solução sem que um interfira no do outro. Para cada $m_i \in M$ está associado um ganho de custo $g(m_i) \in \mathbb{Z}$, que é obtido quando o movimento m_i é aplicado sobre a solução base. O objetivo do PCIPM é determinar o subconjunto $M' \subseteq M$ de movimentos independentes que maximiza o somatório dos pesos associados aos movimentos $m_i \in M'$. Uma vez que o PML é um problema de minimização, este trabalho considera a maximização da seguinte função objetivo: $\max \sum_{m_i \in M'} -g(m_i)$.

O custo de uma solução obtida pela indução da operação de aplicação de movimentos independentes, denotada por $\otimes$, é definido como se segue:

Proposição 1 *Seja s' uma solução obtida pela aplicação simultânea dos movimentos independentes $m_1 \otimes m_2 \otimes \cdots \otimes m_q$ da vizinhança de uma solução base s. O custo da solução resultante s' é dada por $f(s') = f(s) + g(m_1) + g(m_2) + \cdots + g(m_q)$.*

A independência entre dois movimentos de uma vizinhança pode ser determinada em tempo constante pelo cálculo das expressões lógicas relacionadas na Tabela 4.1. Cada célula da tabela apresenta a expressão utilizada para verificar a independência dos dois movimentos $m_1(i_1, j_1)$ e $m_2(i_2, j_2)$ correspondentes. Caso o resultado da expressão seja *verdadeiro*, os movimentos m_1 e m_2 são considerados independentes e o custo da solução resultante s' pode ser calculado como $f(s') = f(s) + g(m_1) + g(m_2)$. Caso contrário, os movimentos são conflitantes e não podem ser aplicados simultaneamente.

A ideia por trás das equações lógicas da Tabela 4.1 é que se dois movimentos não afetam as mesmas arestas da rota, eles podem ser aplicados concomitantemente sem conflitos. No caso da aplicação de duas operações *swap*, quatro clientes e oito arestas da rota são afetados. Portanto, se esses clientes estão alocados em posições que distam mais

$m_1(i_1, j_1)$	$m_2(i_2, j_2)$		
	swap	***2opt***	***oropt-k***
swap	$(\lvert i_1 - i_2\rvert > 1) \wedge$ $(\lvert i_1 - j_2\rvert > 1) \wedge$ $(\lvert j_1 - i_2\rvert > 1) \wedge$ $(\lvert j_1 - j_2\rvert > 1)$	$[(i_1 < i_2 - 1) \vee$ $(i_1 < i_2 - 1)] \wedge$ $[(j_1 < i_2 - 1) \vee$ $(j_1 > j_2 - 1)]$	$[j_1 < \min(i_2, j_2) - 1] \vee$ $[i_1 > \max(i_2, j_2) + k] \vee$ $[i_1 < \min(i_2, j_2) - 1] \wedge$ $j_1 > \max(i_2, j_2) + k]$
2opt	$[(i_2 < i_1 - 1) \vee$ $(i_2 < i_1 - 1)] \wedge$ $[(j_2 < i_1 - 1)] \vee$ $(j_2 > j_1 + 1)]$	$(j_1 < i_2 - 1) \vee$ $(i_1 > j_2 + 1) \vee$ $(j_2 > i_1 - 1) \vee$ $(i_2 > j_1 + 1)$	$(i_1 < \max(i_2, j_2) + k) \vee$ $(j_1 < \min(i_2, j_2) - 1)$
oropt-k	$[j_2 < \min(i_1, i_2) - 1] \vee$ $[i_2 > \max(i_1, i_2) + k] \vee$ $[(i_2 < \min(i_1, i_2) - 1) \wedge$ $(j_2 > \max(i_1, i_2) + k)]$	$(i_2 > \max(i_1, j_1) + k) \vee$ $(j_2 > \min(i_1, j_1) - 1)$	$[\max(i_1, j_1) + k_1 <$ $\min(i_2, j_2)] \vee$ $[\min(i_1, j_1) + k_1 >$ $\max(i_2, j_2) + k_2]$

Tabela 4.1: Expressões lógicas para verificação de independência entre movimentos.

de uma unidade um do outro as arestas que incidem sobre esses clientes são todas distintas e, portanto, os movimentos podem ser aplicados sem interferência mútua. A mesma ideia se aplica para os operadores 2*opt* e *oropt-k*, mas nesse caso o número de arestas afetadas depende dos respectivos operandos dos movimentos. A Figura 4.2 procura ilustrar esse conceito, onde as setas pontilhadas indicam as arestas que foram afetadas após cada movimento.

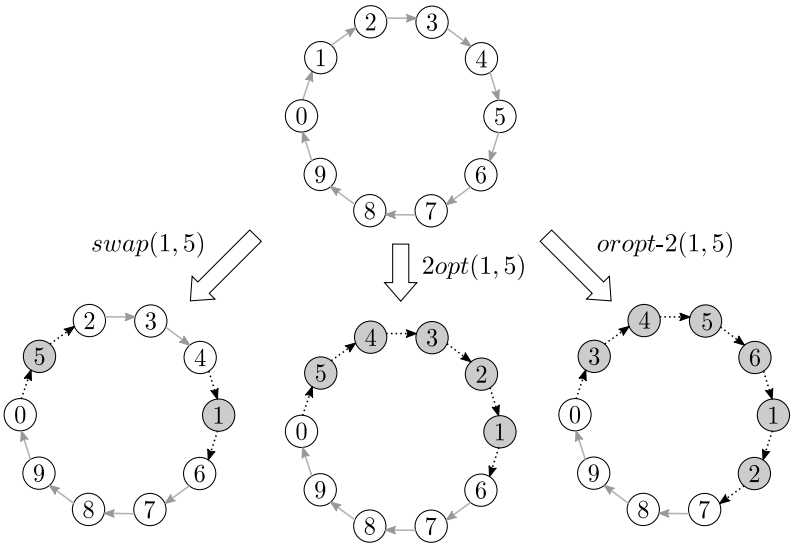


Figura 4.2: Exemplos de como as arestas de uma solução são afetadas pelos operadores de movimento *swap*, 2*opt* e *oropt-k* ($k = 2$).

A Figura 4.3 apresenta um exemplo de uso da estratégia *multi improvement* envolvendo dois pares de movimentos *swap* na vizinhança de uma solução s.

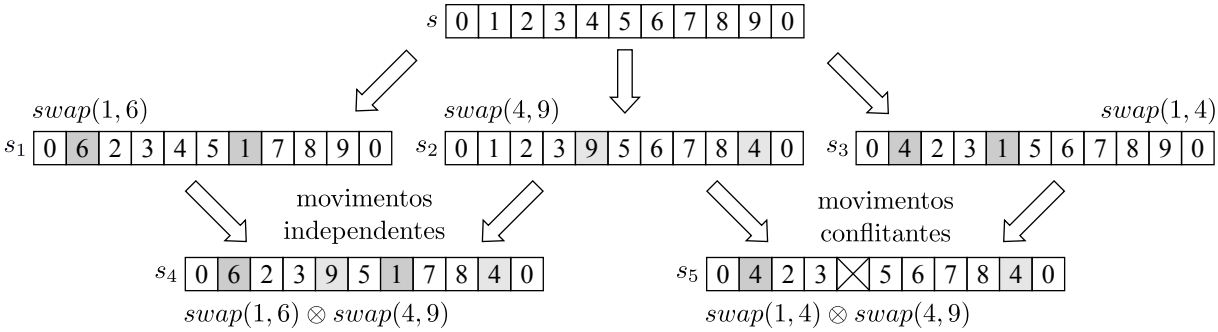


Figura 4.3: Exemplo da estratégia de seleção *multi improvement* em uma vizinhança *swap* envolvendo movimentos independentes e conflitantes.

Neste exemplo, os movimentos $swap(1,6)$ e $swap(4,9)$ são independentes, pois:

$$\begin{aligned} swap(1,6) \otimes swap(4,9) &\Rightarrow (|1-4|>1) \wedge (|1-9|>1) \wedge (|6-4|>1) \wedge (|6-9|>1) \\ &= (3>1) \wedge (8>1) \wedge (2>1) \wedge (3>1) \\ &= \mathrm{V} \wedge \mathrm{V} \wedge \mathrm{V} \wedge \mathrm{V} = \mathrm{V} \end{aligned}$$

enquanto $swap(1,4)$ e $swap(4,9)$ são conflitantes, já que:

$$\begin{aligned} swap(1,4) \otimes swap(4,9) &\Rightarrow (|1-4|>1) \wedge (|1-9|>1) \wedge (|4-4|>1) \wedge (|4-9|>1) \\ &= (3>1) \wedge (8>1) \wedge (0>1) \wedge (5>1) \\ &= \mathrm{V} \wedge \mathrm{V} \wedge \mathrm{F} \wedge \mathrm{V} = \mathrm{F} \end{aligned}$$

Dado que os movimentos $swap(1,6)$ e $swap(4,9)$ são independentes, o custo da solução s_4 pode ser obtido adicionando-se ao custo da solução s os ganhos de custo correspondentes a cada movimento, ou seja, $f(s_4) = f(s) + g(swap(1,6)) + g(swap(4,9))$. Esta situação não se aplica para os movimentos conflitantes $swap(1,4)$ e $swap(4,9)$, já que não é possível aplicá-los concomitantemente na solução s sem que um interfira com o outro.

Convém mencionar que o PCIPM é equivalente ao Problema da Clique com Peso Máximo (*Maximum Weight Clique Problem*) [Bomze *et al.*, 1999]. Ambos os problemas pertencem à classe NP-Difícil e alguns métodos exatos já foram publicados para sua resolução [Nemhauser & Trotter, 1975, Östergard, 1999]. Contudo, para que o mecanismo de busca local introduzido nesse estudo possa utilizar o *multi improvement* é necessário que o PCIPM possa ser resolvido em tempo polinomial, mesmo que a solução obtida não seja ótima. Para realizar essa tarefa é apresentada no Algoritmo 6 uma heurística gulosa rápida e simples para encontrar soluções para o problema.

Algoritmo 6: Pseudocódigo da heurística gulosa para resolução do PCIPM.

```
1  algoritmo PCIPM(R)
2      I ← { };
3      M ← { m' ∈ R | g(m') < 0 };
4      enquanto |M| > 0 faça
5          m ← arg min_{m'∈M} { g(m') };
6          M ← M − { m };
7          I ← I ∪ { m };
8          para cada m' ∈ M faça
9              se conflito(m, m') então
10                 M ← M − { m' };
11             fim
12         fim
13     fim
14     retorna I;
15 fim
```

O algoritmo PCIPM recebe como dado de entrada o conjunto R de movimentos retornados pela GPU após a execução de um *kernel* de avaliação em vizinhança, sendo executado pela função `seleciona_movimento` na linha 10 do Algoritmo 5. O método principia inicializando um conjunto I para armazenamento dos movimentos independentes (linha 2) e um conjunto M contendo apenas os movimentos de R que possibilitam a redução de custo da solução (linha 3). O laço principal (linhas 4 a 13) persiste enquanto houver movimentos em M a serem processados. A cada iteração, a heurística extrai de M o movimento m que representa a maior redução de custo da solução (linhas 5 a 7). Para o movimento independente m selecionado, o método verifica se existem movimentos remanescentes $m' \in M$ que conflitem com m. A cada conflito encontrado, o movimento m' correspondente é removido de M e, portanto, não será mais considerado nas iterações posteriores (linhas 8 a 12). Finalmente, quando não houver mais movimentos a serem processados, o algoritmo retorna o conjunto de movimentos independentes I encontrado (linha 14).

