Exploração Eficiente de Estruturas de Vizinhança em Ambientes Híbridos CPU/GPU

Eyder Rios, Luiz Satoru Ochi, Cristina Boeres
Instituto de Computação - Universidade Federal Fluminense
Rua Passo da Pátria, 156, CEP 24210-240, Niterói, RJ
{eyder, satoru, boeres}@ic.uff.br
Igor M. Coelho
Instituto de Matemática e Estatística - Universidade do Estado do Rio de Janeiro
Rua São Francisco Xavier, 254, CEP 20550-900, Rio de Janeiro, RJ
igor.machado@ime.uerj.br

RESUMO

A busca local é um componente essencial para muitos métodos heurísticos 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. Foram realizados ensaios em quatro diferentes configurações envolvendo as arquiteturas CPU e GPU, ocasião em que importantes aspectos inerentes à implementação dos métodos envolvidos são realçados e avaliados visando incrementar o desempenho da busca em GPU. São obtidos valores de *speedup* de até 410 vezes para uma instância de 1.000 clientes do Problema da Mínima Latência. Os resultados indicam que quando a arquitetura GPU é explorada adequadamente, ganhos de desempenho bastante significativos podem ser alcançados.

PALAVRAS CHAVE: Problema da Mínima Latência, GPU, Busca Local

Áreas Principais: Otimização Combinatória, Processamento Paralelo

ABSTRACT

The local search is an essential component of many heuristic 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. Thus, the classic local search algorithm is combined with an efficient method for neighborhoods evaluation based on movements preprocessing. Experiments were performed in four different configurations involving CPU and GPU architectures, where important aspects related to methods implementation are highlighted and assessed aiming to increase results performance in GPU. Speedup values up to 410 times were obtained to an instance of 1,000 customers for the Minimum Latency Problem. The results indicate that when GPU architecture is properly exploited, quite significant performance gains can be achieved.

KEYWORDS: Minimum Latency Problem, GPU, Local Search

Main areas: Combinatorial Optimization, Parallel Processing

1 Introdução

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 e Potvin, 2010). Atualmente, devido à presença ubíqua de paralelismo nos computadores modernos, o desenvolvimento de heurísticas paralelas tornou-se corrente. Nesse contexto, podem ser encontradas diversas contribuições que buscam explorar o paralelismo presente tanto nos algoritmos quanto nas diversas plataformas de *hardware* existentes (Alba *et al.*, 2013).

Durante a última década, as *Graphics Processing Units* (GPUs) emergiram como uma alternativa de alto poder computacional provido por sua arquitetura massivamente paralela (*manycore*). De fato, 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.

Entretanto, poucos trabalhos na literatura estão focados em reavaliar os mecanismos tradicionais de implementação de métodos heurísticos 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 heurísticas. Tipicamente, o procedimento de busca local representa o esforço computacional dominante empenhado por heurísticas baseadas nesse componente. 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 híbrida. Portanto, heurísticas baseadas em busca local devem levar em conta não só a eficácia do método, mas as características inerentes desta arquitetura híbrida, 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 realçar os aspectos inerentes ao desempenho da busca local 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, onde buscou-se destacar os requisitos relevantes ao desempenho dos elementos que representam o maior impacto na performance. 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. 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 utilizando um eficiente método de avaliação de vizinhança baseado no pré-processamento de movimentos. Foram produzidos quatro diferentes configurações do procedimento de busca local clássico orientados para arquitetura CPU e GPU considerando ou não a metodologia de pré-processamento de movimentos.

2 Arquitetura GPU

Na última década, grandes avanços tecnológicos introduziram recursos paralelos em diversas plataformas alçando-as para um novo patamar em termos de poder computacional. 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.

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 e Wen-mei, 2012). 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 e apresentando um esquema de controle lógico mais elementar cuja finalidade é favorecer o *throughput* da aplicação paralela (Cheng *et al.*, 2014).

Uma aplicação CUDA consiste em uma ou mais etapas que são executadas tanto na CPU quando na GPU. O código GPU é implementado na forma de funções C++ denominadas *kernels*. A aplicação 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. No decorrer da execução de um *kernel*, cada bloco da grade é atribuído a um único multiprocessador, 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 denominados *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 acesso à memória global da GPU é uma das elementos cruciais para o desempenho de uma aplicação CUDA, podendo forçar 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. 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.

