Paralelização do Algoritmo Floyd-Warshall usando GPU

Roussian R. A. Gaioso,
Walid A. R. Jradi,
Lauro C. M. de Paula,
Wanderley de S. Alencar,
Wellington S. Martins,
Hugo Alexandre D. do Nascimento
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Brasil
{roussiangaioso, walid, lauropaula,
wanderley, wellington, hadn}@inf.ufg.br

Edson N. Cáceres
Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
Campo Grande, Brasil
edson@facom.ufms.br

***Resumo*—Este artigo apresenta uma implementação paralela baseada em *Graphics Processing Unit* (GPU) para o problema da identificação dos caminhos mínimos entre todos os pares de vértices em um grafo. A implementação é baseada no algoritmo Floyd-Warshall e tira o máximo proveito da arquitetura *multithreaded* das GPUs atuais. Nossa solução reduz a comunicação entre a *Central Processing Unit* (CPU) e a GPU, melhora a utilização dos *Streaming Multiprocessors* (SMs) e faz um uso intensivo de acesso aglutinado em memória para otimizar o acesso de dados do grafo. A vantagem da implementação proposta é demonstrada por vários grafos gerados aleatoriamente utilizando a ferramenta *GTgraph*. Grafos contendo milhares de vértices foram gerados e utilizados nos experimentos. Os resultados mostraram um excelente desempenho em diversos grafos, alcançando ganhos de até 149x, quando comparado com uma implementação sequencial, e superando implementações tradicionais por um fator de quase quatro vezes. Nossos resultados confirmam que implementações baseadas em GPU podem ser viáveis mesmo para algoritmos de grafos cujo acessos à memória e distribuição de trabalho são irregulares e causam dependência de dados.**

***Palavras-chave*—GPU, Caminho mínimo, Floyd-Warshall, Computação paralela.**

I. Introdução

Na Teoria dos Grafos existem alguns algoritmos sequenciais capazes de identificar os caminhos mínimos entre todos os pares de vértices existentes num grafo conexo, dirigido e com arestas ponderadas. Na literatura, destacam-se dois deles: (1) o de Donald B. Johnson, proposto em 1977; e (2) o de Robert Floyd e Stephen Warshall (Floyd-Warshall), publicado em 1962. Todos possuem uma ampla variedade de aplicações em diversas áreas, tais como: bioinformática, roteamento de tráfego em redes, sistemas distribuídos, ou qualquer problema que possa ser representado por um grafo no qual as arestas sejam ponderadas e cujos valores sejam linearmente acumulados à medida que a rede é percorrida [1].

Geralmente, problemas relacionados a grafos são complexos e exigem bastante esforço computacional para sua resolução. O problema do fecho transitivo e o problema do caminho mínimo, quando aplicado a todos os pares de vértices num grafo conexo e dirigido, são exemplos desta dificuldade [2].

O algoritmo Floyd-Warshall é considerado eficiente em relação ao espaço de armazenamento por possuir uma complexidade O(V^2). Entretanto, possui uma complexidade de tempo O(V^3), onde V representa o número de vértices do grafo [3]. Devido a essa complexidade cúbica de tempo, trabalhos recentes têm explorado o uso de GPUs na tentativa de obter soluções paralelas de melhor desempenho. Dentre eles destacam-se as implementações propostas por Dehne *et al.* [1], Aini *et al.* [2], Harish *et al.* [4], Katz e Kider [5], Ridi *et al.* [6], Borgwardt *et al.* [7] e Jian *et al.* [8].

GPUs são microprocessadores dedicados a realizar operações ligadas a aplicativos gráficos 2D e 3D. Dentre tais aplicativos, podem ser citados os de *Computer Aided Design* (CAD), *Computer Aided Manufacturing* (CAM), jogos, interface gráfica com o usuário. Graças à sua arquitetura altamente paralelizada e especializada, são muito mais eficientes na manipulação de gráficos que as *Central Processing Units* (CPUs), projetadas para a execução de código sequencial. Em essência, as GPUs consistem de vários núcleos primariamente focados em operações de ponto flutuante, massivamente usados nas funções gráficas dos algoritmos de arte-finalização (ou $rendering$). A grande quantidade destes *microchips*, trabalhando em paralelo, é o que permite o alto poder computacional de tais processadores [9].

A partir do ano 2000, as GPUs incorporaram técnicas de *pixel shading*, onde cada $pixel$ pode ser processado por um pequeno programa que inclui texturas adicionais, o que é feito de maneira similar com vértices geométricos, antes mesmo de serem projetados na tela. A fabricante de GPUs $NVIDIA^{®}$ foi a primeira a produzir placas com tais características [10] [11] [12]. À medida que as GPUs evoluem, mais flexibilidade de programação é introduzida, adicionando suporte a programas maiores e de maior complexidade (incluindo controladores de fluxo, tais como: laços, sub-rotinas, *branches*,

etc.), mais registradores e aumento na precisão numérica.