Apesar da heurística PCIPM não necessariamente encontrar o conjunto ótimo de movimentos independentes, a simples combinação de mais que um movimento produz um incremento mais significativo na qualidade da solução do que se apenas uma operação tivesse sido considerada. Como o algoritmo é guloso, existe a garantia de que o melhor movimento será sempre selecionado, o que faz com que o método, no pior caso, replique o comportamento do *best improvement*.

Qualquer procedimento de busca local está sujeito a ficar preso em ótimos locais

dependendo da solução de entrada utilizada. Por esta razão, esses métodos são usualmente validados dentro de uma meta-heurística que atua como um mecanismo de diversificação que pode conduzir o algoritmo a soluções de melhor qualidade e, eventualmente, à solução ótima. Na seção seguinte é apresentado um *framework* baseado na meta-heurística VNS que será utilizando nos experimentos executados neste capítulo.

4.3 *Framework* GVNS

Nesta seção é apresentada um *framework* paralelo orientado para a resolução de problemas de otimização. O algoritmo foi projetado tendo como base uma heurística sequencial bastante eficiente proposta em [Silva *et al.*, 2012] denominada GILS-RVND. Esse algoritmo sequencial foi inspirado na meta-heurística GRASP combinada com um mecanismo de busca local baseado no RVND. Na ocasião de sua publicação, o GILS-RVND apresentou os melhores resultados da literatura para o PML, com destaque para a redução significativa do tempo de execução, além da introdução de novas melhores soluções para o problema. A aceleração do algoritmo foi promovida pela mesma técnica de pré-avaliação de movimentos baseada na concatenação de segmentos descrita no capítulo anterior, o que facultou a abordagem de instâncias de maior porte. Já a obtenção de melhores soluções ficou ao encargo de um mecanismo autoadaptativo que intensifica a busca em torno de soluções promissoras utilizando várias estruturas de vizinhança.

O *framework* paralelo aqui proposto, denominado GVNS, foi projetado de acordo com o modelo mestre-trabalhador. O pseudocódigo da tarefa mestre é apresentado pelo Algoritmo 7, que recebe quatro parâmetros de entrada. O parâmetro p indica o número de tarefas trabalhadoras a ser criado e o conjunto $A = \{ \alpha_1, \ldots, \alpha_{max} \,|\, \alpha_i \in [0,1] \}$ é utilizado pelo construtor de soluções do método. Os dois últimos parâmetros, D_{max} e I_{max} representam, respectivamente, o número de iterações de diversificação e intensificação executado pelas tarefas trabalhadoras. A tarefa mestre começa inicializando uma solução global s^*, de forma que $f(s^*) = \infty$. Em seguida lança assincronamente p tarefas trabalhadoras, as quais realizam efetivamente a busca por soluções. Durante a busca, a solução global é compartilhada entre as *threads* trabalhadoras na memória principal da CPU e seu acesso e atualização são controlados por semáforo. Após o lançamento, a tarefa mestre cria uma barreira de sincronização e aguarda o término das tarefas trabalhadoras retornando, finalmente, o valor atualizado de s^*.

Como mencionado, o número de tarefas da aplicação paralela é controlado por

Algoritmo 7: Pseudocódigo da tarefa mestre do algoritmo GVNS.

```
1 procedimento gvns(p, A, D_max, I_max)
2 |  f(s*) ← ∞;
3 |  para t ← 1 até p faça
4 |  |  Lança assincronamente gvns_thread (A, I_max);
5 |  fim
6 |  Aguarda a conclusão das tarefas gvns_thread ();
7 |  retorna s*;
8 fim
```

p. Tipicamente, este parâmetro de entrada é definido como igual ao número de *cores* existentes na CPU visando explorar todos os recursos paralelos disponíveis. Para efeito de análise de desempenho, é importante ressaltar que ao executar o algoritmo, $p + 1$ tarefas são criadas: 1 tarefa mestre e p tarefas trabalhadoras. Contudo, após a carga da instância do problema e o lançamento das tarefas trabalhadoras, o custo computacional da tarefa mestre é desprezível, já que todos os recursos por ela alocados ficam disponíveis na memória principal da CPU.

O Algoritmo 8 apresenta o pseudocódigo executado pelas tarefas trabalhadoras. Cada tarefa implementa um mecanismo de diversificação baseado na meta-heurística GRASP, que é controlado pelo parâmetro global D_{max} (linhas 2 a 19). O acesso a esse parâmetro é controlado por semáforo (linhas 2 e 3), implementando um esquema que permite que as operações de diversificação de cada tarefa sejam alimentadas sob demanda, à medida que vão terminando. Uma solução s é então criada usando o procedimento de construção clássico do GRASP [Feo & Resende, 1995], cujo grau de aleatoriedade α é selecionado randomicamente do conjunto de entrada A (linhas 4 e 5). A solução s criada no passo anterior é então utilizada como solução inicial de um mecanismo de intensificação inspirado no VNS, cujo número de iterações é controlado pelo parâmetro de entrada I_{max} (linhas 6 a 15). A linha 10 executa uma chamada para um procedimento de busca local tendo como base a solução s. Após a busca, a solução resultante s é avaliada contra a solução corrente da tarefa trabalhadora s' (linha 11). Caso a busca tenha melhorado a qualidade da solução corrente, esta é atualizada e o procedimento de intensificação é reinicializado (linhas 12 e 13). Na sequência, a solução s' é perturbada usando o mecanismo *double-bridge*[1] [Martin *et al.*, 1991] a fim de gerar uma nova solução para a próxima iteração (linha 15). O laço interno persiste até que I_{max} iterações sem atualização da so-

[1] O *double-bridge* consiste em um mecanismo de perturbação que remove quatro arestas da solução e as reconecta de forma que a segunda e quarta sub-rotas sejam permutadas. Por exemplo, se $s = ABCDE$ é uma solução onde as letras indicam sub-rotas contíguas, a solução resultante de uma operação *double-bridge* seria $s = ADCBE$.

lução corrente sejam atingidas. Finalizada a fase de intensificação, a solução corrente s' é avaliada em relação à solução global s^*, que é atualizada caso esta apresente pior custo (linhas 17 a 19). O processo de diversificação prossegue até que toda a quota de iterações seja consumida pelas tarefas trabalhadoras.

Algoritmo 8: Pseudocódigo da tarefa trabalhadora do algoritmo GVNS.

```
1  procedimento gvns_thread (A, I_max)
2      enquanto D_max > 0 faça
3          D_max ← D_max − 1;
4          α ← Valor aleatório de A;
5          s ← construtor(α);
6          s' ← s;
7          i ← 0;
8          enquanto i < I_max faça
9              i ← i + 1;
10             s ← busca_local(s);
11             se f(s) < f(s') então
12                 s' ← s;
13                 i ← 0;
14             fim
15             s ← perturbação(s');
16         fim
17         se f(s') < f(s*) então
18             s* ← s';
19         fim
20     fim
21 fim
```

Vale ressaltar um aspecto interessante em relação ao Algoritmo 8. Apesar do trecho relativo ao procedimento de intensificação (linhas 6 a 16) ter suas iterações limitadas pelo parâmetro I_{max}, este procedimento frequentemente executa uma quantidade maior de iterações, dependendo do número de vezes que a solução s' é atualizada. O mesmo ocorre internamente com o procedimento RVND, que tem sua busca reinicializada sempre que consegue incrementar sua solução corrente. Este comportamento é interessante porque o algoritmo prolonga autonomicamente sua execução à medida que melhores soluções vão sendo encontradas.

Durante o projeto do GVNS, não houve intenção de desenvolver um algoritmo paralelo sofisticado mas, prioritariamente, implementar a abordagem sequencial do GILS-RVND e dispor de um referencial paralelo para comparações de desempenho com os demais algoritmos propostos nesta tese. Apesar disso, os experimentos computacionais realizados com GVNS apresentaram resultados que introduziram novas melhores soluções, além de

uma aceleração significativa do tempo de execução do algoritmo sequencial, especialmente para instâncias de grande porte. O levantamento bibliográfico realizado para a produção desse experimento sugere que esta foi a primeira heurística paralela para o PML.

4.4 Experimentos Computacionais

Os algoritmos utilizados nos experimentos deste capítulo foram implementados em C++ utilizando a biblioteca NVIDIA® CUDA™ versão 7.5 e compilados com o NVCC (NVIDIA *C Compiler*) para uma plataforma de 64-bits executando Linux Ubuntu 14.04. Foram utilizadas as bibliotecas POSIX *Threads* (*pthreads*) e PAPI 5.4 [Mucci *et al.*, 1999], para uma mensuração mais precisa dos tempos de execução em CPU. Os experimentos foram conduzidos em um computador com processador Intel® i7-4820K 3.70GHz, com 4 CPU *cores*, 10MB de *cache* L3 e 16GB memória RAM. Essa máquina também está equipada com duas GPUs NVIDIA® GeForce™ GTX-780, cada uma contendo 3GB de memória global e 12 multiprocessadores. Cada multiprocessador embute 192 GPU *cores*, totalizando 2.304 GPU cores em cada um dos dois dispositivos.

Os experimentos foram executados utilizando oito grupos de instâncias para o PML extraídas da literatura. O primeiro grupo, selecionado a partir de [Abeledo *et al.*, 2013], é composto por 15 instâncias com tamanho variando entre 42 e 107 clientes. O segundo grupo, contendo 9 unidades com dimensões de 70 a 532 clientes, foi selecionado da TSPLIB [Reinelt, 1991]. Os cinco grupos seguintes, com instâncias de tamanhos 10, 20, 50, 100, 200, 500 e 1.000 clientes, cada um com 20 unidades, foram obtidos do trabalho de [Salehipour *et al.*, 2011]. Com exceção do primeiro grupo, cuja função objetivo procura obter o *tour* (circuito) de menor custo, todos os demais grupos estão focados em minimizar o *path*, ou caminho. Essa discrepância, no entanto, é gerenciada naturalmente pelo algoritmo, como foi ressaltado na Seção 3.2. Finalmente, um último grupo com 7 casos foi criado com instâncias selecionadas dos grupos anteriores para fins de experimentações específicas, cujas dimensões variam de 100 a 1.000 clientes.

Uma vez que os métodos baseados em GPU desse trabalho envolvem o cálculo de distâncias entre clientes em tempo de execução, é importante mencionar que nos experimentos realizados foram consideradas apenas instâncias onde os dados de entrada são expressos em coordenadas Euclidianas. Portanto, foram selecionadas 15 das 23 instâncias testadas por [Abeledo *et al.*, 2013], e descartada do segundo grupo apenas uma instância de um total de 10. Todos os demais grupos de instâncias foram utilizados integralmente.

Um sumário de todos os grupos e suas respectivas características é apresentado na Tabela 4.2.

Grupo	Dimensões	Número de Instâncias	Fonte
1	$51 - 107$	15	Abeledo *et al.* (2013)
2	$70 - 439$	9	Reinelt (1991)
3	$10, 20, 50$	60	Salehipour *et al.* (2011)
4	100	20	Salehipour *et al.* (2011)
5	200	20	Salehipour *et al.* (2011)
6	500	20	Salehipour *et al.* (2011)
7	1.000	20	Salehipour *et al.* (2011)
8	$100 - 1.000$	7	Seleção dentre as fontes acima
	Total	**171**	

Tabela 4.2: Os grupos de instâncias do PML utilizadas nos experimentos computacionais, totalizando 171 unidades.

Os experimentos apresentados a seguir foram conduzidos em dois estágios. No primeiro, apresentados na Seção 4.4.1, são reportados e analisados os dados relativos ao desempenho de quatro procedimentos de busca local diferentes. No segundo estágio, contido na Seção 4.4.2, os melhores algoritmos de busca local são testados dentro do *framework* GVNS e os resultados, em termos de desempenho e qualidade da solução, são relatados.