Associada à matriz de multiprocessadores, a GPU possui um espaço de memória próprio e hierarquicamente organizado. Esse recursos, quando convenientemente explorados, configuram o elemento chave para o bom desempenho da aplicação. Apesar de sua ampla capacidade de armazenamento, a *memória global* apresenta alta latência de acesso, em torno de centenas de ciclos de *clock*. Embora o lançamento massivo de *threads* possa teoricamente contribuir para ocultação dessa latência, este mesmo recurso pode facilmente congestionar as vias de transferência de dados conduzindo, eventualmente, alguns multiprocessadores rumo à ociosidade. A transferência de dados de/para a memória global é, portanto, um componente crítico a ser considerado no projeto de aplicações baseadas em GPU. Por sua vez, a *memória local* é composta por pequenas porções de memória associadas a cada multiprocessador que podem ser acessadas em grande velocidade e de forma altamente paralela. Parte da memória local é usada como memória compartilhada que pode ser acedida

por todas as *threads* de um bloco, configurando-se num mecanismo altamente eficiente para o compartilhamento de dados neste escopo.

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. Um estudo mais detalhado sobre a implementação de métodos heurísticos em ambientes CPU/GPU pode ser encontrado em (Rios, 2016).

3 Busca Local e Busca em Vizinhança

A busca local é provavelmente um dos métodos heurísticos mais antigos e elementares. 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. 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.

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 que permitam a redução da complexidade computacional da busca, acelerando o processo de avaliação de soluções vizinhas. No entanto, tal pré-processamento traz 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 trabalho apresenta um estudo sobre a implementação eficiente de busca em vizinhança para um complexo problema de roteamento de veículos.

4 O Problema da Mínima Latência

O Problema da Mínima Latência (PML) é uma variante do Problema do Caixeiro Viajante (PCV) 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 NP-Difícil (Sahni e Gonzalez, 1976). 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 e Karakostas, 2003).

Uma solução para o PML pode ser implementada como um vetor de valores discretos representando uma rota para os clientes a serem visitados. 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: $swap(i,j)$, onde os clientes nas posições i e j da solução são permutados; $2opt(i,j)$, quando a rota delimitada pelos clientes de índices i e j é invertida; e $oropt\text{-}k(i,j)$, onde a partir do índice i, k clientes adjacentes são realocados para a posição j. Foram implementadas três variações desse último operador, para valores de $k \in \{1,2,3\}$. Todos os operadores de movimento são descritos em função dos operandos i e j, indicando a posição (índice) dos clientes na solução de base s.

5 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, já que afetam a latência de clientes subsequentes.

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. Contudo, esse método depende do cômputo de uma EDL constituída por quatro componentes principais, aqui denominados de D, W, T e C e definidos pelas Equações (1) a (4). 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.

$$D(i,j) = d(s_i, s_j) \quad (1)$$

$$W(i,j) = \begin{cases} 0 & ,i = j = 0 \\ 1 & ,i = j \\ |i-j|+1 & ,\text{c.c.} \end{cases} \quad (2)$$

$$T(i,j) = \begin{cases} 0 & ,i = j \\ t_{i,j-1} + d_s(i,j) & ,i < j \\ t_{i,j+1} + d_s(j,i) & ,i > j \end{cases} \quad (3)$$

$$C(i,j) = \begin{cases} 0 & ,i = j \\ c_{i,j-1} + d_s(i,j) & ,i < j \\ c_{i,j+1} + d_s(j,i) & ,i > j \end{cases} \quad (4)$$

$$R(i) = \begin{cases} 0 & , i = 0 \\ d(s_{i-1}, s_i) & , \text{c.c.} \end{cases} \quad (5)$$

$$X(i) = (x_i, y_i) \quad (6)$$

$$L(i) = T(0, i) \quad (7)$$

onde, D, presenta o 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, indica o número de partidas oriundas do depósito necessárias para visitar cada cliente da sub-rota delimitada por i até j; T, é o tempo de viagem acumulado entre dois clientes da solução; e C, representa o custo da viagem entre dois clientes da solução.

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.

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 e Wen-mei, 2012). Esse processo resultou na reestruturação da EDL, que culminou com as Expressões (5) a (7). Os três primeiros componentes da nova estrutura da EDL são definidos pelas Expressões (5) a (7), além da (4). 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), 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 (4).

