Identificação de Vértices Centrais em Grafos com a Aplicação do Algoritmo de Floyd Warshall em GPU

Bruno de Jesus Viana[1,✉], **João Victor Rezende**[3,✉], **and Pedro Leon Paranayba Clerot**[2,✉]

[1]Centro Universitário IESB - Ciência da Computação
[2]Centro Universitário IESB - Ciência da Computação
[3]Centro Universitário IESB - Ciência da Computação

O algoritmo de Floyd-Warshall ou também algoritmo de Floyd, Roy-Warshall ou Roy-Floyd, é um algoritmo para busca do menor caminho aplicado a grafos, podendo apresentar diferentes desempenhos a depender do tamanho, caso e implementação. Com a melhora no desempenho e aumento da capacidade computacional, as Graphics Procesing Units (GPU) se apresentam efetivas para resolução de grandes cálculos e múltiplas tarefas, principalmente as que por sua natureza possam ser paralelizadas. Este trabalho tem como objetivo principal demonstrar a eficiência de uma aplicação em GPU do algoritmo Floyd-Warshall, de forma paralelizada, e seus resultados. O algoritmo será implementado para o cálculo das menores distâncias entre todos os pares de nós de um grafo, com a posterior identificação do vértice central com base na média das distâncias para os demais vértices.

Grafos | Floyd-Warshall | GPU | Paralelismo

Introdução

A crescente geração de dados e consequente maior disponibilidade de informações tem apresentado um desafio constante: processar essa quantidade cada vez maior de dados em intervalos de tempo cada vez mais curtos. Atualmente, a capacidade de processamento das Central Processing Units (CPU) que equipam computadores de alta performance é satisfatória para a resolução de determinados problemas. Contudo, a medida em que se escala a quantidade de dados, bem como em que se aumenta a complexidade dos cálculos e análises a serem realizadas, maior desempenho e respostas mais rápidas são demandadas. Esse obstáculo tem sido frequentemente contornado com a utilização de GPU's, que oferecem processamento mais veloz, especialmente de programas desenvolvidos com base em técnicas de paralelismo. No presente trabalho será apresentada a implementação do algoritmo de Floyd Warshall em uma versão paralelizada e compatível com o processamento em GPU. O algoritmo mencionado é aplicado na resolução de problemas de custo/caminho mínimo, através de estruturas de dados representadas em grafos. Por essa razão, em grafos com maior quantidade de nós e arestas, o tempo de resposta é afetado negativamente. A aplicação a ser desenvolvida dará ênfase à identificação do vértice central de um grafo a partir do cálculo da menor distância existente entre todos os nós componentes da estrutura. O produto desse processamento pode ser aplicado a diferentes problemas práticos, tais como a identificação das melhores localizações para instalação de centros de distrubuição de mercadorias, postos de atendimento de saúde, postos de guarda policial, dentre outras finalidades igualmente relevantes.

Objetivos

Objetivo Geral.

Desenvolver uma versão paralela do algoritmo de Floyd-Warshall para implementação em GPU's, que seja eficiente na resolução de problemas complexos, escaláveis e que demandam redução do tempo de processamento e resposta, especificamente na identificação do vértice central de um grafo analisado.

Objetivos Específicos.

Para a consecução do objetivo geral proposto para o trabalho será necessário o atingimento dos seguintes objetivos específicos:

- Desenvolver uma estrutura de dados que possibilite a aplicação do algoritimo em estudo;
- Implementar o algoritmo, em sua versão sequencial em linguagem C e verificar o seu desempenho na resolução do problema proposto;
- Implementar uma versão do algoritmo utilizando programação paralela na Aquitetura OpenCL, que seja compatível com a implementação em GPU, e verificar o seu desempenho na resolução do problema proposto;
- Comparar os desempenhos auferidos, analisando as vantagens e desvantages, se houverem.

Referencial Teórico

Grafos.

Os grafos, de forma genérica, podem ser definidos como estruturas constituídas por relações entre elementos de determinado conjunto. Esses elementos são representados por nós (ou vértices) que possuem conexões entre si por meio de arestas (ou arcos). Exemplificando, se em um conjunto de

números C = 1, 2, 3, 4, 5, 6, os seus elementos forem relacionados em pares ordenados, tais como (1,2), (1,3), (3,6), (4,5), esses relacionamentos podem ser representados e visualizados na forma de um grafo.

Quando um grafo é constituído por pares ordenados, como no exemplo acima, ele é denominado grafo dígrafo. Outros termos utilizados são orientado, digirido e direcionado. Nos dígrafos, não há simetria entre os dois vértices conectados por uma aresta. Isso significa dizer que, dados dois vértices "a" e "b", o arco (a,b) e (b,a), se existentes, não são iguais. Por outro lado, quando essa relação de simetria ocorre entre os vértices componentes de um arco, o grafo correpondente é chamado de não dirigido ou não direcionado. Nesse caso, (a,b) e (b,a) são iguais, ou seja, a aresta não possui direção.

Um grafo, a depender do seu tamanho, é constituído por um grande número de vértices e arestas. Quando ocorre de todos os vértices possuírem arestas que os ligam entre si, ou seja, não há nenhum vértice que não possa ser conectado a todos os demais, essa estrutura é classificada como um grafo completo. Conforme o número de arestas existentes, um grafo será categorizado como denso ou esparso. No primeiro caso, o número de arcos é elevado, enquanto quw, no segundo, a quantidade de conexões é menor.

Alguns aspectos (características) do grafo são relevantes de serem analisados para sua melhor compreensão. O primeiro deles é a sua ordem, que é a quantidade de vértices que o compõe. Quanto maior a ordem, maior a complexidade da estrutura. Além disso o grau do grafo é outra informação importante. Ele representa o número de conexões que um vértice possui com outros vértices. O grau de entrada de um nó é o número de arestas que o tem como origem. O grau de saída, por sua vez, é a quantidade de arcos que o possui como destino. O grau do grafo como um todo será definido pelo maior grau existente entre seus vértices.

A medida em que são adicionados novos vértices e arestas ao grafo, observa-se que vão se formando sequências de nós que conectam indiretamente um nó "x" e um nó "y", mesmo que não exista um arco entre eles. A essas são sequências é dado o nome de cadeia. Quando há uma cadeia que possibilita sair de um vértice, percorrendo outros pontos do grafo e retornar para o mesmo vértice, sem que haja repetição de arestas pelo caminho, identifica-se um cíclo. Grafos que possuem ao menos uma ocorrência desse tipo são conhecidos como cíclicos. No caso contrario, são acíclicos, sendo conhecidos por DAG (sigla proveniente da expressão em inglês Direct Acyclic Graphs).

As relações entre os vértices, ou seja, as arestas, podem ter pesos ou valores a elas atribuídos. Quando isso ocorre, diz-se que o grafo é ponderado, também conhecido como rede. Os pesos podem se referir a diferentes tipos de informação, a depender do objeto representado pelo grafo. Eles serão particularmente importantes, para análise de custo de caminho que representa o escopo de estudo do presente trabalho. Destaca-se que tais valores são representados na forma numérica, podendo ser positivos ou negativos.