4.4.1 Experimentos com Busca Local

Nesta seção são reportados experimentos realizados com quatro estratégias de busca local, duas inspiradas no RVND clássico e outras duas fundamentadas no algoritmo DVND. Os experimentos foram executados com objetivo de comparar o desempenho relativo entre esses dois métodos quando uma ou múltiplas GPUs estão disponíveis no sistema, e quando a metodologia de seleção em vizinhança é baseada no *best* ou *multi improvement*. A adaptação a partir dos *frameworks* originais do RVND (Algoritmo 3) e do DVND (Algoritmo 5) é realizada por meio do mapeamento das funções `seleciona_movimento` e `aplica_movimento` nos respectivos algoritmos. Para que o RVND utilize a GPU para realizar a avaliação de uma determinada vizinhança, a função `seleciona_movimento` realiza o lançamento do *kernel* responsável pelo operador correspondente. Enquanto que para cambiar a estratégia de seleção de vizinho, a função `aplica_movimento` faz a chamada para o procedimento apropriado.

Como discutido previamente na Seção 4.1, a busca RVND clássica não é adequada

para paralelização, já que sua metodologia se fundamenta na exploração sequencial de vizinhanças. Por esse motivo, a adaptação do RVND utilizada nos experimentos realiza apenas uma avaliação paralela de vizinhança na GPU por vez[2]. Por outro lado, o DVND foi projetado para executar a busca local em paralelo por meio da concorrência e distribuição de tarefas de exploração de vizinhanças.

Para a realização desse ensaio, algumas variações baseadas nesses dois métodos foram implementadas, onde seus nomes seguem o seguinte esquema: {DVND,RVND}-{SG,MG}-{BI,MI}. A primeira parte do nome indica se o mecanismo de busca local utilizado pelo algoritmo foi o DVND ou RVND; o segundo termo determina se o método utiliza uma GPU (SG, *single*-GPU) ou múltiplos dispositivos (MG, *multi* GPU); e a última parte indica qual estratégia de seleção de vizinhança foi utilizada, BI (*best improvement*) ou MI (*multi improvement*).

Nas Tabelas 4.3 a 4.7, cada linha apresenta o resultado médio obtido para 20 execuções para cada instância testada. Os experimentos foram realizados apenas com as instâncias do Grupo 8, já que o objetivo foi verificar o desempenho relativo entre as oito variações dos métodos baseados em RVND e DVND quando a dimensão da instância de entrada muda. Nas tabelas, a primeira coluna indica a instância do MLP utilizada, seguida por duas seções com duas colunas cada. Em cada seção, a primeira coluna indica o tempo de execução da busca local em milissegundos (ms), enquanto a segunda exibe o número de chamadas aos procedimentos de avaliação de vizinhanças realizado na GPU. As duas últimas colunas indicam os resultados relativos entre o método da segunda seção quando comparado com o da primeira. Desta forma, a penúltima coluna mostra o ganho relativo em termos de tempo de execução, enquanto a última mostra o percentual relativo do número de chamadas realizada por cada método.

Observando a Tabela 4.3, é possível verificar que o método RVND-SG-BI executa mais rapidamente que o DVND-SG-BI por um fator de ganho médio de 3,65. Contudo, esse fato já era esperado uma vez que o DVND executa, em média, 439% mais chamadas aos procedimentos de avaliação de vizinhança do que RVND, que as explora sequencialmente. Esta situação é bastante similar quando a estratégia *multi improvement* é empregada, como pode ser visto na Tabela 4.4. Neste caso, o RVND-SG-MI também executa 2,56 vezes mais rápido que o DVND-SG-MI. No entanto, o DVND realiza em média 304% mais chamadas que o RVND. Para todas as sete instâncias testadas, os custos das

[2]Este fato pode ser constatado na linha 6 do Algoritmo 3, onde a operação de seleção de movimentos de uma vizinhança lança um *kernel* GPU para computar os custos dos movimentos vizinhos à solução corrente avaliada.

Instância	DVND-SG-BI Tempo (ms)	DVND-SG-BI Número Chamadas	RVND-SG-BI Tempo (ms)	RVND-SG-BI Número Chamadas	Comparação Ganho	Comparação Chamadas (%)
kroD100	20,50	460,45	6,40	109,50	3,20	420,50
pr226	184,40	1.231,35	53,35	266,30	3,46	462,39
lin318	469,25	1.726,40	139,15	402,65	3,37	428,76
TRP-S500-R1	1.885,65	2755,60	527,60	630,55	3,57	437,02
d657	4.567,30	3.733,45	1.181,25	848,95	3,87	439,77
rat784	7.882,95	4.466,65	1.925,20	1.015,90	4,09	439,67
TRP-S1000-R1	17.881,20	5.986,60	4.498,65	1.337,30	3,97	447,66
				Média	**3,65**	**439,40**

Tabela 4.3: Comparação entre os métodos de busca local DVND-SG-BI e RVND-SG-BI.

Instância	DVND-SG-MI Tempo (ms)	DVND-SG-MI Número Chamadas	RVND-SG-MI Tempo (ms)	RVND-SG-MI Número Chamadas	Comparação Ganho	Comparação Chamadas (%)
kroD100	9,45	185,10	3,90	60,30	2,42	306,97
pr226	62,60	343,70	24,80	108,15	2,52	317,80
lin318	151,90	497,85	59,60	154,85	2,55	321,50
TRP-S500-R1	565,00	758,50	220,50	248,40	2,56	305,35
d657	1.168,40	914,25	458,95	312,00	2,55	293,03
rat784	2.034,95	1.142,65	763,80	392,10	2,66	291,42
TRP-S1000-R1	4.289,00	1.404,50	1.646,25	477,85	2,61	293,92
				Média	**2,56**	**304,28**

Tabela 4.4: Comparação entre os métodos de busca local DVND-SG-MI e RVND-SG-MI.

Instância	DVND-SG-BI Tempo (ms)	DVND-SG-BI Número Chamadas	DVND-SG-MI Tempo (ms)	DVND-SG-MI Número Chamadas	Comparação Ganho	Comparação Chamadas (%)
kroD100	20,50	460,45	9,45	185,10	2,17	59,80
pr226	184,40	1231,35	62,60	343,70	2,95	72,09
lin318	469,25	1726,40	151,90	497,85	3,09	71,16
TRP-S500-R1	1.885,65	2.755,60	565,00	758,50	3,34	72,47
d657	4.567,30	3.733,45	1.168,40	914,25	3,91	75,51
rat784	7.882,95	4.466,65	2.034,95	1.142,65	3,87	74,42
TRP-S1000-R1	17.881,20	5.986,60	4.289,00	1.404,50	4,17	76,54
				Média	**3,36**	**71,71**

Tabela 4.5: Comparação entre os métodos de busca local DVND-SG-BI e DVND-SG-MI.

soluções produzidas por DVND-SG-BI foi, em média, 0,01% menor que os obtidos por RVND-SG-BI.

Na Tabela 4.5, a estratégia *multi improvement* é confrontada com *best improvement* por meio das buscas DVND-SG-MI e DVND-SG-BI. Neste caso, apenas utilizando a estratégia *multi improvement* foi possível reduzir o tempo de execução por um fator de 3,36 em média. Este dado foi induzido pela redução média de quase 72% no número de chamadas efetuadas pelo método baseado em *multi improvement* em relação ao que emprega o *best improvement*. Tal resultado confirma a expectativa de que o *multi improvement* de fato colabora para o melhor desempenho dos métodos baseados em GPU. Isto ocorre porque a aplicação simultânea de movimentos de fato acelera a convergência do método para um ótimo local, o que reduz o número de chamadas aos *kernels* e, consequentemente, restringe o tráfego de dados entre CPU e GPU. Em relação à qualidade, a diferença de custo entre as soluções produzidas por ambos os métodos também ficou abaixo de 0,01% para cada uma das instâncias testadas.

Uma informação interessante que pode extraída da última coluna da Tabela 4.5 é que a contribuição do *multi improvement* tende a evoluir progressivamente à medida que o tamanho da instância aumenta. Esses dados estão destacados no gráfico da Figura 4.4, que mostra a taxa de redução do número de chamadas dos procedimentos de avaliação de vizinhanças. A tendência de crescimento dessa taxa indica que o *multi improvement* pode explorar de maneira eficiente instâncias de porte ainda maior. Isso é possível porque à medida que o porte da solução aumenta a probabilidade de surgimento de movimentos independentes é mais proeminente. Além disso, esse resultado evidencia o grande potencial dessa estratégia para acelerar métodos de busca em outros problemas de otimização.

A Figura 4.5 projeta o comportamento relativo entre o *best* e o *multi improvement* durante a busca local. Em cada gráfico apresentado, o eixo horizontal indica o *gap*[3] entre a solução corrente e a melhor solução conhecida do MLP para a instância considerada. O eixo vertical projeta o tempo necessário por cada estratégia para obter 1% de ganho no valor do custo da solução. O gráfico situado à esquerda apresenta a projeção dos resultados durante toda a busca. A linha vertical indica que a partir daquele ponto as curvas são replicadas mais detalhadamente no gráfico disposto à direita.

Como pode ser observado nos gráficos, o *multi improvement* de fato acelera a busca local obtendo ganhos de custo mais rapidamente que o *best improvement*. Tal

[3]O *gap* entre um custo c e um valor de referência r é dado por $gap(c, r) = (c - r)/r \times 100$.

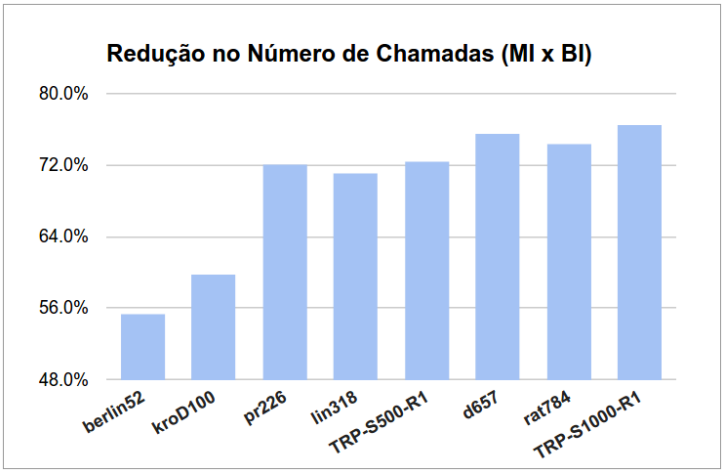


Figura 4.4: Redução do número de avaliações de vizinhança por meio da estratégia *multi improvement* em relação ao *best improvement*.

ganho é mais significativo quando a solução corrente ainda é de má qualidade, e reduz à medida que o *gap* se aproxima de zero. Isto ocorre porque quando a qualidade da solução melhora, os movimentos que apresentam ganho de custo são cada vez mais raros e, consequentemente, menor a possibilidade de existirem ou serem independentes. Nessa situação, o *multi improvement* passa a adotar a função do *best improvement*, como pode ser melhor verificado nos gráficos detalhados. Esse comportamento sugere que a estratégia *multi improvement* poderia ser utilizada como um mecanismo para aferir a convergência da busca local. Nessa ocasião, a estratégia de seleção poderia ser cambiada para *best improvement* que opera mais rapidamente, de forma a contribuir para acelerar um pouco mais o método. Contudo, esse mecanismo não foi implementado nos experimentos aqui apresentados.

