Uma heurística paralela na GPU para o problema do Caixeiro Viajante Duplo com Múltiplas Pilhas

Anolan Milanés, †Arne Løkketangen
Molde University College – Norway
Email: anolan@dcc.ufmg.br

RESUMO

O problema do Caixeiro Viajante Duplo com Múltiplas Pilhas é NP-difícil. Embora vários algoritmos baseados em metaheurísticas foram propostos para resolvê-lo, a escalabilidade continua sendo um desafio na hora de processar grandes instâncias. Este artigo apresenta a paralelização em um ambiente de GPU de um desses algoritmos baseado em programação dinâmica, e discute os problemas relacionados e as estratégias seguidas para resolver as dificuldades encontradas.

PALAVRAS CHAVE. Heurística Paralela. GPU. Programação dinâmica. Caixeiro viajante.

ABSTRACT

The Double Traveling Salesman Problem with Multiple Stacks is a NP-hard problem, and several metaheuristic algorithms have been proposed to solve it. The scalability of these approaches remains a challenge, though, when the necessity for processing larger instances arises. This paper is based on the parallelization, on a GPU setting, of a dynamic programming-based heuristic, discuses the issues involved and the strategies followed to resolve the difficulties encountered.

KEYWORDS. Parallel heuristics. GPU. Dynamic programming. Traveling tournament problem.

1. Introdução

O problema do Caixeiro Viajante Duplo com Múltiplas Pilhas (CVDMP) introduzido por Petersen e Madsen (2009) modela um problema realista em que é necessário transportar bens entre duas cidades. Na cidade fornecedora um veículo coleta e armazena os itens em uma dentre várias pilhas (de tamanho constante e limitado) seguindo a rota de coleta. Posteriormente, na cidade destino, os itens são distribuídos respeitando a rota de entrega e a política de empacotamento aplicada durante a coleta, ou seja, sempre tirar o item a ser entregue do topo de uma das pilhas. Cada rede tem um depósito onde a rota correspondente começa e termina. O problema consiste então em determinar o par de rotas (de coleta e entrega) nas duas redes disjuntas e a alocação de uma pilha para cada bem a ser transportado minimizando a distância total percorrida.

O CVDMP é NP-difícil. Embora vários algoritmos baseados em metaheuristicas foram propostos para resolvê-lo, a escalabilidade continua sendo um desafio na hora de processar grandes instâncias. Este artigo apresenta a paralelização na GPU da heurística proposta por Urrutia e Løkketangen (2012) para o CVDMP: o PD-CVDMP. Nessa abordagem, o cálculo do custo das rotas ótimas de coleta e entrega para diferentes esquemas de carregamento das pilhas é feito usando programação dinâmica. Entretanto, a programação dinâmica é computacionalmente custosa. Isto motiva o uso de uma plataforma massivamente paralela, como é o caso da GPU. Este artigo discute os problemas relacionados a paralelização do algoritmo na GPU e as estratégias seguidas para resolver estes problemas.

2. Um algoritmo de programação dinâmica sequencial para o CVDMP

As instâncias do CVDMP consistem em um número n de clientes, duas matrizes de distância (para as redes de coleta e entrega) especificando as distâncias entre cada par de clientes e a distância de cada cliente ao depósito, e dois valores inteiros constantes s e c que especificam o número de pilhas disponíveis e sua capacidade, respectivamente. O plano de carregamento é uma matriz que determina o arranjo de armazenamento dos clientes uma vez que a coleta foi finalizada. A matriz contém uma permutação dos n clientes e cada fila representa o estado de cada pilha.

Soluções ao problema podem ser vistas como duas permutações dos clientes representando as rotas de coleta e entrega e um plano de carregamento. A figura 1 mostra um plano de carregamento e um par de rotas compatíveis para esse plano de carregamento.

