Uma proposta de solução aproximada para o problema do subgrafo planar de peso máximo

Vinícius de S. Coelho[1], Wellington S. Martins[1], Leslie R. Foulds[1], Elisângela S. Dias[1], Diane Castonguay[1], Humberto J. Longo[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia – GO – Brazil

{viniciusdesousa,wellington,lesfoulds,elisangela,diane,longo}@inf.ufg.br

Abstract. *This work introduces approximate algorithms for the Weighted Maximal Planar Graph Problem (WMPG). The algorithms are based on a "dimpling" approach that successively inserts vertices, within faces of the graph. This method has the advantage that any graph generated by it is a maximal planar graph, thus obviating the need for graph planarity testing. We present both sequential and parallel dimpling algorithms for the WMPG and their implementations. Several tests were performed on instances with up to 85 vertices to evaluate both types of algorithms. Results obtained with the parallel versions, implemented on a manycore architecture, were up to 107 times faster than the sequential version.*

Resumo. *A proposta deste trabalho consiste em uma solução aproximada para o problema do subgrafo planar de peso máximo (WMPG - Weighted Maximal Planar Graph). O algoritmo baseia-se na adição de vértices, aproveitando-se da construção de triangulações nas faces do grafo. A vantagem do uso deste algoritmo dá-se pelo fato que todo grafo gerado por ele é maximal planar, descartando a necessidade de um teste de planaridade. Apresentamos um algoritmo sequencial e um paralelo para o problema WMPG e suas respectivas implementações. Os resultados obtidos com a versão paralela executando em uma arquitetura manycore, com instâncias de até 85 vértices, apresentaram speedups de até 107 vezes em relação à solução sequencial.*

1. Introdução

Seja $G = (V, E)$ um grafo simples completo não direcionado, no qual V é o conjunto de vértices e um E é o conjunto de arestas. Seja $n = |V|$ e $m = |E| = \frac{n \cdot (n-1)}{2}$. A cada aresta $e \in E$ é associado um peso $w(e) \geq 0$. Um grafo é dito planar se for imerso a uma superfície e se quaisquer duas de suas arestas não se cruzam. Um grafo $G' = (V', E')$ é um subgrafo de G se G' é um grafo, tal que $V' \subseteq V$ e $E' \subseteq E$. G' é planar maximal se G' é planar e não é possível adicionar uma aresta sem que a restrição de planaridade seja violada. A terminologia teórica utilizada neste artigo pode ser encontrada em [Foulds 1992].

Obter informações relevantes sobre determinada instância de um problema modelado como um grafo pode tornar-se uma tarefa trabalhosa, sendo necessário que o grafo seja "filtrado". Um dos possíveis resultados do processo de filtragem é o subgrafo planar maximal onde o peso de suas arestas é máximo. Contudo, não existe um algoritmo

de complexidade polinomial no tamanho do grafo capaz de resolver este problema de forma ótima para instâncias de maior porte. De fato, este problema é classificado como NP-Difícil [Garey and Johnson 1979]. Dado esse fato, surgiu a necessidade do uso de heurísticas capazes de obter um resultado quase ótimo em tempo polinomial e que seja satisfatório na maioria das aplicações.

O problema do subgrafo planar de peso máximo (WMPG - *Weighted Maximal Planar Graph*) é um problema clássico da área de planejamento do setor industrial. A sua importância está diretamente relacionada ao projeto de *layouts*, instalações e manuseio de materiais. A modelagem do WMPG pode ser aplicada também na bolsa de valores.

A proposta para este trabalho baseia-se em uma solução aproximada para o problema WMPG. A abordagem gera $\binom{n}{4}$ combinações de vértices, cada uma representando uma estrutura de *tetraedro* (4-*clique*), proposta em [Foulds and Robinson 1978, Foulds and Robinson 1979]. Estes tetraedros são chamados de sementes, uma vez que irão "desenvolver-se" e gerar um subgrafo planar maximal de G.

A cada etapa do algoritmo, vértices serão adicionados às faces do grafo, construindo triangulações. Uma de suas vantagens é não depender de um teste de planaridade. Pelo fato de usar vários tetraedros, a solução final geralmente apresenta uma maior acurácia, comparando-se a outros algoritmos da literatura baseados na estratégia de [Foulds and Robinson 1978, Foulds and Robinson 1979] e que expandem somente uma semente. Ao final da execução, uma destas sementes desenvolvidas representará a melhor solução, ou seja, o subgrafo cuja soma de suas arestas será a maior entre todas analisadas.

Nós apresentamos um algoritmo sequencial que segue estas ideias e, posteriormente, estendemos esta solução para um contexto paralelo com p processadores e uma memória compartilhada. As implementações destes algoritmos foram realizadas em uma arquitetura sequencial convencional (CPU) e em uma arquitetura *manycore* com múltiplas GPUs. Os resultados obtidos usando a arquitetura *manycore*, executando instâncias de até 85 vértices, apresentaram *speedups* de até 107 vezes.

O artigo está organizado da seguinte maneira: na Seção 2 são apresentadas propostas anteriores relacionados a este problema. Os conceitos sobre a arquitetura da GPU são abordados na Seção 3. A descrição dos algoritmos é apresentada na Seção 4. Detalhes sobre as implementações são descritos na Seção 5. Na Seção 6, são reportados estudos experimentais e, na Seção 7, são apresentadas as considerações finais.

