UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

ALLAN FELIPE LEMES

PARALELIZAÇÃO DO PROBLEMA DE GRAPH MATCHING PARA GRAFOS EXATOS UTILIZANDO CUDA

CAXIAS DO SUL

2020

ALLAN FELIPE LEMES

PARALELIZAÇÃO DO PROBLEMA DE GRAPH MATCHING PARA GRAFOS EXATOS UTILIZANDO CUDA

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. Andre Luis Martinotto

CAXIAS DO SUL

2020

ALLAN FELIPE LEMES

Paralelização do problema de graph matching para grafos exatos utilizando CUDA

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em 03/12/2020

BANCA EXAMINADORA

Prof. Dr. Andre Luis Martinotto
Universidade de Caxias do Sul - UCS

Prof. Me. Alexandre Erasmo Krohn Nascimento
Universidade de Caxias do Sul - UCS

Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

RESUMO

O problema de *graph matching* aplicado a grafos exatos consiste em achar a similaridade entre dois grafos, definindo se estes possuem uma similaridade por completo (grafos isomorfos) ou se existe algum subgrafo similar (subgrafos isomorfos). Ainda não foi determinado se esse problema pertence à classe P ou NP-Completo, não existindo algoritmos polinomiais para a solução do mesmo. Desta forma, frequentemente são utilizadas heurísticas para a solução, sendo que entre essas destacam-se o VF2. Dentro deste contexto, neste trabalho foi desenvolvida uma implementação paralela da heurística VF2. Essa foi desenvolvida utilizando a biblioteca CUDA, de forma a explorar o paralelismo em GPUs. Para os testes, foram gerados grafos com três tamanhos distintos, sendo eles: grafos pequenos de 4 vértices, grafos médios com 8 vértices e grafos grandes com 12 vértices. A partir dos testes realizados verificou-se que versão paralela, utilizando grafos pequenos, médios e grande é, respectivamente, 1,66, 1,44 e 1,38 mais rápida que a versão sequencial.

Palavras-chaves: *Graph matching*. CUDA. Paralelização

LISTA DE FIGURAS

Figura 1 – Representação do corpo humano através de grafos 17
Figura 2 – Exemplo de uma função injetora 21
Figura 3 – Exemplo de uma função sobrejetiva 22
Figura 4 – Exemplo de uma relação de bijeção 22
Figura 5 – Exemplo de grafos isomorfos e não isomorfos 23
Figura 6 – Exemplo de grafos isomorfos e não isomorfos 24
Figura 7 – Representação dos tipos de problemas do *graph match* dividido em duas categorias: Grafos exatos e Grafos inexatos 25
Figura 8 – Exemplo de como é criado a matriz (M^0) 27
Figura 9 – Exemplo de como são criados os outros candidatos 28
Figura 10 – Exemplo de como é feito o refinamento 28
Figura 11 – Exemplo de como é gerado o próximo candidato 29
Figura 12 – Exemplo de quando o algoritmo chega ao último nível, caracterizando um isomorfismos em subgrafo 29
Figura 13 – Grafos *G* e *H* 30
Figura 14 – Estado atual: MG = {1,2} e MH = {1,2} 31
Figura 15 – Representação de uma relação de não adjacência 31
Figura 16 – Estado Atual: MG = {1,2} e MH = {1,3} 32
Figura 17 – Estado Atual: MG = {} e MH = {} 32
Figura 18 – Estado final: MG = {1,2,3} e MH = {2,1,3} 33
Figura 19 – Exemplo da canonização em dois grafos 33
Figura 20 – Aplicado uma função de rotação em 180° graus sobre os vértices$\{0, 1\}$ e $\{2, 3\}$ 34
Figura 21 – Partição Inicial do Algoritmo Nauty 35
Figura 22 – Refinamento considerando os graus dos vértices 35
Figura 23 – Exemplo de um refinamento 36
Figura 24 – Árvore de busca final 37
Figura 25 – Um possível candidato para isomorfismo. 38
Figura 26 – Comparativo entre performance das heurísticas 39
Figura 27 – Representação da arquitetura SIMD 43
Figura 28 – Comparação da execução de instruções entre um processador escalar e um processador vetorial 43
Figura 29 – Processador vetorial do tipo registrador-registrador 44
Figura 30 – Imagem representando um processador matricial 45
Figura 31 – Comparativo de poder computacional entre CPU e GPU 46
Figura 32 – Comparativo entre uma arquitetura CPU com uma GPU 46
Figura 33 – Arquitetura Fermi da NVIDIA 47

Figura 34 – Unidade MS da NVIDIA 48
Figura 35 – Exemplo da representação de um grafo 54
Figura 36 – NVIDIA GTX 1650 57
Figura 37 – Tempo de Execução do Método VF2 58
Figura 38 – Uso de memória na GPU 59

LISTA DE TABELAS

Tabela 1 – Hierarquia de memória em uma GPU 50
Tabela 2 – Tamanho dos grafos 58

LISTA DE ALGORITMOS

Algoritmo 1 Algoritmo de soma de matriz, paralelizado 51
Algoritmo 2 Algoritmo para somar dois vetores em paralelo 52
Algoritmo 3 Algoritmo ilustrando a criação de um bloco contendo 10 *threads* 52
Algoritmo 4 Definição das estruturas utilizadas 53
Algoritmo 5 Pseudocódigo da versão sequencial 54
Algoritmo 6 Alocação e transferência dos grafos para a memória da GPU 55
Algoritmo 7 Pseudocódigo da versão paralelizada utilizando CUDA 56
Algoritmo 8 Algoritmo ilustrando a transferência de dados entre GPU e CPU 56

LISTA DE ABREVIATURAS E SIGLAS

GPU	*Graphics Processing Unit*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
P	*Polynomial time*
NP	*Nondeterministic Polynomial time*

LISTA DE SÍMBOLOS

T	Transposição
$\Rightarrow$	implica em
α	Alfa
β	Beta

SUMÁRIO

1 INTRODUÇÃO . . . 17
1.1 OBJETIVO DO TRABALHO . . . 18
1.2 ESTRUTURA DO TRABALHO . . . 19

2 *GRAPH MATCHING* PARA GRAFOS EXATOS . . . 21
2.1 Verificação de Isomorfismo entre Grafos . . . 23
2.2 Heurísticas para Grafos exatos . . . 25
2.2.1 Algoritmo Ullmann . . . 26
2.2.2 Algoritmo VF2 . . . 30
2.2.3 Nauty (ou *Practical Graph Isomorphism*) . . . 33
2.3 Definição da Heurística . . . 38

3 UNIDADES DE PROCESSAMENTO GRÁFICO . . . 41
3.1 Arquitetura SIMD . . . 42
3.2 Unidades de Processamento Gráfico . . . 45
3.3 ARQUITETURA DE UMA GPU . . . 47
3.3.1 Hierarquia de memórias em uma GPU . . . 49
3.3.2 Estrutura de um programa em CUDA . . . 50

4 IMPLEMENTAÇÃO DESENVOLVIDA . . . 53
4.1 Estruturas de Dados . . . 53
4.2 Paralelização do Método VF2 . . . 54
4.3 Testes Realizados e Resultados Obtidos . . . 57
4.3.1 Grafos utilizados para testes . . . 57
4.3.2 Resultados obtidos . . . 58

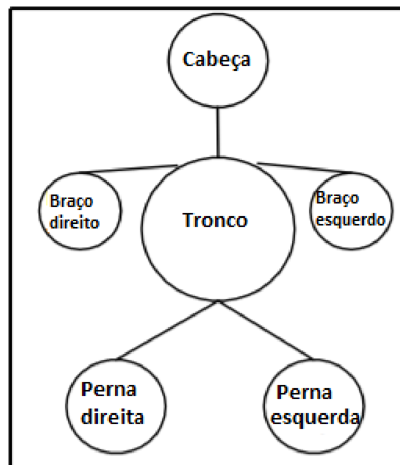
5 CONSIDERAÇÕES FINAIS . . . 61
5.1 Sugestão de Trabalhos Futuros . . . 62

REFERÊNCIAS . . . 63

1 INTRODUÇÃO

O estudo dos grafos é um ramo da matemática muito utilizado em diversas áreas do conhecimento, tais como química, biologia molecular, processamento de imagens, reconhecimento de padrões, entre outras (BENGOETXEA, 2002). Um exemplo de aplicações nestas áreas, são problemas onde imagens e/ou objetos precisam ser processados e regiões da imagem precisam ser identificadas (JüTTNER; MADARASI, 2018). A partir da utilização de grafos é possível representar as regiões de uma imagem. Como pode ser observado na Figura 1, o corpo humano pode ser representado por um grafo, onde as partes do corpo são representadas por vértices e são ligadas umas às outras por meio de arestas.

Figura 1 – Representação do corpo humano através de grafos



Fonte: BENGOETXEA (2002), adaptado

O problema de *graph matching* é uma área de estudo que objetiva encontrar a similaridade entre grafos a partir de um modelo. Por exemplo, considerando que o grafo da Figura 1 representa o corpo humano, tem-se que todos os grafos que representam o corpo humano devem apresentar essa mesma estrutura. Já um grafo que representa um animal, por exemplo, apresentará uma estrutura diferente, ou seja, os dois grafos não são isomorfos.[1].

O grafo que representa um animal, pode ser caracterizado como um grafo não exato (*inexact graph matching*), uma vez que esse não possui nenhuma relação de isomorfismo com o modelo. Porém, quando dois grafos possuem ao menos algum isomorfismo (seja do grafo inteiro ou de algum subgrafo), ele é considerado como sendo um grafo exato (*exact graph matching*) (RIESEN; JIANG; BUNKE, 2010).

[1] Dois grafos G e H são dito isomorfos se existir uma bijeção entre os vértices de G e H que preserva a relação de adjacência entre vértices e arestas. Ou seja, se é possível obter o grafo H a partir de uma nova rotulação dos vértices de G (KöBLER UWE SCHöNING, 1993)

A complexidade do problema de *graph matching* ainda é uma questão em aberto, uma vez que ainda não foi possível provar que esse problema é da classe NP-Completo ou P (BENGOETXEA, 2002). Alguns algoritmos foram desenvolvidos apresentando um tempo de execução polinomial, porém esses algoritmos são específicos para determinados tipos de grafos, tal como os grafos planares (HOPCROFT; WONG, 1974). Até o momento, não existe nenhum algoritmo polinomial conhecido para uso geral (BENGOETXEA, 2002).

A solução do problema de *graph matching* apresenta limitações devido principalmente à complexidade dos métodos de solução. De fato, a solução deste problema apresenta um alto custo computacional principalmente para grafos com um grande número de vértices e arestas (SUMSI, 2008). Uma alternativa para reduzir o tempo necessário para a solução desse problema consiste na utilização de heurísticas de solução [2], bem como o uso de programação paralela (RIESEN; JIANG; BUNKE, 2010).

Dentro deste contexto, neste trabalho foi desenvolvida uma implementação paralela da heurística VF2 para a solução do problema de *graph matching*. Essa implementação foi desenvolvida de forma a explorar o paralelismo em arquiteturas de GPU (*Graphics Processing Unity*) (ESPARRACHIARI; GOMES, 2008). Para desenvolvimento da mesma foi utilizada a biblioteca CUDA (*Compute Unified Device Architecture*) da NVIDIA (CUDA, 2020).

1.1 OBJETIVO DO TRABALHO

O objetivo principal deste trabalho consistiu no desenvolvimento de uma implementação paralela para a solução do problema de *graph matching* para grafos exatos. Essa solução foi desenvolvida para ser executada em uma plataforma de GPU, utilizando a biblioteca CUDA da NVIDIA. Para que o objetivo principal deste trabalho fosse atingido, os seguintes objetivos específicos foram realizados:

- Definição da heurística a ser utilizada para a solução do problema de *graph matching* para grafos exatos.
- Implementação de uma versão sequencial utilizando a heurística definida.
- Paralelização da implementação desenvolvida utilizando a biblioteca CUDA da NVIDIA.
- Realização de testes comparativos entre o tempo execução da versão sequencial e da versão paralela.

[2] Heurísticas são algoritmos que não garantem encontrar a solução ótima de um problema, mas são capazes de retornar uma solução de qualidade em um tempo adequado.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está organizado da seguinte forma:

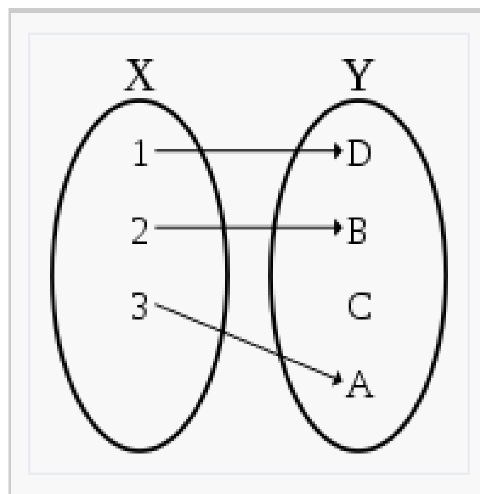
- No Capítulo 1 é feita uma breve introdução do trabalho, apresentando suas motivações e objetivos.
- O Capítulo 2 apresenta o problema de *graph matching* para grafos exatos e algumas heurísticas que podem ser utilizadas para a solução do problema.
- No Capítulo 3 é feita uma breve descrição de arquiteturas paralelas, bem como uma introdução a arquitetura e a programação de GPUs (*Graphics Processing Unit*).
- No Capítulo 4 é descrita a implementação que foi desenvolvida neste trabalho. Além disso, são apresentados os testes realizados e os resultados obtidos.
- Por fim, no Capítulo 5 são apresentadas as considerações finais do trabalho e sugestões para trabalhos futuros.

2 *GRAPH MATCHING* PARA GRAFOS EXATOS

O problema de *graph match*, mais precisamente na categoria de grafos exatos, pode ser definido como em encontrar a similaridade entre dois grafos, ou seja, verificar se eles são isomorfos ou não. Dois grafos $G = (V_1, A_1)$ e $H = (V_2, A_2)$ são ditos isomorfos se existir uma bijeção [1] entre os vértices de G e H que preserva a relação de adjacência entre vértices e arestas. Isto é, se é possível obter o grafo H a partir de uma nova rotulação dos vértices de G.