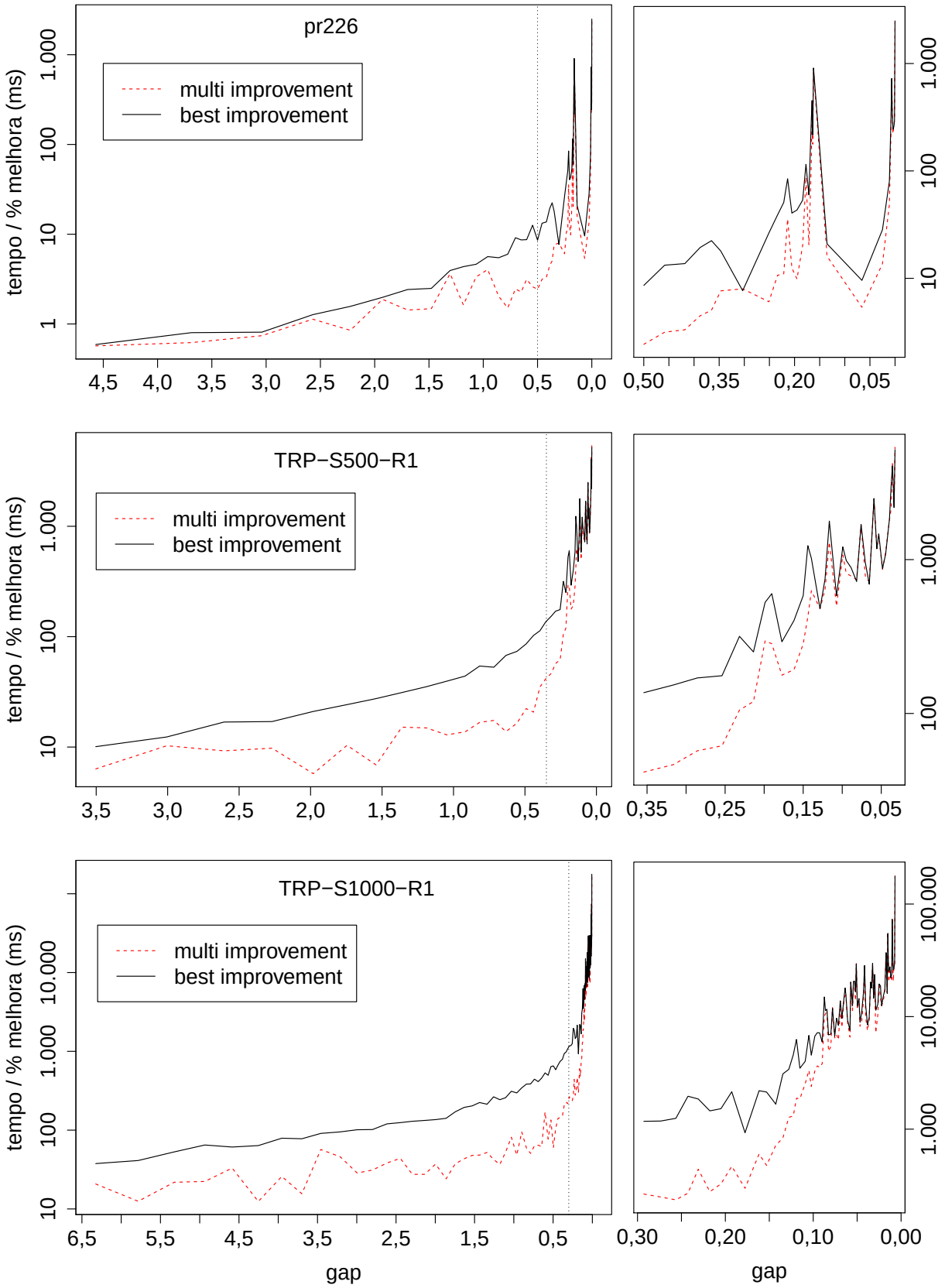


Figura 4.5: Evolução das estratégias *best improvement* e *multi improvement* durante uma busca local para as instâncias pr226, TRP-S500-R1 e TRP-S1000-R1.

Os resultados a seguir mostram a análise do ganho de desempenho do método DVND quando múltiplas GPUs são utilizadas. As Tabelas 4.6 e 4.7 exibem, respectivamente, os dados obtidos utilizando as estratégias *best* e *multi improvement*. A última coluna de cada tabela indica o ganho de desempenho relativo entre os dois métodos comparados. Os métodos SG utilizam uma única GPU, quando as buscas MG dispõem de dois dispositivos idênticos.

Instância	**DVND-SG-BI**	**DVND-MG-BI**	
	Tempo(ms)	**Tempo(ms)**	**Ganho**
kroD100	20,50	14,55	1,41
pr226	184,40	99,35	1,86
lin318	469,25	250,55	1,87
TRP-S500-R1	1.885,65	1.007,80	1,87
d657	4.567,30	2.402,70	1,90
rat784	7.882,95	4.210,40	1,87
TRP-S1000-R1	17.881,20	10.397,50	1,72
		Média	**1,78**

Tabela 4.6: Comparação de desempenho e escalabilidade entre DVND-SG-BI e DVND-MG-BI.

Os dados da Tabela 4.6 indicam que o DVND-MG-BI obteve um ganho de desempenho médio de 1,78 quando o número de GPUs do sistema foi dobrado. Um resultado similar pode ser observado na Tabela 4.7, quando a estratégia de seleção muda para o *multi improvement*. Neste cenário, porém, um desempenho médio ligeiramente inferior, de 1,67, foi obtido. Essa pequena diferença pode ser explicada pelo processamento adicional empenhado pela heurística de detecção de movimentos independentes. Contudo, esses resultados demonstram o potencial de escalabilidade do DVND, independente da estratégia de seleção utilizada.

Instância	**DVND-SG-MI**	**DVND-MG-MI**	
	Tempo(ms)	**Tempo(ms)**	**Ganho**
kroD100	9,45	6,75	1,40
pr226	62,60	37,75	1,66
lin318	151,90	87,10	1,74
TRP-S500-R1	565,00	308,65	1,83
d657	1.168,40	669,90	1,74
rat784	2.034,95	1.185,05	1,72
TRP-S1000-R1	4.289,00	2.624,30	1,63
		Média	**1,67**

Tabela 4.7: Comparação de desempenho e escalabilidade entre DVND-SG-MI e DVND-MG-MI.

4.4.2 Experimentos com o *framework* GVNS

Como já mencionado, os métodos de busca local são dependentes da solução de entrada e podem facilmente ficar confinados em ótimos locais. Por esta razão, com o objetivo de expandir a busca por soluções de melhor qualidade, um mecanismo de diversificação é necessário para complementar a busca local realizada por métodos como o RVND e DVND. Neste contexto, alguns experimentos foram realizados com o *framework* GVNS (Algoritmo 7) utilizando o RVND e DVND como mecanismos de busca local.

Para a execução dos experimentos seguintes, foram resgatados os resultados obtidos pelo algoritmo GILS-RVND publicados em [Silva *et al.*, 2012]. Além disso, foram realizadas duas implementações com o *framework* GVNS para arquitetura CPU, uma sequencial denominada GVNS-RVND-P1, que replica o algoritmo GILS-RVND, e outra paralela identificada como GVNS-RVND-P4. Este último algoritmo foi executado em ambiente CPU com $p = 4$ tarefas paralelas. Durante a implementação do GVNS para essa arquitetura, também foi utilizada a estrutura de dados local (EDL), porém de forma aprimorada, o que contribuiu para a elaboração de uma aplicação mais eficiente. Este fato foi corroborado pelo desempenho apresentado por GVNS-RVND-P1 quando executado sob as mesmas condições (mesmo computador) que a implementação sequencial original GILS-RVND. Tal cenário pôde ser obtido quando o algoritmo GVNS paralelo foi executado com o parâmetro $p = 1$. Desta forma, uma única tarefa trabalhadora é lançada sob condições idênticas às experimentadas pelo algoritmo sequencial.

Em todos os ensaios, os parâmetros de entrada dos algoritmos foram definidos para $D_{max} = 10$ e $I_{max} = min\{100, n\}$, $A = \{0,00; 0,01; 0,02; \ldots; 0,25\}$, onde n representa o número de clientes da instância. Esses valores foram os mesmos utilizados pela heurística sequencial GILS-RVND, com a qual alguns resultados são confrontados. Como medida de avaliação de desempenho dos algoritmos paralelos foram utilizados o *speedup* absoluto e relativo, definidos nas Equações 3.12 e 3.13.

Os resultados dos experimentos são apresentados nas Tabelas 4.8 a 4.13. Essas tabelas mostram os resultados em termos de tempos de execução e qualidade da solução. As primeiras três colunas das tabelas apresentam o nome da instância testada e os respectivos tempos de execução dos algoritmos sequenciais GILS-RVND e GVNS-RVND-P1. As colunas remanescentes estão divididas em três seções de duas colunas cada. Em cada seção, as colunas indicam o tempo de execução e o *speedup* alcançado pelo respectivo método quando comparado com GVNS-RVND-P1. Para facilitar a avaliação, convém mencionar que os números que aparecem nos nomes das instâncias indicam a quantidade

de clientes do problema.

Instância	GILS-RVND	GVNS-RVND-P1	GVNS-RVND-P4		GVNS-RVND-SG-BI		GVNS-DVND-MG-MI	
	tempo	tempo	tempo	$S_R(p)$	tempo	$S_A(p)$	tempo	$S_A(p)$
eil51	0,49	0,40	3,27	0,12	1,72	0,23	3,10	0,13
berlin52	0,46	0,42	3,14	0,13	1,72	0,24	2,85	0,15
st70	1,65	1,05	4,39	0,24	2,32	0,45	4,25	0,25
eil76	2,64	1,61	4,87	0,33	2,91	0,55	5,11	0,32
pr76	2,31	1,46	4,40	0,33	2,62	0,56	4,56	0,32
pr76r	2,41	1,38	4,35	0,32	2,63	0,52	4,44	0,31
rat99	11,27	5,93	5,03	1,18	6,35	0,93	8,92	0,67
kroA100	8,59	4,66	5,44	0,86	4,44	1,05	6,49	0,72
kroB100	9,21	4,16	5,46	0,76	4,44	0,94	7,36	0,56
kroC100	8,17	4,11	5,76	0,71	4,12	1,00	7,05	0,58
kroD100	8,46	4,30	5,74	0,75	4,34	0,99	6,85	0,63
kroE100	8,31	4,04	5,78	0,70	4,46	0,91	6,77	0,60
eil101	12,76	5,78	5,61	1,03	6,20	0,93	8,94	0,65
lin105	8,42	3,90	6,16	0,63	3,85	1,01	6,01	0,65
pr107	10,89	4,94	6,51	0,76	4,56	1,08	6,84	0,72
		Média		**0,59**		**0,76**		**0,48**

Tabela 4.8: Comparação de desempenho entre os melhores métodos heurísticos em termos tempo de execução e qualidade de solução para instâncias do Grupo 1.

