Proposta de paralelização em GPUs CUDA do algoritmo MPS para resolução do problema de encontrar os K-Caminhos Mais Curtos Sem Repetições de Vértices em grafos direcionados e não direcionados

Daniel Morais dos Reis[1], **Álvaro Martins Espíndola** [2], **Sérgio Ricardo de Souza**[1], **Anolan Milanés** [3]

[1] Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Belo Horizonte – MG - Brazil

[2]Universidade Federal de Lavras (UFLA)
Lavras – MG - Brazil

[3]Molde University College
Molde, Noruega

danielmorais@cefetmg.br, alvaromaresp@gmail.com

sergio@cefetmg.br, anmi@himolde.no

Resumo. *O Problema dos K-Caminhos mais Curtos sem Repetições de Vértices (K-*Shortest Loopless Paths *– KSLP) consiste em encontrar $K \mid K > 1$ caminhos, sem repetições de vértices, classificados em ordem crescente de distância, em um grafo $G(V, E)$, no qual V representa o conjunto de vértices e E o conjunto dos arestas que os conecta. Dada a dificuldade de obter bom desempenho para solução do problema, este trabalho propõe a paralelização em GPUs CUDA do algoritmo mais veloz existente na literatura para o problema.*

Abstract. *K-Shortest Loopless Paths (KSLP) consists of finding $K \mid K > 1$ paths without vertex repetitions, ranked in ascending order of distance, in a graph $G(V, E)$, in which V represents the set of vertices and E the set of edges that connects them. Given the difficulty of obtaining good performance to solve the problem, this work proposes the parallelization in CUDA GPUs of the fastest algorithm in the literature for the problem.*

1. Introdução

Proposto por Hoffman e Pavley em 1959 [Hoffman and Pavley 1959], o problema dos K-Caminhos Mais Curtos Sem Repetições de Vértices (K-*Shortest Loopless Paths* – KSLP) consiste em encontrar $K \mid K > 1$ caminhos sem repetições de vértices, classificados em ordem crescente de distância, em um grafo $G(V, E)$, no qual V representa o conjunto de vértices e E o conjunto dos arestas que os conecta, entre um vértice de origem s e um de destino t. Este problema possui várias aplicações em problemas reais de otimização, como roteamento em redes, rastreamento de múltiplos objetos, planejamento de movimentação de robôs, planejamento de layout de circuitos, planejamento de tráfego, bancos de dados, dentre outras [Eppstein 1998, Berclaz et al. 2011,

Chen and Prasanna 2016, Feng 2014, Hershberger et al. 2007, Srivastava et al. 2015, Wen et al. 2013, Yen 1971, Singh and Singh 2015a].

Dado que ainda é desafiador obter bom desempenho na resolução do problema para instâncias de grandes proporções, este trabalho apresenta uma estratégia de solução através da primeira proposta da literatura para paralelização em GPUs Nvidia CUDA© [Cook 2013] do algoritmo MPS [Martins et al. 1999]. Este algoritmo, além de possuir grande aptidão para aplicação de técnicas de paralelização, possui desempenho destacado na literatura e é capaz de operar em grafos direcionados ou não.

O artigo está organizado como segue: a Seção 2 exibe uma revisão bibliográfica sobre o problema; a Seção 3 mostra uma explanação sobre o funcionamento das GPUs NVidia CUDA; a Seção 4 descreve as estruturas de dados para representação do problema; a Seção 5 introduz a proposta de paralelização do algoritmo; os experimentos computacionais são mostrados na Seção 6; e, por fim, a Seção 7 conclui o artigo.

2. Revisão da literatura

[Bock et al. 1957, Pollack 1961, Clarke et al. 1963, Sakarovitch 1968] apresentam soluções com complexidade de tempo exponencial para o KSLP. Em 1971, [Yen 1971] propôs um algoritmo polinomial, de complexidade de tempo $O(Kn(n \log n + m))$, o qual opera reduzindo o KSLP para $O(K)$ instâncias do Problema de Substituição de Caminhos (*Replacement Paths Problem*) [Bernstein 2010]. O algoritmo de Yen é a base do desenvolvimento das estratégias de solução vistas em [Katoh et al. 1982, Martins et al. 1999, Feng 2014].

A complexidade do algoritmo de Yen só foi superada em [Gotthilf and Lewenstein 2009], que apresentaram um algoritmo de complexidade $O(Kn(m + n \log \log n))$. Este algoritmo executa inicialmente um pré-processamento advindo de um dos algoritmos para solução do Problema de Caminhos Mais Curtos para Todos os Pares de Vértices (*All-Pair-Shortest-Paths* – APSP) e sua complexidade final está diretamente ligada à complexidade do algoritmo de APSP escolhido. Para atingir a complexidade mostrada em [Gotthilf and Lewenstein 2009], é utilizado o algoritmo de Pettie [Pettie 2004], o qual possui complexidade $O(n(m + nloglogn))$ e baseia-se na Árvore de Componentes de Thorup [Thorup 1999]. No entanto, devido à dificuldade imposta pela estrutura, ainda não é conhecida nenhuma implementação em sua complexidade correta. [Katoh et al. 1982] apresentam um algoritmo, denominado KIM, que reduz a complexidade de tempo para solucionar o problema para $O(K(m + n \log n))$ em grafos não direcionados.