Uma relação de bijeção é uma relação um para um de um conjunto para outro, de tal forma que, um elemento do conjunto Y deve obrigatoriamente estar vinculada a um único elemento do conjunto X. Podemos afirmar que, uma relação bijectiva é uma relação que possui tanto uma função injectiva quanto uma função sobrejetiva. Por exemplo, os conjuntos apresentados na Figura 2 não possuem uma relação bijeção, uma vez que o elemento C, do conjunto Y, não possui relacionamento com nenhum elemento do conjunto X. Pode-se afirmar ainda que essa função não é sobrejetiva, isso porque, para ser uma função sobrejetiva, todos os elementos de um conjunto devem possuir ao menos uma relação com elementos do outro conjunto. Também pode-se afirmar que essa função é injetora, pois não temos nenhum elemento com mais de uma relação com elementos do outro conjunto. Ou seja, de forma resumida, o exemplo da Figura 2 corresponde a uma função injetora e não sobrejetiva, portanto não bijectiva.

Figura 2 – Exemplo de uma função injetora

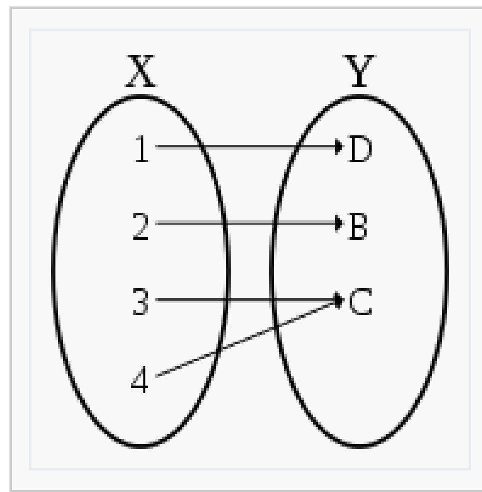


Fonte: CHAPEL (2016)

[1] Bijeção de um conjunto A para um conjunto B é uma correspondência biunívoca entre A e B, isto é, a cada elemento de A corresponde sempre um único elemento de B e reciprocamente.

Na Figura 3, tem-se um outro exemplo de uma relação de não bijeção, isso porque, apesar desta apresentar uma função sobrejetiva, essa não possui uma função injetora. De fato, pois para ser uma função injetora, os elementos de um conjunto não podem estar vinculados a mais de um elemento do outro conjunto. Portanto, essa função é sobrejetiva, porém não é injetora, logo não apresentando uma relação de bijeção.

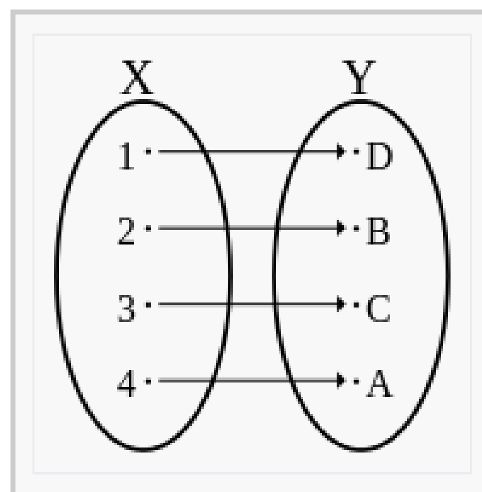
Figura 3 – Exemplo de uma função sobrejetiva



Fonte: CHAPEL (2016)

Já na Figura 4 tem-se um exemplo de uma relação de bijeção, uma vez que os elementos do conjunto X e Y apresentam uma relação de um para um, não apresentando nenhum elemento sem vínculo entre os conjuntos (função injetora) e nenhum elemento está vinculado a mais de um elemento (função sobrejetiva).

Figura 4 – Exemplo de uma relação de bijeção



Fonte: CHAPEL (2016)

2.1 VERIFICAÇÃO DE ISOMORFISMO ENTRE GRAFOS

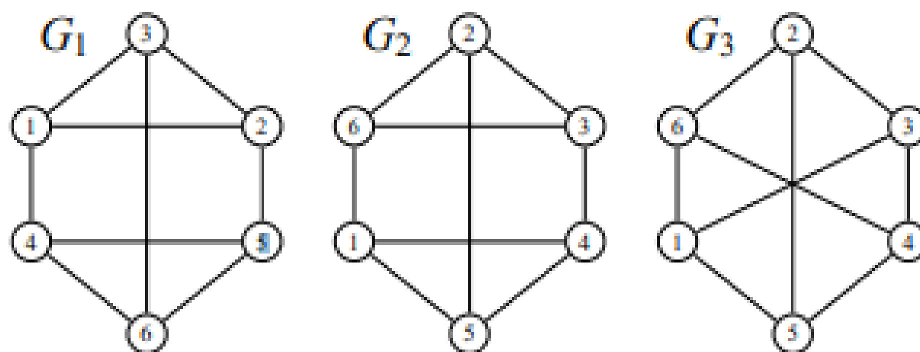
A maior dificuldade encontrada na verificação de isomorfismo entre grafos deve-se em muito à estrutura dos grafos e ao tamanho desses. Desta forma, a abordagem mais usual consiste em primeiramente verificar se os grafos não são isomorfos. Na literatura, essa abordagem muitas vezes é chamada de *Graph Invariants* ou ainda *Graph Properties* [2].

Para determinar que dois grafos não são isomorfos pode-se verificar algumas das suas propriedades e caso alguma dessas não sejam iguais, pode-se determinar que eles não isomorfos. Para que dois grafos sejam isomorfos eles devem possuir as seguintes propriedades:

- Os grafos devem possuir a mesma quantidade de vértices;
- Os grafos devem possuir a mesma quantidade de arestas;
- Os grafos devem possuir a mesma estrutura. Por exemplo, os dois grafos devem ser planares, os dois grafos devem ser bipartidos, etc.;
- Os grafos devem possuir o mesmo grau;

Pode-se facilmente verificar que o grafo G_3 não é isomorfo dos grafos G_1 e G_2 (Figura 5). Apesar do grafo G_3 possuir a mesma quantidade de vértices e arestas, ele não apresenta a mesma estrutura de G_1 e G_2, isto porque, os grafos G_1 e G_2 são grafos planares [3] e o grafo G_3 não é.

Figura 5 – Exemplo de grafos isomorfos e não isomorfos



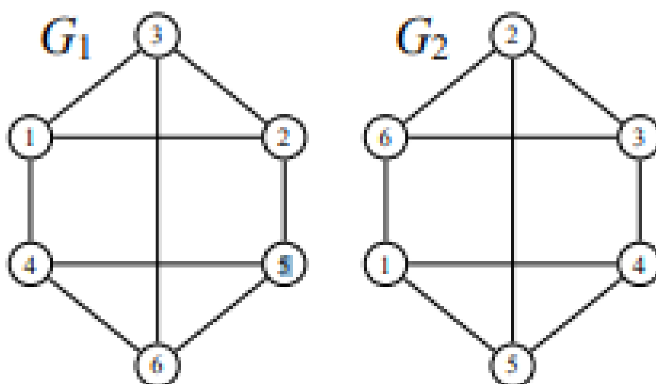
Fonte: RODRIGUES (2011)

[2] Graph Invariants ou Graph Properties são propriedades dos grafos que dependem apenas da sua estrutura e não da sua representação. Um exemplo é a quantidade de vértices.

[3] Um grafo é planar se puder ser desenhado no plano sem que haja arestas que se cruzem.

Já na Figura 6 tem-se um exemplo que apresenta dois grafos G_1 e G_2 que são isomorfos. De fato, estes grafos apresentam a mesma estrutura e através de uma rotulação dos vértices do grafo G_1 pode-se obter o grafo G_2. Por exemplo, o vértice 1 de G_1 pode ser rotulado como vértice 6, o vértice 2 de G_1 como vértice 3, e assim por diante, obtendo uma função bijetora {(1,6),(2,3),(3,2),(4,1),(5,4),(6,5)}.

Figura 6 – Exemplo de grafos isomorfos e não isomorfos



Fonte: RODRIGUES (2011)

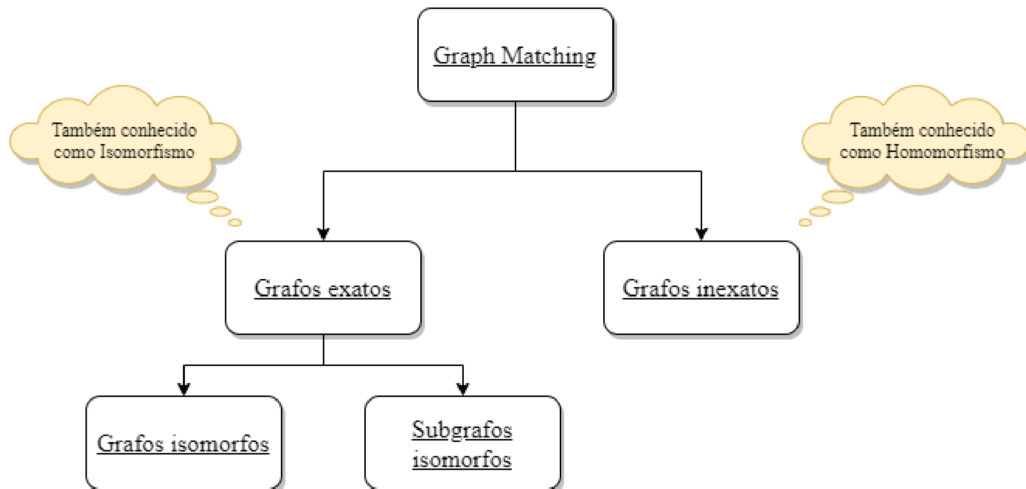
No que se refere ao problema de encontrar a similaridade entre dois grafos tem-se ainda a categoria de grafos inexatos, também chamada de homomorfismo entre grafos. Neste categoria tem-se, por exemplo, dois grafos *G* e *H* que não são isomorfos, por não possuírem o mesmo número de vértices. Assim, procura-se encontrar a melhor correspondência entre eles (BENGOETXEA, 2002). Dentro deste contexto, a categoria de grafos exatos, bem como a categoria de grafos inexatos podem ser subdivididas em (Figura 7):

- Grafos Isomorfos: dois grafos *G* e *H* são ditos isomorfos se existe uma relação de bijeção entre eles, preservando a relação de adjacências e de não adjacências.

- Subgrafos Isomorfos: neste caso, é possível encontrar um isomorfismo entre um grafo e um subgrafo. Por exemplo, todos os vértices de um grafo *G* podem ser mapeados considerando os vértices de um subgrafo de *H*.

- Grafos Inexatos (Homomorfismo): ocorre quando não é possível encontrar uma relação de bijeção entre os grafos *G* e *H*. Neste caso, o objetivo consiste em encontrar a melhor correspondência de não bijeção entre os grafos *G* e *H*, isto é, encontrar uma relação entre *G* e *H* que preserva as adjacências, porém, essa relação não precisa ser bijectiva (injetiva e sobrejetiva).

Figura 7 – Representação dos tipos de problemas do *graph match* dividido em duas categorias: Grafos exatos e Grafos inexatos



Fonte: o autor (2020)

Destaca-se que neste trabalho não serão abordados estratégias para verificação de homomorfismo em grafos inexatos. De fato, todos os algoritmos apresentados nas próximas seções só podem ser utilizados para a verificação de isomorfismos entre grafos e subgrafos.

2.2 HEURÍSTICAS PARA GRAFOS EXATOS

Uma abordagem simples para verificar se dois grafos, com n vértices, são isomorfos consiste em testar todas as possíveis combinações. Embora essa abordagem seja simples, ela é impraticável para grafos com um grande número de vértices, uma vez que o número de testes a serem realizados é igual a $n!$ (CARLETTI, 2016). Desta forma, a abordagem mais utilizada consiste na utilização de alguma heurística de solução. As heurísticas reduzem significativamente o tempo de execução, porém, não garantem alcançar a solução ótima do problema (PEARL, 1984).

Na literatura tem-se diversas heurísticas que podem ser utilizados para verificar o isomorfismo entre grafos, sendo que algumas delas podem ser utilizadas somente para a verificação do isomorfismo completo (grafo-grafo) e outras que podem ser utilizados para verificar o isomorfismo parcial (isomorfismo em subgrafos). Entretanto, todas essas heurísticas utilizam basicamente duas abordagens: a direta e a rotulação canônica.

Em uma abordagem direta, o algoritmo busca verificar o isomorfismo de dois grafos utilizando uma técnica de *backtracking* [4]. Algumas dessas abordagens utilizam ainda técnicas de refinamento para diminuir o espaço de busca, tornando assim o algoritmo mais eficiente (LÓPEZ-PRESA; ANTA, 2009). Nessa categoria destacam-se os algoritmos de Ullmann (ULLMANN, 1976) e VF2 (JüTTNER; MADARASI, 2018).

[4] *Backtracking* é uma técnica de busca sistemática de todas as soluções, garantindo que uma solução será encontrada (COMBA, 2018)

Uma outra abordagem muito utilizada é a da rotulagem canônica, que consiste na aplicação de uma função *C()* nos grafos G e C, retornando uma nova rotulação nos grafos G e C de tal modo que *C(G) = C(H)* (LÓPEZ-PRESA; ANTA, 2009). Isto é, a partir da aplicação de uma função *C()* no grafo G é gerado um novo grafo que é igual ao grafo H. O algoritmo *Practical Graph Isomorphism* (PGI) (MCKAY; PIPERNO, 2013) é o principal exemplo de uma abordagem canônica, porém, ele é mais conhecido na literatura por *Nauty* pela sua utilização na biblioteca de mesmo nome (MCKAY; PIPERNO, 2013).

Nas próximas seções serão descritos os algoritmos de Ullmann, VF2 e *Nauty*. Para a descrição dos mesmos, será utilizada a seguinte notação: o grafo modelo será definido como *G* = *<V,E>*, onde V representa o conjunto de vértices e o E o conjunto de arestas; e o grafo alvo sendo definido como *H* = *<V,E>*.