Hoje em dia, devido ao altíssimo poder computacional oriundo de uma arquitetura intrinsecamente paralela, as GPUs são capazes de manipular enormes cargas de trabalho, com programação flexível e facilitada por meio do uso de diversas *Application Programming Interfaces* (APIs) disponíveis. Isto têm motivado pesquisadores ao redor do mundo a conceberem algoritmos para execução sob estes equipamentos, usando-as como coprocessadores matemáticos para computação de propósito geral [13].

De forma correspondente à evolução do *hardware*, novos modelos de programação capazes de aproveitar o poder desta nova tecnologia têm sido elaborados, destacando-se CUDA (*Compute Unified Device Architecture*) [10] e OpenCL (*Open Computing Language*) [14]. Em ambos, devido a uma ampla disponibilidade de APIs, a implementação de aplicações paralelas eficientes é facilitada, embora as GPUs ainda sejam mais difíceis de serem programadas que as CPUs. Diferentemente da paralelização em CPUs, a organização e o número de $threads$ são gerenciadas manualmente pelo programador [12]. Vale destacar que CUDA foi a primeira arquitetura e interface de programação a permitir que as GPUs pudessem ser usadas para aplicações não gráficas [10] [11].

Este artigo propõe uma solução para a paralelização do algoritmo Floyd-Warshall, implementada utilizando a arquitetura CUDA. Por meio da utilização de um número ideal de blocos de $threads$, a implementação proposta minimiza o uso de comunicações entre a CPU e a GPU, obtendo um desempenho escalonável em relação ao tamanho do problema solucionado, independentemente da arquitetura da GPU utilizada. Este desempenho rivaliza-se e, em certos cenários de problemas, sobrepõe-se aos melhores resultados atualmente disponíveis na literatura dedicada ao tema. Por exemplo, na placa gráfica Tesla C2075, a implementação foi capaz de identificar os caminhos mínimos para todos os pares de vértices de grafos (aleatoriamente gerados) contendo 8192 vértices com 600|V| arestas em menos de 15 segundos, o que resulta num $speedup$ de 149x quando comparado com o algoritmo Floyd-Warshall sequencial. Comparado com a solução proposta por Harish *et al.* [4] e Katz e Kider [5], a média de $speedup$ é de $\approx$ 4x e $\approx$ 1,8x, respectivamente.

O restante deste artigo está organizado da seguinte maneira: na seção II, são descritos os conceitos básicos para o cálculo dos caminhos mínimos num grafo conexo, bem como o algoritmo Floyd-Warshall. A seção III descreve a arquitetura CUDA, em nível de *hardware* e *software*. A seção IV destina-se à apresentação de alguns trabalhos que já abordaram o mesmo tema. Os detalhes da implementação proposta são descritos na seção V. A seção VI lista as características dos recursos computacionais e os métodos usados durante o processo de experimentação. A seção VII descreve os resultados obtidos. Por fim, a seção VIII mostra as conclusões do trabalho.

II. Algoritmo Floyd-Warshall

Seja um grafo conexo e orientado G = (V, E), onde V é o conjunto de vértices e E $\subseteq$ V$\times$V é o conjunto de arestas. O caminho mínimo de um grafo G é o caminho mais curto entre todos os pares de vértices. Esse problema pode ser solucionado, por exemplo, utilizando os algoritmos implementados por Harish *et al.* [4], Hougardy [15], Carvalho [16] e Goldberg *et al.* [17]. O algoritmo Floyd-Warshall utiliza uma técnica conhecida como programação dinâmica [18], [19]. Ele executa em tempo O(V^3), possui complexidade de espaço O(V^2) e pode ser executado num grafo conexo e direcionado G = (V, E), com pesos positivos em suas arestas [20] [21].

O Algoritmo 1 mostra a implementação Floyd-Warshall sequencial. Ele recebe como entrada uma matriz de distâncias, sobre a qual o algoritmo opera para encontrar o menor caminho entre todos os pares de vértices do grafo por ela representado. Tal algoritmo baseia-se numa estrutura de repetição com três laços aninhados. Os dois laços mais internos podem ter a ordem alterada sem afetar o resultado da computação, sempre mantendo o laço mais externo inalterado.

Algoritmo 1 Floyd-Warshall Sequencial.

Entrada: Grafo conexo ponderado G, representado pela matriz de distâncias A_{nxn}.