A Tabela 4.8 apresenta os resultados para as instâncias do Grupo 1. É possível observar que GVNS-RVND-P1 apresenta melhor desempenho que GILS-RVND em todas as instâncias, comportamento que se repete para todos os grupos de instâncias testados. É importante mencionar que o algoritmo GVNS-RVND-P1 foi executado na mesma máquina que GILS-RVND. Isso sugere que o GVNS-RVND-P1, até onde se sabe, representa o método *sequencial* mais rápido para resolução do PML. Por esse motivo, a análise do *speedup* dos métodos paralelos utilizará esse algoritmo como referência. Nesta mesma tabela, é possível verificar que os algoritmos paralelos (GVNS-RVND-P4, GVNS-RVND-SG-BI e GVNS-RVND-MG-MI) apresentam desempenho inferior ou equivalente ao do algoritmo sequencial GVNS-RVND-P1, como pode ser observado na coluna de *speedup* correspondente a cada método. Isso ocorre por causa do pequeno porte das instâncias desse grupo, o que ocasiona uma rápida execução do algoritmo sequencial que supera os abordagens paralelas, atrasados pelos *overheads* associado aos métodos. O mesmo comportamento se repete para as instâncias do Grupo 3, com dimensões que variam de 10 a 50 clientes, razão pela qual as tabelas correspondentes não são apresentadas. Contudo, é importante salientar que todos os algoritmos atingem o mesmo custo da melhor solução conhecida para cada uma das instâncias dos Grupos 1 a 4. Como reportado por [Silva *et al.*, 2012], com exceção de rat99 e eil101 do Grupo 1, os custos das demais instâncias dos Grupos 1 a 3 são resultados ótimos.

Instância	GILS-RVND	GVNS-RVND-P1	GVNS-RVND-P4		GVNS-RVND-SG-BI		GVNS-DVND-MG-MI	
	tempo	tempo	tempo	$S_R(p)$	tempo	$S_A(p)$	tempo	$S_A(p)$
st70	1,51	1,46	0,34	4,29	2,04	0,72	3,62	0,40
rat99	9,47	7,26	2,29	3,17	6,55	1,11	8,59	0,85
kroD100	6,90	5,87	1,49	3,94	4,36	1,35	6,92	0,85
lin105	6,19	4,97	1,38	3,60	3,83	1,30	6,11	0,81
pr107	8,13	6,43	2,00	3,22	5,13	1,25	7,70	0,84
rat195	75,56	55,95	16,89	3,31	21,68	2,58	19,52	2,87
pr226	59,05	54,71	14,60	3,75	16,80	3,26	14,17	3,86
lin318	220,59	198,20	54,68	3,62	44,92	4,41	35,84	5,53
pr439	553,74	657,39	152,25	4,32	96,92	6,78	80,63	8,15
		Média		**3,69**		**2,53**		**2,68**

Tabela 4.9: Comparação de desempenho entre os melhores métodos heurísticos em termos tempo de execução e qualidade de solução para instâncias do Grupo 2.

As instâncias do Grupo 2 foram submetidas aos algoritmos e resultaram nos dados mostrados na Tabela 4.9. Enquanto GVNS-RVND-P4 apresenta um *speedup* mais estável, 3,69 em média, o GVNS-RVND-SG-BI e GVNS-DVND-MG-MI mostram um desempenho melhor nas instâncias maiores, atingindo picos de *speedup* de 6,78 e 8,15, respectivamente. Comparando o resultado desses dois últimos algoritmos, é possível observar que o *speedup* de GVNS-DVND-MG-MI progride à medida que o tamanho da instância aumenta, influenciado pela escalabilidade do método DVND. O comportamento dos algoritmos paralelos para este grupo de instâncias é refletido pelos respectivos resultados médios apresentados nas Tabelas 4.10 e 4.11 quando consideradas as instâncias de dimensões similares, com 100 e 200 clientes, respectivamente.

Nas Tabelas 4.10 a 4.13, para facilitar a apresentação dos dados, foi suprimido do nome de cada instância o prefixo "TRP-", que aparece no cabeçalho da primeira coluna. Nessas mesmas tabelas, o número que aparece após o prefixo "S", indica o número de clientes do problema.

Nas Tabelas 4.12 e 4.13 são exibidos os resultados para as instâncias de maior porte, com 500 e 1.000 clientes, respectivamente. Os métodos baseados em GPU, GVNS-RVND-SG-BI e GVNS-DVND-MG-MI superam em termos de desempenho o algoritmo paralelo baseado em CPU GVNS-RVND-P4, onde atingem maiores *speedups* em cada uma das instâncias dos Grupos 6 e 7. Nesses casos, GVNS-DVND-MG-MI desempenha melhor que GVNS-RVND-SG-BI em ambos os experimentos, sugerindo que o algoritmo DVND se adequa bem para problemas de grande porte. Os melhores resultados foram alcançados por GVNS-DVND-MG-MI para as instâncias com 1.000 clientes, com um valor médio de *speedup* igual a 10,62, chegando a atingir um pico de 13,72.

Instância	GILS-RVND	GVNS-RVND-P1	GVNS-RVND-P4		GVNS-RVND-SG-BI		GVNS-DVND-MG-MI	
TRP-	tempo	tempo	tempo	$S_R(p)$	tempo	$S_A(p)$	tempo	$S_A(p)$
S100-R1	7,05	5,62	1,53	3,67	4,40	1,28	7,19	0,78
S100-R2	7,51	6,04	1,59	3,80	4,85	1,25	7,58	0,80
S100-R3	7,07	5,42	1,59	3,41	4,57	1,19	6,79	0,80
S100-R4	7,27	5,72	1,57	3,64	4,62	1,24	7,66	0,75
S100-R5	8,87	7,33	1,90	3,86	5,33	1,38	7,73	0,95
S100-R6	7,82	6,64	1,73	3,84	5,09	1,30	7,81	0,85
S100-R7	8,74	6,55	2,13	3,08	5,61	1,17	8,06	0,81
S100-R8	7,08	5,34	1,62	3,30	4,53	1,18	7,36	0,73
S100-R9	7,47	6,17	1,68	3,67	4,87	1,27	7,23	0,85
S100-R10	6,78	5,90	1,56	3,78	4,29	1,38	7,02	0,84
S100-R11	7,75	7,19	1,87	3,84	5,30	1,36	7,96	0,90
S100-R12	7,20	5,78	1,62	3,57	4,47	1,29	6,76	0,86
S100-R13	7,78	5,70	1,85	3,08	5,26	1,08	8,18	0,70
S100-R14	6,29	5,47	1,51	3,62	4,28	1,28	7,00	0,78
S100-R15	7,13	6,39	1,66	3,85	4,51	1,42	7,69	0,83
S100-R16	7,97	5,87	1,76	3,34	4,87	1,21	6,60	0,89
S100-R17	9,23	7,08	2,15	3,29	5,55	1,28	7,75	0,91
S100-R18	7,07	5,58	1,63	3,42	4,50	1,24	7,29	0,77
S100-R19	8,08	6,64	1,75	3,79	4,93	1,35	8,36	0,79
S100-R20	8,73	7,45	2,12	3,51	5,82	1,28	9,07	0,82
		Média		**3,57**		**1,27**		**0,82**

Tabela 4.10: Comparação de desempenho entre os melhores métodos heurísticos em termos tempo de execução e qualidade de solução para instâncias do Grupo 4.

Finalmente, na Tabela 4.14, é apresentado um resumo das melhores soluções encontradas pelos métodos implementados neste trabalho. Nessa tabela, as primeiras duas colunas indicam o nome da instância e o respectivo melhor valor de custo conhecido. As próximas colunas reportam, respectivamente, o novo custo encontrado, o *gap* entre os custos e o método que encontrou a solução.

O *framework* GVNS explorou as estruturas de vizinhança do MLP de duas formas diferentes, utilizando a estratégia de seleção clássica *best improvement* e a nova estratégia *multi improvement* introduzida neste capítulo. Essa última estratégia tirou vantagem da grande disponibilidade dados relativos às vizinhanças disponível na memória da GPU, aplicando simultaneamente múltiplos movimentos sobre a solução base. De maneira geral, os métodos baseados no *multi improvement* apresentaram resultados superiores em termos de qualidade de solução e desempenho computacional. Por sua vez, a busca local DVND foi comparada com o eficiente RVND no contexto do *framework* GVNS, e produziram resultado que suplantaram o melhor algoritmo sequencial da literatura. Esses resultados demonstraram a capacidade das técnicas propostas de explorar arquiteturas heterogêneas com o objetivo de lidar com um problema combinatório complexo.

Instância	GILS-RVND	GVNS-RVND-P1	GVNS-RVND-P4		GVNS-RVND-SG-BI		GVNS-DVND-MG-MI	
TRP-	tempo	tempo	tempo	$S_R(p)$	tempo	$S_A(p)$	tempo	$S_A(p)$
S200-R1	73,39	60,97	15,14	4,03	16,16	3,77	17,30	3,52
S200-R2	68,07	58,04	15,88	3,65	17,58	3,30	18,35	3,16
S200-R3	67,11	52,86	16,39	3,23	16,30	3,24	17,43	3,03
S200-R4	72,17	67,15	17,66	3,80	16,72	4,02	19,36	3,47
S200-R5	70,77	55,77	15,05	3,71	16,97	3,29	17,08	3,26
S200-R6	70,52	60,69	16,69	3,64	17,00	3,57	18,63	3,26
S200-R7	72,80	65,01	17,43	3,73	17,91	3,63	17,96	3,62
S200-R8	75,15	64,15	18,95	3,39	18,34	3,50	17,82	3,60
S200-R9	65,45	51,24	15,67	3,27	15,55	3,30	15,58	3,29
S200-R10	74,25	64,58	15,77	4,10	18,08	3,57	18,02	3,58
S200-R11	73,15	58,41	15,43	3,79	16,71	3,50	18,30	3,19
S200-R12	76,74	60,22	17,72	3,40	18,25	3,30	18,02	3,34
S200-R13	72,96	59,12	17,86	3,31	17,40	3,40	17,13	3,45
S200-R14	70,94	67,72	18,12	3,74	17,45	3,88	17,49	3,87
S200-R15	70,41	62,35	14,75	4,23	17,07	3,65	17,19	3,63
S200-R16	77,89	65,55	16,71	3,92	18,51	3,54	18,77	3,49
S200-R17	71,17	53,33	17,10	3,12	16,96	3,14	16,92	3,15
S200-R18	77,03	65,86	18,36	3,59	19,53	3,37	20,05	3,29
S200-R19	71,08	61,54	16,83	3,66	16,57	3,71	17,31	3,55
S200-R20	70,61	59,27	16,77	3,53	17,77	3,34	17,39	3,41
		Média		**3,64**		**3,50**		**3,41**

Tabela 4.11: Comparação de desempenho entre os melhores métodos heurísticos em termos tempo de execução e qualidade de solução para instâncias do Grupo 5.

Instância	GILS-RVND	GVNS-RVND-P1	GVNS-RVND-P4		GVNS-RVND-SG-BI		GVNS-DVND-MG-MI	
TRP-	tempo	tempo	tempo	$S_R(p)$	tempo	$S_A(p)$	tempo	$S_A(p)$
S500-R1	1.738,48	1.480,61	383,19	3,86	225,54	6,56	160,13	9,25
S500-R2	1.476,13	1.453,18	355,07	4,09	211,67	6,87	156,34	9,30
S500-R3	1.557,48	1.263,19	362,24	3,49	214,43	5,89	164,88	7,66
S500-R4	1.597,06	1.294,68	370,51	3,49	205,33	6,31	154,11	8,40
S500-R5	1.530,94	1.182,23	326,29	3,62	209,96	5,63	151,14	7,82
S500-R6	1.576,91	1.242,11	365,15	3,40	213,77	5,81	147,83	8,40
S500-R7	1.584,67	1.460,99	413,90	3,53	220,79	6,62	162,63	8,98
S500-R8	1.565,01	1.170,83	377,73	3,10	212,25	5,52	151,88	7,71
S500-R9	1.409,23	1.231,35	340,45	3,62	201,96	6,10	148,61	8,29
S500-R10	1.621,85	1.242,51	358,97	3,46	219,46	5,66	162,45	7,65
S500-R11	1.530,98	1.182,59	345,35	3,42	202,19	5,85	140,80	8,40
S500-R12	1.554,75	1.159,83	355,29	3,26	212,07	5,47	159,05	7,29
S500-R13	1.598,46	1.156,36	366,82	3,15	210,14	5,50	151,22	7,65
S500-R14	1.701,90	1.277,14	333,54	3,83	223,75	5,71	145,61	8,77
S500-R15	1.623,79	1.304,96	379,62	3,44	218,51	5,97	159,30	8,19
S500-R16	1.583,70	1.241,87	368,67	3,37	217,29	5,72	146,92	8,45
S500-R17	1.549,80	1.340,59	378,90	3,54	213,24	6,29	150,77	8,89
S500-R18	1.620,02	1.207,28	349,91	3,45	205,16	5,88	145,85	8,28
S500-R19	1.602,87	1.266,32	321,32	3,94	192,84	6,57	150,93	8,39
S500-R20	1.507,96	1.227,68	334,54	3,67	201,43	6,09	150,44	8,16
		Média		**3,54**		**6,00**		**8,30**

Tabela 4.12: Comparação de desempenho entre os melhores métodos heurísticos em termos tempo de execução e qualidade de solução para instâncias do Grupo 6.