2.2.1 Algoritmo Ullmann

Uma das primeiras abordagens desenvolvida para verificar o isomorfismo de grafos foi apresentada por Ullmann, em 1976 (ULLMANN, 1976). O grande diferencial do algoritmo de Ullmann em relação ao algoritmos já existentes até então era a utilização de uma técnica de refinamento. De fato, ao contrário dos algoritmos já existentes que faziam uma busca exaustiva de todas as possibilidades, o algoritmo de Ullmann reduz o espaço de busca removendo alguns dos candidatos previamente. Esse algoritmo apesar de ser antigo (1976) ainda é muito utilizado para encontrar isomorfismo em grafos exatos devido principalmente a sua generalidade e efetividade (CORDELLA *et al.*, 2001).

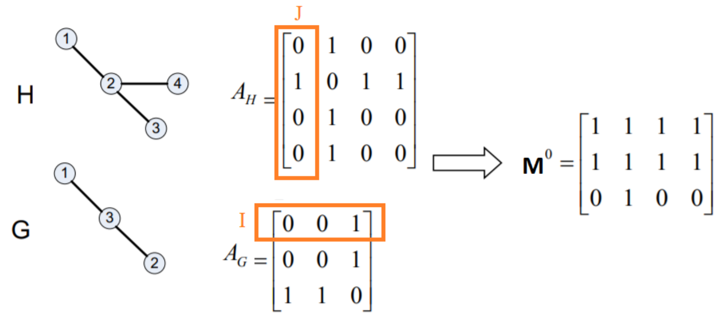
A ideia central do algoritmo de Ullmann é criar, inicialmente, todo o espectro de possíveis candidatos ao isomorfismo, sendo que cada um destes candidatos é representado por um folha em uma árvore de busca. Após, esses nodos são refinados utilizando um algoritmo de busca em profundidade *(Depth-First Search - DFS)* [5], com o objetivo de verificar o isomorfismo ou concluir que este candidato não é apto ao isomorfismo. Neste caso, esse candidato é descartado e todos os candidatos subsequentes a ele também são descartados. Cada um destes candidatos é representado por uma matriz (M^T), sendo que a raiz da árvore, é a matriz (M^0).

A cada novo nodo na árvore, uma linha da matriz é refinada com o objetivo de obter, ao fim de todos os refinamentos, uma matriz de permutação. Uma característica importante da matriz de permutação é que ela deve possuir exatamente uma entrada um (1) em cada linha e coluna, e os demais elementos são todos zeros (0).

[5] Busca em profundidade ou *Depth-First Search - DFS* é um algoritmo de busca em uma árvore ou grafo que, a partir do ramo raiz, explora todo o ramo abaixo dele em uma determinada ordem até o seu fim fazendo sucessivas chamadas recursivas

A matriz representando o candidato (M^T) é uma matriz de dimensões n x m, onde n corresponde ao número de vértices do grafo G, e m corresponde ao número de vértices do grafo *H*. Na Figura 8 tem-se um exemplo de um grafo G (com 3 vértice) e sua matriz de adjacências, correspondendo ao número de linhas da matriz M^T e um grafo H (com 4 vértices) e sua matriz de adjacências, representando o número de colunas da matriz M^T.

Figura 8 – Exemplo de como é criado a matriz (M^0)



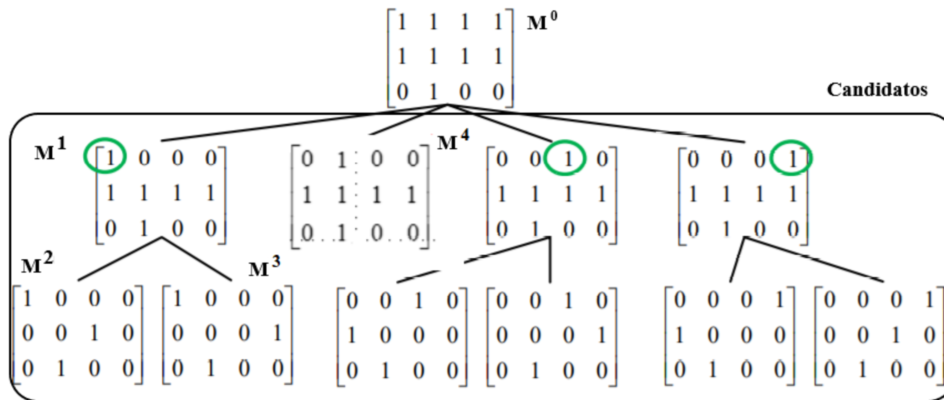
Fonte: MOTTIN; LAZARIDOU (2016), adaptado

A primeira matriz (M^0) deve ser criada seguindo a Equação 2.1. Por exemplo, o elemento $M_{0,0}$ (Figura 8), o grau de V_{G0} é igual a 1 e o grau de V_{H0} também é igual a 1, logo o valor do elemento $M_{0,0}$ será igual a 1. Já para o elemento $M_{2,3}$, tem-se que o grau de V_{G2} é igual a 2 e o grau de V_{H3} é igual a 1, logo o valor do elemento $M_{2,3}$ será igual a zero. A Figura 8 ilustra o processo de criação da matriz (M^0).

$$M_{ij}^0 = \begin{cases} 1, & \text{se grau}(\ V_{Gi}) \ \leq \text{grau}(V_{Hj}) \\ 0, & \text{caso contrário} \end{cases} \tag{2.1}$$

A partir da matriz (M^0) são geradas as outras matrizes, respeitando uma ordenação pós-ordem (todos os nodos a esquerda e após os nodos a direita) alterando todas as células para o valor 0 exceto uma célula de cada linha. Na Figura 9 tem-se exemplificado as matrizes resultantes após a execução do algoritmo. Destaca-se que, a matriz (M^4) só será criada após a matriz (M^3) ter sido devidamente refinada.

Figura 9 – Exemplo de como são criados os outros candidatos

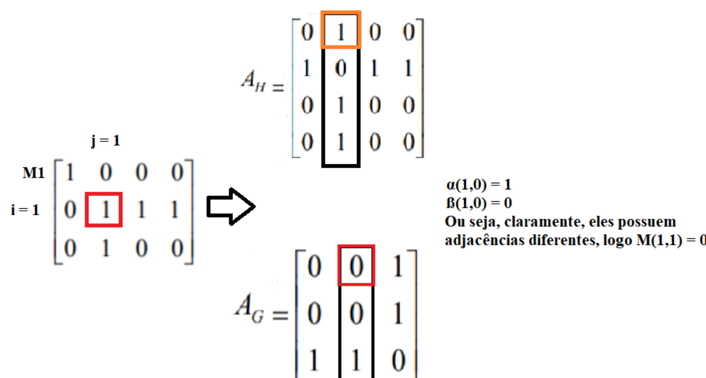


Fonte: o autor (2020)

O maior diferencial do algoritmo de Ullmann é a técnica de refinamento utilizada. O objetivo dessa técnica é eliminar elementos $1's$ da matriz, diminuindo assim o número de possíveis candidatos a serem gerados e reduzindo o tempo de processamento. Durante esse processo de refinamento, é possível antever candidatos inaptos ao isomorfismo. Quando uma determinada linha da matriz de permutação não possui mais nenhum elemento 1, todos os próximos candidatos não serão gerados. Isso porque, quando uma linha da matriz não possui nenhum elemento 1, tem-se que aquele vértice não possui nenhuma adjacência, tornando assim a matriz (e suas subsequentes) inapta ao isomorfismo.

Para a eliminação dos elementos $1's$ da matriz, considera-se que um vértice de G, V_G corresponde a um vértice de H, V_H, então para cada vértice adjacente de V_G, também deve-se ter um vértice em V_H. Por exemplo, como pode ser observado na Figura 10, no grafo H, o vértice 2 (j=1 da matriz) possui adjacência com os vértices 1, 3, 4, porém no grafo G, esse possui apenas adjacência com vértice 3. Desta forma, a relação não pode ser considerada isomórfica, e o elemento $M[1, 1]$ deve ser setado para zero.

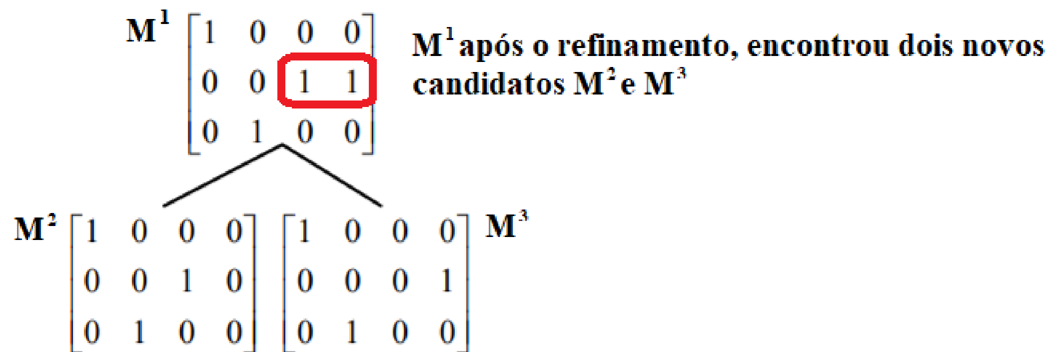
Figura 10 – Exemplo de como é feito o refinamento



Fonte: o autor (2020)

O algoritmo Ullmann segue o processo de refinamento até todos os elementos $1's$ da linha serem setados para 0, exceto um por coluna. Durante o processo de refinamento é verificado constantemente se a linha contém algum elemento 1. Caso a linha possua mais elementos iguais a $1's$, são gerados os próximos ramos da árvore conforme Figura 11. Caso contrário, o processo é interrompido naquele ramo da árvore e o algoritmo executa um *backtracking*, voltando ao próximo candidato válido.

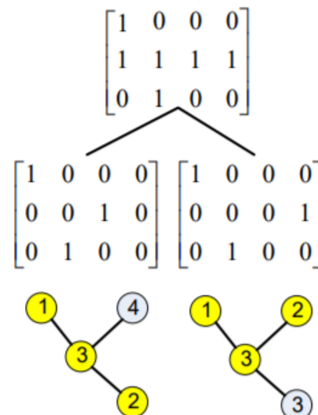
Figura 11 – Exemplo de como é gerado o próximo candidato



Fonte: o autor (2020)

Quando o algoritmo Ullmann chega ao fim do refinamento, isto é, ao nível mais baixo da árvore, a(s) matriz(es) resultante(s) caracterizam-se por ser um isomorfismo de um subgrafo. Porém, apesar do algoritmo ter encontrado um isomorfismo de subgrafo, podem existir ainda outros nodos da árvore que caracterizem um isomorfismo de subgrafos, por isso ele efetua um *backtracking* pós-ordem, até encontrar um próximo candidato que ainda não foi refinado.

Figura 12 – Exemplo de quando o algoritmo chega ao último nível, caracterizando um isomorfismos em subgrafo



Fonte: MOTTIN; LAZARIDOU (2016), adaptado

Durante o processo de refinamento o algoritmo Ullmann verifica constantemente se existe um isomorfismo completo (grafo-grafo) na matriz. No caso da existência do isomorfismo, o processo é interrompido uma vez que o objetivo foi alcançado. O isomorfismo grafo-grafo é encontrado quando a Equação 2.2 é satisfeita. Como a matriz C neste caso é obtida através da multiplicação da matriz de permutação M' pela transposta do resultado da multiplicação da matriz de permutação M' pela matriz de adjacências do grafo H, o isomorfismo é encontrado quando a matriz resultante C for igual a matriz de adjacências do grafo G.

$$\begin{aligned} C &= M'(M'H)^T \\ se\ (G_{ij} = 1) &\Rightarrow (C_{ij} = 1) \end{aligned} \tag{2.2}$$

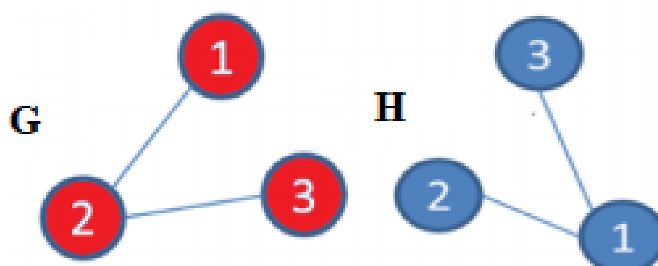
2.2.2 Algoritmo VF2

O algoritmo VF2 foi proposto por Cordella *et al.* (CORDELLA *et al.*, 2001) e, similarmente ao algoritmo de Ullmann, também utiliza uma árvore de busca (DFS). Através do VF2 é possível identificar tanto isomorfismo grafo-grafo como também isomorfismo em subgrafos (CORDELLA *et al.*, 2001).

O algoritmo VF2 é um aprimoramento do algoritmo VF, nesta versão o espaço de busca é organizado de forma a reduzir os requisitos de memória, tornando o algoritmo mais adequado para grafos maiores e também mais rápido para grafos pequenos e médios (CORDELLA *et al.*, 2001).

Considerando os grafos *G* e *H*, apresentados na Figura 13, o algoritmo VF2 tem seu início a partir da criação de dois conjuntos MG ({}) e MH ({}), que inicialmente são vazios. O primeiro conjunto é o conjunto *MG* que representa os vértices do grafo *G* e o segundo o conjunto *MH* que representa os vértices do grafo *H*.