A representação dos grafos pode ocorrer de diferentes formas, sendo as mais utilizadas a lista de arestas, a matriz de adjacências e a lista de adjacências. No primeiro caso, como o próprio nome já diz, é descrita uma lista contendo como elementos as arestas existentes. Essa é a forma mais simples de se representar um grafo.

Na matriz de adjacências, a representação ocorre por meio de um vetor bidimensional que possui entre suas linhas e colunas, a informação de existência ou não de adjacência entre um vértice e outro. A relação de adjacência ocorre, quando um vértice "x" e um vértice "y" são conectados por uma aresta (x,y). Nesse caso, diz-se que "y" é adjacente a "x", em razão do arco existente entre os dois. Na matriz, a representação se dá pelo número "0", quando não há relação de adjacência, e por "1", quando há.

A lista de adjacências, por sua vez, corresponde a uma combinação da lista de vértices componentes do grafo com a relação de nós adjacentes a cada um deles. Na prática, para cada elemento da lista de vértices, ou seja, os nós, háverá uma lista de adjacentes, conforme as relações identificadas pelas arestas existentes. Esta é a forma de representação mais adequada para se trabalhar com grafos grandes e esparsos.

Problema do Caminho Mínimo.

O problema do caminho mínimo (PCM) consiste na identificação da cadeia (sequência) de de nós que conectam indiretamente e com o menor custo um vértice origem e um vértice objetivo em um grafo qualquer. Este é um problema cuja aplicação é visualizada em diversas situações práticas a exemplo de cálculo de rotas de trânsito, expansão de redes elétricas, hidráulicas e de cabeamento de fibra ótica, robótica etc. Conforme menciona Moreira Filho (2017), "O PCM pode ser modelado em grafo e nada mais é do que determinar a melhor rota entre dois pontos."

A rota ótima, assim considerada aquela que apresenta a menor distância entre dois pontos, pode ser identificada sob diferentes óticas a depender do problema que se deseja solucionar. Além do fator distância (localização), pode ser considerada a avaliação de menores custos (em termos financeiros), ou mesmo de tempo, quando o objetivo é identificar rotas mais rápidas, dentre outros possíveis fatores. Dessa forma, pode-se perceber que o escopo de aplicação do problema é grande.

Dreyfus (1969) apud Moreira Filho (2017) apresenta 5 tipos distintos de PCM:

> Determinar o menor caminho entre dois nós específicos em uma rede; Determinar o menor caminho entre todos os pares de vértices em uma rede; Determinar o segundo menor caminho, terceiro, e etc; Determinar o percurso mais rápido através de uma rede com tempos de viagem dependendo da hora de partida; Determinar o menor caminho entre dois pontos específicos onde é necessário a passagem por um nó intermediário.

A complexidade para resolução do cálculo do caminho mínimo em grafos não está concentrada nas operações a serem realizadas, mas sim na escolha do algoritmo mais adequado e eficiente. Vale destacar que isso vai depender do problema prático que se deseja revolver, tendo em vista que há diferentes possibilidades de combinações de sequências de vértices. Neste trabalho, será abordado com maiores detalhes o algoritmo de Floyd-Warshall, que calcula a melhor rota existente entre os todos os pares de vértices (all-pairs). Destaca-se que, além do cálculo das menores distâncias, será implementada a identificação do vértice central de um dado grafo.

Em resumo, um grafo pode apresentar diferentes formas de se chegar a um determinado vértice a partir de um outro qualquer. Nem sempre, a melhor rota é aquela que liga diretamente um até outro, podendo haver sequências de vértice ques apresentam custo total inferior. Exemplificando, em um grafo G = (V, E), quando se deseja identificar o melhor caminho que conecta o par de vértices (i,j), verifica-se todas as possíveis adjacências existentes entre a origem "i" e vértices intermediários que se ligam a "j", selecionando a melhor sequência de nós que minimiza o correspondente custo total (soma dos pesos das arestas). Vale destacar que, em determinados grafos, podem haver pares de vértices cuja conexão é impossível. Nesses casos, diz-se que o custo do caminho é infinito.

Algoritmos Aplicáveis ao Problema do Caminho Mínimo.

Como mencionado no tópico anterior, o algoritmo que será tratado no presente trabalho não é o único que possibilita a resolução do problema do caminho mínimo. Dentre eles, pode-se mencionar o algoritmo de Johnson, o algoritmo de Bellman-Ford e o algoritmo de Dijkstra. Com o objetivo de contextualizar, serão apresentadas breves informações acerca deste último. Ele e o de Bellman-Ford calculam a melhor rota a partir de um vértice origem para todos os demais componentes do grafo (single-destination), apresentando significativa eficiência. O de Johnson, assim como o de Floyd-Warshall, se aplica à identificação do menor caminho entre todos os pares de vértices. Pode citar ainda o algoritmo A* (A Star) que é utilizado quando o objetivo é identificar o custo mínimo entre dois nós específicos.

O algoritmo de Dijkstra foi, originalmente, criado por Edsger Dijkstra em 1959. Para que o cumprimento de sua finalidade mencionada no parágrafo anterior, o algoritmo vai verificar todas as possíveis sequências de nós existentes, indicando aquela que apresentar o menor custo entre a origem e todos os destinos possíveis. Segundo Gaioso (2014), o algoritmo também é aplicável ao cálculo das distâncias entre todos os pares de vértices, mas, para isso, precisa ser aplicado a cada um deles individualmente, o que aumenta substancialmente o tempo e recursos computacionais necessário para o processamento.

O algoritmo de Dijkstra apresenta complexidade de tempo de execução equivalente O(|E|+|V|log|V|), mas que pode variar de acordo com a estrutura de dados utilizada. Dijkstra apenas admite pesos positivos, ou seja, não há possibilita de se trabalhar com o algoritmo em matrizes que possuam arestas com custo inferior a 0. Ele atua utilizando o conceito de relaxamento de arestas, testando seguidamente se é possível a redução da distância do vértice origem para o destino em análise.

Algoritmo de Floyd-Warshall.

O foi originalmente publicado por Bernard Roy (1959) e Setephen Warshall (1962) em um formato um pouco diferente ao amplamente conhecido nos dias atuais. Inicialmente, o algoritmo apenas possibilitava o cálculo do fecho transitivo de um grafo, por meio da técnica de multiplicação de matrizes. Posteriormente, ele foi adaptado por Robert Floyd (1962), no atual formato, que habilita o cálculo das menores distâncias existentes entre todos os pares de vértices de um grafo. De uma forma geral, o algoritmo de Floyd-Warshall é executado sob uma matriz de pesos e adjacências, comparando as distâncias correspondentes às arestas componentes do grafo, a fim de retornar como produto uma matriz de distâncias.