Instância	GILS-RVND	GVNS-RVND-P1	GVNS-RVND-P4		GVNS-RVND-SG-BI		GVNS-DVND-MG-MI	
TRP-	**tempo**	**tempo**	**tempo**	$S_R(p)$	**tempo**	$S_A(p)$	**tempo**	$S_A(p)$
S1000-R1	31.894,19	18.961,03	4.357,74	4,35	1.752,43	10,82	1.382,15	13,72
S1000-R2	30.881,19	15.501,36	4.299,49	3,61	1.825,17	8,49	1.380,19	11,23
S1000-R3	30.184,15	14.180,13	4.378,45	3,24	1.658,31	8,55	1.377,36	10,30
S1000-R4	29.951,12	14.850,17	4.275,61	3,47	1.902,93	7,80	1.484,99	10,00
S1000-R5	30.129,51	15.004,78	4.595,65	3,26	1.988,06	7,55	1.435,99	10,45
S1000-R6	28.161,57	14.518,93	4.769,67	3,04	1.858,29	7,81	1.346,35	10,78
S1000-R7	25.945,41	14.062,64	4.262,94	3,30	1.628,90	8,63	1.301,71	10,80
S1000-R8	26.572,71	13.596,90	4.322,93	3,15	1.847,85	7,36	1.400,15	9,71
S1000-R9	26.330,40	13.992,05	4.568,57	3,06	1.426,42	9,81	1.429,27	9,79
S1000-R10	25.676,31	15.169,59	4.473,15	3,39	1.763,09	8,60	1.413,29	10,73
S1000-R11	26.235,63	14.437,44	4.248,57	3,40	1.769,79	8,16	1.398,15	10,33
S1000-R12	27.910,11	13.108,25	4.178,22	3,14	1.761,83	7,44	1.378,94	9,51
S1000-R13	28.475,89	14.467,44	4.526,23	3,20	1.920,37	7,53	1.451,95	9,96
S1000-R14	27.639,81	14.983,36	4.638,96	3,23	1.750,69	8,56	1.359,93	11,02
S1000-R15	27.633,07	14.349,81	4.706,26	3,05	1.731,02	8,29	1.348,72	10,64
S1000-R16	26.653,16	15.223,09	4.742,39	3,21	1.691,41	9,00	1.414,74	10,76
S1000-R17	27.503,43	15.208,29	4.846,70	3,14	1.819,14	8,36	1.419,82	10,71
S1000-R18	28.808,09	15.190,69	4.418,86	3,44	1.768,00	8,59	1.374,63	11,05
S1000-R19	29.637,49	14.546,70	4.510,98	3,22	1.800,93	8,08	1.298,14	11,21
S1000-R20	27.499,24	13.574,15	4.312,70	3,15	1.754,06	7,74	1.402,81	9,68
		Média		**3,30**		**8,36**		**10,62**

Tabela 4.13: Comparação de desempenho entre os melhores métodos heurísticos em termos tempo de execução e qualidade de solução para instâncias do Grupo 7.

Instância	**[Silva *et al.*, 2012]**	**Este Trabalho**		
	custo	**custo**	***gap*(%)**	**Algoritmo**
TRP-S500-R2	1.816.568	1.815.664	-0,05	GVNS-RVND-SG-BI
TRP-S500-R3	1.833.044	1.826.855	-0,34	GVNS-RVND-SG-BI
TRP-S500-R4	1.809.266	1.804.894	-0,24	GVNS-RVND-SG-MI
TRP-S500-R5	1.823.975	1.821.250	-0,15	GVNS-DVND-SG-MI
TRP-S500-R6	1.786.620	1.782.731	-0,22	GVNS-RVND-SG-MI
TRP-S500-R8	1.820.846	1.819.636	-0,07	GVNS-RVND-SG-MI
TRP-S500-R10	1.762.741	1.761.174	-0,09	GVNS-RVND-P4
TRP-S500-R13	1.873.699	1.863.905	-0,52	GVNS-RVND-SG-BI
TRP-S500-R15	1.791.145	1.785.263	-0,33	GVNS-RVND-P4
TRP-S500-R16	1.810.188	1.804.392	-0,32	GVNS-RVND-SG-MI
TRP-S500-R18	1.826.263	1.825.615	-0,04	GVNS-DVND-SG-MI
TRP-S500-R19	1.779.248	1.776.855	-0,13	GVNS-RVND-P4
TRP-S1000-R4	5.118.006	5.112.465	-0,11	GVNS-RVND-P4
TRP-S1000-R5	5.103.894	5.097.991	-0,12	GVNS-DVND-MG-MI
TRP-S1000-R6	5.115.816	5.109.946	-0,11	GVNS-RVND-SG-BI
TRP-S1000-R7	5.021.383	4.995.703	-0,51	GVNS-RVND-P4
TRP-S1000-R9	5.052.599	5.046.566	-0,12	GVNS-RVND-P4
TRP-S1000-R10	5.078.191	5.060.019	-0,36	GVNS-RVND-P4
TRP-S1000-R11	5.041.913	5.031.455	-0,21	GVNS-DVND-MG-MI
TRP-S1000-R14	5.099.433	5.092.861	-0,13	GVNS-RVND-P4
TRP-S1000-R15	5.142.470	5.131.013	-0,22	GVNS-RVND-P4
TRP-S1000-R16	5.073.972	5.064.094	-0,19	GVNS-DVND-MG-MI
TRP-S1000-R17	5.071.485	5.052.283	-0,38	GVNS-RVND-P4
TRP-S1000-R18	5.017.589	5.005.789	-0,24	GVNS-RVND-SG-BI
TRP-S1000-R19	5.076.800	5.064.873	-0,23	GVNS-RVND-P4

Tabela 4.14: Novas soluções para o Problema da Mínima Latência encontradas pelos métodos paralelos.

Capítulo 5

Considerações Finais e Trabalhos Futuros

Este trabalho procurou identificar e mensurar diversos aspectos relacionados ao desempenho na implementação de meta-heurísticas por meio do estudo de mecanismos de busca local projetados para ambientes híbridos compostos por CPU e GPUs. Para tanto, foi utilizado como referência um método sequencial bastante eficiente para avaliação de vizinhanças, que foi decomposto, adaptado e sistematicamente avaliado na forma de várias combinações de alocação de seus componentes nessa plataforma. Na ocasião, diversos elementos relacionados ao desempenho foram destacados, aferidos e avaliados. Além disso, foram propostos dois novos métodos para auxiliar heurísticas baseadas em busca local. Uma técnica de seleção em vizinhança se mostrou bastante promissora para a aceleração e abordagem de problemas de grande porte. Um novo método de busca local projetado para ambientes híbridos CPU/GPU também foi introduzido e apresentou bons desempenho e escalabilidade em plataformas com múltiplas GPUs.

Apesar do poder computacional proporcionado pela arquitetura massivamente paralela das GPUs, o projeto e implementação de meta-heurísticas para essas plataformas híbridas ainda não é uma tarefa trivial. Muitos aspectos inerentes à essa arquitetura precisam ser considerados para que algoritmos heurísticos possam explorar adequadamente seu potencial. Por esse motivo, um minucioso estudo de desempenho foi realizado durante a implementação de uma estratégia sequencial de aceleração da avaliação em vizinhança baseada no pré-processamento de movimentos e no uso de estruturas de dados locais (EDL). Durante a adaptação desse método para GPU, seus componentes essenciais foram desmembrados e remodelados para se adequarem às especificidades da arquitetura GPU e explorarem integralmente seu potencial. Uma vez que ainda não existem mecanismos bem definidos para comparação de desempenho entre algoritmos projetados para CPU e GPU, já que se tratam de arquiteturas bastante distintas, um interessante ensaio envol-

vendo a Lei de Amdahl foi realizado. Neste experimento, um componente essencial para o desempenho global do mecanismo de pré-processamento de vizinhanças, o de atualização da EDL, teve seu nível de paralelismo testado e avaliado. Resultados de experimentos demonstraram que o desempenho da versão paralela para GPU ficou apenas a 7,6% do limite teórico de paralelismo que poderia ser alcançado, evidenciando que o emprego de técnicas de implementação orientadas para a arquitetura GPU pode, de fato, contribuir significativamente para um melhor desempenho nessa plataforma. Os ensaios envolvendo diversas configurações dos demais componentes do método revelaram que o mecanismo de pré-processamento não se mostrou promissor quando executado na GPU. Mesmo assim, a versão onde o pré-processamento não foi considerado, obteve *speedups* entre 50 a 410 vezes durante a avaliação da vizinhança de cinco operadores clássicos de movimento.

Estes resultados corroboraram o fato de que a movimentação de dados entre tarefas paralelas é realmente um importante quesito no projeto de algoritmos com memória distribuída, seja ela realizada entre os níveis da rica hierarquia de memória da GPU, ou na comunicação entre CPU e GPU. Visando a redução desse tráfego de dados, foi introduzida uma nova técnica de avaliação de vizinhança, o *multi improvement*, que se vale da grande quantidade de dados de avaliação de movimentos disponível na memória da GPU para aplicar múltiplos movimentos sobre uma mesma solução base. Os ensaios mostraram que o uso dessa estratégia possibilitou uma convergência mais rápida da busca local e uma redução máxima de até 4 vezes no número de avaliações de vizinhança, o que limitou o tráfego entre CPU e GPU. Um outro aspecto interessante do *multi improvement* é a sua tendência de melhorar esse desempenho à medida que o tamanho do problema aumenta.

Uma nova estratégia de busca local DVND foi introduzida e comparada com o eficiente RVND combinado com a estratégia de pré-avaliação de movimentos baseada em EDL. Esse último é reconhecidamente um método sequencial muito efetivo para aceleração da busca em vizinhança. Mesmo assim, o DVND se mostrou como uma proposta paralela robusta capaz de realizar uma busca até 4 vezes mais intensiva que o RVND, explorando paralelamente o espaço de soluções. Quando combinado com o *multi improvement* foi capaz de acelerar a busca 3,4 vezes em média quando comparado com sua versão baseada no *best improvement*. Um outro aspecto onde o DVND se mostrou promissor foi no quesito escalabilidade, onde os experimentos demostraram uma aceleração proporcional a 1,8 quando o número de GPUs foi dobrado, mesmo utilizando um esquema estático de balanceamento de carga. Um levantamento bibliográfico sugeriu que o DVND foi o primeiro método paralelo baseado no VND.