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. O menor tamanho da EDL implica na redução do volume de dados envolvidos em transferências entre CPU e GPU o que, obviamente, reduz a latência desse processo. 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.

6 Experimentos Computacionais

A principal finalidade dos experimentos apresentados a seguir é procurar ressaltar os aspectos relevantes acerca da implementação da atualização da EDL e da busca nas cinco estruturas de vizinhança previamente descritas. Os experimentos buscam demostrar que acelerações significativas podem ser obtidas quando os aspectos intrínsecos da arquitetura alvo são considerados.

Os algoritmos utilizados nos experimentos foram implementados em C++ utilizando a biblioteca NVIDIA® CUDA™ versão 7.0 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 um computador com processador Intel® i7-4820K 3.70GHz, com 4 CPU *cores*, 10MB de *cache* L3 e 16GB memória RAM. A máquina está equipada uma GPU modelo NVIDIA® GeForce™ 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 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.

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. 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 grande velocidade e, ao final da execução do *kernelEDL*, copiados para a memória global. Durante a carga, os dados são organizados de tal forma que cada *thread* de um bloco acesse um banco de memória compartilhada distinto, evitando o problema do conflito de bancos. A combinação dessas técnicas, conduz ao uso eficiente dos dispositivos de memória mais rápidos da GPU. 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.

Foi realizado um ensaio para estimar o nível de paralelismo da implementação por meio da aferição da 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 $S_T(p) = 1/((1-f) + f/p)$. As comparações de desempenho entre algoritmos seriais e paralelos utilizam o conceito de *speedup absoluto*.

Para calcular o *speedup* teórico máximo 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 e denominadas a seguir: K, que contém código correspondente ao *overhead* de lançamento do *kernel*; P_r e P_w, representado as sequências paralelas responsáveis pela leitura e escrita dos dados da solução e da instância de/para a memória global; e S, denotando o laço serial que realiza a atualização de uma linha da matriz C. 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 ($10^{-6}s$). 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. Os tempos coletados para as 10 execuções de cada instância foram bastante estáveis, com desvio padrão sempre muito próximo de zero, o que justifica o número reduzido de amostras.

A Tabela 1 apresenta os resultados de aceleração alcançados pelo *kernelEDL* quando executado na GPU 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 $S_R(p)$.

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 1: Comparação entre *speedup* teórico máximo e *speedup* relativo obtidos para o *kernelEDL* na GTX-780, com $p = 2.304$ *cores*.

Para a GPU GTX-780, as operações de leitura/escrita paralela equivalem, em média, a 68,03% do tempo de execução do *kernel*. Portanto, uma aceleração média máxima de 3,13 seria esperada se todos os blocos da grade tivessem sido executados paralelamente nos 2.304 *cores* do dispositivo. Na prática, devido à latência de acesso à memória e a eventual serialização no escalonamento de blocos, um *speedup* 6,83% inferior foi obtido, o que ainda representa um resultado bastante satisfatório. Mesmo com parte do código serializado, um notável nível de paralelismo foi obtido em *kernelEDL*, 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. 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%. 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 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. A Figura 1 apresenta os resultados do experimento quando o *kernelEDL* é executado na GTX-780. Conforme a expectativa, a configuração serial B1 apresenta a pior performance, induzida pela baixa taxa de ocupação do multiprocessador. O melhor balanceamento para esta GPU foi obtido pela configuração B384. Por questão de clareza, o gráfico 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.

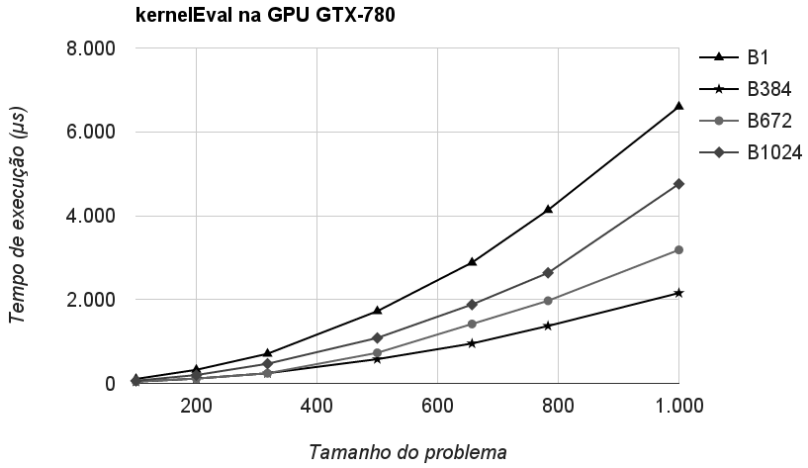

Figura 1: Análise da taxa de ocupação do kernelEDL na GPU GTX-780.