Através de alterações feitas no algoritmo de Yen e em técnicas advindas do algoritmo de KIM, em [Martins et al. 1997, Martins et al. 1999] foi desenvolvido o algoritmo MPS, que opera através de um atalho na obtenção de caminhos candidatos em suas iterações. Para obter este atalho, após uma etapa inicial de pré-processamento, na qual se calcula os caminhos mais curtos (*shortest paths*) a partir do vértice de destino t para todos os demais vértices $v \in G$ e aproveita-se tais caminhos na obtenção de novos, reduzindo, assim, a complexidade de tempo da solução do problema para $O(m \log n + Kn)$ no pior caso.

Em [Singh and Singh 2015b] é proposta a paralelização do algoritmo de Yen em GPUs Nvidia CUDA©, chegando a atingir um *speedup* de $6,5$ vezes em relação à versão

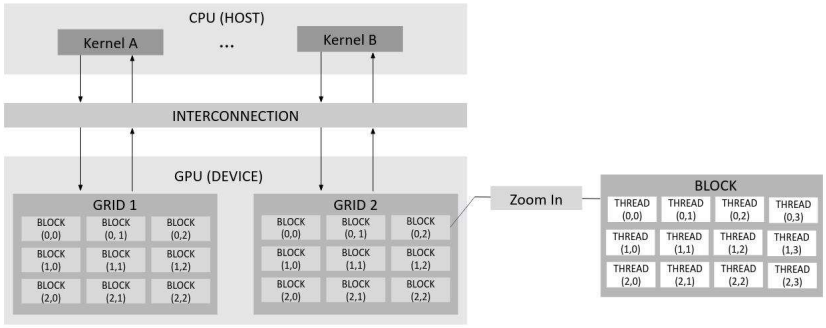

Figura 1. Arquitetura CUDA.

serial; entretanto, os experimentos foram realizados com instâncias que possuem dimensões de $|V|$ e $|E|$ adequadas para existirem *threads* suficientes para realizar o processamento do grafo por completo em paralelo em única iteração.

3. GPUs Nvidia CUDA

As *Graphic Processing Units* (GPUs) operam através do processamento simultâneo de instruções em uma coleção de *Streaming Processors* (SM) [Cook 2013], o que as tornam eficientes na execução de algoritmos que possuem padrões recorrentes de acesso a dados e alta intensidade de execução aritmética [Luebke 2008, Cook 2013]. As GPUs Nvidia CUDA© permitem a implementação de *kernels* nestes dispositivos (*devices*), que, por sua vez, executam múltiplas *threads*. Estes *kernels* são invocados pelo processador *host* (CPU) após a execução, normalmente serial, de código para carregamento de dados [Cook 2012], conforme mostrado na Figura 1.

Na GPU, a grade de processamento é dividida em blocos, os quais possuem *threads*. Toda grade, bloco e *thread* é identificado por um ID, advindo da API CUDA, e que é utilizado para estabelecer as atribuições de cada *thread*, seus momentos de execução e respectivos dados. Cada SM executa um conjunto de 32 *threads* (*warp*) por vez. A disponibilização de dados acontece em uma estrutura hierárquica de memória, na qual cada nível possui um tempo e restrição [Nvidia 2018], sendo: (i) *Private* ou *Register memory*: mais veloz e com acesso privado a cada *thread*, existente normalmente em menor quantidade nos *devices*; (ii) *Shared Memory*: segundo mais veloz, compartilhado entre as *threads* de um bloco e existente em segunda menor quantidade nos *devices*; (iii) *Global Memory*: mais lento entre os níveis, é compartilhado entre todos os blocos de uma grade, mas disponível em maior quantidade para utilização nos *devices*; (iv) *Texture* e *constant memory*: dados somente leitura disponíveis na memória DRAM do *device*.

4. Estruturas de dados

Neste artigo referências a elementos específicos de *arrays* unidimensionais são dadas por $Array[i]$ ou $Array_{[i]}$, ou seja, o i-ésimo elemento de $Array$. Apesar de, na prática, serem implementados em *arrays* unidimensionais contínuos, *arrays* aninhados têm seus elementos referenciados por $Array_{[i,y]}$, ou seja o y-ésimo elemento do i-ésimo *array* armazenado em $Array$. Além disso, este artigo usa as seguintes expressões: (i) s: vértice de origem de um caminho tal que $s \in V$; (ii) t: vértice de destino de destino de um caminho tal que $t \in V$; (iii) T^k: *array* aninhado contendo caminhos candidatos a *caminho mais curto*