2. Propostas anteriores

[Tumminello et al. 2005] propuseram uma técnica onde extrai-se um subgrafo contendo as relações de maior importância, chamada de PMFG (*Planar Maximally Filtered Graph*). O uso deste procedimento com ações do mercado de capitais norte-americano demonstrou que o grafo obtido contém relações significantes referentes à estrutura e propriedades do mercado. Em [Massara et al. 2015] é feito um comparativo com o algoritmo PMFG, de complexidade $O(n^3)$, e apresentado o algoritmo TMFG (*Triangulated Maximally Filtered Graph*), de complexidade $O(n^2)$. A implementação do algoritmo apresentou um melhor desempenho computacional comparado ao PMFG.

Apesar do objetivo simples das aplicações mencionadas, o problema é bem complicado na prática. Para instâncias pequenas, a quantidade de combinações possíveis

já é enorme. Mesmo com o uso de algum algoritmo de teste de planaridade eficiente, seja ele em tempo linear, no pior caso, seria necessário testar (para $n = 10$) todas as $3.773.655.750.150$ possibilidades, sendo que nem todas são planares.

Isso inviabiliza a solução, pois em um cenário da vida real, a quantidade de vértices do problema é bem maior que 10. [Foulds and Robinson 1976] descreveram um algoritmo baseado em derivação e limitação (*branch-and-bound*) para obter resultados ótimos. Apesar das restrições aplicadas, o tempo computacional necessário para apenas 12 vértices ainda é enorme. O que se tem na literatura são heurísticas capazes de obter um resultado quase ótimo em tempo polinomial. Uma das soluções pioneiras foi proposta em [Foulds and Robinson 1978, Foulds and Robinson 1979]. A ideia consiste em extrair de um grafo completo o grau de importância de cada vértice, somando os pesos das arestas de cada vértice. Dessa forma, os 4 vértices mais "pesados" representarão uma solução inicial, consistindo em uma estrutura de tetraedro [Foulds and Robinson 1979], que em suma é um grafo simples completo (K_4).

Uma característica particular de subgrafos planares maximais é a presença de regiões triangulares, as faces, cada uma composta por exatamente 3 vértices. Tendo isso em mente, a solução é construída a partir das faces do grafo. Para cada vértice que não esteja presente na solução atual, será calculada a importância da inserção deste vértice em uma das faces. A importância é calculada pela soma das 3 arestas que serão adicionadas caso o vértice seja escolhido. Este método descrito em [Foulds and Robinson 1979], chamado por muitos autores de *the deltahedron method*, é também conhecido como *dimpling*, criando espécies de "covinhas" no grafo. A sua vantagem é a ausência de um teste de planaridade, pois ela é mantida a cada etapa de inserção de vértices. O método é repetido até que não existam vértices a serem escolhidos. Por ser uma estratégia gulosa, o algoritmo procura a combinação de maior peso a cada etapa de escolha. No entanto, é possível que a escolha de maior peso não produza a melhor solução.

A nossa proposta segue a linha de raciocínio de [Foulds and Robinson 1979] e [Massara et al. 2015]. Dado um tetraedro inicial, é calculado o ganho da inserção de um vértice em cada uma das faces do grafo, somando-se o peso das arestas que serão adicionadas. Junto com o vértice, são adicionadas 3 arestas adjacentes aos vértices da face. A face escolhida é removida, dando origem a 3 novas faces.

A nossa proposta apresenta, em relação às anteriores, três inovações. Primeiro, ao invés de escolher apenas um tetraedro, exploramos todo o conjunto combinatorial de possíveis tetraedros. Dessa forma, é possível obter uma solução mais próxima da ótima, visto que diferentes tetraedros podem resultar em diferentes soluções. Em segundo lugar, evitamos uma busca sequencial (linear), quando da escolha do par de vértice e face, através do uso de uma fila de prioridades (*heap*). Por último, usamos uma estrutura de dados que reúne o conjunto de faces, facilitando o rápido acesso a qualquer uma delas.

3. Paralelismo e a arquitetura *manycore* das GPUs

Processamento paralelo é uma tendência crescente na indústria de computadores visto que a velocidade de operação dos processadores individuais tem sido fortemente afetada pelos limites físicos envolvidos na sua construção. Praticamente todos os processadores hoje são sistemas paralelos e esta tendência tem sido seguida pelas chamadas arquiteturas *manycore*, como as GPUs (*Graphic Processing Units*). Processadores *many-*

core utilizam núcleos mais simples e mais lentos, mas em contagens muito maiores, da ordem de milhares de núcleos.

O baixo custo e a facilidade de aquisição de aceleradores na forma de GPUs tem tornado estas plataformas bastante atraentes para a implementação de algoritmos paralelos. No entanto, as GPUs têm uma arquitetura e organização de memória diferente e, para explorar plenamente as suas capacidades, é necessário um alto grau de paralelismo (dezenas de milhares de *threads*) e um uso adequado dos seus recursos de *hardware*. Isto impõe algumas restrições em termos de projeto de algoritmos paralelos apropriados, que exige a concepção de novas soluções e novas abordagens de implementação.

A arquitetura CUDA reune três conceitos principais (uma descrição detalhada pode ser encontrada em [Kirk and Hwu 2011]): Processadores de Fluxo (SP), Multiprocessadores de Fluxo (SM) e uma hierarquia de memória (global, compartilhada e registradores). Os SPs, conhecidos também como CUDA *cores*, são definidos como núcleos de processamento e são constituídos de unidades lógicas e aritméticas (ULA) e unidades de ponto flutuante (FPU). Os SPs são agrupados em estruturas que compartilham vários recursos do *hardware*. Essas estrututas são chamadas de Multiprocessadores de Fluxo (SM), onde cada uma é responsável por uma grande quantidade de *threads*. Embora todos os SMs não trabalhem sincronizadamente, os SPs contidos em único SM processam os dados de maneira síncrona.