6.2 Avaliação da Vizinhança

Os experimentos apresentados a seguir têm como objetivo 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.

6.2.1 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. 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.

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. Neste contexto, foram realizados experimentos similares aos que resultaram na Figura 1, de onde foi concluído que a melhor configuração foi de 384 *threads*/bloco para a GTX-780.

A seguir são apresentados os resultados computacionais obtidos pelos *kernels* implementados. A Tabela 2 apresenta valores médios obtidos por 11 execuções dos *kernels* testados nesse trabalho na GPU GTX-780. 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 três estruturas de vizinhança testadas. Para a vizinhança 2*opt*, quando a avaliação da vizinhança é realizada sem o uso de EDL (colunas 2 e 3), um *speedup* máximo de 410,83 é alcançado pela GPU GTX-780. Esse resultado foi possível porque a avaliação de uma grande quantidade de computação, $O(n^3)$, tirou proveito do paralelismo massivo empenhado pela GPU. Conforme esperado, quando a CPU utiliza a EDL o tempo de execução do algoritmo cai drasticamente, de forma que a estratégia puramente baseada em CPU (coluna 4) é mais rápida que a realizada pela GPU, com um *speedup* máximo de 2,04, que representa um desempenho modesto frente ao potencial paralelo oferecido pelo dispositivo. Este resultado sugere que o montante de trabalho paralelo executado na GPU não foi suficiente para compensar os custos de transferências de dados e alcançar acelerações mais significativas. Apesar disso, uma comparação entre as colunas $S_A(p)$ mostra que o *speedup* é incrementado à medida que o número de *cores* GPU aumenta.

n	**2*opt***				***oropt*-1**				***swap***			
	sem EDL		**com EDL**		**sem EDL**		**com EDL**		**sem EDL**		**com EDL**	
	tempo CPU	S_A **GPU**	**tempo CPU**	S_A **GPU**	**tempo CPU**	S_A **GPU**	**tempo CPU**	S_A **GPU**	**tempo CPU**	S_A **GPU**	**tempo CPU**	S_A **GPU**
100	531	31,24	52	0,68	470	24,74	78	0,72	28	2,00	73	1,24
200	4.258	118,28	231	1,29	2.990	57,50	311	1,28	127	6,35	324	2,19
318	13.102	187,17	506	1,44	9.319	66,56	735	1,60	354	12,21	849	2,72
500	52.426	274,48	1.300	1,71	44.750	133,58	1.840	2,01	979	16,88	2.068	2,84
657	130.607	363,81	2.243	1,85	111.171	145,89	3.210	2,24	1.590	19,63	3.407	2,78
783	220.227	340,38	3.162	1,84	205.228	103,34	4.518	2,26	2.118	16,81	4.333	2,15
1000	489.296	**410,83**	5.424	**2,04**	473.436	**150,68**	10.462	**3,43**	3.177	16,05	10.197	3,47
Méd		**246,60**		**1,55**		**97,47**		**1,94**		**12,85**		**2,48**

Tabela 2: Resultados da avaliação da vizinhança do 2*opt*, *oropt*-1 e *swap* na GPU GTX-780 ($p = 2.304$).

6.2.2 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. 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 movimento implica na relocação de $|i - j| + k$ clientes da solução, o que leva a um número de operações da ordem de $O(n)$. Uma vez que 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*.

Os resultados apresentados a seguir seguem a mesma metodologia dos experimentos realizados para o *kernel2Opt*. A Tabela 2 exibe os resultados para execução do *kernelOrOpt* para o operador *oropt*-1 ($k = 1$). Como pode ser visto na tabela, um *speedups* absolutos máximos de 150,68 foi obtido, respectivamente, pela GPU GTX-780 para a instância de maior porte. Mais uma vez, o método baseado em EDL apresenta pior desempenho 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.