Figura 13 – Grafos *G* e *H*

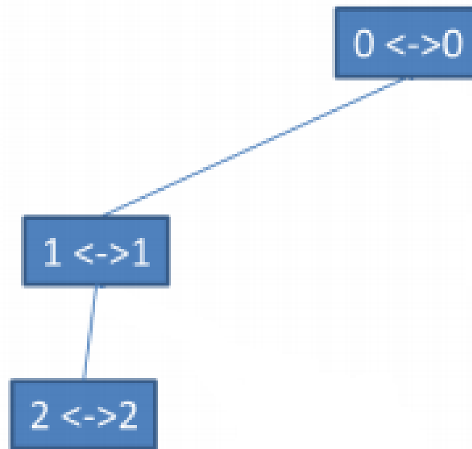


Fonte: MAKSOV; LI; BUTLER (2015), adaptado

Inicialmente, o algoritmo VF2 compara os conjuntos *MG* ({}) e *MH* ({}) , retornando verdadeiro uma vez que ambos os conjuntos são vazios. Depois são incluídos os vértices 1 dos grafos G e H, resultando nos conjuntos *MG*={1} e *MH*={1}. Uma vez que eles são isomorfos, a

condição é verdadeira e o algoritmo passa para o próximo elemento, que neste caso é o vértice 2. O algoritmo repete o mesmo procedimento, utilizando o vértice 2. Uma vez que existe um aresta entre os vértices 1 e 2 em ambos os grafos, os vértices são adicionados na árvore, conforme pode ser observado na Figura 14, e o vértice 2 é inserido nos conjuntos MG=$\{1, 2\}$ e MH=$\{1, 2\}$.

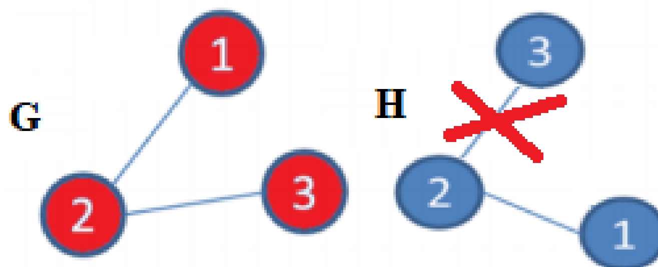
Figura 14 – Estado atual: MG = {1,2} e MH = {1,2}



Fonte: MAKSOV; LI; BUTLER (2015), adaptado

O algoritmo VF2 prossegue objetivando inserir o vértice 3 dos grafos G e H nos conjuntos MG=$\{1, 2, 3\}$ e MH=$\{1, 2, 3\}$, porém neste caso a condição é falsa, uma vez que a aresta $\{2, 3\}$ do grafo G não existe no grafo H, como pode ser observado na Figura 15.

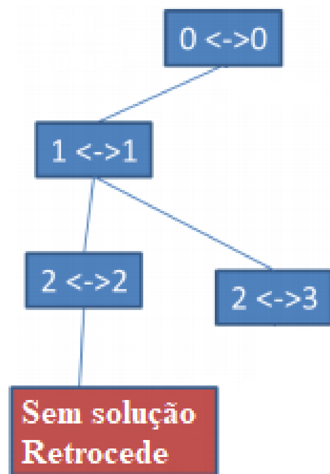
Figura 15 – Representação de uma relação de não adjacência



Fonte: o autor, (2020)

Como o algoritmo não encontrou essa adjacência, o algoritmo efetua um *backtracking* removendo os vértices $\{2, 3\}$ de ambos os conjuntos, resultando nos conjuntos MG=$\{1\}$ e MH=$\{1\}$. Após, o algoritmo insere os vértices 2 e 3 nos conjuntos MG e MH. Neste caso como existe as arestas $\{1, 3\}$ e $\{1, 2\}$ nos grafos G e H, os vértices 2 e 3 são inseridos nos conjuntos MG e MH, respectivamente, conforme pode ser observado na Figura 16.

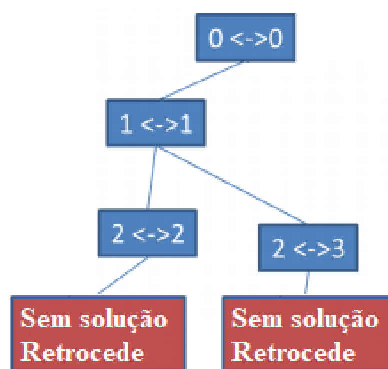
Figura 16 – Estado Atual: MG = {1,2} e MH = {1,3}



Fonte: MAKSOV; LI; BUTLER (2015), adaptado

Posteriormente, são inseridos os vértices $\{3\}$ e $\{2\}$ nos conjuntos *MG* e *MH*, respectivamente. Entretanto, como o vértice 3 não possui uma aresta com vértice 2 no grafo *H*, a verificação retorna falso e o algoritmo efetua um novo *backtracking* retornando ao início e removendo todos os vértices dos conjuntos *MG* e *MH*, resultando nos conjuntos MG ({}) e MH ({}). Como pode ser visualizado na Figura 17.

Figura 17 – Estado Atual: MG = {} e MH = {}



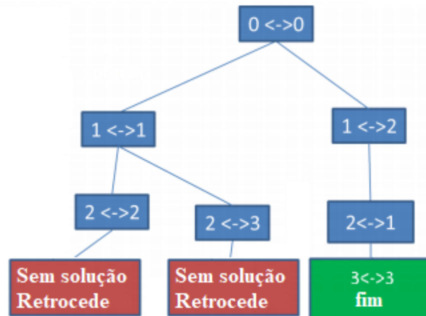
Fonte: MAKSOV; LI; BUTLER (2015), adaptado

O algoritmo prossegue inserindo os vértices 1 e 2 nos conjuntos *MG* e *MH*. Uma vez que os conjuntos encontram-se vazios, a condição é satisfeita e os vértices são inseridos nos conjuntos, resultando em MG=$\{1\}$ e *MH*=$\{2\}$. Depois são inseridos os vértices 2 e 1, dado que existe uma relação em ambos os grafos (existe a aresta $\{1,2\}$ nos grafos G e H) , é retornado verdadeiro e os vértices são inseridos nos conjuntos, resultando em *MG*=$\{1,2\}$ e *MH*=$\{2,1\}$. Após, o vértice 3 é inserido nos conjuntos, uma vez que existe a aresta $\{2,3\}$ no grafo *G* e a aresta $\{1,3\}$ no grafo *H*, resultando nos conjuntos *MG*=$\{1,2,3\}$ e *MH*=$\{2,1,3\}$.

Por fim, o algoritmo verifica se ainda existem vértices que não foram inseridos nos

conjuntos. Como não existem mais vértices a serem inseridos o algoritmo é concluído retornando que os grafos G e H são isomorfos com a relação MG=$\{1, 2, 3\}$ e MH=$\{2, 1, 3\}$.

Figura 18 – Estado final: MG = {1,2,3} e MH = {2,1,3}



Fonte: MAKSOV; LI; BUTLER (2015), adaptado

2.2.3 Nauty (ou *Practical Graph Isomorphism*)

O *Nauty* é um algoritmo que pode ser utilizado exclusivamente para a identificação do isomorfismo completo em grafos, ou seja, dados dois grafos G e H, o algoritmo busca verificar se eles são isomorfos ou não. Apesar da nomenclatura correta para a heurística ser *Practical Graph Isomorphism*, em grande parte da literatura esse é identificado como *Nauty* (No AUTomorphisms, Yes?), que é o nome da biblioteca que implementa esse algoritmo (MCKAY; PIPERNO, 2019).

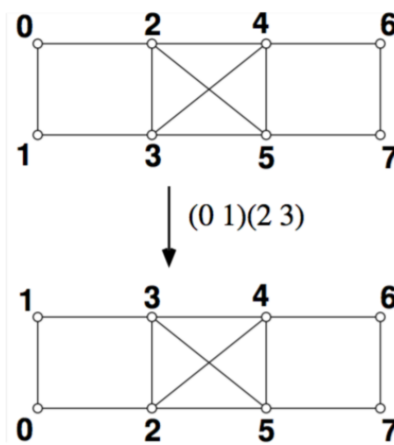
Para a identificação do isomorfismo em grafos, o *Nauty* faz uso de um processo chamado de rotulagem canônica, que consiste em renomear todos os vértices do grafo (canonização), de tal forma que, os grafos serão isomorfos, se e somente se, após a canonização os mesmos forem idênticos (MCKAY; PIPERNO, 2019). Na Figura 19 tem-se um exemplo do processo de canonização de dois grafos.

Figura 19 – Exemplo da canonização em dois grafos

Fonte: MCKAY; PIPERNO (2019)

O *Nauty* utiliza uma técnica de automorfismo para reduzir o espaço de busca do algoritmo. O automorfismo pode ser definido como um isomorfismo de um grafo em relação a ele mesmo. Em outras palavras, pode-se fazer uma permutação dos vértices de tal forma que as relações de adjacências e não adjacências sejam preservadas, como pode ser observado na Figura 20. Neste caso foi aplicada uma função de rotação de 180° graus sobre os vértices $\{0, 1\}$ e $\{2, 3\}$, sendo que o resultado manteve as adjacências e não adjacências, sendo assim os vértices $\{0, 1\}$ e $\{2, 3\}$ são um automorfismo. Um candidato a isomorfismo é descartado quando um automorfismo é encontrado, pois isso significa que uma representação similar já foi processada.

Figura 20 – Aplicado uma função de rotação em 180° graus sobre os vértices$\{0, 1\}$ e $\{2, 3\}$



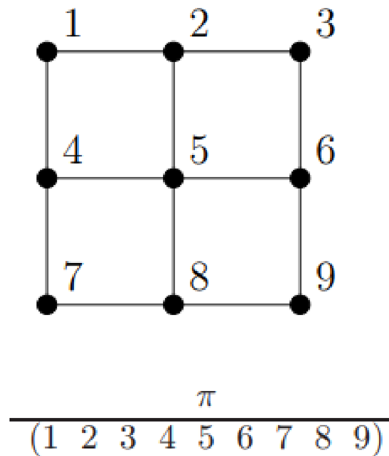
Fonte: MCKAY; PIPERNO (2019)

O funcionamento do *Nauty* baseia-se no conceito de partições, sendo que uma partição [6] π pode ser definida como sendo um conjunto de elementos, onde cada elemento deste conjunto representa um vértice do grafo. Dentro de cada partição, podemos ter n subconjuntos denominados células. Uma partição é dita discreta quando o conjunto possui apenas elementos triviais, isto é, quando a célula possui apenas um vértice. Por exemplo, dado uma partição $\pi = (1, 2, 3)$, ela será discreta quando $\pi = (1|2|3)$.

A partição inicial π será um conjunto com n elemento(s), onde n é a quantidade de vértices do grafo. Na Figura 21, tem-se uma representação do grafo original e a partição inicial que é formada por todos os vértices do grafo.

[6] No trabalho original, o autor denomina essas partições como *coloração* e para cada nova partição abaixo dela, representa uma coloração mais fina

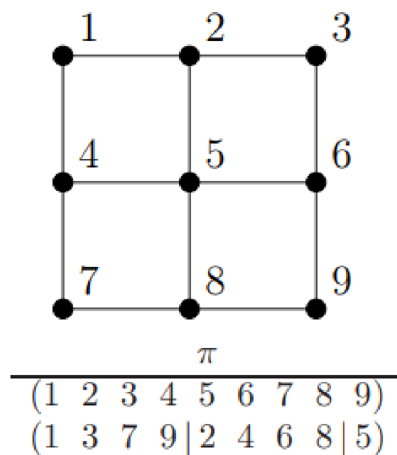
Figura 21 – Partição Inicial do Algoritmo Nauty



Fonte: HARTKE; RADCLIFFE (2013)

Dado a partição inicial, o objetivo é obter uma partição equitativa onde todos os elementos da célula respeitem uma mesma regra. No caso do *Nauty*, todos elementos da célula devem possuir o mesmo grau, isto é $dG(u, Vj) = dG(v, Vi)$ onde Vj e Vi são células de π. Caso o grau dos elementos forem diferentes, o elemento será dividido gerando uma nova célula.

Figura 22 – Refinamento considerando os graus dos vértices



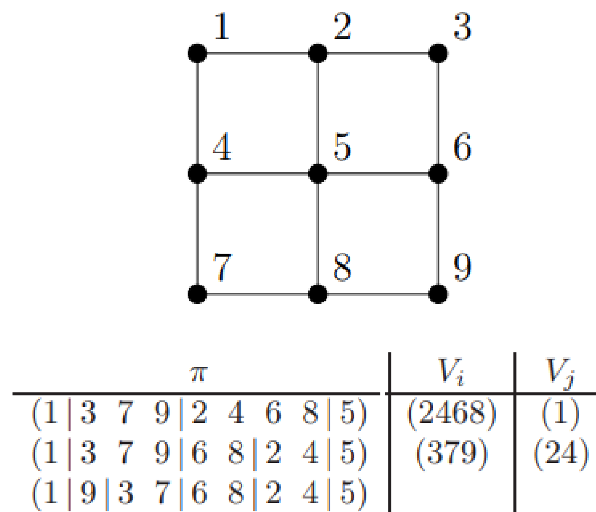
Fonte: HARTKE; RADCLIFFE (2013)

Como a partição π inicialmente tem apenas uma única célula, ela será dividida de acordo com o grau dos vértices gerando assim 3 novas células, a primeira delas com vértices de grau dois (π = (1 3 7 9)) a segunda formada com vértices de grau três (π = (2 4 5 8)) e a terceira com vértices de grau quatro (π=(5)).

Após, é aplicada uma técnica chamada de distinção artificial onde, dado uma célula não trivial, obtém-se uma nova célula com o primeiro elemento desta célula. No exemplo, a primeira célula não trivial é $\pi = (1379)$ e o primeiro elemento é o (1). Logo, esse elemento será dividido gerando uma nova célula, resultando em $(1|379)$. Essa técnica tem como objetivo reduzir a quantidade de ramos na árvore, visto que sem essa técnica, teríamos muitos ramos equivalentes que futuramente seriam descartados.

Posteriormente, obtém-se um elemento de uma determinada célula V_j e a partir dessa é realizada a divisão de uma célula V_i. Essa divisão ocorre verificando se os elementos de V_i possuem uma relação de adjacência com o elemento de V_j, Por exemplo, se $V_j = (1)$ e $V_i = (2468)$, tem-se que apenas os elementos $\{2, 4\}$ apresentam uma relação de adjacência com o elemento $V_j = (1)$, logo V_i será dividida originando uma nova célula contendo $(68|24)$, conforme pode ser observado na Figura 23.