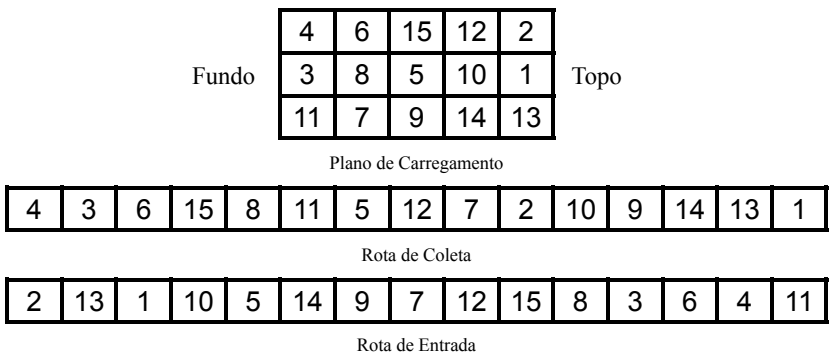

Figura 1: Um plano de carregamento e um par de rotas de coleta e entrega compatíveis.

Casazza et al. (2012) mostraram que é possível calcular rotas ótimas no que diz respeito a um determinado plano de carregamento em tempo polinomial (contanto que o número de pilhas seja fixo) através de programação dinâmica. Urrutia e Løkketangen (2012) propuseram uma heurística para o CVDMP baseada em uma busca local no plano de carregamento. Um algoritmo de programação dinâmica computa as rotas ótimas de coleta e entrega para o plano de carregamento resultante, dai o nome PD-CVDMP para a heurística. Basicamente, a proposta é um algoritmo multi-start que usa busca tabú seguido por um único passo de pesquisa em uma vizinhança *swap* para melhorar as soluções iniciais. O algoritmo 1 mostra o pseudo-código da heurística proposta.

Algoritmo 1: A heurística PD-CVDMP

entrada: Instancia CVDMP
saida : melhor solução encontrada antes do critério de parada ser satisfeito

```
1 while critério de parada não satisfeito do
2 |   sol ← HeuristicaConstrutivaRnd
3 |   repeat
4 |   |   sol ← Busca_Tabu(sol)
5 |   |   sol ← Passo_Unico_Swap(sol)
6 |   until ambos os procedimentos de melhora falham melhorando sol
```

Repare que para um plano de carregamento dado, a computação das rotas ótimas de coleta e entrega podem ser executadas independentemente com duas chamadas a programação dinâmica.

A abordagem PD-CVDMP tem várias vantagens quando comparada com abordagens baseadas em rotas:
(i) o tamanho do espaço de solução é bem menor: $O(n!)$ em lugar de $O((n!)^2)$;
(ii) a abordagem não sofre de problemas de inviabilidade.

Duas vizinhanças simples foram propostas nesse trabalho para explorar o espaço de soluções dos plano de carregamento: *swap* (tamanho da vizinhança $n \cdot (n-1)/2$) e $3-exchange$ (tamanho da vizinhança $n \cdot (n-1) \cdot (n-2)/3$).

Em ambas as vizinhanças, a evaluação dos movimentos é feita computando os custos das rotas ótimas de coleta e entrega para o plano de carregamento resultante, que por sua vez são determinadas chamando duas vezes a programaçào dinâmica (uma para a rota de coleta e outra para a de entrega). Portanto, já que o algoritmo de programação dinâmica executa em tempo $O(s \cdot c^s)$, o custo computacional de evaluar a vizinhança completa é $O(s \cdot c^s \cdot n^2)$ para a vizinhança swap e $O(s \cdot c^s \cdot n^3)$para a 3-exchange. Se s se considera fixo e igual a 3, o custo computacional é $O(n^5)$ e $O(n^6)$ respectivamente.

Para atacar esta complexidade, Urrutia e Løkketangen (2012) incluiram uma função de estimação de custo do vizinho para assistir ao procedimento de busca local na identificação de vizinhos promissores. O estimador de custo determina, em O(1) por vizinho, um limite superior no custo que é obtido através da programação dinâmica após o movimento.