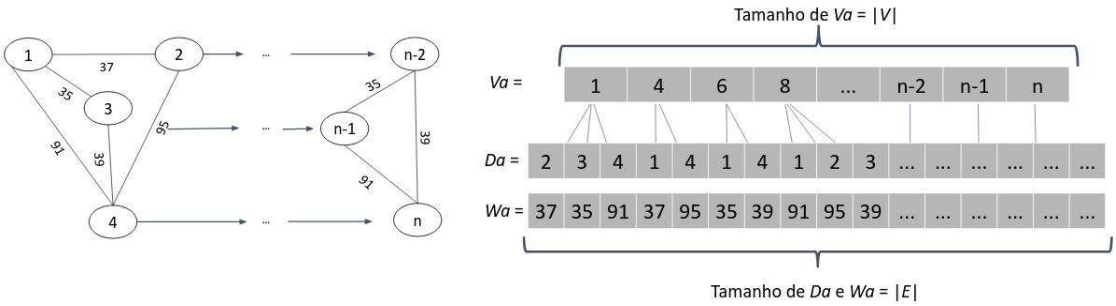

Figura 2. Grafo representado em uma lista de adjacências compacta.

na k-ésima iteração; (iv) T^*: *array* aninhado, contendo o *caminho mais curto* de cada vértice de G até t; (v) X: *array* aninhado com os k*-caminhos mais curtos* exclusivos e ordenados; (vi) $C^{[index]}$, $L^{[index]}$ e $A^{[index]}$: *arrays* aninhados em que cada índice contém, respectivamente, o custo, quantidade de arestas e disponibilidade do respectivo caminho em um *array* aninhado.

Por questões de eficiência no processamento e armazenamento, nas GPUs os grafos devem ser representados em matrizes esparsas do tipo *linha esparsa comprimida* (*compressed sparse row* - CSR) ou *coluna esparsa comprimida* (*Compressed sparse column* - CSC) [Golub and Van Loan 1996, Hildebrand 1987], compondo uma lista de adjacências compactada, conforme pode ser observado na Figura 2. O *array* Va contém em cada posição um valor que representa o índice inicial das informações de suas arestas, como os vértices de destino em Da e peso em W_a. No grafo *não direcionado* representado na Figura 2, cada aresta é composta por dois arcos, um $< i \rightarrow j >$ e outro $< j \rightarrow i >$. Opcionalmente criados, os *arrays* Sa e Ra, de tamanho $|E|$, respectivamente, contém os vértices de origem dos arcos e o índice de seus opostos, o que permite a eliminação de *loops* dos algoritmos na busca por informações do grafo.

5. Paralelização CUDA do algoritmo MPS

Algoritmos para localizar o caminho mais curto em grafos normalmente possuem a etapa de relaxamento de vértices inerentemente sequencial e sua recorrente utilização é comum nos demais algoritmos para o KSLP. O pré-processamento realizado pelo MPS exclui a utilização posterior destes algoritmos, uma vez que caminhos localizados na $(k-1)$-ésima iteração são decompostos e concatenados aos caminhos encontrados na fase de pré-processamento para compor novos candidatos. Invertendo a direção das arestas de G e em seguida executando um algoritmo de caminho mais curto tendo t como origem, o MPS cria, em sua fase de pré-processamento, a árvore de caminhos T_t^*. A implementação paralela do algoritmo de caminho mais curto aqui utilizada é vista em [Harish and Narayanan 2007, Dijkstra 1959].

O algoritmo de Dijkstra paralelizado mostrado em [Harish and Narayanan 2007] retorna o *array* c_a, contendo a distância total do caminho mais curto da origem até cada respectivo vértice representado por seus índices. Para extrair o caminho mais curto de um determinado vértice até t, utilizamos o Algoritmo 1.

O Algoritmo 1 recebe, como parâmetros, os *arrays* Sa, Da, Wa, ca e o *array* $prev$, de tamanho $|V|$. Após a comparação do peso de cada aresta $e \in E$ somado ao peso

Algoritmo 1: Função Device ExtractPreviousEdges

```
   Entrada: Sa, Da, Wa, ca, prev
1  private<var> tid ← getThreadId();
2  private or shared<var> stride ← getStrideSize();
3  shared or global<array of size |V|> prev;
4  for th ← tid; th < |E|; th ← th + stride do
5  |   if (Wa[th] + ca[Sa[th]] = ca[Da[th]]) and (ca[Da[th]] < ∞) then
6  |   |   prev[Da[th]] ← th;
7  |   end
8  end
9  barrier();
10 return prev;
```

Algoritmo 2: Função Device ExtractPath

```
   Entrada: o, t, Sa, Wa, ca, prev
1  private<var> cost ← 0, length ← 0;
2  private<array of size |V| > path ←;
3  length ← 1;
4  next ← t;
5  while true do
6  |   path[length] ← prev[next];
7  |   cost ← cost + Wa[length];
8  |   if (Sa[path[length]] = o) then
9  |   |   break;
10 |   else
11 |   |   next = Sa[path[length]];
12 |   |   length ← length + 1;
13 |   end
14 end
15 return path, cost, length;
```

final do caminho mais curto do seu vértice de origem em ca, o *array* $prev$ é retornado, contendo, em cada posição, o índice da última aresta que compõe o caminho mais curto do respectivo vértice.