Saída: Grafo conexo ponderado G com caminho mínimo entre todos os pares de vértices, representado pela matriz de distância A_{nxn}.

```
1: n ←| A_nxn |
2: para k = 0 até n − 1 faça
3:    para i = 0 até n − 1 faça
4:       para j = 0 até n − 1 faça
5:          se a_ik + a_kj<a_ij então
6:             a_ij ← a_ik + a_kj
7:          fim se
8:       fim para
9:    fim para
10: fim para
```

III. Arquitetura CUDA

CUDA pode ser definida em duas partes: *hardware* e *software*. Quanto ao *hardware*, a arquitetura básica da GPU consiste num conjunto de multiprocessadores (SMs), cada um contendo vários processadores (SPs). Todos os SPs dentro de um SM executam as mesmas instruções sobre dados diferentes, conforme o modelo SIMD (*Single Instruction, Multiple Data*). A quantidade de SPs em cada SM varia de acordo com a arquitetura da placa gráfica. A arquitetura disponibiliza uma hierarquia de memória na GPU, que pode ser acessada por todos os núcleos de processamento (SPs), embora os tempos de acesso sejam diferentes em cada nível.

Quanto à parte de *software*, CUDA é construída com base em um conjunto de bibliotecas implementadas na linguagem C, desenvolvidas pela $NVIDIA^{®}$. Elas permitem explorar integralmente os recursos da GPU. O modelo de programação oferecido por CUDA envolve fases de computação executadas

tanto na CPU quanto na GPU. Na GPU, a linguagem utiliza uma série de funções ($kernels$) que são executadas pelas $threads$ nos núcleos de processamento da GPU. Cada fase é definida por uma grade ($grid$), que consiste em todas as $threads$ executando a mesma função. Cada $grid$ consiste em um número de blocos de $threads$, de modo que todas as $threads$ num bloco executam em um único SM, e as $threads$ de cada bloco têm acesso comum à memória compartilhada [10] [22].

IV. Trabalhos Anteriores

A solução sequencial proposta por Venkataraman *et al.* [3] faz com que o algoritmo Floyd-Warshall utilize a memória *cache* da CPU de forma eficiente. O método particiona a matriz de distâncias em ladrilhos. A figura 1 ilustra uma matriz dividida em 4×4 ladrilhos. Dentro de cada ladrilho, múltiplas iterações são executadas. Os ladrilhos são executados em uma certa ordem, onde o ladrilho primário, situado ao eixo diagonal da iteração correspondente, define a ordem da execução dos ladrilhos. Isso facilita, de certa forma, a localidade na memória *cache*, diminuindo valores constantes do tempo computacional. Entretanto, não altera a complexidade do algoritmo Floyd-Warshall. O ganho de $speedup$ situa-se entre 1,6x e 2x [3]. A figura 2 representa, de forma geral, a ordem de execução dos ladrilhos de uma matriz de distâncias em cada iteração. O primeiro ladrilho calculado é o de cor vermelha e, em seguida, os ladrilhos cinzas e, por último, os ladrilhos brancos.

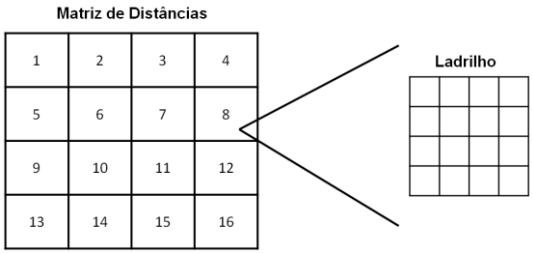

Figura 1. Divisão de uma matriz de distâncias em ladrilhos.

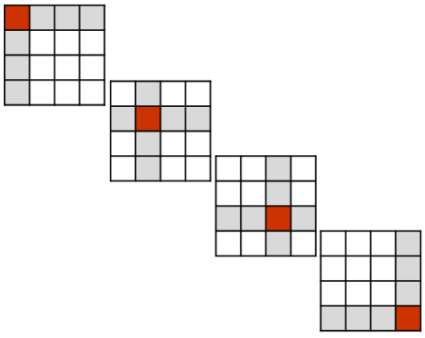

Figura 2. Solução sequencial proposta por Venkataraman *et al.* [3].

A abordagem paralela utilizada por Harish *et al.* [4] divide o algoritmo Floyd-Warshall em duas partes. Uma parte é executada na GPU, onde uma única função $kernel$ é executada por O(V) $threads$. A função $kernel$ é chamada O(V) vezes sequencialmente pela CPU, logo, várias comunicações são realizadas. Contudo, uma única transferência de dados significativa é feita, pois os dados são mantidos na memória da GPU até o final do processamento. De acordo com Xiao e Feng [23], uma grande parte do tempo de execução de uma aplicação CUDA é gasto no lançamento do $kernel$ e na sincronização. Isso ocorre quando há um uso intensivo de comunicação entre CPU e GPU. Entretanto, apesar desta limitação, essa solução oferece um ganho de $speedup$ sobre a implementação sequencial [4].

A segunda parte do algoritmo é executada na CPU, onde uma função $kernel$ é chamada várias vezes, como mostrado no Algoritmo 2.

Algoritmo 2 Implementação do algoritmo Floyd-Warshall proposta por Harish *et al.* [4].