Em Petersen e Madsen (2009) foram introduzidos três conjuntos de instâncias de benchmark de 12, 33 e 66 clientes com 20 instâncias cada. Posteriormente foram

estudadas também instâncias com 132 clientes. Todas as instâncias têm 3 pilhas e a capacidade das pilhas é tal que todas as pilhas estão cheias quando a rota de coleta foi completada, ou seja, a capacidade das pilhas é 11 para instâncias com 33 clientes, e 22 para as instâncias de 66 clientes.

O PD-CVDMP tem proporcionado resultados computacionais competitivos para instâncias de tamanho médio com 33 clientes. No entanto, na sua forma actual, não é adequado para casos muito maiores. Uma forma de superar este problema está no processamento paralelo. Este trabalho concentra-se na forma de resolver em paralelo a seção de programação dinâmica do algoritmo, pois ela é a sua operação computacionalmente mais pesada.

3. Contexto

O interesse no uso de unidades de processamento gráfico (GPUs) como arquiteturas de computação de propósito geral vem crescendo nos últimos anos. GPUs foram projetadas focando na computação paralela de alto desempenho. Em consequência, apresentam uma quantidade massiva de processadores e a cache e o controle são pequenos quando comparados às CPUs modernas. O modelo de execução garante que enquanto threads estão bloqueadas esperando por dados, outras podem ser escalonadas para execução (dado um um grau de ocupação suficiente).

Atualmente, vários frameworks facilitam o acesso dos programadores às capacidades de computação paralela da GPU. Neste documento nos referiremos especificamente ao *Compute Unified Device Architecture*, CUDA (NVIDIA (2012)).

O modelo de computação GPU + CPU é heterogêneo: a GPU atúa como co-processador da CPU para acelerar operações computacionalmente intensivas. As tarefas são submetidas pela CPU (*host*) na GPU (*device*) como chamadas a função com uma sintaxe específica que inicia a execução de um *kernel*: o código a ser executado pelas threads da GPU. CUDA organiza a computação paralela usando as abstrações de threads, blocos e grades. Blocos são grupos de threads. Grade é um grupo de blocos. O código é executado na GPU em grupos de threads que executam a mesma construção simultaneamente. Esses grupos são chamados de *warps*. Nas GPUs NVIDIA atuais, o tamanho de um warp é de 32 threads.

A hierarquia de memória da arquitetura CUDA é composta por vários tipos de memória: local, compartilhada, constante, de textura e a global. Elas são otimizadas para diferentes padrões de acesso e diferem no tamanho máximo, a visibilidade e a persistência entre chamadas ao kernel. Por exemplo, o acesso à memória compartilhada é mais rápido que o acesso à memória global em duas ordens de magnitude, mas o tamanho dela é bem menor. Diferente da memória global, a memória compartilhada não persiste entre chamadas a kernel.

4. Desenvolvendo um algoritmo paralelo de programação dinâmica para a GPU

Existem vários algoritmos conhecidos para a paralelização de problemas de programação dinâmica (Grama et al. (2003)). Entretanto, a maior parte dos trabalhos focam na execução da programação dinâmica em sistemas baseados em CPU em lugar de unidades de processamento gráfico. Trabalhos recentes (Luong et al. (2013),Schulz (2013)) tratam o problema da paralelização na GPU de problemas de otimização. Outros (Xiao et al. (2009), Boyer et al. (2011)) desenvolveram algoritmos para resolver problemas de programação dinâmica na GPU. O presente trabalho pretende combinar ambas idéias na resolução do problema CVDMP.

As estratégias para a paralelização da programação dinâmica podem ter granularidade fina ou grossa. Uma estratégia de paralelização de grau fino consiste em distribuir a avaliação de cada vizinho entre p processadores. Uma análise da computação da tabela de custo da programaçào dinâmica revela que a computação dos estados em todas as etapas depende somente dos resultados nos estágios imediatamente precedentes. No entanto, a necessidade de sincronização no final da execução de cada estágio pode afetar a capacidade da GPU para ocultar a latência de memória intercalando a execução de threads bloqueadas com prontas.