O algoritmo admite custo negativo nas arestas, contudo não permite a existência de ciclos de peso negativo, ou seja, cadeias de arcos que tenham como custo total valores inferiores a zero. Vale destacar que este é um algoritmo que utiliza programação dinâmica para o cálculo das menores distâncias, o que siginifica dizer que ele utiliza busca pela solução ótima a partir daquela previamente identificada como tal, até o fim do processamento. Isso evita que a computação retorne a passos anteriores repetidas vezes, retardando a conclusão dos cálculos.

Para encontrar a menor distância entre todos os pares de vértices, o algoritmo de Floyd-Warshall utiliza o conceito de vértice intermediário. Moreira Filho (2017) menciona que "o vértice intermediário em um caminho S = v1, v2, ..., vf é qualquer vértice que não seja v1 (vértice inicial) ou vf (vértice final), mas que conectam esses dois nós."

Para melhor entendimento, considerando-se um grafo G = (V, E), o melhor caminho existente entre um vértice "i" e um vértice "j", será identificado a partir da verificação de todos os possíveis nós "k" que são intermediários. Caso seja constatado que a soma da distância existente entre a origem (i) e um intermediário qualquer (k) e deste ao destino (j), a menor distância entre "i" e "j" passa a ser o resultado desse somatório. Essa avaliação é realizada repetidas vezes até que todas as possibilidades existentes tenham sido consideradas.

Um detalhe importante acerca do algoritmo é que ele resulta na menor distância existente, mas não guarda a sequência de vértices correpondente ao caminho, informando apenas o último predecessor. Ele possui complexidade de tempo de execução correspondente a O(|V|3), tendo em vista que é necessário o processamento de três laços de repetição. Maiores detalhes sobre o algoritmo serão discutidos ao tratar-se das implementações sequencial e paralela propostas

no presente trabalho. Abaixo é descrito um exemplo de pseudocódigo proposto por Gaioso et. al (2013):

Entrada: Grafo conexo ponderado G, representado pela Matriz de Adjacências Anxn.
Saída: Grafo ponderado G com caminho mínimo entre todos os pares de vértices, representado pela Matriz de Distâncias Anxn.

```
1: n <- |Anxn|
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 Aik + Akj < Aij
6:                 Aij <- Aik + Akj
7:             fim se
8:         fim para
9:     fim para
10: fim para
```

Medidas de Centralidade.

A centralidade do grafo é definida pelo vértice que representa a melhor origem (que detém as melhores rotas) para todos os seus demais vértices. Difícilmente será possível identificar o nó que reúna o maior número conexões e todas as rotas ótimas. Contudo, é possível definir aquele que ponderadamente representa o centro da estrutura. Para isso, as principais medidas de centralidade existentes são as seguintes:

- **Centralidade pelo Grau**: Nesse caso, o vértice central é obtido a partir do maior número de arestas existentes para e a partir dele, ou seja, aquele nó que se conecta com a maior quantidade de nós grafos. Essa medida é útil e mais adequada quando se considera como fator mais relevante a existências de caminhos ligando um vértice a outro, com menor consideração à distância entre eles. Conforme aponta Borba (2013), para a sua identificação é necessário, calcular os graus de entrada, saída e total de cada vértice;

- **Centralidade por Proximidade**: Essa medida apresenta o vértice central considerando aquele que reúne a menor distância média para os demais nós do grafo. Para a sua identificação, é necessário o cálculo do caminho de menor custo de todos para todos os nós da estrutura, o que é possiblitado pela aplicação do algoritmo de Floyd Warshall. Nota-se que este método só é aplicável em grafos ponderados, ou seja, quando as arestas possuem pesos definidos;

- **Centralidade por Intermediação**: Diferentemente dos outros dois métodos, aqui a ênfase está no nó que atua como intermediário no maior número de cadeias (sequências). Em outras palavras, o vértice central será aquele que integra indiretamente o caminho para o maior número de vértices no grafo. Notadamente, essa medida é mais adequada quando o que se procura é pontos de parada ou de conexão nas rotas analisadas.

No presente trabalho, optou-se pela implementação do cálculo do centro do grafo pelo cálculo da média das distância de cada vértice para todos os demais nós. Essa medida combina dois dos métodos mencionados acima: primordialmente, a proximidade, quando considera o somatório das distâncias existentes entre um vértice e os demais; e o grau (de entrada), tendo em vista que esse atributo é utilizado para o cálculo da média correpondente.

$$MédiaDistâncias[v] = \frac{SomaDistâncias[v]}{GrauEntrada[v]} \quad \textbf{(1)}$$

Programação Paralela.

Usualmente, os programas de computadores são escritos pelos desenvolvedores com base uma sequência de tarefas a serem executadas de forma sequencial pelos processadores. Contudo, é notável que muitas dessas tarefas não são interdependentes e, por essa razão, não necessitariam de esperar a execução de todos os passos anteriores listados no código para então iniciar a sua execução. Em outras palavras, há a possibilidade de que partes de um programa sejam processados simultâneamente, permitindo ganhos de desempenho. A essa alternativa de processamento de instruções dá-se o nome de computação paralela.

Essa metodologia de computação tem se tornado cada vez mais relevante, dada a crescente necessidade por processamento de grandes quantidades de dados e com maior rapidez. Conforme bem menciona De Paula (2014), "nos últimos anos, a expectativa da ciência e do mercado indica a necessidade de redes de computadores cada vez mais rápidas, sistemas distribuídos altamente escaláveis e arquiteturas de computadores multiprocessados, mostrando claramente que o processamento paralelo é o futuro da computação." Por essa razão, é o que paralelismo tem sido mais frequentemente utilizado na resolução de problemas computacionais, considerando a sua superioridade aos modelos sequenciais em termos de velocidade.

A computação paralela foi possibilitada em função do desenvolvimento de computadores compostos por mais de um processador, bem como por unidades de processamento com variados núcleos. Nessas arquiteturas, é possível a execução de instruções de programas de forma simultânea, haja vista a existência de unidades de processamento que trabalham em paralelo, o que seria inviável em máquinas compostas por único processador. A esse respeito, vale destacar o que é apontado por Gaioso (2014) quando menciona que "a estrutura de uma máquina paralela envolve considerações sobre diversas características tanto dos processadores quanto da rede de comunicação". Isso quer dizer que, foram necessárias adaptações também na rede de comunicação dos computadores para compatibilização com o programação paralela.

No que se refere às diferentes arquiteturas de computadores, Michael J. Flynn, em 1966, apresentou uma classificação dos modelos existentes no que viria a ser conhecido como a Taxanomia de Flynn. Conforme De Paula (2014), a conceituação trazida por Flynn pode ser assim resumida:

a) Single Instruction Single Data (SISD): uma única unidade de controle é responsável por processar um único fluxo de instruções num único fluxo de dados;

b) Single Instruction Multiple Data (SIMD): projetos de arquitetura onde uma única instrução é executada simultaneamente em múltiplos fluxos de dados;

c) Multiple Instruction Single Data (MISD): este tipo de arquitetura é fonte de divergência entre pesquisadores da área de arquitetura de computadores. Navaux (1989) afirma que nenhum sistema conhecido se encaixa nesta categoria. Entretanto, Quinn (2003) aponta como exemplo para esta categoria o systolicarray;

d) Multiple Instruction Multiple Data (MIMD): arquiteturas que apresentam múltiplas unidades de processamento que manipulam diferentes fluxos de instrução com diferentes fluxos de dados.