A presença de centenas de registradores dentro dos SMs permite que as trocas de contextos entre as *threads* sejam mais eficientes, maximizando o desempenho de processamento dos SMs. Isso possibilita que a GPU tenha uma solução eficaz em relação à latência requerida para acessar a memória, no qual é gasto centenas de ciclos de *clock* para buscar e armazenar os dados. Quando um grupo de 32 *threads* executando em um SM fica ocioso por acessar a memória ou por alcançar uma barreira de sincronização, outro grupo de *threads* é escalonado instantaneamente para o SM. Isso é realizado pelos escalonadores de *warps* contidos nos SMs.

No modelo de programação CUDA, uma aplicação desenvolvida não é totalmente paralelizada, pois requer duas fases: uma sequencial e uma paralela. Elas se alternam entre si uma ou várias vezes em uma única execução. A CPU, também conhecida como *host*, inicia a execução, de forma sequencial. A GPU, chamada de *device*, é responsável pela fase paralela. Essa fase é realizada por funções, chamadas de *kernels*. Quando a execução de um *kernel* é concluída, o controle é redirecionado para a fase sequencial na CPU. Na fase paralela, milhares de *threads* podem ser lançadas na GPU. A quantidade de *threads* é determinada no momento do lançamento do *kernel*.

As *threads* são organizadas em grades e blocos de *threads*. Cada bloco de *threads* da grade possui identificadores em relação à sua posição nos eixos x e y na grade e toda *thread* de um bloco também possui um identificador de sua posição nos eixos x, y e z em relação ao bloco. Uma grade está associada a um único *kernel* e, consequentemente, os blocos de *threads* da grade contêm o mesmo código. As *threads* que estão dentro de uma grade são organizadas em uma lista de blocos de *threads*. Essa lista pode ser de uma ou duas dimensões. Todos os blocos de uma mesma grade possuem o mesmo tamanho. Esse tamanho é definido pela quantidade de *threads*, sendo que a quantidade máxima pode variar de acordo com as gerações das arquiteturas.

As *threads* fazem uso da hierarquia de memória. Cada *thread* tem acesso à memória privada e, *threads* de um mesmo bloco enxergam o mesmo espaço de endereçamento da memória compartilhada. Por sua vez, as *threads* de uma grade alcançam toda a memória global, cuja a latência é a maior entre todas as memórias da GPU.

4. Algoritmos propostos

O Algoritmo 1, a nossa solução sequencial, começa com vários tetraedros gerados a partir de $\binom{n}{4}$ combinações. Para cada semente, cria-se uma lista $\mathcal{S}$ contendo todos os vértices que não estão na semente (Linha 4), uma matriz de adjacências $\mathcal{E}$ com a solução atual (Linha 5), uma lista F contendo o conjunto de faces da semente (Linha 6) e uma estrutura de *heap* $\mathcal{H}$ contendo o par de vértice e face de maior ganho (Linha 7). Em seguida, para cada semente é calculada a soma dos pesos de suas arestas e este valor é armazenado em $TmpMax$ (Linha 8).

A cada passo é extraido de $\mathcal{H}$ o par de vértice e face de maior pontuação (Linha 10). O movimento de *dimpling* é então aplicado ao par de vértice e face escolhido. O vértice escolhido é removido de $\mathcal{S}$ e a face é removida de F, dando lugar a 3 novas faces (Linha 13). O processo de *dimpling* é ilustrado na Figura 1. Em seguida, é adicionado a $TmpMax$ o valor das 3 arestas inseridas na semente (Linha 15). É necessário que $\mathcal{H}$ seja atualizado para a próxima iteração, calculando-se o ganho de cada um dos vértices restantes quando inseridos nas 3 novas faces (Linha 16). Este processo é repetido enquanto $\mathcal{S}$ não estiver vazio. Concluída a etapa anterior, é então verificado se, para o grafo gerado, a soma do pesos de suas arestas é maior que o valor armazenado em $MaxWeight$. Em

Algoritmo 1: Algoritmo Sequencial

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n\times n}$ com arestas de pesos não negativos.
Saída: Grafo planar maximal representado por $\mathcal{M}'$, matriz esparsa filtrada de $\mathcal{M}$.

1 $\mathcal{C} \leftarrow$ Conjunto de possíveis tetraedros $\{v_1, v_2, v_3, v_4\}$, com $v_i \in V$;
2 $MaxWeight \leftarrow 0$;
3 **para** $k \leftarrow 1 \leq |\mathcal{C}|$ **faça**
4 $\quad \mathcal{S} \leftarrow V - \mathcal{C}_k$;
5 $\quad \mathcal{E} \leftarrow \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$;
6 $\quad F \leftarrow \{\{v_1, v_2, v_3\}, \{v_1, v_2, v_4\}, \{v_1, v_3, v_4\}, \{v_2, v_3, v_4\}\}$;
7 $\quad \mathcal{H} \leftarrow \forall s \in \mathcal{S}$, calcule o ganho máximo de $\{s, f\}, \forall f \in F$;
8 $\quad TmpMax \leftarrow \sum \mathcal{E}$;
9 $\quad$ **enquanto** *$\mathcal{S}$ não estiver vazio* **faça**
10 $\quad\quad \{v_i, f_i\} \leftarrow \mathcal{H} - \max\{\mathcal{H}\}$, $v_i \in \mathcal{S}$ e $f_i \in F$;
11 $\quad\quad$ Execute o *dimpling*;
12 $\quad\quad \mathcal{S} \leftarrow \mathcal{S} - \{v_i\}$;
13 $\quad\quad F \leftarrow (F - \{f_i\}) \cup \{f_a, f_b, f_c\}$;
14 $\quad\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{\{v_i, v_x\}, \{v_i, v_y\}, \{v_i, v_z\}\}$, com $v_x, v_y, v_z \in f_i$;
15 $\quad\quad TmpMax \leftarrow TmpMax + \{v_i, v_x\} + \{v_i, v_y\} + \{v_i, v_z\}$
16 $\quad\quad \mathcal{H} \leftarrow \mathcal{H} \cup \{\forall s \in \mathcal{S}$, calcule o ganho máximo de $\{s, f_a\}, \{s, f_b\}, \{s, f_c\}\}$;
17 $\quad$ **se** $TmpMax \geq MaxWeight$ **então**
18 $\quad\quad MaxWeight \leftarrow TmpMax$;
19 $\quad\quad \mathcal{M}' \leftarrow \mathcal{E}$;
20 **retorna** $\mathcal{M}'$.