Uma estratégia simples de granularidade grossa para a paralelização do PD-CVDMP consiste na tradução direta da implementação sequencial apresentada por Urrutia e Løkketangen (2012) para a GPU. Esta abordagem é embaraçosamente paralela. O passo de sincronização entre estágios não é necessário, mas ainda é precisa sincronizar a computação dos custos das rotas ótimas de coleta e entrega para cada solução pois eles devem ser reduzidos (somados) para determinar a rota de menor custo.

Uma formulação deste algoritmo utiliza tantos processadores (p) quanto vizinhos (n). Neste caso, todos os vizinhos podem ser processados simultaneamente. Uma variação pode ser considerada, em que o modelo anterior é convertido para um modelo menor com p' processadores, com p'<p. Então, a vizinhança é dividida em n/p' grupos. Cada passo de computação pode ser seguido por uma redução, ou a redução pode seguir o fim da avaliação da coleta e a entrega. O algoritmo 2 implementa este segundo caso.

Algoritmo 2: O algoritmo paralelo

entrada: Matrix das pilhas
saida : Indice e custo do melhor movimento

```
1 while bucketNumber < vizinhos/bucketSize do
2     submeter kernel-PD (a ser executado por bucketSize threads)
3     bucketNumber ← bucketNumber + 1
4 submeter kernel de redução
5 ler da memória da GPU o custo e índice do melhor movimento não tabu
6 executar o melhor dos movimentos não tabu avaliados na matriz de pilhas
```

4.1. A representação da tabela de custos

Um dos problemas mais complexos a resolver no cálculo do custo das soluções do PD-CVDMP paralelo é a dimensão da tabela de custos da programação dinâmica. Esta tabela de custos deve ser mantida de forma a computar as rotas ótimas de coleta e entrega. Ela pode ser representada como uma matriz de dimensão $(c+1)\cdot(c+1)\cdot(c+1)\cdot s$ para a coleta e outra para a entrega, para cada vizinho.

Uma vez que o acesso a memória compartilhada na GPU é mais rápido que a memória global, podemos considerar implementar uma cache da tabela de custos na memória compartilhada a fim de aumentar a velocidade de execução. No entanto, o tamanho da memória partilhada varia de 16kb a 48kb, muito pequeno, mesmo para o exemplo 33 clientes. Em consequência, nessa abordagem a matriz deve ser alocada em memória local ou global. Estes problemas relacionados com a tabela de

custos afetam a escalabilidade, e forçam a recorrer a técnicas, tais como a divisão da vizinhança quando aumenta o tamanho da instância como mostrado no algoritmo 2.

4.2. Comunicação entre as threads e coordenação CPU-GPU

Abordagens baseadas em paralelismo de dados são adequadas para a GPU. Quando uma tarefa pára enquanto aguarda por dados, a latência de memória pode ser escondida intercalando a execução de outras tarefas. Isso poderia ser dificultado em casos em que as tarefas estão fortemente ligadas, pois pode acontecer de não existir tarefas prontas para intercalação. Outra razão para evitar a comunicação das threads tanto quanto é possível é quando a comunicação envolve diferentes blocos, uma vez que, neste caso, o único meio é a memória global, com acesso lento. Outro aspecto importante para reduzir o impacto da comunicação no desempenho é a procura de formas de sobrepor computação e comunicação.

Depois que o kernel executando a avaliação do custo retorna, é necessário somar cada par de custos correspondente a coleta e a entrega para cada vizinho avaliado, a fim de determinar o custo mínimo e seu índice. A fim de manter esses procedimentos fracamente acoplados, um novo kernel é iniciado após o retorno da avaliação da vizinhança. Esse kernel irá adicionar cada par correspondente aos custos de coleta e de entrega e encontrar o melhor vizinho.