O Algoritmo 2 extrai, através do *array* $prev$, o caminho mais curto de um vértice o até t. Este algoritmo executa uma estrutura de repetição que compõe o caminho com o índice de suas arestas no sentido $t \rightarrow o$, o armazena no *array* $path$, juntamente com seus dados de custo total e quantidade de arestas nas variáveis $cost$ e $lenght$. A descoberta de uma aresta que compõe um caminho é precedida de outra, o que faz o algoritmo ser sequencial, no entanto, podendo ser executadas várias instâncias simultaneamente.

5.1. *Forward Sorted Star Form*

O *Forward Sorted Star Form* permite que o MPS, a cada iteração na composição de novos caminhos candidatos, utilize a aresta seguinte disponível do vértice v_i. Este formato é dado pelo grafo G, com todas as suas arestas ordenadas de maneira crescente em seus respectivos vértices v_i, de acordo com o seu custo reduzido [Martins et al. 1999]. Por sua vez, o custo reduzido de uma aresta $< i, j >$ é dado por $\overline{c}^{<i,j>} = \pi(j) - \pi(i) + Wa_[< i, j >]$, no qual $\pi(j)$ e $\pi(i)$ representam o custo do respectivo caminho do vértice até t em $T_t^\star$. A ordenação neste formato foi utilizada pela primeira vez em [Eppstein 1998] e repetida com sucesso em [Martins et al. 1999].

Ordenar as arestas de cada vértice de forma independente é análogo ao problema de *ordenação segmentada* (*segmented sort*) [Schmid et al. 2016]. Trabalhos na literatura fazem estudos sobre estratégias de *segmented sort* para GPUs CUDA. *Benchmarks* para

a ordenação com utilização de paralelismo dinâmico [Nvidia 2018] são demonstrados em [Bergstrom and Reppy 2012]. Utilizando a função `radix sort` da biblioteca `Thrust` [Nvidia 2018], os autores discutem também o *segmented sort*, mas não apresentam os resultados de testes de desempenho para o problema. A `Parallel Fix Strategy` mostrada em [Schmid et al. 2016] apresenta até então o melhor desempenho para o *segmented sort* em GPUs. Ela consiste em pré-processar todo o *array* com os segmentos a serem ordenados, de forma que somente uma execução do algoritmo de ordenação seja necessária. A adaptação da estratégia para a etapa de pré-processamento do MPS é apresentada no Algoritmo 3 e exemplificada na Figura 3.

Algoritmo 3: Função Device MPS Parallel Forward Sorted Star Form

Entrada: G, K, s, t
shared<var> msb $\leftarrow \emptyset$;
for th $\leftarrow$ tid; th $< |E|$; th $\leftarrow$ th + strideSize **do**
  $\overline{C}^{Ea}_{[th]} \leftarrow C^{T^*}_{[Da[th]]} - C^{T^*}_{[Ea[th]]} + C^{Wa}_{[th]}$;
end
barrier();
msb $\leftarrow$ encontrar via redução o maior valor armazenado em $\overline{C}^{Ea}$;
barrier();
if tid = 0 **then**
  msb $\leftarrow \log_2 msb$;
end
barrier();
for th $\leftarrow$ tid; th $< |E|$; th $\leftarrow$ th + strideSize **do**
  fix $\leftarrow Ea_{[th]} + shiftLeft(Sa_{[th]}, msb)$;
  $\overline{C}^{Ea}_{[th]} \leftarrow \overline{C}^{Ea}_{[th]}$ + fix;
end
barrier();
Call Um algoritmo de ordenação paralelo para ordenar o *array* E_a de acordo com os valores armazendos em $\overline{C}^{Ea}$ com todas as threads disponíveis;

Vertex = [Binary]	1 = [1]			2 = [10]				3 = [11]		4 = [100]					...
Da	2	3	4	1	4	5	6	1	4	1	2	3	5	7	...
$\overline{C}^{Ea}_{[Si,j>]}$ *(divided by 1000 and rounded up)*	21	20	48	21	50	19	20	20	22	48	50	22	36	**52 (highest value)**	...
Binary of $\overline{C}^{Ea}_{[Si,j>]}$	10101	10100	110000	10101	110010	10011	10100	10100	10110	110000	110010	10110	100100	110100	...
1st - MSB Index	$[log_2^{52}] \rightarrow$ **6**														
2nd - Pre sort	**10**10101	**10**10100	**11**10000	**100**10101	**10**110010	**100**10011	**100**10100	**110**10100	**110**10110	**100**110000	**100**110010	**1000**10110	**100**100100	**100**110100	...
3rd - Sorted	**10**10100	**10**10101	**11**10000	**100**10011	**100**10100	**100**10101	**10**110010	**110**10100	**110**10110	**1000**10110	**100**100100	**100**110000	**100**110010	**100**110100	...
4rd - Post sort	10100	10101	110000	10011	10100	10101	110010	10100	10110	10110	100100	110000	110010	110100	...
Da	3	2	4	5	6	1	4	1	4	3	5	1	2	7	...