Conforme se pode notar, as arquiteturas que possibilitam o processamento de múltiplos dados (SIMD e MIMD) são aquelas compatíveis com a computação paralela. Vale destacar que, a arquitetura MIMD possibilita não apenas o processamento paralelo de dados, o que também é realizado pela SIMD, mas também a execução simultânea de diferentes instruções, ou seja, programas diferentes.

Uma importante diferença entre os computadores que realizam o processamento sequencial e os que possibilitam a computação paralela é o tipo de memória utilizado. Enquanto nos primeiros, que são classificados basicamente como SISD, são utilizadas as memórias de acesso randômico (RAM), nestes últimos, é utilizada uma alternativa modificada chamada máquina paralelo de acesso randômico (PRAM). Segundo Gaioso (2014):

> A PRAM é constituída por vários processadores independentes que, geralmente, são vistos como sendo da mesma arquitetura e são conectados por uma memória compartilhada. Esses processadores trabalham sincronizados em paralelo e utilizam a memória compartilhada para computação e comunicação entre eles.

Considerando que a finalidade principal de se utilizar a programação paralela é o ganho de desempenho em relação à execução sequencial dos programas, faz necessária a definição de uma forma de aferição desse fator. À medida de ganho de desempenho obtido com os código paralelos dá-se o nome de speedup. Gaioso (2014) argumenta que o speedup é "medido, normalmente, pelo tempo que leva para completar uma tarefa num único processador sobre o tempo necessário para completar a mesma tarefa em N processadores paralelos". Em decorrência disso, a seguinte fórmula seria aplicável para a aferição do ganho de desempenho:

$$Speed - up = \frac{TempoSequencial(1)}{TempoParalelo(n)} \quad \textbf{(2)}$$

Graphic Processing Unit (GPU).

Os computadores mais comumente utilizados pelas pessoas em geral são compostos por uma CPU que realiza o processamento de programas e dados, além de uma unidade específica para o processamento gráfico, ou seja na geração das imagens transmitidas ao usuário pela tela da máquina. Esse componente que atua conjuntamente com a CPU na execução dessa tarefa é conhecida como GPU. Considerando que as aplicações gráficas exigem maior desempenho para a sua execução satisfatória, as GPU's são desenvolvidas de forma que superem as CPU em termos de velocidade de processamento.

Para que a velocidade de processamento mencionada no parágrafo anterior seja obtida, as GPU's são baseadas essencialmente em computação paralela. Conforme aponta Gaioso (2014), "a arquitetura basica da GPU consiste num conjunto de multiprocessadores (SMs), cada um contendo vários processadores (SPs)". Dessa forma, a própria constituição de unidades de processamento desse tipo são mais compatíveis com a utilização de paralelismo quando comparadas às CPUS.

É importante observar que a GPU é dependente da CPU para o seu perfeito funcionamento. Ocorre que os dados a serem processados estão armazenados na memória principal do computador. Diante disso, para que as tarefas possam ser executadas na GPU é necessário primeiro que esses dados sejam transferidos para a sua memória específica. Tal procedimento é realizado pela CPU quantas vezes forem necessárias e exigidas pelas instruções nela contidas. É fácil perceber que, em determinadas situações, essa constante fluxo de informações, pode resultar em perda de desempenho.

Vale destacar que a utilização das GPU's para outras finalidades além de processamente gráfico tem se tornado cada vez mais usual, em função do ganho de desepenho observado. Isso é observável principalmente em atividades que envolvam o processamento de enormes quantidades de dados ou com aplicação forte de paralelismo. Como exemplo podem ser citadas as tarefas realizadas em áreas como ciência de dados e inteligência artificial. A esse respeito De Paula (2014) resume bem quando afirma que "a computação com GPU pode ser considerada como sendo o uso de uma unidade de processamento gráfico como um coprocessador para acelerar as Central Processing Units (CPU) para computação científica e de propósito geral."

Padrão OpenCL.

O Open Computing Language (OpenCL) é um padrão para progamação paralela desenvolvido pela Apple Inc. e posteriormente submetido ao Krhonos Group para padronização no ano de 2008. O OpenCL foi elaborado com o objetivo de possibilitar, de forma aberta (opensource), o desenvolvimento de programas aptos a serem executados em plataformas de diferentes fabricantes, sem a necessidade de utilização de solução proprietária. Considerando a crescente uti-

lização de paralelismo nas aplicações, a criação desse padrão mostrou-se bastante relevante.

Para melhor entendimento do padrão e arquitetura OpenCL, alguns conceitos devem ser observados. Conforme aponta Silveira et al. (2010):

> Um ambiente computacional OpenCL é integrado por um hospedeiro (host), que agrega um ou mais dispositivos (devices). Cada dispositivo possui uma ou mais unidades de computação (compute units), sendo estas compostas de um ou mais elementos de processamento (processing elements).

A informação apresentada acima corresponde ao modelo de plataforma da arquitetura. Além deste, outros três aspectos são considerados para compreensão do OpenCL, o primeiro deles é o modelo de execução, do qual destaca-se os seguintes elementos:

- Kernel: que segundo Silveira et al (2010) é "executado em um espaço de índices de 1, 2 ou 3 dimensões, denominado NDRange (N-Dimensional Range)";
- Itens de Trabalho (work-itens): que correspondem às instâncias do kernel e possuem identificadores globais formados por uma tupla de índices;
- Grupos de Trabalho (work-groups): agrupamentos de itens de trabalho identificados internamente no grupo por identificadores locais;
- Contexto: que representa a relação de plataformas, dispositivos, estruturas e objetos próprios do padrão OpenCL que vão possibilitar o desenvolvimento de aplicações compatíveis;
- Fila de Comando: que constituem filas de instruções a serem passadas pelo hospedeiro ao dispositivo onde serão processadas, incluindo transferência de dados entre eles;
- Objetos de Memória: correspondendo aos locais onde os dados estão ou serão armazenados no hospodeiro e no dispositivo, possibilitando a realização de operações de leitura e escrita;
- Objetos de Programa: cuja função é o encapsulamento do código-fonte do kernel que será executado pela aplicação;

Tem-se ainda o modelo de programação, no qual há a possibilidade de execução de kernels por meio de paralelismo de dados (um kernel executado por vários itens de trabalho) ou paralelismo de tarefas (múltipos kernels executados cada um por um item de trabalho exclusivo). Por fim, o padrão é caracterizado pelo seu modelo de memória que pode ser: global, quando compartilhada por todos os itens de trabalho; local, onde o compartilhamento ocorre apenas entre itens de um mesmo grupo; privada, que é restrita a um único item; e contante, similar à global, porém habilitada apenas para operações de leitura.