Entrada: Grafo G representado pela matriz de distâncias A_{nxn}.

Saída: Caminho mínimo do grafo G representado pela matriz de distâncias A_{nxn}.

1: $bloco\_size \leftarrow$ número máximo de threads por bloco
2: Aloca memória para a matriz A_{nxn} na GPU
3: Copia a matriz A_{nxn} para GPU
4: $tam\_bloco \leftarrow bloco\_size \times bloco\_size$
5: $tam\_grade \leftarrow \lceil(\frac{n}{bloco\_size})\rceil \times \lceil(\frac{n}{bloco\_size})\rceil$
6: **para** $estagio = 0$ até $n - 1$ **faça**
7: Invoca o kernel $<tam\_grade,\ tam\_bloco\ >(A_{nxn},\ estagio)$ na GPU
8: **fim para**
9: Copia a matriz A_{nxn} da GPU para CPU
10: Libera o espaço na memória da GPU ocupado pela matriz A_{nxn}

O algoritmo paralelo proposto por Katz e Kider [5] é baseado no algoritmo sequencial implementado por Venkataraman *et al.* [3]. O algoritmo particiona a matriz de distâncias em blocos de tamanhos iguais. Cada bloco fica com 16×16 vértices de tamanho. Dessa forma, o bloco pode ser facilmente mapeado para os blocos de $threads$. O algoritmo procede em estágios, cada um constituído por três fases. As fases são executadas por $kernels$. Em todos os estágios, um bloco primário é definido. Esse bloco primário fica situado ao longo do eixo diagonal da matriz de distâncias, que se inicializa na localização (0, 0), e o último bloco fica em (|V|-16, |V|-16), onde V é o conjunto de vértices. Assim, o algoritmo consiste em $\frac{|V|}{16}$ iterações.

A Fase 1 é a mais simples. Somente o bloco primário atual executa o algoritmo, e um único SM da GPU permanece ativo. O bloco é obtido somente uma vez na memória global e levado para a memória compartilhada, de modo a acelerar a fase tanto quanto possível.

A Fase 2 manipula os blocos cujos valores são dependentes do bloco primário. Esses blocos estão localizados na mesma linha (eixo y) e coluna (eixo x) do bloco primário

computado. Tais blocos são chamados blocos simplesmente-dependentes [5]. Para a realização dessa fase, o bloco primário da matriz de distâncias é carregado para a memória compartilhada. Por exemplo, para uma matriz com n blocos por eixo, $2\times(n$ - $1)$ blocos da matriz são computados e, consequentemente, o mesmo número de blocos de $threads$ são organizados no $grid$.

A Fase 3 computa os blocos restantes da matriz, conhecidos por blocos duplamente-dependentes, por dependerem da computação da linha e coluna do bloco primário. A Figura 3 fornece uma visualização de alto nível da execução das três fases. Vale ressaltar que cada fase é executada sequencialmente, resultando numa comunicação constante entre a CPU e a GPU.

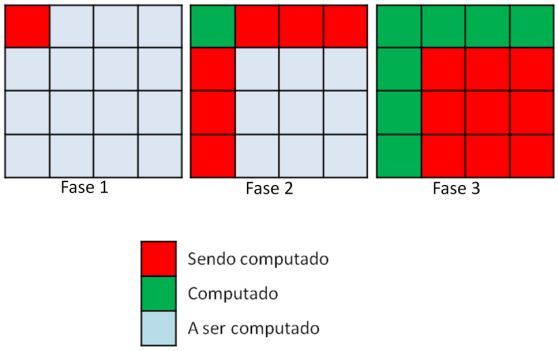

Figura 3. Visão geral do algoritmo proposto por Katz e Kider [5].

V. Proposta do Trabalho

O cálculo do menor caminho entre todos os vértices de um grafo necessita passar por uma série de iterações, sendo que uma iteração depende do resultado da iteração anterior para que possa ser efetivada. Devido a essa dependência de dados, algoritmos como Floyd-Warshall e outros com a mesma característica não podem ser executados totalmente em paralelo [11] [12].

Com o objetivo de obter um melhor desempenho computacional em relação ao estado da arte das implementações paralelas do algoritmo Floyd-Warshall, propõe-se neste trabalho uma abordagem cuja solução faz uso de dois aspectos principais da GPU: localidade de dados e ocupação intensiva dos SMs.

A localidade de dados relaciona-se com a hierarquia de memória disponível na arquitetura CUDA, onde o acesso à memória global possui uma latência muito alta em relação à memória compartilhada. Para uma melhor localidade dos dados, a implementação proposta baseia-se naquela sequencial concebida por Venkataraman *et al.* [3]. A matriz de distâncias é dividida em blocos de tamanhos iguais. Esses blocos são levados inteiramente para a memória compartilhada e, somente então, é realizado o cálculo do caminho mínimo. O algoritmo segue em estágios, semelhante à implementação de Katz e Kider [5]. Em cada estágio, que é subdividido em duas fases, um bloco primário é definido.