Figura 3. Segmented Sorting através da Parallel Fix Strategy

O Algoritmo 3 calcula o custo reduzido das arestas, localiza o maior deles via redução e associa o *bit mais significativo* (do inglês *most significant bit* - MSB). Em seguida, adiciona, após o MSB, os bits correspondentes ao número do segmento somado ao resultado da Algoritmo `ShifLeft(`msb`)`, a qual é responsável por movimentar bits para a esquerda. O *array* Da é ordenado de acordo com seu novo custo reduzido por um algoritmo de ordenação paralelo. A Figura 3 mostra o grafo da Figura 2 em *forward sorted star form*, antes e após a aplicação do Algoritmo 3.

5.2. CUDA *based* MPS

A implementação paralelizada em CUDA do algoritmo MPS proposta neste trabalho é exibida nos Algoritmos 4 e 5. Após a fase de pré-processamento, o caminho mais curto

Algoritmo 4: Kernel MPS

Entrada: K, Sa, Da, Wa, Ra, X, L, C, A, s, t

shared<var> $k \leftarrow 1$, $flag \leftarrow false$, $tid \leftarrow$ getThreadId(), $stride \leftarrow$ getStrideSize();
shared<arrays of size $|V| > ca$, $prev$;
$ca \leftarrow$ **Call** algoritmo Dijkstra para encontrar distância do *caminho mais curto* de t até cada vértice $v \in V$ de G;
if $ca[t]$!= ∞ **then**
    $prev \leftarrow$ ExtractPreviousEdges(Sa, Da, Wa, ca);
    barrier();
    for th $\leftarrow$ tid; th $< |V|$; th $\leftarrow$ th + stride **do**
        $T^*_{[th,t]}, C^{T^*}_{[th,t]}, L^{T^*}_{[th,t]} \leftarrow$ ExtractPath(th, t, Sa, Wa, ca, $prev$);
    end
end
barrier();
if tid = 0 **then**
    $L^X_{[k]} \leftarrow L^{T^*}_{[s,t]}$; $C^X_{[k]} \leftarrow C^{T^*}_{[s,t]}$; $L^{T^k}_{[k]} \leftarrow L^{T^*}_{[s,t]}$; $C^{T^k}_{[k]} \leftarrow C^{T^*}_{[s,t]}$;
end
for th $\leftarrow$ tid; th $< L^{T^*}_{[s,t]}$; th $\leftarrow$ th + stride **do**
    $X_{[k,th]} \leftarrow T^*_{[s,t,th]}$; $T^k_{[k,th]} \leftarrow T^*_{[s,t,th]}$;
end
barrier();
Call algoritmo Forward Sorted Star Form para reorganizar G (arrays Sa, Da, Wa, Ra);
barrier();
while (k < K) and (flag = false) **do**
    flag $\leftarrow$ true;
    barrier();
    Call T^k, C^{T^k}, L^{T^k}, A^{T_k}, count $\leftarrow$ MPS Generate Candidates($tmppc$, k, Va, Sa, Da, Wa, Ra, X, L, C, A, s, t);
    barrier();
    $k \leftarrow k + 1$;
    $sht \leftarrow$ através de redução encontrar o *caminho mais curto* disponível em $T_{[k]}$ através da análise de A^{T^k} e C^{T^k};
    barrier();
    if tid = 0 **then**
        $L^X_{[k]} \leftarrow L^{T^k}_{[sht]}$;
    end
    for th $\leftarrow$ tid; th $< L^{T^k}_{[sht]}$; th $\leftarrow$ th + strideSize **do**
        $X_{[k,th]} \leftarrow T^k_{[sht,th]}$;
    end
end
return X;

de s a t é extraído de T^*_t e armazenado em X, sendo, então, o primeiro dos K caminhos mais curtos. Para a geração de caminhos candidatos, a partir do $(k-1)$-ésimo caminho disponível em X dá-se origem a um novo, composto por uma *caminho raiz*, ou seja, os vértices entre s e v_i e unidos a um *caminho complementar* advindo de $T^*_{v_{i+1}}$, sendo v_{i+1} o vértice de destino da próxima aresta no vértice v_i do *caminho raiz* não utilizada em T^k. Caso o caminho candidato não possua repetição de vértices, é então armazenado em T^k. O processo se repete até que todos os vértices de v_i até t do $(k-1)$-ésimo caminho sejam analisados. Antes de iniciar a próxima k iteração, o caminho mais curto disponível em T^k é encontrado via redução, analisando-se os *arrays* aninhados C^{T_k} e A^{T_k}, sendo, então, transferido para X e marcado como indisponível em $A^{T_k}_{[sht]}$.