A sintaxe do padrão OpenCL será melhor detalhada na seção de apresentação do algoritmo paralelo desenvolvido. É importante destacar que ela adota por base a Linguagem da especificação C99, com algumas modificações (restrições e acréscimos). A princípio, são apresentadas as funções que são utilizadas para a implementação de aplicação:

- get-global-id: retorna o índice global do item de trabalho na dimensão informada como parâmetro;
- clGetPlatformIDs: identifica as plataformas OpenCL existentes no hospedeiro (computador host);
- clGetDeviceIDs: identifica os dispositivos componentes do hospedeiro que possibilitam a execução de aplicações desenvolvidas com o Padrão OpenCL;
- clCreateContext: cria um contexto OpenCL, considerando os dispositivos e plataformas existentes no hospedeiro;
- clCreateCommandQueue: cria uma fila de comandos para ser executado no dispositivo e contexto selecionados;
- clCreateProgramWithSource: cria um objeto de programa a partir do código-fonte descrito para o kernel que será executado com paralelismo;
- clBuildProgram: compila o programa criado para execução nos dispositivos selecionados;
- clCreateKernel: cria o kernel a partir do programa compilado com o seu correspondente código-fonte;
- clCreateBuffer: cria um objeto de memória do tipo buffer destinado a possibilitar a transferência de dados entre a memória do hospedeiro e os dispositivos integrantes do contexto;
- clEnqueuReadBuffer: inclui uma instrução de leitura de dados armazenados no dispositivo para a memória do hospedeiro em uma fila de comandos selecionada;
- clEnqueueWriteBuffer: inclui, em uma fila de comandos existente, uma instrução de escrita de dados que estão na memória do hospedeiro para a memória do dispositivo selecionado;
- clSetArgKernel: corresponde à configuração dos argumentos da função kernel que será executada, indicando os endereços onde estão armazenados os dados na memória do hospedeiro;
- clEnqueueNDRangeKernel: envia o kernel para execução no dispositivo selecionado, inserindo-o em uma fila de comandos atribuída a esse dispositivo;
- clFinish: bloqueia a execução de novas instruções presentes no hospedeiro enquanto não houver a conclusão de todos os comandos existentes na fila em processamento no dispositivo selecionado;

- Além das funções mencionadas, há uma adicional de nome "clRelease" que libera os objetos OpenCL que foram criados durante a execução do progama. Ao nome da função acrescenta-se a identificação do tipo de objeto. Exemplo: "clReleaseKernel".

Materiais e Métodos

O presente trabalho se caracteriza como uma pesquisa aplicada, no que se refere à sua finalidade, tendo em vista que busca implementar uma aplicáção prática do algoritmo proposto. Para tanto, foram elaboradas versões sequencial e paralela do algoritmo utlizando equipamento com as seguintes características de hardware e software:

- Processador Intel Core i5-8250U 1.60 GHz com 4 núcleos e 8 threads;
- Placa de Vídeo integrada Intel UHD Graphics 620 com 4Gb de memória;
- 8Gb de Memória RAM;
- Sistema Operacional Windows 10;
- Linguagem C para implementação do algoritmo sequencial;
- OpenCL 3.0 para implementação do algoritmo paralelo.

Implementação

Estrutura de Dados Representativa do Grafo.

A estrutura de dados a ser utilizada no presente trabalho, conforme já mencionado anteriormente é o grafo orientado e ponderado. A representação escolhida e que melhor se adequa à implementação do algoritmo de Floyd-Warshall é a Matriz de Adjacências, com sua correspondente Matriz de Pesos. O código referente à estrutura de dados foi desenvolvido utilizando a linguagem C.

Para a geração da matriz representativa do gráfico, foi criada uma função que retorna um ponteiro para ponteiro de uma variável do tipo interio "allocate-matrix". Essa função recebe como parâmetro um número interio que corresponde ao número de vértices desejados (NumberOfVert). Na sua execução, é declarada um ponteiro para ponteiro com o nome de "matrix", que irá receber um endereço de memória alocado dinamicamente pela utilização da função "malloc" do tamanho de um inteiro multiplicado pelo número de vértices informado.

Na sequência, é executado uma iteração "for" partindo de 0 até o número total de vértices, que irá atribuir também por alocação dinâmica, para cada índice da matriz, um endereço de memória do tamanho de um inteiro multiplicado pelo número de vértices. Ao final da execução da iteração, a função irá retornar o ponteiro para ponteiro corrrespondente à matriz.

Em seguida, os pesos da matriz são gerados aleatóriamente por meio da função "rand", contendo valores de 1 a 500, sendo preenchido com valor infinito para os casos em que não há aresta entre os dois vértices.

Algoritmo Sequencial.

O algoritmo de Floyd-Warshall na forma sequencial foi implementando utilizando-se a linguagem de programação C. O código consiste em uma função que recebe como parâmetros de entrada o grafo gerado de forma aleatória, conforme apresentado no tópico anterior, representado por sua matriz de adjacências ponderada. Ao final de sua execução, a função resultará na matriz de distâncias do grafo processado.

Inicialmente, são declaradas três variáveis do tipo inteiro "i", "j" e "k" que representarão os contadores necessários à execução dos laços de repetição do algoritmo. Em seguida, são iniciados os referidos laços. Primeiro, um laço "for" partindo de 0 até o número total de vértices do grafo (DATASIZE), que percorrerá todas as linhas da matriz de adjacências. Na sequência, um outro laço "for" aninhado, com os mesmos parâmetros de partida e parada do anterior, que percorrerá, para cada uma das linhas individualmente, todas a colunas da matriz de adjacências. Isso significa que, nesse momento, todos os possíveis arcos do grafo serão visitados.

O terceiro laço "for", também com os mesmos parâmetros de partida e parada que os dois anteriores, irá percorrer, para cada coluna (k) em cada linha (i) do grafo, novamente todas as colunas (j). Isso é feito para que o programa possa comparar os valores dos pesos de todos os arcos "i", "j" com os correspondentes e "k","j", a fim de verificar, seguidamente até a última combinação possível, a existência de caminhos mais curtos entre os vértices "i" e "j".

Por fim, é feita uma nova verificação de condição, onde a distância entre os vértices "i" e "j" é comparada à soma dos pesos das arestas entre "i" e "k" e entre "k" e "j". Caso isso seja verdadeiro, a distância entre "i" e "j" recebe o valor da soma mencionada, significando que um caminho mais curto foi encontrado. Como dito anteriormente, essa comparação é feita sucessivamente até que todas as combinações possíveis de adjacências sejam verificadas, obtendo-se a matriz de distâncias como resultado da execução da função.