Figura 23 – Exemplo de um refinamento



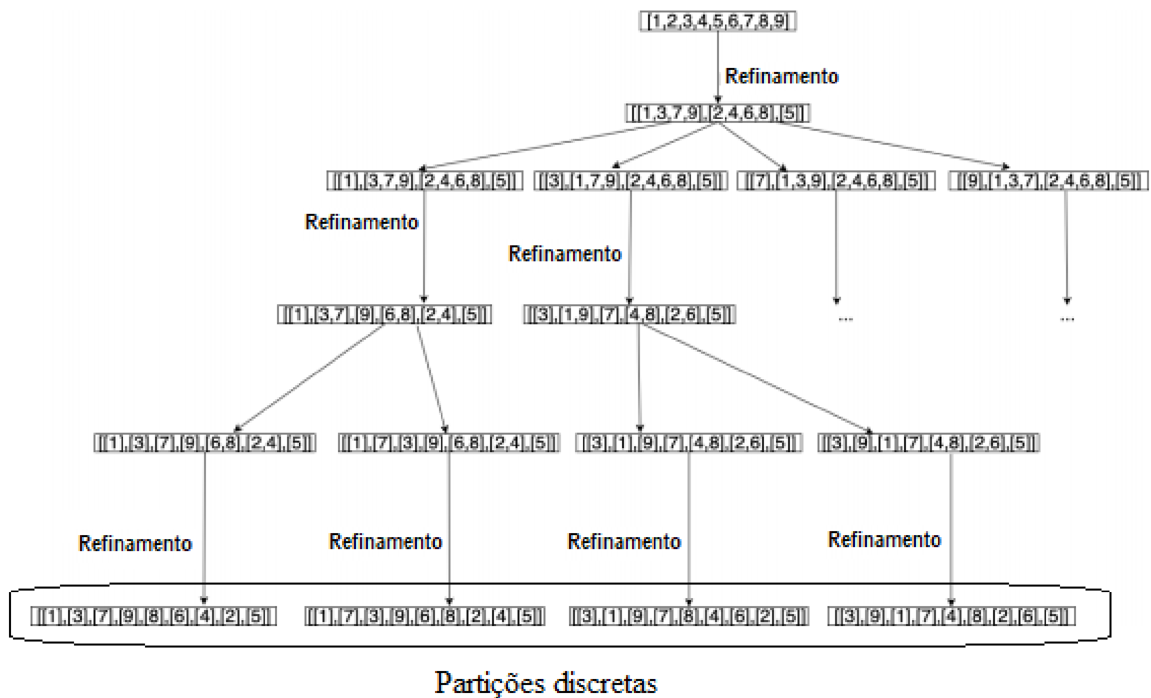
π	V_i	V_j
$(1\,|\,3\ 7\ 9\,|\,2\ 4\ 6\ 8\,|\,5)$	(2468)	(1)
$(1\,|\,3\ 7\ 9\,|\,6\ 8\,|\,2\ 4\,|\,5)$	(379)	(24)
$(1\,|\,9\,|\,3\ 7\,|\,6\ 8\,|\,2\ 4\,|\,5)$		

Fonte: HARTKE; RADCLIFFE (2013)

Uma vez que não é possível refinar nenhuma outra célula a partir de V_j, tem-se a escolha de um novo valor para V_j. Por exemplo, tem-se a escolha da célula (24) como o novo V_j e a partir desse é realizado a divisão da célula (379) (Figura 23). Esse processo é repetido até que todas as células tenham sido divididas. Assim, quando não for possível dividir mais nenhuma célula tem-se uma partição equitativa e a partir dela são gerados as partições filhas, que também podem ser chamadas de partições folhas.

Para a geração da rotulagem canônica, cada partição equitativa é armazenada em uma estrutura de árvore de busca, onde cada partição é representada como um nodo na árvore (Figura 24). A raiz da árvore é a partição π inicial refinada. Essa árvore é utilizada tanto para a geração da rotulagem canônica bem como para o descarte dos nodos que caracterizam um automorfismo.

Figura 24 – Árvore de busca final

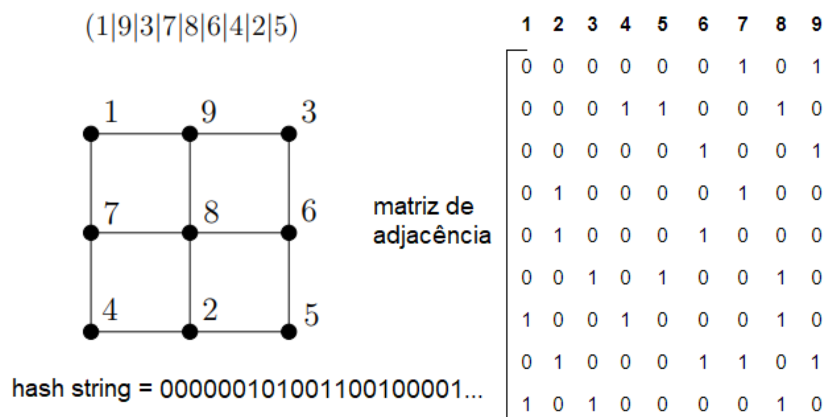


Fonte: TABAK (2020), adaptado

A partir do nodo inicial, o *Nauty* percorre a árvore utilizando uma busca em profundidade pós-ordem, gerando os nodos folhas. Esses nodos folhas são gerados retirando o primeiro elemento da primeira célula não trivial e os colocando em uma nova célula. Por exemplo, para o conjunto $(1|9|37|68|24|5)$, tem-se que a primeira célula não trivial é a célula (37). Essa será dividida gerando a célula $(3|7)$, resultando na partição folha $(1|9|3|7|68|24|5)$. Esse processo é realizado até que nenhuma célula possa ser dividida, restando apenas células triviais, caracterizando π como sendo uma partição discreta. Cada partição discreta gerada indica uma nova rotulação dos vértices. Por exemplo, caso a partição discreta resultante for $\pi = (1|3|2)$, tem-se que o vértice 3 pode ser trocado com vértice 2, mantendo-se o vértice 1. Na Figura 24, tem-se uma representação da árvore de busca criada para algumas partições discretas.

Quando uma partição discreta é encontrada, o algoritmo gera uma representação binária para essa partição. Essa representação binária é gerada a partir da porção triangular superior da matriz de adjacência. É importante ressaltar que essa representação, que pode ser chamada de *hash string*, não é única, ou seja, duas partições distintas podem ter um mesmo *hash string*. Quando isso ocorre, indica que ambas partições π possuem a mesma matriz de adjacências e, consequentemente, indica a existência de um automorfismo.

Figura 25 – Um possível candidato para isomorfismo.



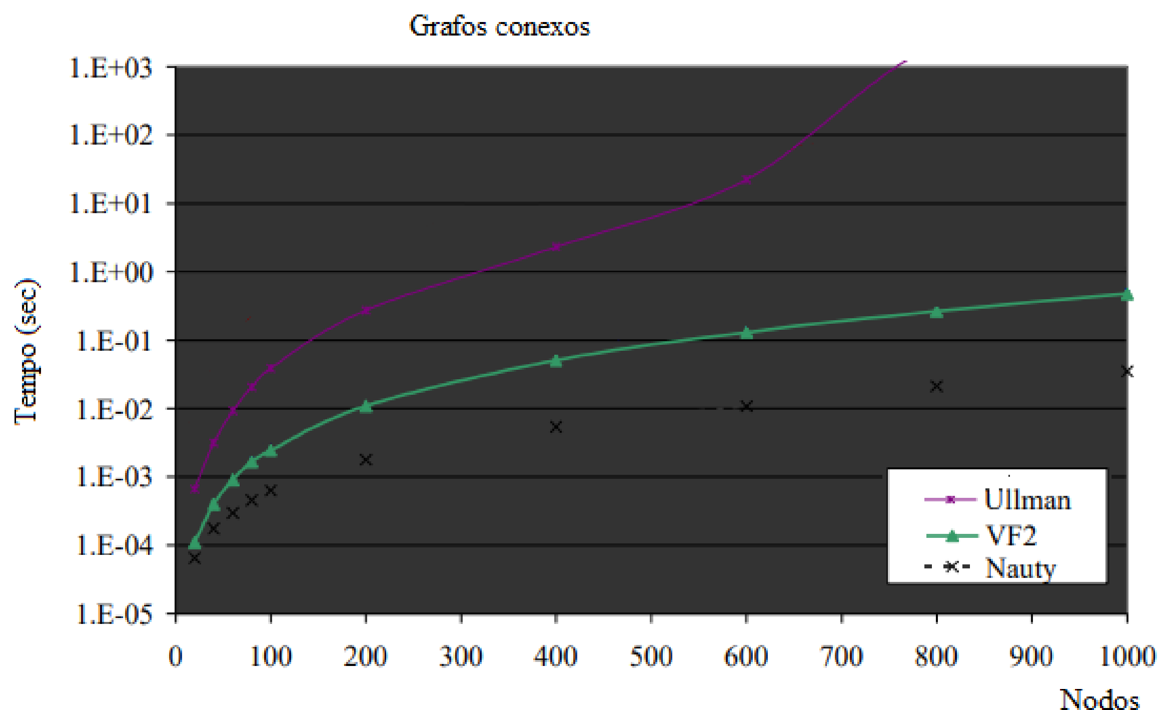
Fonte: o autor (2020)

Dado que os nodos da árvore de busca são geradas em uma busca em profundidade, pode-se ter casos onde uma uma representação binária já foi previamente gerada. Neste caso, tem-se que ambas rotulagens são iguais e, consequentemente, representam uma mesma matriz de adjacência. Assim, pode-se interromper a verificação deste ramo da árvore, uma vez que o automorfismo já foi encontrado anteriormente e, desta forma, tem-se uma redução no tempo de execução do método.

2.3 DEFINIÇÃO DA HEURÍSTICA

Na Figura 26 pode-se verificar um comparativo realizado por Foggia, Sansone e Vento (FOGGIA; SANSONE; VENTO, 2001b) entre os tempos de execução das heurísticas: Ullmann, VF2 e Nauty. Para esse comparativo, utilizou-se grafos conexos de diferentes tamanhos que foram gerados randomicamente. Como pode ser observado na Figura 26 os algoritmos VF2 e Nauty apresentaram tempos de execução com ordens de grandeza similares. Já o algoritmo de Ullmann mostrou-se inviável para grafos com mais de 700 vértices.

Figura 26 – Comparativo entre performance das heurísticas



Fonte: FOGGIA; SANSONE; VENTO (2001b), adaptado

A partir dos resultados obtidos por Foggia, Sansone e Vento (FOGGIA; SANSONE; VENTO, 2001b), optou-se pela utilização do algoritmo VF2. Apesar deste ter apresentado tempos de execução superiores ao *Nauty*, optou-se pela utilização do VF2 pois este apresenta um tempo de execução aceitável, mesmo para grafos com um grande número de vértices, e possibilita a verificação de isomorfismo em subgrafos. Já o algoritmo de *Nauty* pode ser utilizado exclusivamente para a identificação de isomorfismo completo em grafos.

3 UNIDADES DE PROCESSAMENTO GRÁFICO

Na últimas décadas observou-se uma grande evolução no poder computacional dos processadores. Essa evolução se deve, em grande parte, a capacidade dos fabricantes de microprocessadores em diminuir o tamanho dos transistores e, consequentemente, aumentar o número de transistores por processador.

Em 1965, Gordon Earle Moore, um dos cofundadores da Intel, previu que a densidade dos transistores nos *chips* dobraria em um período de cerca de 24 meses pelo mesmo custo e, como resultado, o poder computacional também dobraria nesse período. Essa previsão acabou se confirmando e ficou conhecida como a Lei de Moore (Moore, 1965) .

Entretanto, nos últimos anos os fabricantes estão encontrando dificuldades em aumentar o poder computacional dos processadores devido, principalmente, a questões físicas. De fato, quando diminuímos o tamanho dos transistores tem-se um aumento na quantidade desses e, consequentemente, tem-se um aumento em problemas relacionados à dissipação de calor e a comunicação entre esses. De forma a contornar essas limitações, foram criadas as arquiteturas paralelas, que se caracterizam pela presença de múltiplos processadores que cooperam entre si, para resolver um determinado problema (STALLINGS, 2017).

Em 1966, Michael J. Flynn criou um dos primeiros sistemas de classificação para computadores paralelos e sequenciais, sendo que essa classificação é conhecida como taxonomia de Flynn. De acordo com a taxonomia de Flynn, os computadores podem ser divididos considerando o número de fluxos de instruções e o número de fluxos de dados que estão sendo processados simultaneamente. A partir dessas características os sistemas computacionais foram classificados em 4 categorias:

- *Single Instruction Single Data (SISD)*: é o tipo de arquitetura mais simples onde tem-se um único fluxo de instruções sobre um único conjunto de dados sendo esta, uma arquitetura estritamente sequencial, onde uma única operação é realizada por vez. Nesta categoria encontram-se os computadores com processadores de um único núcleo (*monocores*).
- *Single Instruction Multiple Data (SIMD)*: neste tipo de arquitetura tem-se apenas um fluxo de instrução atuando sobre múltiplos dados. A arquitetura SIMD explora o paralelismo a nível de dados, mas ainda segue sendo uma estrutura sequencial. Essa arquitetura está presente em grande parte dos processadores matriciais, sendo que um exemplo de processador que tem recebido destaque nos últimos anos são as GPUs (*Graphics Processing Unit*, ou Unidade de Processamento Gráfico).

- *Multiple Instruction Single Data (MISD)*: esse tipo de arquitetura se caracteriza por várias instruções sendo executadas sobre um único dado. Não existem exemplos dessa arquitetura, sendo que a maioria dos autores a consideram como sendo apenas uma categoria teórica, sem nenhum exemplo real. Outros autores citam como exemplo da categoria MISD os *pipelines* de instruções, que encontram-se presentes nos processadores modernos.

- *Multiple Instruction Multi Data (MIMD)*: essa arquitetura é caracterizada pela execução de múltiplos fluxos de instruções sobre múltiplos dados. Como exemplo de sistemas que encontram-se nessa categoria pode-se citar os processadores *multicore*, computadores multiprocessados e os *clusters* de computadores.