O Algoritmo 5, em conjunto com o Algoritmo 6, agrupa as *threads*, de forma que cada grupo atue em um vértice diferente do *caminho raiz* e o MPS gere um ou vários caminhos candidatos simultaneamente. O Algoritmo 5 é utilizada previamente à iteração da função de geração de decomposição de caminhos e armazena o resultado na variável tps.

Algoritmo 5: Função Device MPS Generate Candidates

Entrada: $tmppc, k, Va, Sa, Da, Wa, Ra, X, L, C, A, s, t$

shared *or* **private**<var> tpc ← 0, smc ← 0, tid ← getThreadId(), stride ← $\frac{L^X_{[k-1]}}{tpc}$, gridSize ← getGridSize();
shared<array set of size $|V| * L^X_{[k]} - k$ > loopless ← true;
if tid = 0 **then**
    tpc ← getThreadsSetsSize($tmppc$, $L^X_{[k]} - k$); smc ← $\lfloor \frac{gridSize}{tpc} \rfloor$;
end
barrier();
for th ← $\lfloor \frac{tid}{tpc} \rfloor$; th ≤ $L^X_{[k]} - k$; th ← th + smc **do**
    private<var> cn ← $\lfloor \frac{th}{tpc} \rfloor$, $vertex \leftarrow Sa[X_{[k,cn]}]$;
    if $A^{T^k}_{[cn]}$ ¡$Va[vertex]$ **then**
        edge ← $A^{T^k}_{[cn]} + 1$; $A^{T^k}_{[cn]} \leftarrow A^{T^k}_{[cn]}$ + 1;
    else
        next th;
    end
    barrier();
    for thc ← modulus($\frac{tid}{tpc}$); thc ≤ $k + L^{T^*}_{[Da[edge]]}$ and (loopless[cn] = false); thc ← thc + tpc **do**
        if (thc ≤ k) and (loopless[cn] = false) **then**
            $q_{[k,cn,thc]} \leftarrow X_{[k,thc]}$;
            $count_{[k,cn,Sa[X_{[k,thc]}]]} \leftarrow count_{[k,cn,Sa[X_{[k,thc]}]]}$ + 1;
            if $count_{[k,cn,Sa[X_{[k,thc]}]]} > 1$ **then**
                begin atomic loopless[cn] ← false; **end atomic**
            end
        else if (thc - k) < $L^{T^*}_{[Da[edge]]}$ **then**
            $q_{[k,cn,thc]} \leftarrow T^*_{[Da[edge],(thc-k)]}$;
            $count_{[k,cn,Sa[T^*_{[Da[edge],(thc-k)]}]]} \leftarrow count_{[k,cn,Sa[T^*_{[Da[edge],(thc-k)]}]]}$ + 1;
            if $count_{[k,cn,Sa[T^*_{[Da[edge],(thc-k)]}]]} > 1$ **then**
                begin atomic loopless[cn] ← false; **end atomic**
            end
    end
    barrier();
    if loopless[cn] = true **then**
        if modulus($\frac{tid}{tpc}$) = 0 **then**
            $L^{T^k}_{[(k+1),cn]} \leftarrow k + L^{T^*}_{[Da[edge]]}$; count ← count + 1;
        end
        barrier();
        for thc ← modulus($\frac{tid}{tpc}$); (thc < $L^{T^k}_{[(k+1),cn]}$) and (loopless[cn] = true; thc ← thc + tpc **do**
            $T^k_{[(k+1),cn,thc]} \leftarrow q_{[k,cn,thc]}$;
        end
    end
    $cn \leftarrow cn + smc$;
end
return T^k, C^{T^k}, L^{T^k}, A^{T_k}, *count*;

O valor a ser alocado em tps é dado de acordo com a quantidade de *threads* disponíveis e a quantidade de vértices a serem decompostos no caminho. Já o Algoritmo 6 recebe via parâmetro o percentual mínimo de *threads* a serem utilizadas em cada vértice.

6. Experimentos

Para comprovar o desempenho da proposta de paralelização do MPS, duas modalidades de experimentos foram realizadas: (i) experimento comparativo com diferentes configurações do CUDA-MPS via múltiplas gerações de caminhos candidatos e a versão CUDA-YEN; (ii) experimento comparativo entre os Algoritmos CUDA-MPS e CUDA-YEN. A implementação do algoritmo CUDA-YEN utilizada está disponível em

Algoritmo 6: Função Device: getThreadsSetsSize

Entrada: $tmpps, nsets$

```
1  private<var> grid ← getGridSize();
2  if tmpps != ∅ then
3      if grid ≥ nsets * |V|) then
4          tps ← ⌊grid / (nsets*|V|)⌋;
5      else if ⌊grid / nsets⌋ > (grid * tmpps) then
6          tps ← ⌊grid / nsets⌋;
7      else
8          tps ← ⌈grid * tmpps⌉;
9  else if grid / nsets > 1 then
10     if grid / nsets > nsets * |V| then
11         tps ← |V|;
12     else
13         tps ← ⌈grid / nsets⌉;
14 return tps;
```

Tabela 1. Resultados do experimento comparativo com diferentes configurações de múltiplas gerações de caminhos candidatos do CUDA-MPS vs CUDA-YEN ($K = 100$).