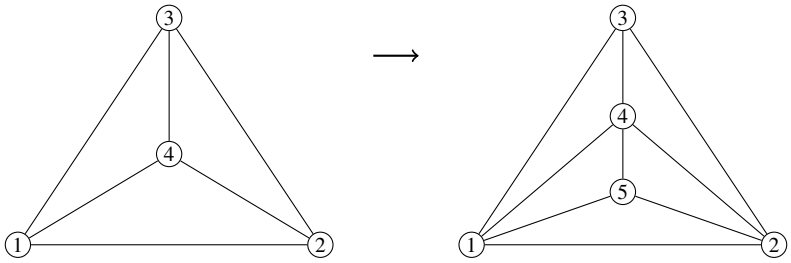


Figura 1. Inserção de um vértice em uma face, resulta em 3 novas faces.

caso afirmativo, o valor é atualizado e o grafo $\mathcal{E}$ que gerou a resposta é então armazenado em $\mathcal{M}'$ (Linhas 18 e 19).

Como o algoritmo sequencial processa os tetraedros de maneira totalmente independente, e com alto custo computacional resultante, nós estendemos este algoritmo para um ambiente paralelo com p processadores e memória compartilhada. Embora direta, esta extensão requer um gerenciamento das sementes e dos resultados produzidos por cada instância. Além disso, existe o desafio de se trabalhar com uma estrutura irregular (grafo) em uma abordagem altamente paralela onde os custos de comunicação (acesso à memória compartilhada) costumam ser altos para acessos não aglutinados. Por conta disso, nós estruturamos os dados de faces e vértices como um vetor de estruturas para garantir acesso aglutinado (*coalesced*) por vários processadores.

O algoritmo paralelo resultante é apresentado em Algoritmo 2. Cada semente terá um identificador paralelo, com uma lista de vértices, um conjunto de faces e uma estrutura de fila de prioridade. Dada uma lista dos vértices que não pertencem à semente, um vértice é escolhido de forma que a sua inserção em uma face produza a maior pontuação a cada iteração (Linha 5). O vértice e face são removidos de suas listas e dão origem a 3 novas faces a partir do vértice inserido (Linha 6). Ao final da iteração, cada identificador atualizará o valor do peso máximo caso este seja maior que o valor mais atual (Linha 9). Esta operação é crítica e, por isso, apenas um identificador deve executá-la por vez.

Algoritmo 2: Algoritmo Paralelo

Entrada: Grafo completo $G = (V, E)$.
Saída: Subgrafo Planar Maximal e o peso máximo obtido.

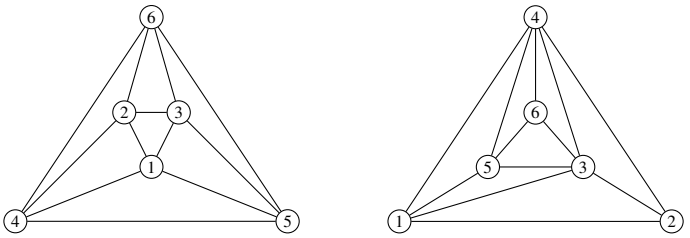
```
1  para todo id representando uma semente faça em paralelo
2      Obtenha os vértices que não estão na semente;
3      Construa o conjunto de faces e a fila de prioridade;
4      enquanto existirem vértices a serem inseridos faça
5          Encontre o vértice e face de ganho máximo;
6          Execute o dimpling;
7          Remova o vértice escolhido da lista;
8          Adicione o peso das arestas inseridas ao peso máximo da instância;
9      Atualize o grafo planar maximal e o valor do peso máximo se, atráves de uma operação
       atômica, o valor obtido na instância for maior que o peso máximo;
10 retorna Subgrafo Planar Maximal e peso máximo.
```

Um detalhe importante relacionado às soluções encontradas pelos Algoritmos 1 e 2, e de outros baseados em *dimpling*, é o fato de não garantirem que uma solução ótima

seja obtida. Isso acontece porque esse tipo de abordagem sempre gera soluções com pelo menos um vértice de grau 3. Este fato pode ser visto na Tabela 1, que corresponde a matriz de adjacências de um grafo K_6. O Algoritmo 1 gera a solução da Figura 2(b), com pelo menos um vértice de grau 3. A soma do peso de suas arestas é igual a 23. No entanto, a solução ótima é a mostrada na Figura 2(a), onde todos os vértices possuem grau 4. A soma do peso de suas arestas é igual a 24.

	2	**3**	**4**	**5**	**6**
1	2	2	2	2	1
2		2	2	1	2
3			1	2	2
4				2	2
5					2

Tabela 1. Matriz de adjacências de um grafo K_6.