Os sistemas paralelos pode ser divididos ainda de acordo com a organização de memória utilizada e de acordo com a forma como é feita a comunicação entre os núcleos de processamento. Neste caso, as arquiteturas paralelas são comumente divididas em:

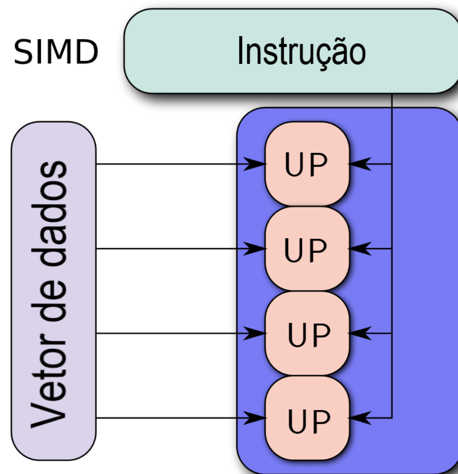
- *Arquitetura com Memória Compartilhada:* todos os núcleos de processamento acessam uma mesma área de memória que é utilizada para a troca de informações. Essa abordagem é muito utilizada quando o paralelismo é realizado em um mesmo computador.

- *Arquiteturas com Memória Distribuída:* abordagem utilizada em sistemas onde os núcleos de processamento não compartilham memória. Desta forma, a comunicação entre os núcleos é realizada através do envio e do recebimento de mensagens operando em uma rede de computadores.

3.1 ARQUITETURA SIMD

A arquitetura SIMD foi introduzida na década de 70 e era muito utilizada nos super-computadores da época que utilizavam processadores vetoriais. Esses processadores eram assim chamados pois efetuavam uma única instrução sobre um vetor de dados. Nesse tipo de arquitetura, uma única unidade de controle (UC) transmite uma instrução para as unidades de processamento (UP) que irão trabalhar em paralelo. Na Figura 27 tem-se uma representação da estrutura de uma arquitetura do tipo SIMD.

Figura 27 – Representação da arquitetura SIMD



Fonte: o autor (2020)

Esse tipo de arquitetura apresenta vantagens se comparada a uma arquitetura SISD. Por exemplo, considerando um programa que tem como objetivo somar os elementos de dois vetores com N elementos (Figura 28). Em uma arquitetura SISD, para cada um dos N elementos será realizada uma operação de READ para a busca na memória, depois é realizada a operação de soma e por fim é feito o STORE do resultado. Em uma arquitetura SIMD tem-se um único LOAD de todos os N elementos dos vetores e é realizada uma instrução de soma sobre os elementos dos vetores. Por fim, é executado uma operação de STORE com todos os resultados. Conclui-se que, em uma arquitetura do tipo SIMD, os dados são tratados como um bloco e as operações são realizadas sobre o bloco completo.

Figura 28 – Comparação da execução de instruções entre um processador escalar e um processador vetorial

soma_dois_vetores.c

```
for (i = 0; i < N; i++)
{
  A[i] = B[i] + C[i];
}
```

processador escalar

```
INITIALIZE I = 1
READ B(I)
READ C(I)
ADD B(I) + C(I)
STORE A(I)
INCREMENT I = I+ 1
IF I <= N GO TO 10
STOP
```

processador vetorial

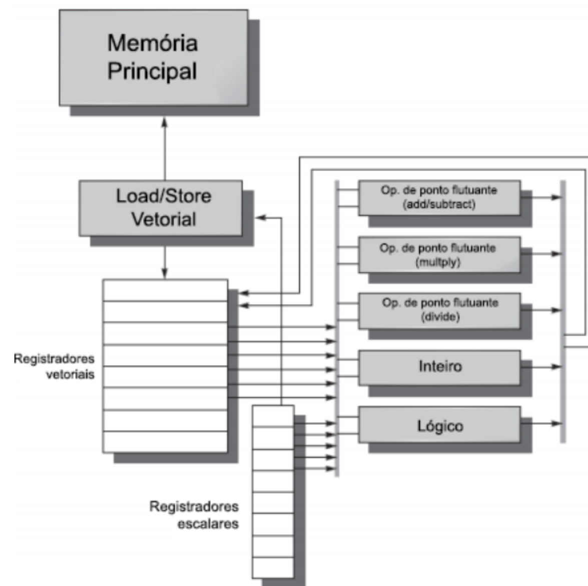
```
TEMP(1:N) = A(2:N + 1)
A(1:N) = B(1:N) + C(1:N)
B(1:N) = 2 * TEMP(1:N)
```

Fonte: o autor (2020)

Existem duas categorias de processadores presentes na arquitetura SIMD, os processadores matriciais (*array processors*) e os processadores vetoriais (*vector processors*). Esses são diferenciados pela forma como é efetuada a concorrência de instruções e pela forma como é realizado o compartilhamento dos dados entre as instruções (BASU, 2016).

Um processador vetorial é um processador de uso geral que pode efetuar operações sobre escalares, ao contrário de um processador matricial que não possui tais funções e que depende de uma outra unidade de controle para efetuar essas operações. Os processadores vetoriais utilizam ULAs (Unidade lógica e aritmética) trabalhando em um *pipeline*[1] ou em paralelo. Na Figura 29 podemos visualizar uma representação genérica deste tipo de arquitetura. O compartilhamento de dados é feito ou através de uma memória, ou ainda através de registradores.

Figura 29 – Processador vetorial do tipo registrador-registrador

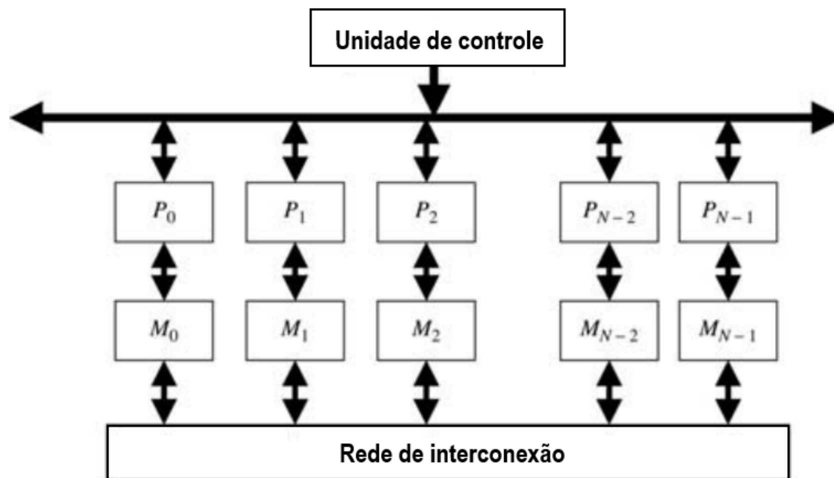


Fonte: DAHLBERG (2012) adaptado

Já um processador matricial não possui um processador escalar (unidade de controle), sendo assim, esse tipo de arquitetura não é independente. Nesta arquitetura cada unidade de processamento é constituída de uma ULA contendo a sua respectiva memória local. Para o compartilhamento dos dados entre as unidades de processamento é utilizada um canal de interconexão que encontra-se ligado a todas as unidades de processamento. Na Figura 30 tem-se uma representação deste tipo de arquitetura.

[1] É uma técnica para implementar paralelismo a nível de instrução em um único processador. Ao invés do processador executar toda a instrução de uma vez só, essa instrução é dividida em estágios. Assim que termina o primeiro estágio, executa-se o segundo e uma nova instrução é executada no primeiro estágio. Dessa forma, o processador está quase sempre ocupado, diminuindo o tempo de processamento das instruções.

Figura 30 – Imagem representando um processador matricial



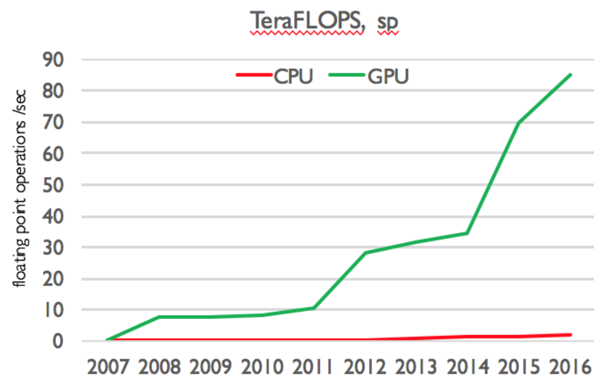
Fonte: ABD-EL-BARR; EL-REWINI (2005), adaptado

3.2 UNIDADES DE PROCESSAMENTO GRÁFICO

Apesar da arquitetura SIMD ter sido concebida para ser utilizada em supercomputadores, atualmente, ela se mostra presente em um grande número de computadores pessoais através das Unidades de Processamento Gráfico (GPU - *Graphics Processing Unit*). O uso das GPUs cresceu muito nas últimas décadas, em grande parte devido a indústria de jogos de computadores, que tornaram os jogos cada vez mais realistas, exigindo GPUs cada vez mais poderosas.

Com o passar do tempo, as grandes empresas viram um grande potencial no uso de GPUs em aplicações de propósito geral, como por exemplo, aplicações da área de bioinformática, dinâmica molecular, exploração de petróleo e gases, finanças computacionais, processamento de áudio e sinal, entre outras (STALLINGS, 2017). Atualmente, já são comercializadas GPUs sem saída de vídeo, sendo utilizadas exclusivamente para processamento paralelo (AMORIM; ARAÚJO, 2011). A Figura 31 exibe um comparativo da capacidade de processamento de CPUs (processador intel) e GPUs com relação ao número de operações de ponto flutuante realizadas.

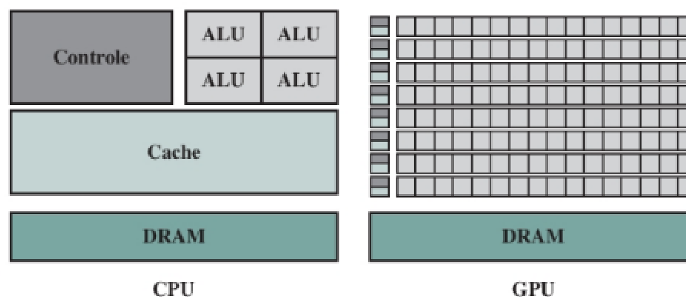
Figura 31 – Comparativo de poder computacional entre CPU e GPU



Fonte: OMNISCI (2017)

O maior poder de processamento das GPUs se deve ao fato das GPUs terem sido projetadas para maximizar o paralelismo das tarefas (OMNISCI, 2017). Por exemplo, em uma CPU *multicore* é possível executar apenas algumas *threads* em simultâneo. Já em uma GPU tem-se a possibilidade de executar um número muito maior de *threads*. De fato, na Figura 32, tem-se um exemplo de uma CPU, que possui apenas 4 ULAs e de uma arquitetura GPU que possui 128 ULAs.

Figura 32 – Comparativo entre uma arquitetura CPU com uma GPU



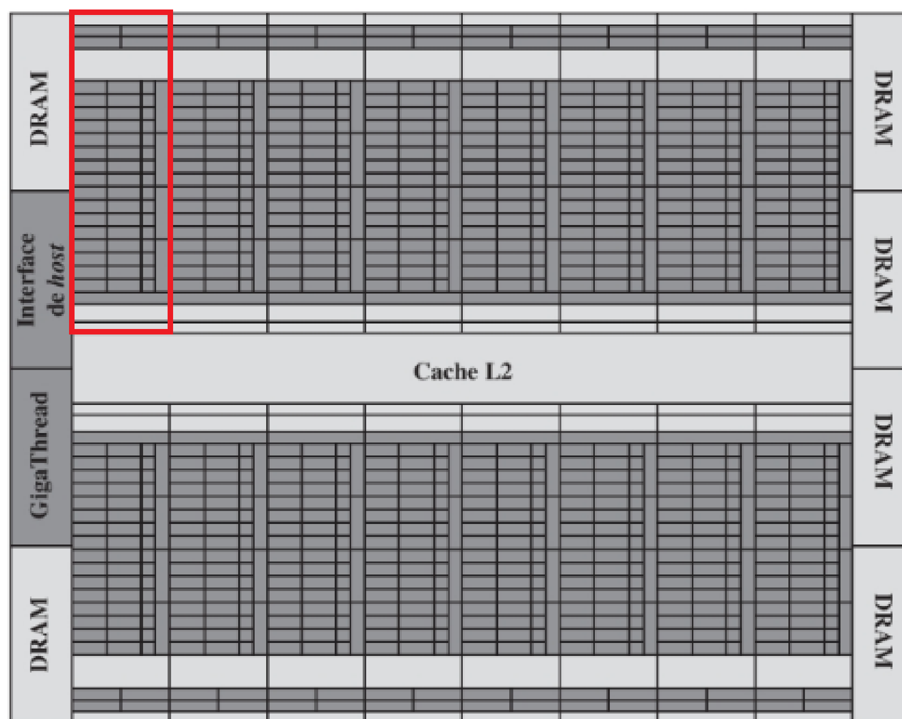
Fonte: STALLINGS (2017)

Esse tipo de uso de GPUs acabou se popularizando, tanto que em 2006 a NVIDIA lançou a primeira solução para computação de propósito geral em GPUs, a plataforma NVIDIA CUDA (*Compute Unified Device Architecture*) (CUDA, 2020). Essa plataforma permitiu que programadores pudessem utilizar as GPUs da NVIDIA para a paralelização de aplicações das mais diversas áreas. Posteriormente, em 2008 foi lançada a plataforma de código aberto OpenCL (*Open Computing Language*) (OPENCL, 2020), inicialmente criada pela Apple, e posteriormente gerida pelo consórcio Khronos Group. O OpenCL é uma plataforma de desenvolvimento genérica, isto é, essa pode ser utilizada em GPUs de todos os fabricantes, ao contrário da NVIDIA CUDA que só pode ser utilizada em GPUs da própria NVIDIA.

3.3 ARQUITETURA DE UMA GPU

A estrutura básica de uma GPU é formada basicamente por um conjunto de MS (*Streaming Multiprocessor*), sendo que cada um desses MS possui um determinado número de CUDA *cores*. Na Figura 33 tem-se uma representação da arquitetura Fermi da NVIDIA. Essa arquitetura apresenta: 16 MSs, sendo que cada um deles contém um grupo de 32 *cores* CUDA, totalizando 512 cores CUDA; 6 unidades de memória DRAM (*Dynamic Random Access Memory*); uma interface de *host* que permite a conectividade entre a CPU e a GPU; e uma unidade *GigaThread* que é responsável por distribuir as *threads* em um MS.