O desempenho do DVND e do RVND também foram testados no contexto de um *framework*, o GVNS, composto por mecanismos de diversificação inspirado em GRASP e de uma estratégia de intensificação orientada pelo VNS. O GVNS foi adaptado para ambas as arquiteturas CPU e GPU e utilizado para testar seis configurações envolvendo o DVND e RVND, com *best e multi improvement*, em sistemas utilizando uma ou duas GPUs. A versão paralela do GVNS também foi testada num ambiente puramente baseado em arquitetura CPU, onde apresentou desempenho superior ao melhor algoritmo da literatura quando testado sob condições idênticas em termos de configuração e plataforma de *hardware*. Esse resultado foi possível devido, principalmente, à otimização da implementação realizada para CPU, onde algumas das técnicas evidenciadas nos experimentos para GPU foram utilizadas, como o acesso coalescente à memória e estruturadas de dados reduzidas e projetadas para tirar o máximo proveito do rápido acesso à memória *cache* da CPU. Até onde se sabe, esse foi o primeiro algoritmo paralelo para o Problema da Mínima Latência.

Finalmente, os métodos introduzidos nesse trabalho foram avaliados num contexto mais amplo, quando embutidos no *framework* heurístico GVNS. Mais uma vez diversas configurações envolvendo DVND/*multi improvement* e RVND/*best improvement* foram testadas. De maneira geral, o DVND apresentou melhor desempenho quando submetido a instâncias de grande porte sugerindo que esse método se acomoda bem em cenários mais complexos. Novamente o DVND se mostrou bastante efetivo quando combinado com o *multi improvement* e submetido a uma plataforma multi-GPU, ocasião em que obteve seus melhores resultados, com *speedup* médio de 10,6 e chegando a picos de 13,7 para as maiores instâncias. Adicionalmente, o *framework* paralelo GVNS foi capaz de encontrar 25 novas soluções para o Problema da Mínima Latência para instâncias de 500 e 1.000 clientes.

Como geralmente ocorre quando se investiga mais profundamente um tema, novas perspectivas afloram ampliando os horizontes de pesquisa. Não foi diferente no contexto das propostas concebidas nessa tese. O estudo preliminar de desempenho da estratégia *multi improvement* indica que esse método utilizado em conjunto com o DVND se beneficiam mutuamente. Entretanto, a escolha da estratégia de busca em vizinhança empregada pela busca local está relacionada com o *trade-off* entre qualidade de solução e tempo de execução. Segundo [Talbi, 2009], a estratégia *first improvement* parece conduzir a uma convergência mais rápida quando a solução base é gerada aleatoriamente (tipicamente de má qualidade), enquanto o *best improvement* pode conduzir a melhores resultados quando é aplicado um método construtivo que produz uma solução mais robusta. Os

resultados dos experimentos realizados já demonstraram que o *multi improvement* é mais eficaz quando utilizado no início da busca, quando a qualidade da solução ainda é inferior. Neste contexto, um estudo comparativo da estratégia *multi improvement* com esses métodos clássicos pode ajudar na avaliação de sua influência na qualidade da solução ao longo da busca. Desta forma, seria possível verificar se a trajetória de convergência mais acentuada proporcionada pelo *multi improvement* pode negligenciar ou deixar de explorar regiões promissoras do espaço de busca, especialmente quando submetida a problemas de grande porte.

Um outro aspecto pertinente ao *multi improvement* é a verificação da influência da heurística de detecção de movimentos independentes na busca. Nos testes realizados foi utilizada uma heurística gulosa que priorizou a velocidade de execução em detrimento da maximização do ganho de custo. Uma vez que a resolução do subproblema relacionado à detecção de movimentos independentes já é estudado pela literatura, a avaliação de métodos mais eficientes pode ser considerado. O experimento de outros métodos — heurísticos ou exatos — pode ser realizado com o objetivo de avaliar o comportamento da busca global neste contexto.

Os resultados dos experimentos realizados indicam que o DVND reage bem quando o número de GPUs no sistema aumenta, indicando uma boa escalabilidade. Desta forma, um caminho natural para exploração dessa característica é sua aplicação em *clusters* de computadores compostos por CPU e GPUs. Essa expansão de escopo pode proporcionar o poder computacional necessário para atacar problemas mais complexos e vizinhanças com maior cardinalidade. Nesse contexto outros aspectos do algoritmo podem ser explorados como, por exemplo, a implementação de um mecanismo de balanceamento dinâmico de carga. Tal mecanismo poderia considerar a distribuição equitativa de carga nos três níveis dessa arquitetura, considerando tanto o balanceamento dentro dos nós, envolvendo múltiplas GPUs e tarefas da CPU, quanto entre as máquinas do *cluster*.

Um fato relevante que merece ser evidenciado sobre o DVND é que seu potencial ainda não foi completamente explorado, já que o algoritmo utilizou a seleção *multi improvement* apenas dentro do espaço de busca de uma mesma vizinhança. No entanto, como foi apresentado na ocasião, os movimentos independentes podem ser aplicados concomitantemente quando são oriundos de vizinhanças distintas. Essa característica aventa a possibilidade de combinar operações heterogêneas para produzir uma única grande vizinhança que pode ser explorada paralelamente por uma ou mais GPUs. Na realidade, alguns passos nessa direção já foram dados e uma proposta incipiente já foi concebida. Ela

consiste na generalização do DVND que consiste, a grosso modo, em expandir a função do histórico para que registre a indução de movimentos independentes e seus eventuais conflitos quando vizinhanças distintas são considerados. Nesse caso, o histórico seria promovido da função de mero repositório de soluções para se tornar o componente central do método.

O sucesso no emprego de técnicas relacionadas ao desempenho na implementação de métodos heurísticos também motivam o desenvolvimento de propostas para outros problemas de otimização. Um destaque pode ser dado a problemas onde a tradicional arquitetura CPU pode ser um fator limitante para a abordagem de problemas mais complexos. Uma pesquisa preliminar identificou pelo menos um problema estudado na literatura que se atende a essa condição. Trata-se do Problema de Rotulação Cartográfica de Pontos, do inglês *Point-Feature Cartographic Label Placement*, que possui muitas aplicações em geração de mapas e sistemas de georreferenciamento. Até onde se sabe, ainda não existem aplicações para esse problema orientadas para GPU e especialistas no problema tem apontado essa arquitetura como uma alternativa viável para a abordagem de instâncias mais complexas ou para a redução do tempo computacional. Experimentos preliminares realizados com *kernels* GPU conseguiram obter acelerações bastante impressionantes quando comparados com seu equivalente para CPU. Apesar da incipiência dos ensaios, o problema reúne características que são adequadas à arquitetura GPU, tornando promissor o investimento no desenvolvimento de métodos para sua resolução.

APÊNDICE A – Trabalhos Submetidos e Publicados

- **Artigos Publicados**
  - Rios, E.; Ochi, L. S. e Boeres, C. Uma Heurística Paralela Eficiente para o Problema da Mínima Latência. In: XLV Simpósio Brasileiro de Pesquisa Operacional (SBPO), 2013.
  - Rios, E.; Ochi, L. S.; Boeres, C.; Coelho, I. M. and Farias, R. A Performance Study on GPU-based Neighborhood Search Algorithms for Vehicle Routing. In: 6th Workshop on Applications for Multi-core Architectures (WAMCA), held in conjuction with SBAC-PAD, 2015.
- **Artigos Aceitos**
  - Rios, E.; Ochi, L. S.; Boeres, C.; Coelho, I. M.; Coelho, V. N. and Mladenovic, N. A Performance Study on Multi Improvement Neighborhood Search Strategy. In: 4th International Conference on VNS, 2016.
  - Rios, E.; Coelho, I. M.; Ochi, L. S.; Boeres, C. Exploração Eficiente de Estruturas de Vizinhança em Ambientes Híbridos CPU/GPU. In: XLVIII Simpósio Brasileiro de Pesquisa Operacional (SBPO), 2016.
  - Coelho, I. M.; Rios, E.; Silva, M. M.; Urosevic, D. and Mladenovic, N. Iterated VND and GVNS for Traveling Repairman Problem: head to head comparison. In: 4th International Conference on VNS, 2016.
- **Artigo Submetido**
  - Rios, E.; Ochi, L. S.; Boeres, C.; Coelho, I. M. and Farias, R. Exploring parallel multi-GPU local search strategies in a metaheuristic framework. Journal of Parallel and Distributed Computing (JPDC), 2016.

- **Colaboração**
  - Coelho, V. N.; Coelho, I. M.; Rios, E. and Guimaraes, F. G. A deep learning hybrid fuzzy model using GPU disaggregated function evaluations. In: Applied Energy Symposium and Summit (REM2016), 2016.

Referências

[Abeledo *et al.*, 2013] Abeledo, H., Fukasawa, R., Pessoa, A., & Uchoa, E. The time dependent traveling salesman problem: polyhedra and algorithm. *Math. Program. Comput.*, 5(1), 27–55, (2013).

[Alba, 2005] Alba, E., (2005). *Parallel Metaheuristics: A New Class of Algorithms*. Wiley.

[Alba & Dorronsoro, 2005] Alba, E. & Dorronsoro, B. The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 9(2), 126–142, (2005).

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

[Alba & Tomassini, 2002] Alba, E. & Tomassini, M. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5), 443–462, (2002).

[Amdahl, 1967] Amdahl, G. M., (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (pp. 483–485).: ACM.

[Anderson, 2005] Anderson, S. E. Bit twiddling hacks. *URL: http://graphics.stanford.edu/~seander/bithacks.html*, (2005).

[Angel-Bello *et al.*, 2013] Angel-Bello, F., Alvarez, A., & Garc\'ia, I. Two improved formulations for the minimum latency problem. *Applied Math. Modelling*, 37(4), 2257–2266, (2013).

[Applegate *et al.*, 2007] Applegate, D. L., Bixby, R. E., Chvatal, V., & Cook, W. J., (2007). *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton, NJ, USA: Princeton University Press.

[Archer *et al.*, 2008] Archer, A., Levin, A., & Williamson, D. P. A faster, better approximation algorithm for the minimum latency problem. *SIAM-JC*, 37(5), 1472–1498, (2008).

[Arora & Karakostas, 2003] Arora, S. & Karakostas, G. Approximation schemes for minimum latency problems. *SIAM Journal on Computing*, 32(5), 1317–1337, (2003).

[Ausiello *et al.*, 2000] Ausiello, G., Leonardi, S., & Marchetti-Spaccamela, A., (2000). On salesmen, repairmen, spiders, and other traveling agents. In G. Bongiovanni, G. Gambosi, & R. Petreschi (Eds.), *Algorithms and Complexity*, volume 1767 of *Lect. Notes in Comput. Sci.* (pp. 1–16).: Springer-Verlag Berlin.

[Bianco *et al.*, 1993] Bianco, L., Mingozzi, A., & Ricciardelli, S. The traveling salesman problem with cumulative costs. *Networks*, 23(2), 81–91, (1993).

[Blum *et al.*, 1994] Blum, A., Chalasani, P., Coppersmith, D., Pulleyblank, B., Raghavan, P., & Sudan, M., (1994). The minimum latency problem. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing* (pp. 163–171).: ACM.

[Blum & Roli, 2003] Blum, C. & Roli, A. Metaheuristics in combinatorial optimization. *ACM Computing Surveys*, 35(3), 268–308, (2003).

[Blum *et al.*, 2005] Blum, C., Roli, A., & Alba, E., (2005). An introduction to metaheuristic techniques. In E. Alba (Ed.), *Parallel metaheuristics: a new class of algorithm* chapter 1, (pp. 3–42). Wiley-interscience.

[Bomze *et al.*, 1999] Bomze, I. M., Budinich, M., Pardalos, P. M., & Pelillo, M., (1999). The maximum clique problem. In *Handbook of combinatorial optimization* (pp. 1–74). Springer.