(a) (b)

Figura 2. (a) solução ótima correspondente ao grafo da Tabela 1 e (b) solução obtida pelo Algoritmo 1.

5. Implementações propostas

Na implementação do Algoritmo 1, foi necessário criar uma matriz para armazenar cada uma das sementes. Ou seja, para instâncias de 85 vértices, foram necessárias $\binom{85}{4}$ combinações, resultando em $2.024.785$ sementes. Como cada semente é composta de 4 vértices, foi preciso alocar $8.099.140$ posições de memória. Como cada semente é construída sequencialmente, é necessário apenas uma única instância de uma matriz $\mathcal{E}$ para armazenar a solução atual, uma matriz $\mathcal{M}'$ para a solução final, uma matriz F para o conjunto de faces, um conjunto S com os vértices que não estão presentes na solução atual e uma fila de prioridade $\mathcal{H}$ para armazenar os pares de vértice e face. Dessa forma, a quantidade de memória necessária para execução é compensada.

O Algoritmo 2 foi mapeado para executar em uma arquitetura *manycore* representada por uma GPU. A preparação dos dados e o recebimento do resultado são descritos na rotina "Pré e Pós-processamento". É necessário alocar uma quantidade de memória que corresponda ao tamanho do grafo $\mathcal{M}$, para o conjunto de sementes $\mathcal{C}$, para F' que armazenará o índice da semente que resultou a solução, para $MaxWeight$ contendo o maior peso obtido dentre os grafos gerados e para uma estrutura $\mathcal{T}$ responsável por armazenar as instâncias de F e $\mathcal{S}$ para cada semente. Em seguida, o grafo e as sementes são copiadas para a memória alocada na GPU. A chamada ao *kernel* é então realizada, passando todos os parâmetros alocados anteriormente. Conforme experimentos realizados, os melhores tempos de execução foram obtidos quando a quantidade de *threads* por bloco foi de $2^{10} = 1024$ e a quantidade de blocos foi igual a $\left\lceil \frac{|\mathcal{C}|}{2^{10}} \right\rceil$.

Na primeira versão proposta, descrita pela Implementação 1, cada *thread* é responsável pelo crescimento de uma semente de forma paralela. Após a execução da barreira de sincronização, uma operação atômica de máximo é realizada, comparando os valores armazenados em $TmpMax$ e $MaxWeight$, de forma a garantir que somente uma *thread*

Pré e Pós-processamento: Passo da CPU

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n \times n}$ com arestas de pesos não negativos.

Saída: Conjunto de faces F', representando um grafo planar maximal e $MaxWeight$ com o peso máximo obtido.

Aloca memória para $MaxWeight$ na GPU;
Aloca memória para F' na GPU;
Aloca memória para a matriz $\mathcal{M}_{n \times n}$ na GPU;
Aloca memória para as sementes $\mathcal{C}$ na GPU;
Aloca memória para uma estrutura $\mathcal{T}$, para cada instância de $\mathcal{C}$, composto de F e $\mathcal{S}$, na GPU;
Cópia da matriz $\mathcal{M}_{n \times n}$ e das sementes $\mathcal{C}$ para a GPU;
$dimBloco \leftarrow 2^{10}$;
$dimGrid \leftarrow \lceil \frac{|\mathcal{C}|}{dimBloco} \rceil$;
Invoca o kernel $< dimGrid, dimBloco >(\mathcal{M}_{n \times n}, \mathcal{C}, \mathcal{T}, MaxWeight, F')$;
Cópia de F' e $MaxWeight$ da GPU para a CPU;
Libera o espaço da memória da GPU ocupado por $\mathcal{M}_{n \times n}$, $\mathcal{C}$, F', $\mathcal{T}$ e $MaxWeight$;
retorna F' e $MaxWeight$.

Implementação 1: Kernel - Algoritmo Paralelo

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n \times n}$ com arestas de pesos não negativos e um conjunto de sementes $\mathcal{C}$.

Saída: Índice do conjunto de faces F' que resultou no grafo planar maximal e $MaxWeight$, com o peso máximo obtido.

$idx \leftarrow$ Índice de uma *thread*;
para todo $idx \leq |\mathcal{C}|$ **faça**
    $\mathcal{T}_{\mathcal{S}_{idx}} \leftarrow V - \mathcal{C}_{idx}$;
    $\mathcal{E} \leftarrow \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$;
    $\mathcal{T}_{F_{idx}} \leftarrow \{\{v_1, v_2, v_3\}, \{v_1, v_2, v_4\}, \{v_1, v_3, v_4\}, \{v_2, v_3, v_4\}\}$;
    $\mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \forall s \in \mathcal{T}_{\mathcal{S}_{idx}}$, calcule o ganho máximo de $\{s, f\}, \forall f \in \mathcal{T}_{F_{idx}}$;
    $\mathcal{T}_{TmpMax_{idx}} \leftarrow \sum \mathcal{E}$;
    enquanto $\mathcal{T}_{\mathcal{S}_{idx}}$ *não estiver vazio* **faça**
        $\{v_i, f_i\} \leftarrow \mathcal{T}_{\mathcal{H}_{idx}} - \max\{\mathcal{T}_{\mathcal{H}_{idx}}\}$, $v_i \in \mathcal{T}_{\mathcal{S}_{idx}}$ e $f_i \in \mathcal{T}_{F_{idx}}$;
        Execute o *dimpling*;
        $\mathcal{T}_{\mathcal{S}_{idx}} \leftarrow \mathcal{T}_{\mathcal{S}_{idx}} - \{v_i\}$;
        $\mathcal{T}_{F_{idx}} \leftarrow (\mathcal{T}_{F_{idx}} - \{f_i\}) \cup \{f_a, f_b, f_c\}$;
        $\mathcal{T}_{TmpMax_{idx}} \leftarrow \mathcal{T}_{TmpMax_{idx}} + G_{v_i,v_x} + G_{v_i,v_y} + G_{v_i,v_z}$, com $v_x, v_y, v_z \in f_i$;
        $\mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \mathcal{T}_{\mathcal{H}_{idx}} \cup \{\forall s \in \mathcal{T}_{\mathcal{S}_{idx}}$, calcule o ganho máximo de $\{s, f_a\}, \{s, f_b\}, \{s, f_c\}\}$;
    Barreira de Sincronização;
    se $\mathcal{T}_{TmpMax_{idx}} \geq MaxWeight$ **então**
        $MaxWeight \leftarrow \mathcal{T}_{TmpMax_{idx}}$;
        $F' \leftarrow idx$;
retorna F' e $MaxWeight$.