Com relação à ocupação intensiva dos SMs, evita-se que qualquer SM fique ocioso devido à dependência de dados no momento da computação. Ao processar a dependência de dados, ocupa-se um SM com o processamento e os outros ficam ociosos. Para atingir a ocupação intensiva, a dependência é replicada para todos SMs. Para isso, um $kernel$ é lançado com um número de blocos de $threads$ múltiplo da quantidade de blocos (1 a 8 vezes) executados concorrentemente em cada SM. Um bloco de $threads$ é executado em único SM, que processa um ou mais blocos de $threads$ simultaneamente. Desta forma, evita-se que blocos de $threads$ fiquem na fila de escalonamento. Consequentemente, cada bloco de $threads$ poderá operar em mais de uma parte da matriz de entrada, quando esta for maior que o $grid$ de blocos de $threads$ do $kernel$. Por exemplo, seja uma GPU com quatro SMs, onde cada um suporta somente um bloco de $threads$ em uma unidade de tempo. Logo, o $kernel$ será lançado com quatro blocos de $threads$, conforme mostra a Figura 4.

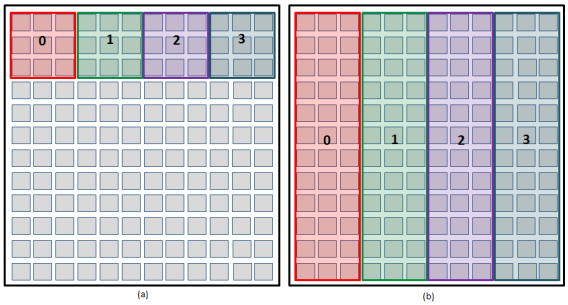

Figura 4. Exemplo de ocupação intensiva dos SM's: (a) Número de blocos de $thread$; (b) Divisão das partes da matriz por bloco de $threads$.

A Fase 1 do algoritmo realiza o cálculo do caminho mínimo no bloco primário e nos seus respectivos eixos x e y, ou seja, nos blocos que pertencem à linha e à coluna do bloco primário. Para tal, aplica-se uma ocupação intensiva dos SMs e uma divisão em etapas dessa fase, onde todas as etapas são executadas em um único $kernel$.

A primeira etapa consiste no armazenamento do bloco primário na memória compartilhada e, em seguida, na execução do algoritmo neste bloco primário, em cada bloco de $threads$. Como o tempo de vida das $threads$ é alterado para persistir até o final da execução do $kernel$, cada bloco de $threads$ busca e armazena uma única vez o bloco primário atual, onde encontra-se a dependência de dados. Portanto, o bloco primário é replicado conforme a quantidade de blocos de $threads$ lançados no $kernel$. A Figura 5 exemplifica essa primeira etapa. Como pode ser observado, há 3 SMs na GPU, e cada um executa um único bloco de $threads$.

Na segunda etapa, os blocos de $threads$ executam o algoritmo na linha e coluna do atual bloco primário. Entretanto, um bloco de $threads$ executa em mais de um bloco da matriz de distâncias, sendo definida a sequência de tarefas antes da execução. Ressalta-se que um SM diferente a cada estágio é responsável por armazenar o atual bloco primário. A Figura 6 ilusta essa segunda etapa, onde em (a) a linha está sendo executada; e em (b) a coluna está sendo executada.

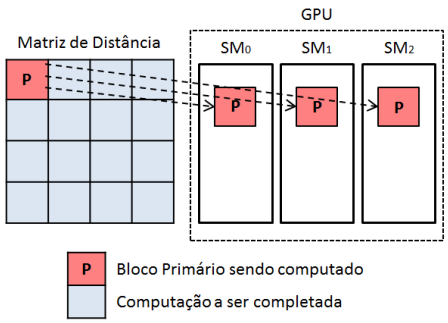

Figura 5. Primeira etapa da Fase 1 da implementação.

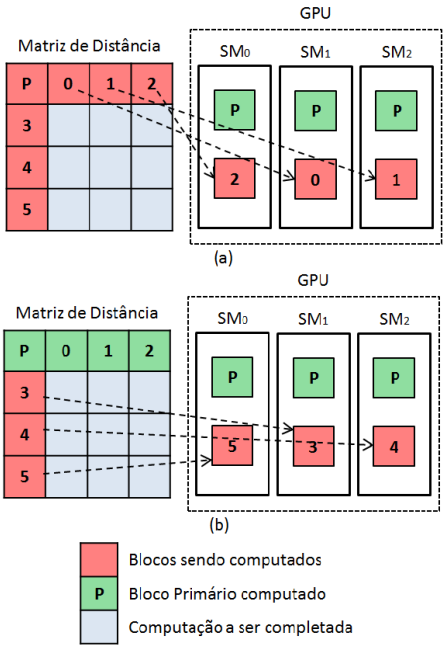

Figura 6. Segunda etapa da Fase 1 da implementação.