O vértice central do grafo é identificado pela chamada da função "calcula-centralidade", que recebe como parâmetro o grafo em análise representada pela sua matriz de distâncias gerada pela execução do algoritmo de Floyd-Warshall. Na função, primeiramente são declarados duas variáveis do tipo inteiro "i" e "j" que serão os contadores das iterações a serem realizadas, além de dois ponteiros para variáveis do tipo inteiro: "soma", que correponde a um vetor que armazenará o somatório das distâncias de um vértice para todos os demais; e "grau", que será também um vetor ao qual serão atribuídos os graus de saída (números de arestas das quais o nó é orgiem) de cada nó. Além desses, um terceiro ponteiro é declarado "media", desta vez do tipo float, que também

será um vetor que registrará o resultado da divisão das somas das distâncias pelo grau de cada nó. Destaca-se que aos três ponteiros são atribuídos endereços de memória alocada dinamicamente. São declaradas também três variáveis do tipo float "node-central-id", "node-central-media" e "node-central-grau", cuja finalidade é o armazenamento dos atributos do vértice que representa o centro do grafo após a sua identificação.

Após isso, um laço de repetição "for" é iniciado a partir de 0 até o número total de vértices do grafo (DATA-SIZE), percorrendo todas as linhas da matriz de distâncias. Para cada índice "i", os vetores "soma" e "grau" são inicializados com valor 0. Em seguida, outro laço "for" com os mesmos parâmetros percorre, para cada linha (i), todas as colunas (j) da matriz. Uma condição "if" é então verificada: se "i" e "j" são diferentes e se o valor da distância entre os dois vértices é diferente de infinito. Em caso positivo, é calculada a soma das distâncias de todas as arestas que tenham o vértice "i" como origem e o valor do índice correspondente do vetor grau é incrementado em uma unidade. Ao término das iterações do segundo laço "for" (j), é realizada a divisão da soma das distâncias verificada para o vértice "i" pelo seu correspondente grau de saída, ou seja, a quantidade de arestas que tem o nó como origem. O resultado dessa divisão corresponde à média das distâncias do nó "i" para todos os demais.

Concluída essa etapa, primeiro, duas das variáveis declaradas no início da função (node-central-media e node-central-grau) recebem o valor da média e do grau do nó de índice 0 no grafo, e em seguida, um novo laço de repetição "for" percorre todos os vértices, comparando as suas correspondentes médias das distâncias. Um condicional "if" verifica se a média do vértice que está sendo comparado é menor que a daquele que está no momento armazenado na variável node-central-media. Sendo o resultado verdadeiro, aquele passa a ser o vértice central com as novas comparações sendo realizadas sucessivamente até que o laço de repetição seja finalizado. Caso os valores sejam iguais, um novo condicional compara os graus, sendo selecionado como nó central aquele que apresentar a maior media. Ao final do processamento, o nó que estiver sendo indicado pela variável node-central-id é o vértice central. A função imprime na tela a idenficação do nó central, seu grau e sua média das distâncias e então é finalizada.

Algoritmo Paralelo.

O algoritmo em sua versão paralelizada utilizando o padrão OpenCL, é descrito como uma função kernel declarada como um ponteiro para uma variável do tipo const char "ProgramSource". Ela vai receber como parâmetro um ponteiro para o buffer (pathDistanceBuffer) que contém a matriz a ser processada pelo algoritmo de Floyd-Warshall armazenada na memória do hospedeiro, além de outro ponteiro para o buffer (pathBuffer) que replicará a mesma matriz na memória do dispositivo. Além disso, são recebidos como parâmetros uma variável do tipo inteiro "numNodes", que representa o número total de vértices que compõem o grafo que será processado", e outra variável do tipo inteiro chamada "pass" que corresponde ao índice da matriz equivalente à primeira iteração (k) do algoritmo sequencial. Isso decorre do fato de que as outras duas iterações (i e j) são dependentes da primeira, o que implica na impossibilidade de que os três índices sejam processados paralelamente. É necessário que o primeiro seja executado por meio de um "for" sequencial, chamando a execução do kernel repetidas vezes e passando o índice como parâmetro (pass), para que a matriz possa ser percorrida de forma paralela nos outros dois índices que compõem o algoritmo.

Dentro da função, duas variáveis do tipo inteiro são declaradas "xValue" e "yValue". A ambas é atribuído o índice gerado pela função "get-global-id". A variável "xValue" representa a iteração "j" do algoritmo sequencial, percorrendo as colunas da matriz (dimensão 0). A variável "yValue" corresponde ao "i" da versão sequencial e percorre as linhas (dimensão 1). Além disso, é declarada uma outra variável do tipo inteiro "k" que recebe o valor da parâmetro recebido como "pass" mencionado no parágrafo anterior.

Na sequência, duas novas variáveis do tipo inteiro são declaradas "oldWeight" e "tempWeight". A primeira recebe como valor o peso inicial correspondente a cada aresta do grafo. A segunda recebe a soma dos pesos existente entre os arcos de "yValue" a "k" e deste a "xValue". Em seguida, há uma verificação de condição: caso o valor de "tempWeight" seja menor que o de "oldWeight", a distância da aresta entre "yValue" e "xValue" passa a ser o valor indicado pela primeira. Essa verificação se repete, de forma paralela, até que todos as combinações de pares de vértices sejam processadas.

Descrito o kernel do algoritmo, passa-se as especificações do padrão OpenCL para execução no dispositivo acelerador. Primeiramente, é feita a declaração das variáveis que vão armazenar a indicação do contexto (context), propriedades do contexto (properties[3]), kernel, fila de comandos (command-queue), programa (program), número de plataformas (num-of-platforms), idenficadores das plataformas (platform-id), número de devices (num-of-devices), identificadores dos devices (device-id), os buffers (path-disbuffer e path-buffer), as dimensões dos itens de trabalho global e local (global[2] e local[2]), o tamanho dos blocos de execução (block-size) e uma variável destinada ao armazenamento do código de erro a ser retornado pelas funções OpenCL que serão utilizadas.

O identificador das plataformas é obtido a partir da função "clGetPlatformIDs", passando-se como parâmetros o número de plataformas desejadas e os endereços das variáveis "num-of-platforms" e "platform-id". O mesmo procedimento é realizado para obtenção das informações do device, desta vez por meio da função "clGetDeviceIDs". Ela recebe como função, o identificador da plataforma gerado anteriormente, o tipo de device desejado (neste caso GPU) e os endereços das variáveis num-of-devices e device-id.

Em seguida, é criado o contexto, utilizando a função "clCreateContext, passando como parâmetros as propriedades do contexto, o número de dispositivos desejado e os endereço da variável "device-id" e da variável para armazenamento do código de erro a ser retornado pela função.

Após, cria-se a fila de comandos com a função clCreateCommandQueue. São passados como parâmetros o contexto, o identificador do device a ser utilizado e o endereço da variável de erro.

Então, cria-se o programa pela utilização da função "clCreateProgramWithSource", fornecendo os parâmetros correspondentes ao contexto, o número de strings que compõe o array do código-fonte, o endereço da variável que o armazena (ProgramSource) mencionada no início desta seção e a variável para armazenamento do erro. Com o programa criado, é feita a sua compilação a partir da função "clBuildProgram". Os parâmetros informados são o programa recém criado e o valor "0" para que a compilação seja aplicada a todos os dispositivos presentes no contexto. Finalmente, o kernel é criado com a utilização da função "clCreateKernel", informando-se o programa, o nome da função kernel no formato de uma string e o endereço da variável de erro.