realize a escrita em $MaxWeight$ por vez. A matriz $\mathcal{E}$ foi removida para economizar memória, sendo necessário apenas o valor da soma das arestas que compõem a semente. Por isso, caso o valor de $MaxWeight$ seja atualizado, o índice da *thread* que realizou a atualização é armazenado em F'. Dessa forma, é possível reconstruir o grafo resultante a partir do índice da *thread* responsável por construir a melhor solução.

Uma das otimizações realizadas, descrita na Implementação 2 consiste em utilizar um espaço alocado na memória compartilhada da GPU para armazenar o grafo. Esta memória, embora pequena, é mais rápida do que a memória global. Dessa forma, ao invés de acessar a memória global, o acesso é realizado na memória compartilhada, acelerando o trabalho de todas as *threads*.

Outro detalhe está na presença de um acesso aglutinado no conjunto de faces, uma vez que este está armazenado na GPU e será utilizado com muita frequência. Tal organização de dados consiste em colocar todo um conjunto de faces de um determinado índice sequencialmente. Isto é, em cada instrução de *warp*, todas as *threads* buscarão o mesmo índice do conjunto de faces. Dessa forma, diminui-se a latência de acesso à memória global.

A última melhoria, correspondendo a uma terceira implementação, consiste em dividir a tarefa entre várias GPUs. Cada GPU fica responsável por uma fatia das sementes, de forma que quanto mais GPUs estejam disponíveis, maior a distribuição e mais rápido a execução. Após concluída a etapa de execução do *kernel*, os valores de F' e $MaxWeight$ são copiados para a memória principal e cada valor de $MaxWeight$ é comparado com os obtidos em cada GPU, sendo o maior deles a resposta. A matriz resultante é então reconstruída atráves do F' que resultou no maior $MaxWeight$.

Implementação 2: Kernel - Algoritmo Paralelo com Memória Compartilhada

Entrada: Grafo $G = (V, E)$, representado pela matriz de adjacências $\mathcal{M}_{n \times n}$ com arestas de pesos não negativos e um conjunto de sementes $\mathcal{C}$.

Saída: Índice do conjunto de faces F' que resultou no grafo planar maximal e $MaxWeight$, com o peso máximo obtido.

1 $idx \leftarrow$ Índice de uma *thread*;
2 Aloca G' na memória compartilhada;
3 $G' \leftarrow G$;
4 Barreira de Sincronização;
5 **para todo** $idx \leq |\mathcal{C}|$ **faça**
6 $\quad \mathcal{T}_{\mathcal{S}_{idx}} \leftarrow V - \mathcal{C}_{idx}$;
7 $\quad \mathcal{E} \leftarrow \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$;
8 $\quad \mathcal{T}_{F_{idx}} \leftarrow \{\{v_1, v_2, v_3\}, \{v_1, v_2, v_4\}, \{v_1, v_3, v_4\}, \{v_2, v_3, v_4\}\}$;
9 $\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \forall s \in \mathcal{T}_{\mathcal{S}_{idx}}$, calcule o ganho máximo de $\{s, f\}, \forall f \in \mathcal{T}_{F_{idx}}$;
10 $\quad \mathcal{T}_{TmpMax_{idx}} \leftarrow \sum \mathcal{E}$;
11 $\quad$ **enquanto** $\mathcal{T}_{\mathcal{S}_{idx}}$ *não estiver vazio* **faça**
12 $\quad\quad \{v_i, f_i\} \leftarrow \mathcal{T}_{\mathcal{H}_{idx}} - \max\{\mathcal{T}_{\mathcal{H}_{idx}}\}$, $v_i \in \mathcal{T}_{\mathcal{S}_{idx}}$ e $f_i \in \mathcal{T}_{F_{idx}}$;
13 $\quad\quad$ Execute o *dimpling*;
14 $\quad\quad \mathcal{T}_{\mathcal{S}_{idx}} \leftarrow \mathcal{T}_{\mathcal{S}_{idx}} - \{v_i\}$;
15 $\quad\quad \mathcal{T}_{F_{idx}} \leftarrow (\mathcal{T}_{F_{idx}} - \{f_i\}) \cup \{f_a, f_b, f_c\}$;
16 $\quad\quad \mathcal{T}_{TmpMax_{idx}} \leftarrow \mathcal{T}_{TmpMax_{idx}} + G'_{v_i,v_x} + G'_{v_i,v_y} + G'_{v_i,v_z}$, com $v_x, v_y, v_z \in f_i$;
17 $\quad\quad \mathcal{T}_{\mathcal{H}_{idx}} \leftarrow \mathcal{T}_{\mathcal{H}_{idx}} \cup \{\forall s \in \mathcal{T}_{\mathcal{S}_{idx}}$, calcule o ganho máximo de $\{s, f_a\}, \{s, f_b\}, \{s, f_c\}\}$;
18 $\quad$ Barreira de Sincronização;
19 $\quad$ **se** $\mathcal{T}_{TmpMax_{idx}} \geq MaxWeight$ **então**
20 $\quad\quad MaxWeight \leftarrow \mathcal{T}_{TmpMax_{idx}}$;
21 $\quad\quad F' \leftarrow idx$;
22 **retorna** F' e $MaxWeight$.