Instância	YEN	MPS						
		1.0	0.5	0.3333	0.25	0.2	0.1	0.05
Abilene	41.91	35.06	29.06	20.02	15.03	14.09	13.95	~~14.85~~
Cost-266	312.87	285.593	207.593	195.513	174.073	159.713	152.813	~~153.003~~
France	189.98	150.14	121.14	116.08	92.96	79.54	79.4	~~80.39~~
Germany50	714.24	517.66	443.66	434.62	429.63	428.69	421.79	421.6
India35	617.98	508.12	400.12	388.04	366.6	352.24	352.1	351.12
NewYork	417.24	339.81	276.81	271.75	248.63	235.21	228.31	~~228.42~~
Norway	542.78	343.4	256.4	247.36	242.37	241.43	241.29	240.31
Pioro40	611.37	600.61	496.61	484.53	463.09	448.73	441.83	~~449.02~~
Polska	578.32	482.13	384.13	379.07	355.95	342.53	342.39	342.2
Ta1	478.45	329.85	245.85	236.81	231.82	230.88	223.98	216
Zib54	649.56	461.261	390.261	378.181	356.741	342.381	342.241	328.541

[Singh and Singh 2015a]. Ambos os experimentos foram executados em um computador dotado de processador Intel Core i7, 16 GB RAM, GPU Geforce GTX 1060 e drive NVMe. Como instâncias de dados utilizadas para alimentar os algoritmos, foi utilizado o acervo *Survivable Network Design Library* (SNDLib) [Zuse-Institute 2020].

A Tabela 1 exibe os resultados do experimento comparativo do desempenho do Algoritmo de Yen paralelizado e do Algoritmo MPS. As colunas com porcentagens sob o MPS indicam a quantidade mínima de *threads* aplicada para geração de cada solução candidata. Conforme nota-se nesta Tabela, somente após a redução do percentual mínimo de *threads* por candidato para 5% é que ocorre o fim da progressão de desempenho do MPS; no entanto, ainda assim é superior ao algoritmo de Yen. Estes resultados encontram-se com o texto em formato tachado.

A Figura 4 mostra a comparação do desempenho dos algoritmos de Yen e MPS paralelizados em CUDA para a instância Cost266 com diferentes valores de K. Conforme pode ser notado, o MPS tem sempre *Speedup* sublinear.

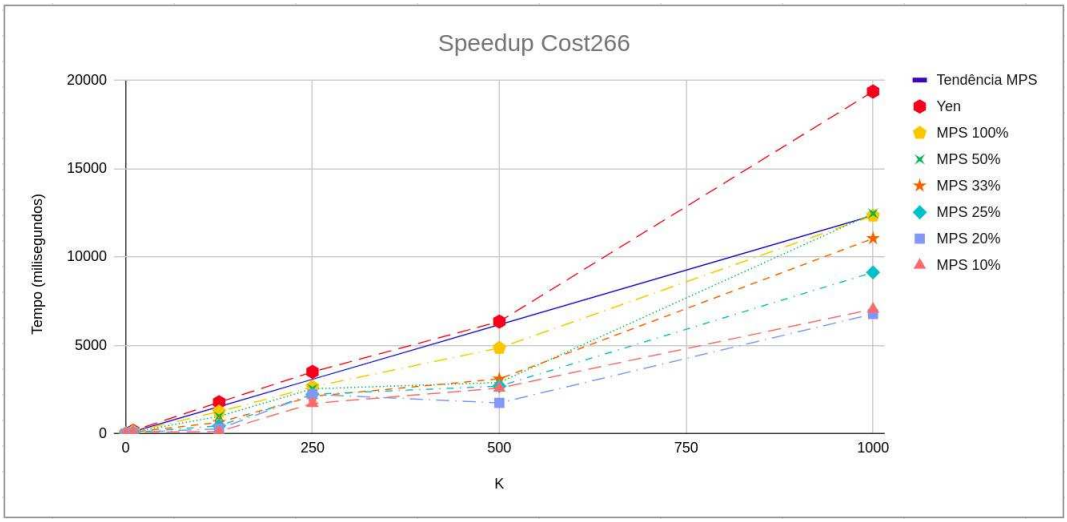

Figura 4. Tempo de execução dos algoritmos com diferentes valores de K - Instância Cost266.

7. Conclusão

Dada a importância e ampla aplicação prática na resolução do problema de encontrar os K-Caminhos Mais Curtos Sem Repetições de Vértices, este artigo propôs a paralelização de um algoritmo já conhecido para o problema, qual seja, o algoritmo MPS, usando as GPUs Nvidia CUDA. A paralelização do MPS leva vantagem sobre o outro algoritmo previamente paralelizado em GPUs presente na literatura, dada sua menor quantidade de execuções do algoritmo de caminho mais curto, o qual é inerentemente sequencial em sua etapa de relaxamento de vértices.