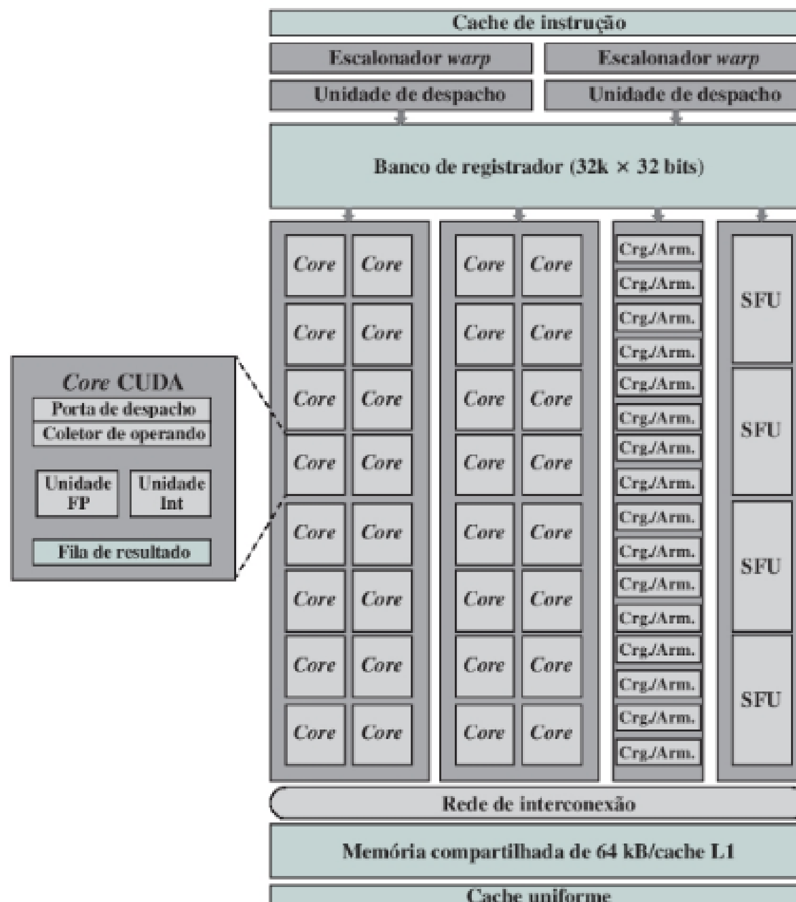
Figura 33 – Arquitetura Fermi da NVIDIA



Fonte: STALLINGS (2017), adaptado

O MS (*Streaming Multiprocessor*) é uma unidade que contém: os CUDA *cores*; as unidades de função especial (SFUs - do inglês, *Special Function Units*); as unidades de carga/armazenamento de instruções; e uma memória *cache* de nível 1 (L1). Na arquitetura Fermi, cada MS possui 32 CUDA *cores*, 4 SFUs, 16 unidades de carga/armazenamento e uma memória L1 compartilhada com 64 Kbytes. A Figura 34 apresenta a arquitetura de uma unidade MS.

Figura 34 – Unidade MS da NVIDIA



Fonte: STALLINGS (2017)

Como pode ser observado na Figura 34 um MS apresenta 32 CUDA *cores*, sendo que cada um deles possui dois *pipelines*, o primeiro deles um *pipeline* de operações de inteiro (INT) e um segundo para operações de ponto flutuante (FP). Porém, apenas um destes *pipelines* pode ser utilizado em um unidade de *clock*. Já as SFUs, são utilizadas para a realização de operações mais complexas como, por exemplo, operações de cosseno, seno, recíproca e raiz quadrada, sendo que essas operações são realizadas em um único ciclo de *clock* (STALLINGS, 2017).

Além disso, um MS (Figura 34) possui ainda os escalonadores *warp* que operam em conjunto com as unidades de despacho. Essas duas unidades compõem o que a NVIDIA chama de *Single Instruction Multiple Thread (SIMT)*, que é uma técnica para a execução de *threads* em grupos de 32, que são denominados de *warps*. Todas as *threads* de um *warp* executam o mesmo conjunto de instruções, possuindo um contador de programas próprio e o seu conjunto de registradores. Esses dois componentes trabalham em conjunto com o escalonador *GigaThread*, que é o responsável pelo envio de um *warp* para um dos SM disponíveis.

Por fim, como pode ser observado na Figura 33, temos o componente interface de *host*, esse componente é utilizada pela CPU para o envio de instruções e dados para a GPU. Em grande parte das GPUs do mercado, essa interface possui um hardware de acesso direto à memória (DMA), que permite a transferência de dados de modo eficiente entre a memória do sistema *host* e a GPU (HWU; KIRK, 2011). A interface *host* é muito importante para a performance da GPU, pois uma interface com uma baixa taxa de transferência provocará uma queda significativa de performance. O ideal é balancear a capacidade de processamento da GPU com a taxa de frequência do barramento.

3.3.1 Hierarquia de memórias em uma GPU

Um fator significante no desenvolvimento de aplicações para GPUs são os diferentes níveis da hierarquia de memória. Em uma GPU tem-se diversos níveis de memória, com diferentes tamanhos, escopo e tempos de acesso. Segundo Stallings (2017), os níveis de memória que frequentemente se encontram disponíveis em uma arquitetura de GPU são:

- *Registradores*: são as memórias mais rápidas de uma GPU, sendo que cada MS tem o seu respectivo conjunto de registradores. Neste caso, cada *thread* só pode acessar os dados que encontram-se armazenados nos seus respectivos registradores.
- *Memória compartilhada*: localizada dentro de cada MS e pode ser acessada por todas as *threads* de um determinado *warp*. Assim, essas *threads* podem utilizar a memória compartilhada para a troca de informações ou como uma *cache*, sem a necessidade de utilizar a memória principal, reduzindo a latência de acesso a memória (KINDRATENKO, 2014)
- *Memórias locais*: são utilizadas como uma memória secundária dos registradores e como tal, também só podem ser acessadas pela própria *thread*. Esse tipo de memória é utilizada para alocar as variáveis automáticas [2]. Um exemplo de variável automática é quando uma variável possui um tamanho superior ao que pode ser armazenado em um registrador. Neste caso, a memória local é utilizada para armazenar a variável.
- *Memória global (DRAM)*: é a memória principal da GPU, sendo que essa área de memória é compartilhada por todas as *threads* de todos os MS. A memória global é acessível ainda pela CPU.
- *Memória de constante*: é um tipo específico de memória com baixa latência, porém esse tipo de memória só opera no modo de leitura (*read-only*). Esse tipo de memória é de baixa capacidade, sendo que a arquitetura Fermi possui apenas 64 KB e encontra-se junto com a memória global, ou seja, não existe um *chip* específico para ela.

[2] Variáveis automática são variáveis locais que são alocadas e desalocadas automaticamente quando o programa entre em um determinado escopo.

- *Memória de textura:* é uma memória somente de leitura e que é implementada no mesmo espaço físico da memória global e, portanto, a latência de acesso é a mesma da memória global. Essa possui ainda modos de endereçamento diferentes ou filtros para alguns formatos específicos de dados. De fato, essa é otimizada para acesso de dados organizados no formato 1D (dimensão), 2D ou 3D.

Na Tabela 1 tem-se um resumo de todos os tipos de memórias que se encontram presentes na arquitetura de uma GPU, bem como o tempo de acesso e o escopo dessas.

Tabela 1 – Hierarquia de memória em uma GPU

Tipo de memória	Tempos de acesso relativo	Tipo de acesso	Escopo	Vida útil dos dados
Registradores	O mais rápido. On-chip	L/E	*Thread* única	*Thread*
Compartilhada	Rápido. On-chip	L/E	Todas as *threads* em um bloco	Bloco
Local	Entre 100 e 150 vezes mais lento que a compartilhada e o registrador. Off-chip	L/E	*Thread* única	*Thread*
Global	Entre 100 e 150 vezes mais lento que a compartilhada e o registrador. Off-chip	L/E	Todas as *thread* e *host*	Aplicação
Constante	Entre 100 e 150 vezes mais lento que a compartilhada e o registrador. Off-chip	L	Todas as *thread* e *host*	Aplicação
Textura	Entre 100 e 150 vezes mais lento que a compartilhada e o registrador. Off-chip	L	Todas as *thread* e *host*	Aplicação

3.3.2 Estrutura de um programa em CUDA

Em uma aplicação desenvolvida em CUDA, deve-ser ter uma um distinção do trecho de código que é executado pela CPU (*host*) e o que é executado pela GPU (*device*). Normalmente, o *host* fica responsável pela execução das partes seriais do programa e o *device* pela execução das partes de código que são paralelizáveis.

Em CUDA tem-se a possibilidade de criar funções a serem executadas pela GPU, que são denominadas de funções *Kernel*. A definição de uma função *Kernel* é realizada através da palavra chave __global __antes da definição da função. No Algoritmo 1 tem-se um exemplo da definição de uma função *Kernel*.

Algoritmo 1 – Algoritmo de soma de matriz, paralelizado

```
1  __global__ add_matrix
2  (float* a, float* b, float* c, int N){
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      int j = blockIdx.y * blockDim.y + threadIdx.y;
5      int index = i + j * N;
6      if (i < N && j < N)
7          c[index] = a[index] + b[index];
8  }
9  int main (){
10     add_matrix<<<NBLOCKS, NTHREADS>>>>(a,b,c,N);
11 }
```

No momento da definição de uma função *Kernel* é necessário explicitar as configurações de execução, o que implica em informar ao compilador quantos blocos e *threads* serão utilizadas naquela execução. Na linha 10 do Algoritmo 1 pode-se verificar um exemplo de configuração de um *Kernel*. O primeiro parâmetro é referente a quantidade de blocos na execução, já o segundo, a quantidade de *threads* presentes em cada bloco.

O bloco pode ser inicializado com uma, duas ou até três dimensões e corresponde a quantidade de *blocos* que serão utilizados na execução. O segundo parâmetro define a quantidade de *threads* que serão utilizados em um único bloco, isto é, caso eu inicialize o *Kernel* como 3x3 (3 blocos e 3 *threads*), isso implica que a minha função *Kernel* será executada com 9 *threads* operando em concorrência.

Além das palavras chaves __global__ (função *Kernel*) tem-se ainda o qualificador __device __ e qualificador __host __. O qualificador __device__ é utilizado para indicar que a função será executada exclusivamente pela GPU. Já o qualificador __host __define que a função será executada apenas pela CPU.

Além dos qualificadores de funções, tem-se ainda os qualificadores de tipo de variável que são: __device __, __constant __ e __shared __. O qualificador __device __indica que a variável deverá ser alocada na memória principal da GPU, tornando assim a variável acessível a todas as *threads* da execução. O qualificador __constant __é utilizado para alocar variáveis na memória constante. Por fim, tem-se as variáveis __shared __, que são alocadas na memória compartilhada da MS e que são acessíveis por todas as *threads* de um *warp*.

Em CUDA tem-se ainda as variáveis embutidas, que só podem ser utilizadas dentro de um *Kernel*. Essas são frequentemente utilizadas para a divisão do processamento entre as *threads*, pois permite que as *threads* e blocos sejam diferenciados, isto é, funciona como um identificador único. As variáveis embutidas permitem que divisão seja realizada a nível de *thread* ou de bloco. O índice de uma *thread* em um bloco pode ser acessado utilizando as variáveis *threadId.x* e *threadId.y* (para blocos com mais de uma dimensão).

Quando o *Kernel* é inicializado contendo blocos de apenas uma dimensão, tem-se que

variável embutida *threadIdx.x* irá retornar a posição das *threads* dentro do referido bloco. Caso seja criado um bloco com duas dimensões (2x2), a variável *threadIdx.x* irá retornar a posição de uma *thread* em relação à dimensão X, e variável *threadIdx.y* a posição de uma *thread* em relação a dimensão Y. No Algoritmo 2, tem-se um exemplo de uso da variável embutida *threadId.x* (linha 3) para o acesso de dados em um vetor.

Algoritmo 2 – Algoritmo para somar dois vetores em paralelo

```
1
2   __global__ void addWithCuda(int *c, const int *a, const int *b)
3       int i = threadIdx.x;
4       c[i] = a[i] + b[i];
5   }
6   int main()
7   {
8       const int arraySize = 5;
9       const int a[arraySize] = { 1, 2, 3, 4, 5 };
10      const int b[arraySize] = { 10, 20, 30, 40, 50 };
11      int c[arraySize] = { 0 };
12
13      // soma dois vetores em paralelo
14      addWithCuda<<<NBLOCKS, NTHREADS>>>>(c, a, b);
15  }
```

Por fim, pode-se obter as dimensões do bloco através da variável *blockDim*. No caso de um bloco criado em uma única dimensão pode-se obter o tamanho através da variável *blockDim.x*. No caso de um bloco criado em duas dimensões obtém-se as dimensões do bloco através das variáveis *blockDim.x* e *blockDim.y*, como pode ser observado no Algoritmo 3.

Algoritmo 3 – Algoritmo ilustrando a criação de um bloco contendo 10 *threads*

```
1   __global__ void addKernel()
2   {
3       printf("blockDim.x %d", blockDim.x);
4       printf("blockDim.y %d", blockDim.y);
5   }
6
7   int main()
8   {
9       addKernel <<<10, 1 >>> ();
10  }
```

4 IMPLEMENTAÇÃO DESENVOLVIDA

A implementação sequencial foi desenvolvida utilizando a linguagem de programação C++ e o ambiente de desenvolvimento *Microsoft Visual Studio*. Para o desenvolvimento da versão sequencial, inicialmente, foram utilizados recursos nativos da linguagem C++, tais como, *vectors* e *string*. Porém, a plataforma CUDA não apresenta suporte para esses recursos, suportando apenas estruturas nativas da linguagem C. Desta forma, foi necessário substituir esses recursos por outros equivalentes da linguagem de programação C. Por exemplo, a classe *vector* C++ teve que ser substituída por um vetor, bem como os métodos implementados pela classe *vector*, tais como o método de ordenação *merge sort*, tiveram que ser implementados.