6. Experimentos realizados

6.1. Configuração dos experimentos

Os testes computacionais foram realizados com um conjunto de 16 grafos completos, com pesos associados às arestas. O número de vértices variou de 10 até 85, em múltiplos de 5, e os pesos foram gerados de forma aleatória no intervalo $[0, 199]$.

Os experimentos foram conduzidos em um computador com processador Intel Xeon E5-2620 2GHz, 16GB de ECC RAM, sistema operacional CentOS 7.2.1511 64-bits, com uma e quatro (4) GeForce Zotac Nvidia GTX Titan Black, com 6GB de RAM e 2.880 CUDA *cores* cada. Foram usados os compiladores GCC 4.9.2 e CUDA Toolkit 7.5 para os códigos da CPU e GPU, respectivamente, com a *flag* $O3$.

Os tempos calculados levam em consideração a execução do programa como um todo, incluindo entrada e saída, e não apenas o cálculo do subgrafo planar de maior peso. Cada teste foi executado 10 vezes para cada instância de cada versão utilizada. Medidos os tempos de execução das 10 execuções, foi calculado o tempo médio com intervalos de confiança de 95%.

6.2. Resultados e análise

Na Figura 2, é feito um comparativo entre a versão sequencial e a versão paralela mais eficiente. A versão sequencial apresenta uma curva de alto crescimento com tempo próximo a 30 minutos de execução, enquanto a versão multi-GPU, com 4 GPUs, permanece praticamente inalterada, resolvendo em apenas 17 segundos a maior das instâncias (85 vértices). A Figura 3 mostra um comparativo de tempo entre as implementações mencionadas neste artigo. É notável o desempenho da implementação multi-GPU, que apresenta um aumento de poucos segundos de uma instância para outra, enquanto a versão mais simples tem um aumento de quase 3 minutos de 75 a 85 vértices.

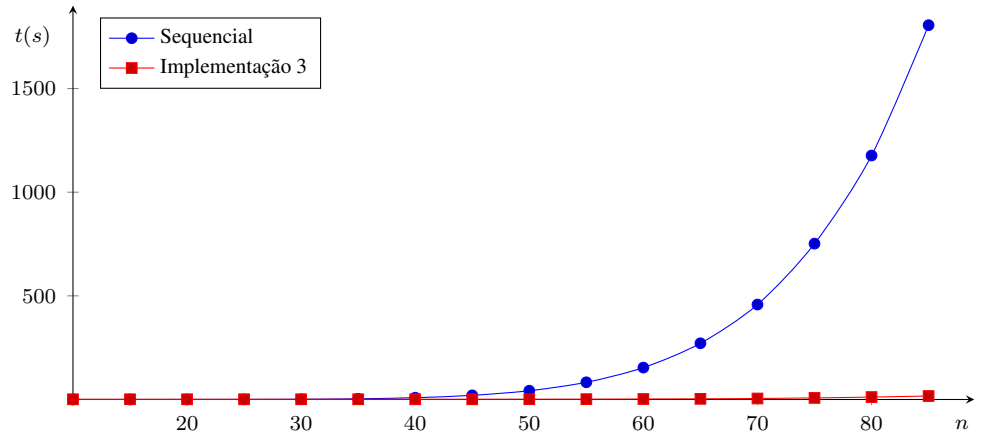


Figura 2. Comparativo de tempo entre a implementação sequencial e a implementação paralela multi-GPU.

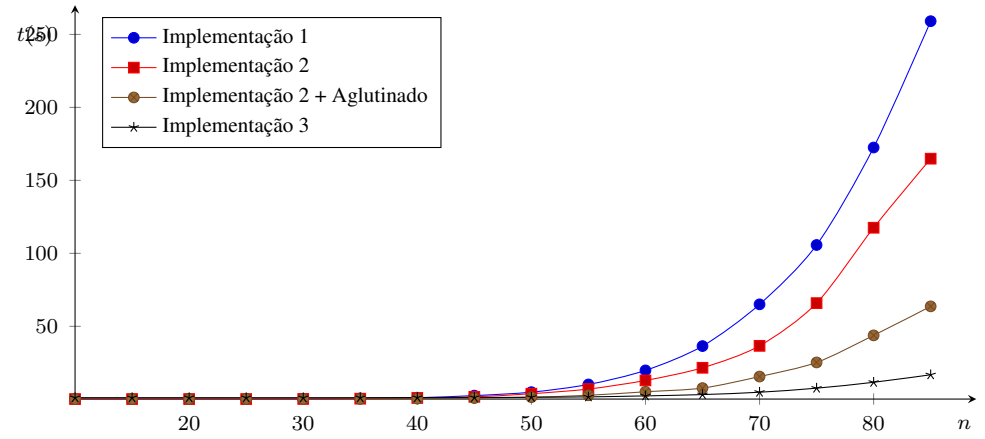


Figura 3. Comparativo de tempo entre as versões paralelas e suas melhorias.