Os resultados referentes aos operadores *oropt*-2 e *oropt*-3 não serão discutidos porque refletem o desempenho obtido por *oropt*-1. 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.

6.2.3 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*.

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 $O(n^2)$. Os resultados apresentados na Tabela 2 corroboram essa expectativa. Por conta da menor complexidade, a estratégia baseada em CPU 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 GPU sem EDL (coluna 3) alcance um *speedup* máximo de 19,63, essa aceleração é menor que as obtidas por *kernel2Opt* e *kernelOrOpt*. Como esperado, a estratégia GPU baseada em EDL apresenta os resultados menos relevantes. Esse comportamento foi induzido pelos custos de computação e a alta latência de acesso à memória global envolvendo a EDL, que foram mais lentos que a computação dos mesmos dados quando necessário.

7 Conclusões

Este trabalho lidou com o Problema da Mínima Latência (PML), um problema de otimização NP-Difícil. O poderoso paralelismo encontrado nas GPUs modernas foi explorado com o objetivo de acelerar a busca local, o componente computacionalmente mais intensivo de muitos métodos heurísticos. Para pôr à prova o potencial paralelo da busca em GPU foi considerada uma eficiente técnica de aceleração da avaliação de vizinhanças baseada no pré-processamento de movimentos. Para permitir a avaliação de movimentos em tempo constante, esse método depende do cômputo de Estruturas de Dados Locais (EDL). Neste contexto, foram experimentados quatro diferentes estratégias de busca, orientadas para CPU e GPU e envolvendo ou não o uso de EDL.

Durante os experimentos foram realçados importantes aspectos relativos ao desempenho

na implementação da busca em vizinhança na arquitetura GPU. Nesse processo, um interessante ensaio demonstrou que as técnicas de implementação foram capazes de atingir *speedups* muito próximos do limite teórico máximo determinado pela Lei Amdahl. Cinco estruturas de vizinhança clássicas foram avaliadas: *swap*, 2*opt*, *oropt*-1, *oropt*-2 e *oropt*-3. Uma aceleração máxima bastante significativa de 410,83 vezes foi obtida para uma instância de 1.000 clientes para o MLP. A avaliação da vizinhança do operador *swap* apresentou desempenho modesto frente ao potencial paralelo da GTX-780 ocasionado, principalmente, pelo pequeno montante de trabalho atribuído a cada *thread* GPU. Apesar do método baseado em EDL ser bastante eficiente na plataforma CPU, sua adaptação para a arquitetura GPU não apresentou uma aceleração proporcional aos recursos paralelos disponibilizados. 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.

8 Agradecimentos

Este trabalho foi parcialmente financiado pelo CNPq, FAPERJ e UESPI.

Referências

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

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

Arora, S. e Karakostas, G. (2003), Approximation schemes for minimum latency problems. *SIAM Journal on Computing*, v. 32, n. 5, p. 1317–1337.

Blum, A., Chalasani, P., Coppersmith, D., Pulleyblank, B., Raghavan, P. e Sudan, M. The minimum latency problem. *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, p. 163–171. ACM, 1994.

Cheng, J., Grossman, M. e McKercher, T. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

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

Gendreau, M. e Potvin, J.-Y. *Handbook of metaheuristics*. Springer, 2nd edição, 2010.

Kirk, D. B. e Wen-mei, W. H. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2nd edição. ISBN 9780124159921, 2012.

Lenstra, J. K. *Local search in combinatorial optimization*. Princeton Univ. Press, 1997.

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

Mucci, P. J., Browne, S., Deane, C. e Ho, G. Papi: A portable interface to hardware performance counters. *Proceedings of the department of defense HPCMP users group conference*, p. 7–10, 1999.

Rios, E. *Exploração de estratégias de busca local em ambientes CPU/GPU*. Tese de doutorado, Universidade Federal Fluminense, 2016.

Sahni, S. e Gonzalez, T. (1976), P-complete approximation problems. *Journal of the ACM*, v. 23, n. 3, p. 555–565.

Silva, M. M., Subramanian, A., Vidal, T. e Ochi, L. S. (2012), A simple and effective metaheuristic for the minimum latency problem. *EJOR*, v. 221, n. 3, p. 513–520.