Na Fase 2 do algoritmo, os blocos restantes da matriz de distâncias são calculados de forma convencional proporcionado pelo modelo de programação CUDA. O tempo de vida das $threads$ não é persistente até o fim do $kernel$. Caso a matriz seja maior que a capacidade de $threads$ em execução, haverá $threads$ esperando para serem escalonadas. Esta fase é realizada somente em um único $kernel$. Portanto, os blocos da linha e da coluna correspondentes ao bloco de $threads$ a ser computado são armazenados na memória compartilhada.

O Algoritmo 3 mostra a estratégia da implementação no código da CPU, onde a variável $qtdPartes$ indica a proporção entre o tamanho da matriz de distâncias (tamanho do grafo) com o número de SMs da GPU e tamanho do bloco de $threads$. Na GPU, ela indicará quantas partes da matriz de distâncias que cada bloco de $threads$ irá executar. A quantidade exata de blocos de $threads$ para cada $kernel$ é ditado pelas variáveis $dimGrid\_fase1$ e $dimGrid\_fase2$.

Algoritmo 3 Código da CPU - Implementação proposta do algoritmo Floyd-Warshall

Entrada: Grafo G representado pela matriz A_{nxn}, $bloco\_size$, $numeroSM$.
Saída: Caminho Mínimo do grafo G representado pela matriz A_{nxn}.

1: $n \leftarrow \mid A_{nxn} \mid$;
2: $numeroBlocos\_SM \leftarrow numeroSM$;
3: $numeroBlocos \leftarrow \lceil (\frac{n}{bloco\_size}) \rceil$;
4: $qtdPartes \leftarrow (\frac{n}{(bloco\_size \times numeroBlocos\_SM)})$;
5: Aloca memória para a matriz A_{nxn} na GPU;
6: Copia a matriz A_{nxn} para GPU;
7: $dimBloco \leftarrow bloco\_size \times bloco\_size$;
8: $dimGrid\_fase1 \leftarrow numeroBlocos\_SM$;
9: $dimGrid\_fase2 \leftarrow numeroBlocos \times numeroBlocos$;
10: **while** $estagio = 0$ até n **faça**
11: Invoca o kernel_fase_1 $<dimGrid\_fase1, dimBloco>(A_{nxn}, n, numeroBlocos\_SM, qtdPartes)$;
12: Invoca o kernel_fase_2 $<dimGrid\_fase2, dimBloco>(A_{nxn}, n, estagio)$;
13: $estagio \leftarrow estagio + bloco\_size$
14: **fim while**
15: Copia a matriz A_{nxn} da GPU para CPU;
16: Libera o espaço na memória da GPU ocupado pela matriz A_{nxn};

VI. Materiais e Métodos

Neste trabalho, a implementação proposta foi comparada com os trabalhos de Harish *et al.* [4], Katz e Kider [5] e o algoritmo Floyd-Warshall sequencial.

Os algoritmos descritos neste trabalho foram testados em uma máquina com 6 GB de memória RAM, sistema operacional Linux (Ubuntu 12.04 x86), processador Intel Core i5 2300 (2.80GHz) e uma placa gráfica $NVIDIA^{®}$ Tesla C2075, com 4 GB de memória e 448 núcleos de processamento oferecendo até 515 Gflops de desempenho em ponto flutuante de precisão dupla.

As propostas de Harish *et al.* [4] e Katz e Kider [5] foram implementadas conforme suas descrições. Na execução do algoritmo de Katz e Kider [5], a dimensão do bloco de $threads$ foi de 32×32, onde obteve-se os melhores tempos comparados com outras dimensões. Os melhores tempos obtidos com o algoritmo de Harish *et al.* [4] foram com a dimensão 16×16. Já na solução proposta, a dimensão do bloco de $threads$ que alcançou melhores resultados foi de 32×32, obtendo 66% de ocupação da GPU.

Todos os grafos foram gerados aleatoriamente utilizando o gerador $random$ da ferramenta GTgraph [24] e, posteriormente, gravados em arquivo com o objetivo de utilizar os mesmos grafos em todos testes. Foram realizados experimentos com grafos de diferentes números de vértices (1024, 2048, 3072, 4096, 5120, 6144, 7168, 8192 e 16384) e com números de arestas iguais a $6|V|$, $60|V|$ e $600|V|$. Como não houve grandes diferenças de tempo entre os grafos de diferentes

arestas, a média aritmética foi realizada. Para cada grafo, o algoritmo foi executado 10 vezes e a média dos tempos computada.