Para diminuir os custos de comunicação, o cálculo final do custo mínimo é feito na GPU. É possível deslocar a matriz de resultados para a CPU e calcular o mínimo no host. Entretanto, a sobrecarga devido ao movimento de dados entre CPU e GPU é alto (Gregg e Hazelwood (2011)). A execução em um kernel separado garante que todas as tarefas tenham terminado sem a necessidade explícita de sincronização, e minimiza a sobrecarga de transmissão de dados, porque todos os dados a serem reduzidos já estão disponíveis na memória global. Nesse caso, apenas o resultado final (o melhor movimento) é enviado de volta para o host. O custo da submissão do kernel é a motivação para executar o kernel de redução somente no final da computação da programação dinâmica toda em lugar de após a avaliação de cada bucket.

4.3. Configuração

A determinação do tamanho do bloco não é simples. Em geral, é preferível utilizar um tamanho grande de bloco, a fim de maximizar a ocupação do dispositivo. No entanto, existe um compromisso entre o tamanho do bloco e o número de registros e a quantia de memória compartilhada disponíveis para as threads do bloco, uma vez que esses recursos são compartilhados. Em cada caso, o tamanho do bloco mais eficaz foi determinado experimentalmente.

A placa de vídeo Geforce GTX 690 inclui duas GPUs. É possível tirar proveito disso através da execução de um kernel de coleta em um dispositivo e um kernel de entrega no outro. Essa abordagem tem a vantagem de se trabalhar com uma única tabela de distâncias em cada GPU. No entanto, a fase de redução exigiria a comunicação entre os dois dispositivos, ou trazer todos os dados para o host. Já que a comunicação na GPU é uma operação onerosa, foi escolhida a opção de submeter kernels que executam tanto a coleta quanto a entrega em ambos os dispositivos. Os resultados da computação ficam armazenados na GPU e são posteriormente processados por um kernel de redução no final da execução. Esse kernel é responsável por somar cada par correspondente de custos de coleta e entrega e encontrar o melhor movimento. Em seguida, o host recebe os melhores resultados calculados em cada dispositivo e determina finalmente o melhor movimento.

Detalhes de implementação são fundamentais no desempenho da programação dinâmica paralela.

Em Cuda, é possível configurar a memória, a fim de favorecer a cache da memória global contra a quantidade de memória compartilhada. Já que nossa implementação não faz uso da memória compartilhada, optamos por aumentar o espaço de cache, o que representou uma melhoria substancial no desempenho.

Outro problema que deve ser tratado é a presença de divergências no código executado pelas threads de um mesmo warp. Divergências na avaliação de uma condição pelas threads de um warp são tratadas executando em sequência, primeiramente as threads do warp que avaliam verdadeiro e seguidamente as que avaliaram falso a condição. Isto claramente afeta o paralelismo da aplicação. Para eliminar no possível divergencias no código, os casos de fronteira da programação dinâmica foram tratados separadamente no início do procedimento, eliminando a necessidade de acessá-los no restante do código. O código já foi desenrolado na implementação seqüencial do PD-CVDMP, com ganho considerável em desempenho. Por outro lado, isto pode limitar o número de execuções simultâneas possíveis e, consequentemente, comprometer o desempenho da aplicação.

5. Resultados computacionais

O algoritmo PD-CVDMP foi codificado em C++ e Cuda 5.0 e executado na CPU e na GPU sobre 20 instâncias da literatura com 66 clientes e 3 stacks cada. Os experimentos foram conduzidos em um computador com processador Intel Core i7, 2.67GHz e 7,8 GB de RAM, e uma GPU Geforce GTX 690 com capacidade computacional 3.0, 2 dispositivos, 8 Multiprocessadores cada e 192 CUDA Cores por multiprocessador.