De acordo com os experimentos mostrados, observa-se a ampliação do desempenho do algoritmo MPS quando configura-se apropriadamente a geração simultânea de múltiplos caminhos candidatos, de acordo com a instância a ser analisada.

Referências

Berclaz, J., Fleuret, F., Turetken, E., and Fua, P. (2011). Multiple object tracking using k-shortest paths optimization. *IEEE transactions on pattern analysis and machine intelligence*, 33(9):1806–1819.

Bergstrom, L. and Reppy, J. (2012). Nested data-parallelism on the GPU. In *ACM SIGPLAN Notices*, volume 47, pages 247–258. ACM.

Bernstein, A. (2010). A nearly optimal algorithm for approximating replacement paths and K shortest simple paths in general graphs. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 742–755, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Bock, F., Kantner, H., and Haynes, J. (1957). *An algorithm (the r-th best path algorithm) for finding and ranking paths through a network.* Armour Research Foundation.

Chen, R. and Prasanna, V. K. (2016). Accelerating equi-join on a CPU-FPGA heterogeneous platform. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 212–219. IEEE.

Clarke, S., Krikorian, A., and Rausen, J. (1963). Computing the N best loopless paths in a network. *Journal of the Society for Industrial and Applied Mathematics*, 11(4):1096–1102.

Cook, S. (2012). *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

Cook, S. (2013). *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Dijkstra, E. W. (1959). A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269:271.

Eppstein, D. (1998). Finding the k shortest paths. *SIAM Journal on computing*, 28(2):652–673.

Feng, G. (2014). Improving space efficiency with path length prediction for finding k shortest simple paths. *IEEE Transactions on Computers*, 63(10):2459–2472.

Golub, G. H. and Van Loan, C. F. (1996). *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, USA.

Gotthilf, Z. and Lewenstein, M. (2009). Improved algorithms for the k simple shortest paths and the replacement paths problems. *Inf. Process. Lett.*, 109:352–355.

Harish, P. and Narayanan, P. (2007). Accelerating large graph algorithms on the GPU using CUDA. In *International conference on high-performance computing*, pages 197–208. Springer.

Hershberger, J., Maxel, M., and Suri, S. (2007). Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms (TALG)*, 3(4):45.

Hildebrand, B. F. (1987). *Introduction to Numerical Analysis: 2nd Edition*. Dover Publications, Inc., USA.

Hoffman, W. and Pavley, R. (1959). A method for the solution of the n th best path problem. *Journal of the ACM (JACM)*, 6(4):506–514.

Katoh, N., Ibaraki, T., and Mine, H. (1982). An efficient algorithm for k shortest simple paths. *Networks*, 12(4):411–427.

Luebke, D. (2008). CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE International Symposium on Biomedical imaging: from nano to macro*, pages 836–838. IEEE.

Martins, E. d. Q. V., Pascoal, M. M. B., and dos Santos, J. L. E. (1997). A new algorithm for ranking loopless paths. Technical report, Research Report (CISUC), Universidade de Coimbra.

Martins, E. d. Q. V., Pascoal, M. M. B., and dos Santos, J. L. E. (1999). Deviation algorithms for ranking shortest paths. *International Journal of Foundations of Computer Science*, 10(03):247–261.

Nvidia (2018). CUDA toolkit documentation v10.1.105.

Pettie, S. (2004). A new approach to all-pairs shortest paths on real-weighted graphs. *Theor. Comput. Sci.*, 312(1):47–74.

Pollack, M. (1961). Letter to the editor - the k-th best route through a network. *Operations Research*, 9(4):578–580.

Sakarovitch, M. (1968). The k shortest chains in a graph. *Transportation Research*, 2(1):1 – 11.

Schmid, R., Pisani, F., Borin, E., and Cáceres, E. (2016). An evaluation of segmented sorting strategies on GPUs. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1123–1130. IEEE.

Singh, A. P. and Singh, D. P. (2015a). Implementation of k-shortest path algorithm in GPU using CUDA. *Procedia Computer Science*, 48:5–13.

Singh, A. P. and Singh, D. P. (2015b). Implementation of k-shortest path algorithm in GPU using CUDA. *Procedia Computer Science*, 48:5 – 13. International Conference on Computer, Communication and Convergence (ICCC 2015).

Srivastava, A., Chen, R., Prasanna, V. K., and Chelmis, C. (2015). A hybrid design for high performance large-scale sorting on fpga. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE.

Thorup, M. (1999). Undirected single source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394. Announced at FOCS'97.

Wen, Q., Zhou, Q., Zhao, C., and Ma, X. (2013). Fixed-point realization of lattice-reduction aided mimo receivers with complex k-best algorithm. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5031–5035. IEEE.

Yen, J. Y. (1971). Finding the k shortest loopless paths in a network. *management Science*, 17(11):712–716.

Zuse-Institute (2020). Survivable fixed telecommunication network design. `http://sndlib.zib.de/home.action`, Visitado pela última vez em 10 de julho, 2020.