VII. Resultados e Discussão

Analisando os gráficos nas Figuras 7 e 8, observa-se que houve, de fato, ganhos de eficiência computacional utilizando o algoritmo proposto. O ganho obtido foi por meio da estratégia aqui utilizada, que diminui a comunicação entre a CPU e a GPU, fazendo com que as *threads* permaneçam ativas por um tempo maior nos núcleos de processamento. Vale ressaltar que a utilização dessa estratégia altera somente constantes do tempo do algoritmo e não na complexidade do mesmo.

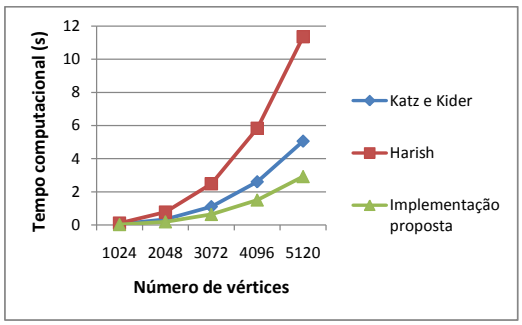

Figura 7. Tempo de execução dos algoritmos para grafos com 1024, 2048, 3072, 4096 e 5120 vértices.

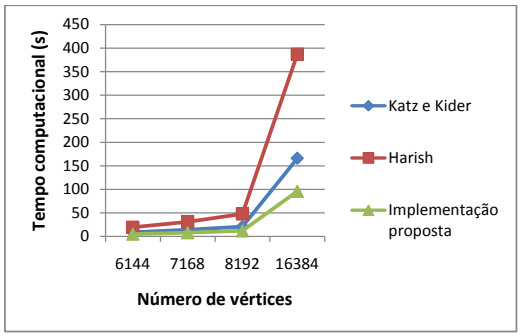

Figura 8. Tempo de execução dos algoritmos para grafos com 6144, 7168, 8192 e 16384 vértices.

Os algoritmos apresentados por Harish *et al.* [4] e Katz e Kider [5] apresentaram um tempo maior de execução para grafos com um número maior que 2048 vértices, devido ao uso intensivo de comunicação entre os dispositivos (CPU e GPU). As Figuras 9 e 10 mostram os ganhos obtidos utilizando grafos com vários números de vértices. Por exemplo, para o cálculo do caminho mínimo em um grafo com 16384 vértices, o algoritmo proposto se mostrou 4x mais rápido que o algoritmo de Harish *et al.* [4] e até 195x mais rápido que a implementação sequencial do algoritmo Floyd-Warshall (algoritmo 1).

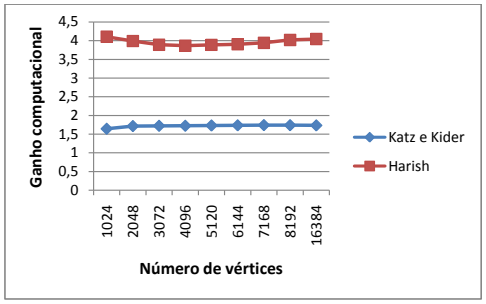

Figura 9. Ganhos de *speedup* em relação às implementações de Harish *et al.* [4] e Katz e Kider [5].

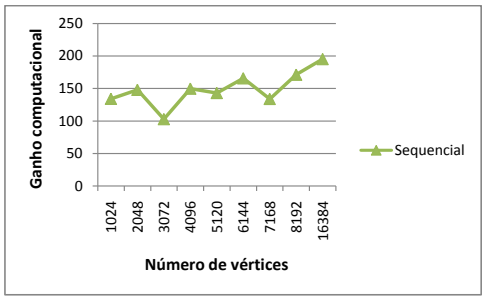

Figura 10. Ganhos de *speedup* em relação à implementação sequencial do algoritmo Floyd-Warshall.

VIII. Conclusão

Apresentou-se neste trabalho uma implementação paralela escalonável para o algoritmo Floyd-Warshall, cujo o objetivo foi a identificação dos caminhos mínimos entre todos os possíveis pares de vértices em grafos conexos, digiridos e com arestas ponderadas. O algoritmo aqui apresentado mostrou um ganho de eficiência computacional (*speedup*) de $\approx$ 149x em relação ao algoritmo sequencial. Quando comparado com as implementações de Harish *et al.* [4] e Katz e Kider [5], a média de *speedup* foi de $\approx$ 4x e $\approx$ 1,8x, respectivamente.

Foi possível concluir que este *speedup* pôde ser obtido por meio da utilização de todos os recursos da tecnologia CUDA, aliado com melhores práticas arquiteturais durante o desenvolvimento. Além disso, concluiu-se que CUDA colabora, de maneira significativa, para a obtenção de versões paralelas eficientes por meio da exploração de características paralelizáveis intrínsecas ao problema e adaptáveis às especificações e modo de funcionamento desta arquitetura.

Trabalhos seguintes nessa linha de pesquisa poderão solucionar o problema do caminho mínimo em grafos com uma maior quantidade de vértices. Adicionalmente, pretende-se utilizar as estratégias aqui descritas no software PETGYN [25].