Os passos seguintes são a criação dos objetos de memória buffer pela utilização da função "clCreateBuffer". É criado um buffer (path-dis-buffer) correspondente ao local onde estarão armazenados os dados da matriz na memória. O objeto é habilitado para operações de leitura e escrita (CL-MEM-READ-WRITE) e recebe como parâmetro de tamanho o quadrado do número de vértices (DATA-SIZE). Após gerado, o buffer é inserido na fila de comando (command-queue) para escrita dos dados nas variável declarada com esse objetivo (path-dis-mat), copiando os dados para a memória da GPU. Esse procedimento é realizado com a utilização da função "clEnqueueWriteBuffer". Entre os parâmetros passados, destaca-se o comando CL-TRUE, cuja finalidade é garantir que a fila de comandos só seja encerrada e passe para a próxima etapa após o processamento de todos os itens de trabalho.

Feito isso, passa-se então à configuração dos argumentos do kernel que será executado. A função utilizada é a "clSetKernelArg", sendo invocada para cada um dos parâmetros definidos no código-fonte do kernel. É necessário informar, além do kernel a que se refere, o índice do argumento (0 a n), o tamanho correspondente, e o endereço de onde os dados estão armazenados. Concluída mais essa etapa, o programa está apto a processar o algoritmo de forma paralela.

Conforme mencionado anteriormente, a execução do kernel é realizada multiplas vezes por meio de uma iteração "for". Esse laço é definindo com início em 0 até o número total de vértices (DATA-SIZE). Desse modo, são gerados os índices "k" do algoritmmo, chamando, para cada um deles de forma sequencial, a execução do kernel de forma paralela, por meio da função "clEnqueueNDRangeKernel". É necessário destacar que, antes desse último comando, o quarto argumento do kernel é novamente configurado, uma vez que ele corresponde ao próprio índice proveniente da iteração. Depois de concluído o processamento de todos os dados, a fila de comando é finalizada.

O procedimento seguinte é a leitura dos dados do buffer para sua transferência da memória da GPU para a do hospedeiro. Tais dados correspondem a matriz de distâncias calculada pela execução do algoritmo de Floyd-Warshall. Isso é feito pela invocação da função "clEnqueueReadBuffer", sobrescrevendo os dados originais.

Concluído o processamento do algoritmo e tendo a matriz de distâncias sido gerada, passa-se então aos cálculos necessários à identificação do centro do grafo. Nos testes realizados com a execução sequencial, verificou-se que o tempo necessário para idenficação do vértice central é ínfimo, o que torna a paralelização desnecessária e de pouco benefício. Por essa razão, optou-se por mantê-la de forma sequencial. Apenas foram criadas duas funções: "calcula-centralidade-buffer" e calcula-centralidade. A primeira realiza os cáculos a partir dos dados que estão no buffer. A segunda, por sua vez, faz o mesmo utilizando as informações armazenadas na variável "matrix" declarada no início do código.

Após o processamento dos dados, identificação do vértice central e aferição dos tempos de execução, os recursos de memória ocupados pelos objetos OpenCL que foram gerados são liberados utilizando a função "clRelease".

Apresentação e Análise dos Resultados

Após a compliação do código desenvolvido, foi realizada a sua execução sob grafos gerados aletatóriamente contendo entre 100 e 9.000 vértices no total. Conforme se observa na Figura 1, abaixo, para estruturas com total de nós inferior a 1.500, o processamento paralelo (linha azul claro) não apresenta ganho de desempenho, pois, nesse nível, a execução sequencial (linha azul escuro) também apresenta boa performance. Contudo, ao se ultrapassar essa número de vértices, o processamento sequencial potencializa o tempo necessário para a sua conclusão, enquanto que o paralelo também cresce, porém, de forma moderada.

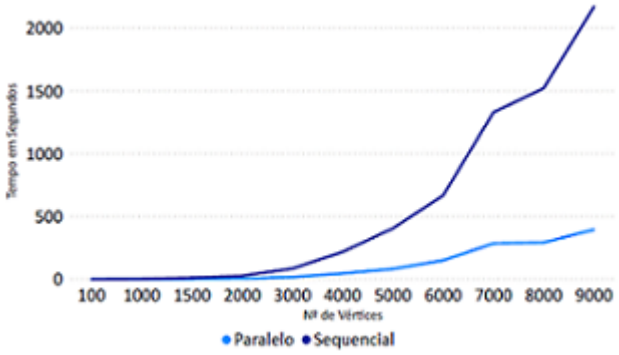

Fig. 1. Tempos de Execução - Paralelo x Sequencia

O ganho de desempenho (speed-up) observado na execução do algoritmo paralelizado varia entre 4 e 6 vezes em

relação ao sequencial, conforme o número de véritces componentes do gráfico. A média observada corresponde a 5,10, o que pode ser considerado com um resultado muito bom. Conforme pode ser visto na Figura 2, abaixo, o maior nível de speed-up é atingido logo na execução de 1.500 nós, decaindo um pouco, conforme se aumenta a quantidade até o processamento de 6.000 nós, quando volta a subir novamente:

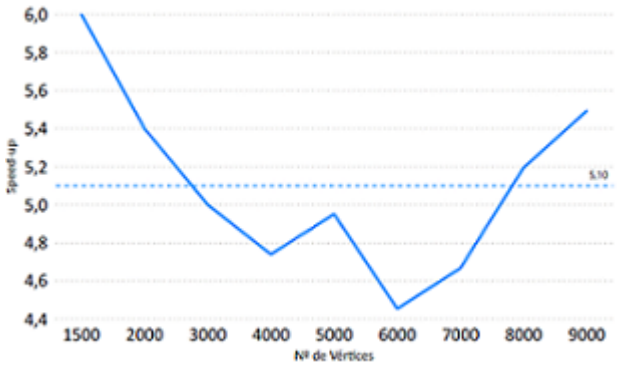

Fig. 2. Sepeed-up

Quando comparados os desempenhos das aplicações paralela e sequencial proporcionalmente ao processamento de cada vértice, nota-se que os tempos observados são muito próximos aos relativos a todo o grafo. As Figuras 3 e 4 apresentadas na sequência fornecem a visualização dos dados que comprovam essa afirmação:

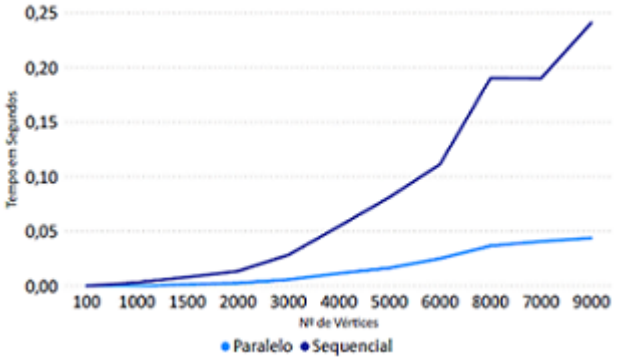

Fig. 3. Tempos de Execução por Vértice - Paralelo x Sequencia

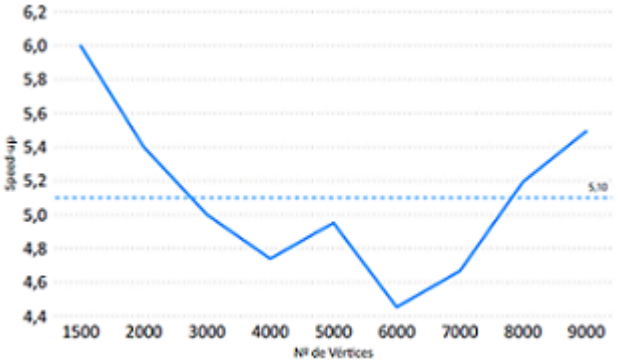

Fig. 4. Speed-up por Vértice

Diante dos resultados observados com a execução da aplicação, verifica-se que a paralelização do algorítimo de Floyd-Warshall apresenta relevante ganho de velocidade de processamento em relação à sua execução sequencial. A Tabela 1, abaixo, apresenta o detalhamento dos dados de tempo de execução e speed-up registrados a partir das simulações realizadas:

Vértices	Sequencial	Paralelo	Speed-up	P/Nó - Seq	P/Nó - Par	Speed-up	Centro
100	0	0	NaN	0,00	0,00	NaN	0
1000	3	0	0,00	0,00	0,00	0,00	0
2000	27	5	5,40	0,01	0,00	5,40	0
2500	52	10	5,20	0,02	0,00	5,20	0
3000	85	17	5,00	0,03	0,01	5,00	0
3500	153	29	5,28	0,04	0,01	5,28	0
4000	218	46	4,74	0,05	0,01	4,74	0
4500	302	66	4,58	0,07	0,01	4,58	0
500	0	0	NaN	0,00	0,00	NaN	0
5000	406	82	4,95	0,08	0,02	4,95	0
5500	546	122	4,48	0,10	0,02	4,48	0
6000	668	150	4,45	0,11	0,03	4,45	0
6500	833	174	4,79	0,13	0,03	4,79	0
7000	1330	285	4,67	0,19	0,04	4,67	0
7500	1302	261	4,99	0,17	0,03	4,99	0
8000	1522	293	5,19	0,19	0,04	5,19	0
8500	1825	334	5,46	0,21	0,04	5,46	0
9000	2170	395	5,49	0,24	0,04	5,49	0

Fig. 5. Tabela 1 - Resumo das Simulações (Tempos e Speed-up)

Conforme pode ser observado, para o grafo contendo 9.000 vértices (81.000.0000 de arestas), o tempo total gasto na execução pelo algoritmo sequencial corresponde a 2.170 segundos, o que equivale a um pouco mais de 36 minutos. Por outro lado, com esse mesmo número de nós, o algoritmo paralelizado precisou de apenas 395 segundos, equivalente a 6 minutos e meio. Nota-se uma redução de aproximadamente 80/100 do tempo total, ratificando a vantajosidade do modelo que utiliza paralelismo em GPU. Outra informação relevante que deve ser destacada é que o tempo para processamento da função que calcula o centro do grafo é bem pequeno, sendo inferior a 1 segundo.

Conclusão

Considerando os resultados obtidos com a implementação da aplicação proposta pelo presente trabalho, pode-se concluir com elevado grau de assertividade que a utilização de programação paralela para processamento em GPU otimiza o tempo de execução do algoritmo de Floyd-Warshall. O ganho médio de 5 vezes no processamento dos dados é um resultado muito bom e relevante, haja vista que as iterações realizadas pelo algoritmo são elevadas ao cubo a medida em que o número de vértices avaliado aumenta, o que leva a intervalos necessários para conclusão excessivamente longos.

O estudo também demonstrou que o Padrão OpenCL pode ser de grande utilidade para desenvolvedores de aplicações que utilizam paralelismo em GPU. A arquitertura se apresenta como uma alternativa open source a soluções proprietárias, que pode ser utilizada em diferentes dispositivos compatíveis, reduzindo os custos de implementação e conferindo maior acessibilidade a ferramentas de programação paralela e de alto desempenho.

Pode-se afirmar, também, que o algoritmo de Floyd-Warshall, na sua forma paralelizada, pode ser utilizado para solução de problemas práticos de organizações empresariais e da sociedade de maneira geral. Como exemplo, cita-se a definição dos melhores locais para instalação de pontos de atendimento de saúde, em função da proximidade em relação à população a ser atendida. Em situações como a vivida nos dias atuais, em que o mundo enfrenta uma pandemia com implicações sem precedentes, uma aplicação que propicia a tomada de decisão de maneira rápida e assertiva a partir do processamento de dados complexos e em grande escala, pode ser muito relevante e de grande contribuição para a sociedade. Por essa razão, sugere-se como trabalhos futuros que podem ser desenvolvidos sobre o tema, a utilização do algoritmo paralelizado para solução de problemais reais no sentido do exemplo apresentado.

Referências

BORBA, Elizandro Max. Medidas de Centralidade em Grafos e Aplicações em redes de dados. 2013. Disponível em: <https://www.lume.ufrgs.br/handle/10183/86094> Acesso em: 21 nov. 2020.

DE PAULA, Lauro Cássio Martins. CUDA vs. OpenCL: uma comparação teórica e tecnológica. ForScience, v. 2, n. 1, p. 31-46, 2014. Disponível em: <http://forscience.ifmg.edu.br/forscience/index.php/forscience/article/view/53> Acesso em: 21 nov. 2020.

GAIOSO, R. R. et al. Paralelização do algoritmo Floyd-Warshall usando GPU. SIMPÓSIO EM SISTEMAS COMPUTACIONAIS. XIV, 2013. Disponível em: <https://www.academia.edu/download/32856600/Versao-publicada.pdf> Acesso em: 21 nov. 2020.

GAIOSO, Roussian Di Ramos Alves. Implementações paralelas para os problemas do fecho transitivo e caminho mínimo APSP na GPU. 2014. Disponível em: <https://repositorio.bc.ufg.br/tede/handle/tede/3471> Acesso em: 21 nov. 2020.

MOREIRA FILHO, Roberval Gonçalves. Heurística para Determinação de Caminho Mínimo com Parada Intermediária, Aplicada ao Resgate Médico de Urgência. 2017. Disponível em: <https://di.uern.br/tccs2019/html/ltr/PDF/013002724.pdf> Acesso em: 22 nov. 2020.

SILVEIRA, César LB; DA SILVEIRA JR, Luiz G.; CAVALHEIRO, Gerson Geraldo H. Programação em OpenCL: Uma introdução prática. 2010. Disponível em: <http://www.sbgames.org/papers/sbgames10/computing/tuto-comp2.pdf> Acesso em: 21 nov. 2020.