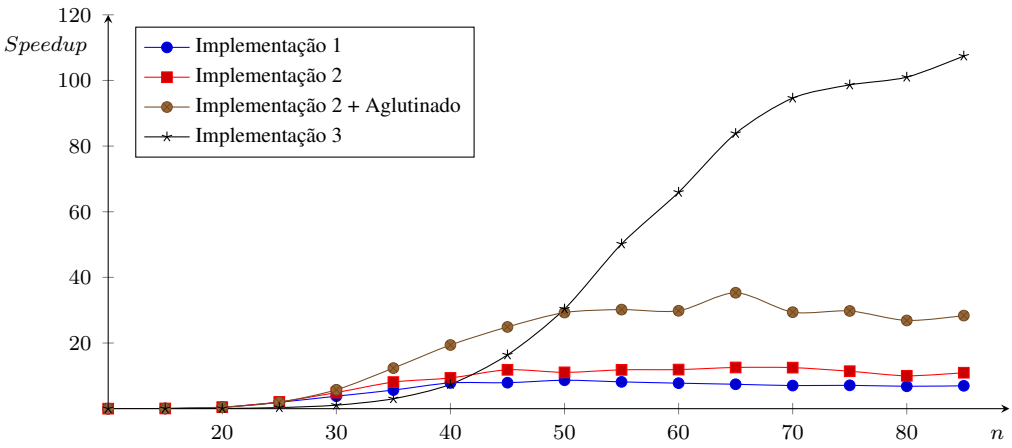


Figura 4. *Speedup* das versões paralelas em relação ao algoritmo sequencial.

Tabela 2. *Speedups* em relação ao algoritmo sequencial, em função de n.

n	Implementação 1	Implementação 2	Implementação 2 + Aglutinado	Implementação 3
10	0,006	0,006	0,006	0,001
15	0,059	0,059	0,055	0,009
20	0,444	0,471	0,442	0,072
25	1,947	2,056	1,979	0,335
30	3,806	4,917	5,576	1,065
35	5,667	8,095	12,364	3,030
40	7,847	9,366	19,399	7,418
45	7,913	11,846	24,900	16,386
50	8,665	11,063	29,281	30,368
55	8,165	11,829	30,185	50,190
60	7,769	11,913	29,828	65,928
65	7,437	12,553	35,314	83,846
70	7,040	12,503	29,398	94,624
75	7,112	11,411	29,770	98,642
80	6,822	10,013	26,872	101,004
85	6,970	10,953	28,356	107,447

É possível perceber que a Implementação 2 começa a obter *speedup* após instâncias de 25 vértices, conforme apresentado na Figura 4 e Tabela 2. A curva de *speedup* projeta um forte crescimento para instâncias de até 65 vértices. A partir de 70 vértices, o *speedup* decai um pouco e estabiliza-se. Para a versão multi-GPU, a curva de *speedup* tem forte crescimento até 85 vértices. Não foram utilizados grafos maiores devido à quantidade de memória necessária para instâncias acima de 85 vértices ser superior a 6GB, inviabilizando os testes usando somente uma GPU. Contudo, a versão multi-GPU é altamente escalável, devido a possibilidade de particionamento das sementes entre mais GPUs, sendo assim possível realizar testes com instâncias maiores.

7. Conclusões e considerações finais

A solução aproximada para o problema WMPG apresentada neste trabalho considera várias sementes (tetraedros) e assim permite que o espaço de busca desta heurística seja totalmente analisado. Esta abordagem também permitiu uma extensão paralela que mostrou-se capaz de trabalhar de maneira eficiente com instâncias maiores do problema. Além da implementação sequencial e paralela, foram apresentadas otimizações importantes que contribuiram para uma solução escalável em uma plataforma multi-GPU.

Um ponto importante é que a solução proposta garante que o grafo resultante tenha a soma do peso de suas arestas maior ou igual que as propostas de [Foulds and Robinson 1978, Massara et al. 2015], uma vez que são testadas todas as possibilidades de tetraedros. Além disso, o uso de paralelismo permite que instâncias maiores, de utilidade prática, sejam processadas em um tempo razoável. Por exemplo, o processamento de um grafo com 85 vértices usando 4 GPUs leva apenas 17 segundos, enquanto a implementação sequencial leva cerca de meia hora. No entanto, testar todas as possibilidades é um fator limitante para a solução do problema, devido à enorme quantidade de memória necessária para sua execução.

Referências

Foulds, L. R. (1992). *Graph Theory Applications*. Springer, New York.

Foulds, L. R. and Robinson, D. F. (1976). A Strategy for Solving The Plant Layout Problem. *Journal of the Operational Research Society*, 27(4):845–855.

Foulds, L. R. and Robinson, D. F. (1978). Graph theoretic heuristics for the plant layout problem. *International Journal of Production Research*, 16(1):27–37.

Foulds, L. R. and Robinson, D. F. (1979). Construction Properties of Combinatorial Deltahedra. *Discrete Applied Mathematics*, 1:75–87.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York.

Kirk, D. B. and Hwu, W.-M. W. (2011). *Programando Para Processadores Paralelos: Uma Abordagem Prática à Programação de GPU*. Elsevier.

Massara, G. P., Di Matteo, T., and Aste, T. (2015). Network Filtering for Big Data: Triangulated Maximally Filtered Graph. *CoRR*, pages 1–18.

Tumminello, M., Aste, T., Di Matteo, T., and Mantegna, R. N. (2005). A tool for filtering information in complex systems. *Proceedings of the National Academy of Sciences of the United States of America*, 102(30):10421–10426.