Agradecimentos

Os autores agradecem à CAPES e à FAPEG pelo apoio a esta pesquisa na forma de bolsas de mestrado e doutorado. Agradecem também à *NVIDIA*® pelo fornecimento da

placa Tesla C2075 e pela escolha da UFG como um Centro de Ensino CUDA (*CUDA Teaching Center*).

REFERÊNCIAS

[1] F. Dehne and K. Yogaratnam, "Exploring the limits of gpus with parallel graph algorithms," *CoRR*, vol. abs/1002.4482, 2010.

[2] A. Aini and A. Salehipour, "Speeding up the floyd–warshall algorithm for the cycled shortest path problem," *Applied Mathematics Letters*, vol. 25, no. 1, pp. 1–5, 2012.

[3] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A blocked all-pairs shortest-paths algorithm," *Journal of Experimental Algorithmics (JEA)*, vol. 8, 2003.

[4] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," *High performance computing–HiPC*, pp. 197–208, 2007.

[5] G. J. Katz and J. T. Kider, "All-pairs shortest-paths for large graphs on the gpu," *Proceedings of the 23rd ACM SIGGRAPHEUROGRAPHICS symposium on Graphics Hardware*, pp. 47–55, 2008.

[6] L. Ridi, J. Torrini, and E. Vicario, "Developing a scheduler with difference-bound matrices and the floyd-warshall algorithm," *Software, IEEE*, vol. 29, no. 1, pp. 76–83, 2012.

[7] K. M. Borgwardt and H.-P. Kriegel, "Shortest-path kernels on graphs," in *Data Mining, Fifth IEEE International Conference on*. IEEE, 2005, pp. 8–pp.

[8] J. Ma, K.-p. Li, and L.-y. Zhang, "A parallel floyd-warshall algorithm based on tbb," in *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*. IEEE, 2010, pp. 429–433.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, 2008.

[10] N. CUDA, *NVIDIA CUDA C Programming Guide*, 4th ed. 2701 San Tomas Expressway Santa Clara, CA 95050: NVIDIA Corporation, 2011.

[11] L. C. M. de Paula, L. B. S. de Souza, L. B. S. de Souza, and W. S. Martins, "Aplicação de processamento paralelo em método iterativo para solução de sistemas lineares," *X Encontro Anual de Computação*, pp. 129–136, 2013.

[12] L. C. M. de Paula, A. da Silva Soares, T. W. Soares, W. S. Martins, A. R. G. Filho, and C. J. Coelho, "Partial parallelization of the successive projections algorithm using compute unified device architecture," *19th International Conference on Parallel and Distributed Processing Techniques and Applications*, 2013.

[13] D. Luebke and G. Humphreys, "How gpus work," *Computer*, vol. 40, pp. 96–100, February 2007.

[14] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, J. Son, and S. Miki, *The OpenCL Programming Book*. Fixstars, 2010.

[15] S. Hougardy, "The floyd-warshall algorithm on graphs with negative cycles," *Information Processing Letters*, vol. 110, no. 8, pp. 279–281, 2010.

[16] B. M. P. S. Carvalho, "Algoritmo de dijkstra," Master's thesis, Departamento de Engenharia Informática Universidade de Coimbra, Portugal, 2003.

[17] A. V. Goldberg and T. Radzik, "A heuristic improvement of the bellman-ford algorithm," *Applied Mathematics Letters*, vol. 6, no. 3, pp. 3–6, 1993.

[18] R. Bellman, *Dynamic Programming*, ser. Dover Books on Computer Science Series. Dover Publications, Incorporated, 2003.

[19] S. R. Eddy, "What is dynamic programming?" *Nature Biotechnology*, vol. 22, no. 7, pp. 909–910, Jul. 2004. [Online]. Available: http://dx.doi.org/10.1038/nbt0704-909

[20] C. Papadimitriou and M. Sideri, "On the floyd-warshall algorithm for logic programs," *The journal of logic programming*, vol. 41, no. 1, pp. 129–137, 1999.

[21] S. Pettie and V. Ramachandran, "Computing shortest paths with comparisons and additions," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2002, pp. 267–276.

[22] N. $CUDA^{TM}$, *NVIDIA CUDA C Programming Best Practices Guide*. 2701 San Tomas Expressway Santa Clara, CA 95050: NVIDIA Corporation, 2009.

[23] S. Xiao and W.-c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[24] D. A. Bader and K. Madduri, "Gtgraph: A synthetic graph generator suite," *Atlanta, GA, February*, 2006.

[25] W. Jradi, H. do Nascimento, H. Longo, and B. Hall, "Simulation and analysis of urban traffic the architecture of a web-based interactive decision support system," in *Intelligent Transportation Systems, 2009. ITSC '09. 12th International IEEE Conference on*, 2009, pp. 1–6.