[Brodtkorb *et al.*, 2010] Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., & Storaasli, O. O. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1), 1–33, (2010).

[Brodtkorb *et al.*, 2013] Brodtkorb, A. R., Hagen, T. R., Schulz, C., & Hasle, G. Gpu computing in discrete optimization. part i: Introduction to the gpu. *EURO Journal on Transportation and Logistics*, 2(1-2), 129–157, (2013).

[Catala *et al.*, 2007] Catala, A., Jaen, J., & Mocholi, J. A., (2007). Strategies for accelerating ant colony optimization algorithms on graphical processing units. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on* (pp. 492–500).: IEEE.

[Cecilia *et al.*, 2011] Cecilia, J. M., Garcia, J. M., Ujaldon, M., Nisbet, A., & Amos, M., (2011). Parallelization strategies for ant colony optimisation on gpus. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on* (pp. 339–346).: IEEE.

[Chaudhuri *et al.*, 2003] Chaudhuri, K., Godfrey, B., Rao, S., & Talwar, K., (2003). Paths, trees, and minimum latency tours. In *Found. of Comput. Sci. Proc. 44th Annual IEEE Symposium on* (pp. 36–45).

[Cheng *et al.*, 2014] Cheng, J., Grossman, M., & McKercher, T., (2014). *Professional Cuda C Programming*. John Wiley & Sons.

[Coelho *et al.*, 2016] Coelho, I., Munhoz, P., Ochi, L., Souza, M., Bentes, C., & Farias, R. An integrated cpu–gpu heuristic inspired on variable neighbourhood search for the single vehicle routing problem with deliveries and selective pickups. *International Journal of Production Research*, 54(4), 945–962, (2016).

[Crainic & Gendreau, 2010] Crainic, T. G. & Gendreau, M. *Parallel Metaheuristics*, (2010), volume 146, chapter 17, (pp. 497–541). Springer US, 2nd edition.

[Crainic & Toulouse, 2003] Crainic, T. G. & Toulouse, M., (2003). Parallel strategies for meta-heuristics. In F. Glover & G. Kochenberger (Eds.), *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science* (pp. 475–513). Springer US.

[Cung *et al.*, 2002] Cung, V.-D., Martins, S. L., Ribeiro, C. C., & Roucairol, C., (2002). Strategies for the parallel implementation of metaheuristics. In V.-D. Cung, S. L. Martins, C. C. Ribeiro, & C. Roucairol (Eds.), *Essays and Surveys in Metaheuristics* chapter 13, (pp. 263–308). Springer US.

[Diego *et al.*, 2012] Diego, F. J., Gómez, E. M., Ortega-Mier, M., & García-Sánchez, Á., (2012). Parallel cuda architecture for solving de vrp with aco. In *Industrial Engineering: Innovative Networks* (pp. 385–393). Springer.

[Ezzine *et al.*, 2010] Ezzine, I. O., Semet, F., & Chabchoub, H., (2010). New formulations for the traveling repairman problem. In *8th Int. Conference of Modeling and Simulation*.

[Feo & Resende, 1995] Feo, T. A. & Resende, M. G. C. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2), 109–133, (1995).

[Fischetti *et al.*, 1993] Fischetti, M., Laporte, G., & Martello, S. The delivery man problem and cumulative matroids. *Oper. Res.*, 41(6), 1055–1064, (1993).

[Flynn, 1972] Flynn, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9), 948–960, (1972).

[Gendreau & Potvin, 2010] Gendreau, M. & Potvin, J.-Y., (2010). *Handbook of metaheuristics*, volume 146. Springer, 2nd edition.

[Glover, 1986] Glover, F. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5), 533–549, (1986).

[Glover & Kochenberger, 2003] Glover, F. & Kochenberger, G. A., (2003). *Handbook of metaheuristics*. Springer.

[Gregg & Hazelwood, 2011] Gregg, C. & Hazelwood, K., (2011). Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on* (pp. 134–144).: IEEE.

[Harris, 2007] Harris, M. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2(4), (2007).

[Holland, 1975] Holland, J. H., (1975). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press.

[Hoos & Stützle, 2004] Hoos, H. H. & Stützle, T., (2004). *Stochastic local search: Foundations & applications*. Elsevier.

[Janiak *et al.*, 2008] Janiak, A., Janiak, W. A., & Lichtenstein, M. Tabu search on gpu. *J. UCS*, 14(14), 2416–2426, (2008).

[Johnson *et al.*, 1988] Johnson, D. S., Papadimitriou, C. H., & Yannakakis, M. How easy is local search? *Journal of computer and system sciences*, 37(1), 79–100, (1988).

[Kindervater & Savelsbergh, 1997] Kindervater, G. A. & Savelsbergh, M. W. Vehicle routing: handling edge exchanges. *Local search in combinatorial optimization*, (pp. 337–360)., (1997).

[Kirk & Wen-mei, 2012] Kirk, D. B. & Wen-mei, W. H., (2012). *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2nd edition.

[Lee *et al.*, 2010] Lee, V. W., Hammarlund, P., Singhal, R., Dubey, P., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., & Chennupaty, S. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH Computer Architecture News*, 38(3), 451, (2010).

[Lenstra, 1997] Lenstra, J. K., (1997). *Local search in combinatorial optimization*. Princeton University Press.

[Martin *et al.*, 1991] Martin, O., Otto, S. W., & Felten, E. W., (1991). *Large-step Markov chains for the traveling salesman problem*. Citeseer.

[McCool *et al.*, 2012] McCool, M. D., Robison, A. D., & Reinders, J., (2012). *Structured parallel programming: patterns for efficient computation*. Elsevier.

[Melab *et al.*, 2010] Melab, N., Talbi, E.-G., et al., (2010). Gpu-based island model for evolutionary algorithms. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation* (pp. 1089–1096).: ACM.

[Minieka, 1989] Minieka, E. The delivery man problem on a tree network. *Annals of Oper. Res.*, 18(1), 261–266, (1989).

[Mladenović & Hansen, 1997] Mladenović, N. & Hansen, P. Variable neighborhood search. *Computers & Operations Research*, 24(11), 1097–1100, (1997).

[Mladenović *et al.*, 2013] Mladenović, N., Urošević, D., & Hanafi, S. Variable neighborhood search for the travelling deliveryman problem. *4OR*, 11(1), 57–73, (2013).

[Moscato *et al.*, 1989] Moscato, P. et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826, 1989, (1989).

[Mucci *et al.*, 1999] Mucci, P. J., Browne, S., Deane, C., & Ho, G., (1999). Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference* (pp. 7–10).

[Nemhauser & Trotter, 1975] Nemhauser, G. L. & Trotter, L. E. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8(1), 232–248, (1975).

[Ngueveu *et al.*, 2010] Ngueveu, S. U., Prins, C., & Wolfler Calvo, R. An effective memetic algorithm for the cumulative capacitated vehicle routing problem. *C&OR*, 37(11), 1877–1885, (2010).

[NVIDIA, 2016] NVIDIA, (2016). Cuda toolkit documentation.

[Östergard, 1999] Östergard, P. R. A new algorithm for the maximum-weight clique problem. *Electronic Notes in Discrete Mathematics*, 3, 153 – 156. 6th Twente Workshop on Graphs and Combinatorial Optimization, (1999).

[Papadimitriou & Steiglitz, 1998] Papadimitriou, C. H. & Steiglitz, K., (1998). *Combinatorial optimization: algorithms and complexity*. Courier Dover Publications.

[Pardalos & Xue, 1994] Pardalos, P. M. & Xue, J. The maximum clique problem. *Journal of global Optimization*, 4(3), 301–328, (1994).

[Penna *et al.*, 2013] Penna, P. H. V., Subramanian, A., & Ochi, L. S. An iterated local search heuristic for the heterogeneous fleet vehicle routing problem. *Journal of Heuristics*, 19(2), 201–232, (2013).

[Reinelt, 1991] Reinelt, G. Tsplib: A traveling salesman problem library. *ORSA J. on Computing*, 3(4), 376–384, (1991).

[Sahni & Gonzalez, 1976] Sahni, S. & Gonzalez, T. P-complete approximation problems. *Journal of the ACM*, 23(3), 555–565, (1976).

[Salehipour *et al.*, 2011] Salehipour, A., Sorensen, K., Goos, P., & Braysy, O. Efficient {GRASP+VND} and {GRASP+VNS} metaheuristics for the traveling repairman problem. *4OR-A Quarterly J. of Oper. Res.*, 9(2), 189–209, (2011).

[Schulz, 2013] Schulz, C. Efficient local search on the gpu - investigations on the vehicle routing problem. *Journal of Parallel and Distributed Computing*, 73(1), 14–31, (2013).

[Silva *et al.*, 2012] Silva, M. M., Subramanian, A., Vidal, T., & Ochi, L. S. A simple and effective metaheuristic for the minimum latency problem. *EJOR*, 221(3), 513–520, (2012).

[Simchi-Levi & Berman, 1991] Simchi-Levi, D. & Berman, O. Minimizing the total flow time of n jobs on a network. *IIE transactions*, 23(3), 236–244, (1991).

[Sitters, 2006] Sitters, R., (2006). *The minimum latency problem is NP-hard for weighted trees*. Springer.

[Souza *et al.*, 2010] Souza, M., Coelho, I., Ribas, S., Santos, H., & Merschmann, L. A hybrid heuristic algorithm for the open-pit-mining operational planning problem. *European Journal of Operational Research*, 207(2), 1041 – 1051, (2010).

[Stuart & Podolny, 1996] Stuart, T. E. & Podolny, J. M. Local search and the evolution of technological capabilities. *Strategic Management Journal*, 17(S1), 21–38, (1996).

[Subramanian *et al.*, 2013] Subramanian, A., Uchoa, E., & Ochi, L. S. A hybrid algorithm for a class of vehicle routing problems. *Computers & Operations Research*, 40(10), 2519–2531, (2013).

[Talbi, 2009] Talbi, E.-G., (2009). *Metaheuristics: From Design to Implementation*. Wiley Publishing.

[Toth & Vigo, 2001] Toth, P. & Vigo, D., (2001). *The vehicle routing problem*. Society for Industrial and Applied Mathematics.

[Tsitsiklis, 1992] Tsitsiklis, J. N. Special cases of traveling salesman and repairman problems with time windows. *Networks*, 22(3), 263–282, (1992).

[Van Luong *et al.*, 2013] Van Luong, T., Melab, N., & Talbi, E.-G. Gpu computing for parallel local search metaheuristic algorithms. *Computers, IEEE Transactions on*, 62(1), 173–185, (2013).

[Vidal & Alba, 2010] Vidal, P. & Alba, E., (2010). A multi-gpu implementation of a cellular genetic algorithm. In *IEEE Congress on Evolutionary Computation* (pp. 1–7).: IEEE.

[Vidal *et al.*, 2011] Vidal, T., Crainic, T. G., Gendreau, M., & Prins, C., (2011). *A unifying view on timing problems and algorithms.* Technical report, CIRRELT.

[Vidal *et al.*, 2013] Vidal, T., Crainic, T. G., Gendreau, M., & Prins, C. Heuristics for multi-attribute vehicle routing problems: A survey and synthesis. *EJOR*, 231(1), 1–21, (2013).

[Wilkinson & Allen, 2005] Wilkinson, B. & Allen, M., (2005). *Parallel Programming: techniques and applications using networked workstations and parallel computers*. Prentice Hall, 2nd edition.

[Zhu & Curry, 2009] Zhu, W. & Curry, J., (2009). Parallel ant colony for nonlinear function optimization with graphics hardware acceleration. In *2009 IEEE International Conference on Systems, Man and Cybernetics* (pp. 1803–1808).: IEEE.