Instance	GPU (ms)	CPU (ms)
R00	586.33	63313.46
R01	587.74	63340.14
R02	586.99	63193.46
R03	588.07	63227.28
R04	583.93	63311.6
R05	592.70	63279.36
R06	585.71	63339.7
R07	591.98	63232.8
R08	584.57	63283.2
R09	586.78	63274.86
R10	589.87	63305.08
R11	587.20	63286.28
R12	585.57	63296.2
R13	592.89	63255.26
R14	583.10	63299.76
R15	588.94	63244.72
R16	592.53	63268.9
R17	588.50	63265.48
R18	583.53	63273.02
R19	590.53	63279.12
Normalized Std Deviation	0.53%	0.06%

Tabela 1: Tempo de execução do PD-CVDMP na GPU e CPU - 66 clientes

A tabela 1 mostra o tempo de execução do PD-CVDMP paralelo nas 20 instâncias para um plano de carregamento fixo sem integração com a heurística.

Os resultados mostram que a execução da programação dinâmica na GPU em todos os casos foi duas ordens de magnitude mais rápida do que na CPU, para a vizinhança 3-exchange completa. Isto representa um total de 183040 execuções da programação dinâmica.

6. Considerações finais

Este trabalho apresenta uma implementação paralela do PD-CVDMP. O nosso interesse está na avaliação da vizinhança completa em lugar de apenas uma porção como a heurística seqüencial faz. Dessa forma esperamos encontrar boas soluções que teriam sido descartadas. Resultados computacionais preliminares mostram que a implementação do algoritmo de GPU supera a implementação seqüencial CPU. Esses resultados podem ser esperados uma vez que a GPU é adequada para a execução de milhares de threads. Atualmente estamos trabalhando na integração do algoritmo com a heurística. Aspectos que precisam ser definidos incluem se o processamento do status tabu dos movimentos deve ser executado na GPU ou na CPU.

A outra linha de trabalho está na implementação da abordagem de grau fino. Essa abordagem é mais complexa por precisar de sincronização. Por outro lado, tem várias vantagens quando comparada a versão de grau grosso que incluem o maior grau de paralelismo, a possibilidade de usar a memória compartilhada e a maior escalabilidade.

Agradecimentos

Este trabalho foi parcialmente financiado pelo CNPq Brasil e o projeto Dominant II Noruega.

Referências

Boyer, V., Baz, D. E. e Elkihel, M. (2011), Dense dynamic programming on multi gpu, *in* Y. Cotronis, M. Danelutto e G. A. Papadopoulos, eds, 'PDP', IEEE Computer Society, pp. 545–551.

Casazza, M., Ceselli, A. e Nunkesser, M. (2012), 'Efficient algorithms for the double traveling salesman problem with multiple stacks', *Computers & OR* (39), 1044–1053.

Grama, A., Karypis, G., Kumar, V. e Gupta, A. (2003), *Introduction to Parallel Computing (2nd Edition)*, 2 edn, Addison Wesley.

Gregg, C. e Hazelwood, K. (2011), Where is the data? Why you cannot debate CPU vs. GPU performance without the answer, *in* 'Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software', ISPASS '11, IEEE Computer Society, pp. 134–144.

Luong, T. V., Melab, N. e Talbi, E.-G. (2013), 'Gpu computing for parallel local search metaheuristic algorithms', *IEEE Trans. Computers* **62**(1), 173–185.

NVIDIA (2012), 'CUDA programming guide 4.2'.

Petersen, H. L. e Madsen, O. B. G. (2009), 'The double travelling salesman problem with multiple stacks - formulation and heuristic solution approaches', *European Journal of Operational Research* **198**(1), 139–147.

Schulz, C. (2013), 'Efficient local search on the GPU - Investigations on the vehicle routing problem', *J. Parallel Distrib. Comput.* **73**(1), 14–31.

Urrutia, S. e Løkketangen, A. (2012), A dynamic programming based local search approach for the double travelling salesman problem with multiple stacks, *in* 'Matheuristics2012'.

Xiao, S., Aji, A. M. e Feng, W.-c. (2009), On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit, *in* 'Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems', ICPADS '09, IEEE Computer Society, Washington, DC, USA, pp. 26–33.