4.1 ESTRUTURAS DE DADOS

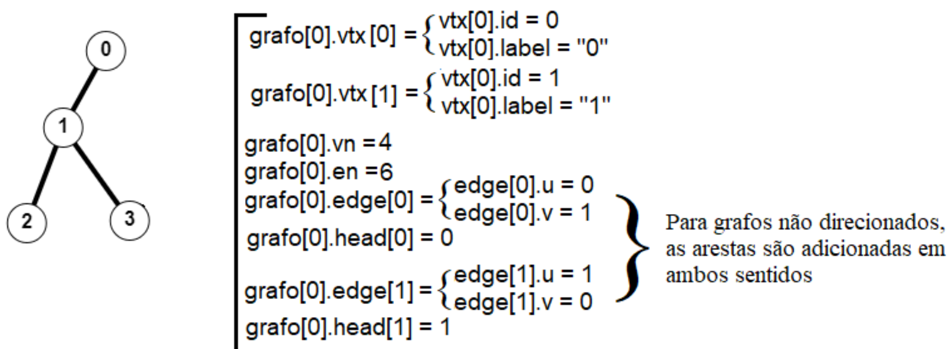
Como pode ser observado no Algoritmo 4, os grafos foram representados por estruturas (*structs*), sendo que cada grafo possui dois *arrays* de *structs*. O primeiro deles representando uma lista de vértices (*Vertex *vtx* - linha 19) e outro representando a lista de arestas (*Edge *edge* - linha 20). Por fim, tem-se um terceiro *array* que é utilizado para manter a relação das arestas que já foram inseridos nos conjuntos pelo método VF2 (*int *head* - linha 16). Essas estruturas foram utilizadas tanto para os grafos modelos, quanto para os grafos alvos.

Algoritmo 4 – Definição das estruturas utilizadas

```
1   struct Vertex
2   {
3       int id;
4       int label;
5   };
6
7   struct Edge
8   {
9       int u;
10      int v;
11      int label;
12      int next;
13  };
14  struct Graph
15  {
16      int *head;
17      int vn; //quantidade de vertices
18      int en; //quantidade de arestas
19      Vertex *vtx;
20      Edge *edge;
21  };
```

Na Figura 35 tem-se um grafo que foi representado utilizando as estruturas implementadas neste trabalho. Como pode ser observado, o grafo *0* é composto por *4* vértices (*vn = 4*) e 6 arestas (*en = 6*). Uma vez que os grafos utilizados nesse trabalho são não-direcionados, as arestas são adicionadas em ambos os sentidos, resultando em *6* arestas. As arestas são armazenadas na estrutura como um vetor (*edge*), onde o campo *u* é referente ao vértice origem e o campo *v* corresponde ao vértice de destino. Os vértices também são armazenadas utilizando um vetor onde o campo *id* é utilizado para identificar o vértice e o campo *label* é utilizado para atribuir um nome ao vértice. Por fim, tem-se vetor denominado *head*, que armazena as arestas que já foram inseridas nos conjuntos pelo método VF2.

Figura 35 – Exemplo da representação de um grafo



4.2 PARALELIZAÇÃO DO MÉTODO VF2

Na abordagem sequencial, para cada grafo alvo, a implementação desenvolvida compara as relações entre os vértices desse grafo com todos os vértices dos grafos modelos, procurando verificar a existência de um algum isomorfismo. No algoritmo 5 tem-se um um pseudocódigo da implementação sequencial.

Algoritmo 5 – Pseudocódigo da versão sequencial

```
1
2      //percorre os grafos a serem processados
3      for(int i = 0; i < sizeQuery;i++)
4          initVariaveisComplementares();
5          //percorre os grafos modelos
6          for (int j = 0; j < sizeDB; j++){
7              if (query(i,j)){
8                  //caso retorne verdadeiro, encontrou um isomorfismo
9                  matches++;
10             }
11         }
12     }
```

Para o desenvolvimento da versão paralelizada, optou-se pelo armazenamento de todos os grafos na memória global da GPU. Para tanto, as estruturas dos grafos foram alocadas na GPU através da função *cudaMalloc* e após, a transferência foi realizada através da função *cudaMemcpy* passando como parâmetro a diretiva *cudaMemcpyHostToDevice* indicando ao compilador de que a transferência é realizada da memória do *host* para a memória da GPU (*device*). Optou-se pelo armazenamento de todos os grafos na memória global da GPU, de forma a evitar o acesso a memória principal do computador, que apresenta um tempo de acesso maior. No Algoritmo 6 tem-se uma exemplificação de como é feito o processo de alocação e transferência dos grafos para a memória principal da GPU.

Algoritmo 6 – Alocação e transferência dos grafos para a memória da GPU

```
1
2   int sVtx = sizeof(Vertex), sEdg = sizeof(Edge), sGph = sizeof(Graph);
3
4   for(k=0;k < graphSize;k++){
5    //aloca os vetores dinâmicos
6    cudaMalloc((void **)&vtx, grafo[k].vn * sVtx);
7    cudaMalloc((void **)&edge, grafo[k].en * sEdg);
8    cudaMalloc((void **)&head, grafo[k].en * 4);
9
10   //transfere
11   cudaMemcpy(vtx,grafo[k].vtx,grafo[k].vn * sVtx,cudaMemcpyHostToDevice);
12   cudaMemcpy(edge,grafo[k].edge,grafo[k].en * sEdg,cudaMemcpyHostToDevice);
13   cudaMemcpy(head,grafo[k].head,grafo[k].en * 4,cudaMemcpyHostToDevice);
14
15   graphHost[k].vtx = vtx;
16   graphHost[k].edge = edge;
17   graphHost[k].head = head;
18  }
19
20  //aloca e transfere o grafo para GPU
21  cudaMalloc((void **)&graphCUDA, graphSize * sGph);
22  cudaMemcpy(graphCUDA,graphHost,(sGph * graphSize),cudaMemcpyHostToDevice);
```

Posteriormente, a função de *Kernel* foi criada utilizando 256 blocos, sendo que cada bloco contém 32 *threads*, totalizando 8192 *threads*. O total de *threads* foi definido considerando o número de grafos modelos que foram utilizados nos testes. De fato, nos testes foram utilizados 8192 grafos modelos sendo que cada grafo modelo é processado independentemente por uma *thread*.

As *threads* foram organizadas utilizando blocos de uma única dimensão. Assim, as *threads* são indexadas utilizando apenas uma dimensão e a a identificação do grafo modelo a ser processado por cada *thread* é realizada a partir de uma combinação do identificador da *thread* (*threadIdx.x*), do identificador do bloco (*blockIdx.x*) e da quantidade de *threads* por bloco (*blockDim.x*), como pode ser observado na linha 3 do Algoritmo 7. No momento que uma determinada *thread* identifica a existência de um isomorfismo (grafo inteiro ou subgrafo), essa armazena o resultado em um vetor que encontra-se armazenado na memória global da GPU (vetor *dev_matches* - linha 11). Esse vetor é inicializado no *host* e é compartilhado por todas as *threads*. De forma, a evitar problemas concorrência, o acesso a esse vetor é realizado através de uma operação atômica. No Algoritmo 7 tem-se um pseudo-código da paralelização realizada.

Algoritmo 7 – Pseudocódigo da versão paralelizada utilizando CUDA

```
1
2   //variável que armazena o ID único da thread
3   int init = threadIdx.x + blockIdx.x * blockDim.x;
4
5   while(controle[init] < sizeQuery){
6       int j = controle[init]
7       initVariaveisComplementares();
8       //percorre os grafos modelos
9       for (int x = init; x < sizeDB; x += NTHREADS * NBLOCKS){
10          if (query(queryGraph, modelGraph)){
11              atomicAdd(&dev_matches[j],1);
12          }
13      }
14      controle[init]++;
15  }
```

Por fim, o vetor de resultados *dev_matches* é transferido para a memória do *host* e as estruturas são desalocadas. Como pode ser observado na linha 6 do Algoritmo 8 a transferência é realizada através da função *cudaMemcpy* utilizando a diretiva *cudaMemcpyDeviceToHost* indicando que a transferência será feita do *device* (GPU) para o *host* (CPU) e as estruturas são desalocadas utilizando a função *cudaFree*, conforme linha 7 do Algoritmo 8.

Algoritmo 8 – Algoritmo ilustrando a transferência de dados entre GPU e CPU

```
1   //chama método paralelizado
2   solve <<< NBLOCKS,NTHREADS >>>(queryGraphCUDA,dbGraphCUDA,matchesCUDA);
3   cudaDeviceSynchronize();
4
5   int TAM = sizeQuery * sizeof(int);
6   cudaMemcpy(matches,matchesCUDA,TAM,cudaMemcpyDeviceToHost);
7   cudaFree(queryGraphCUDA);
8   cudaFree(dbGraphCUDA);
9   cudaFree(matchesCUDA);
```

4.3 TESTES REALIZADOS E RESULTADOS OBTIDOS

Todos os testes foram realizados utilizando um computador com um processador Intel Core i5-9300H contendo 4 núcleos de processamento operando a uma frequência de 2.4 GHZ. Este processador possui 256 KB de cache L1, 1 MB de cache L2 e 8 MB de cache L3 . Além disso, esse computador apresenta uma memória RAM DDR4 de 8 GB operando a 2666 MHz e um disco rígido SSD de 120 GB. O sistema operacional utilizado é o Windows 10 Home de 64 bits.

Para o desenvolvimento e para a realização de testes foi utilizada uma placa de vídeo GTX 1650 da NVIDIA. A placa possui 896 CUDA *cores* contidos em 14 *Streaming Multiprocessor* (MS) e 56 unidades de textura. A placa de vídeo possui 4 GB de memória principal GDDR5 operando a uma frequência de 2000 MHZ, podendo atingir 8000 MHZ devido ao uso tecnologia GDDR. A placa possui ainda uma interface de memória 128 bits com largura de banda de 128 GB por segundo. Além disso, essa possui 64 KB reservados para a memória constante e 48 KB de memória compartilhada. Na Figura 36 tem-se uma imagem da placa de vídeo GTX 1650 da NVIDIA.

Figura 36 – NVIDIA GTX 1650

4.3.1 Grafos utilizados para testes

Os grafos utilizados para testes foram gerados a partir do algoritmo desenvolvido por FOGGIA; SANSONE; VENTO (2001a). Esse algoritmo foi desenvolvido especificamente para a geração de grafos que podem ser utilizados em estudos envolvendo problemas de *graph matching*. Desta forma, a partir de parâmetros iniciais, são gerados dois grafos, sendo um grafo de modelo e outro o grafo alvo.

Neste trabalho foram realizados testes utilizando 3 tamanhos distintos de grafos, sendo os grafos pequenos com 4 vértices, os grafos medianos com 8 vértices e os grafos grandes contendo 12 vértices (Tabela 2). Para cada tamanho de grafos foram gerados 10 grafos diferentes, sendo que cada grafo foi comparado com 8192 grafos modelos. Os grafos modelos apresentam um número de vértices variando entre 1 e 40. Os grafos modelos são maiores devido ao interesse também na identificação do isomorfismo em subgrafos.

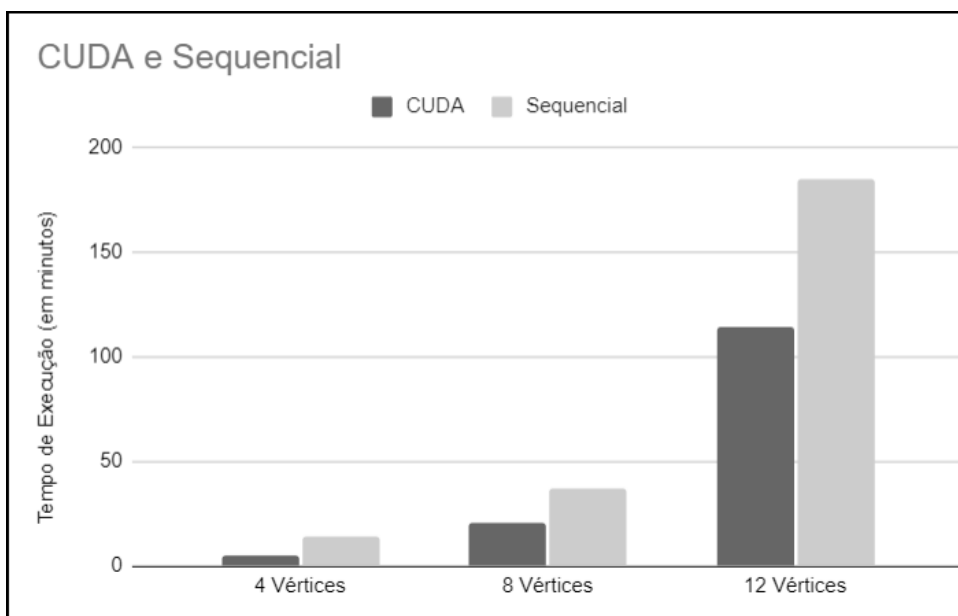
Tabela 2 – Tamanho dos grafos

Tipo	**Quantidade**
Grafos pequenos	4 vértices
Grafos medianos	8 vértices
Grafos grandes	12 vértices

4.3.2 Resultados obtidos

Na Figura 37 tem-se um comparativo dos tempos de execução do método VF2 utilizando grafos com 4 vértices, 8 vértices e 12 vértices. Para cada grafo foram realizadas 5 execuções e tempo de execução foi obtido a partir da média aritmética dessas 5 execuções. Como pode ser observado, a implementação paralela em GPU, apresenta um tempo de execução inferior ao tempo de execução do VF2 em CPU, porém o desempenho da aplicação vai decaindo com o aumento no número de vértices dos grafos. De fato, para os grafos pequenos a implementação foi 1,66 vezes mais rápida. Já para grafos médios e grandes a implementação foi 1,44 e 1,38 vezes mais rápida, respectivamente.

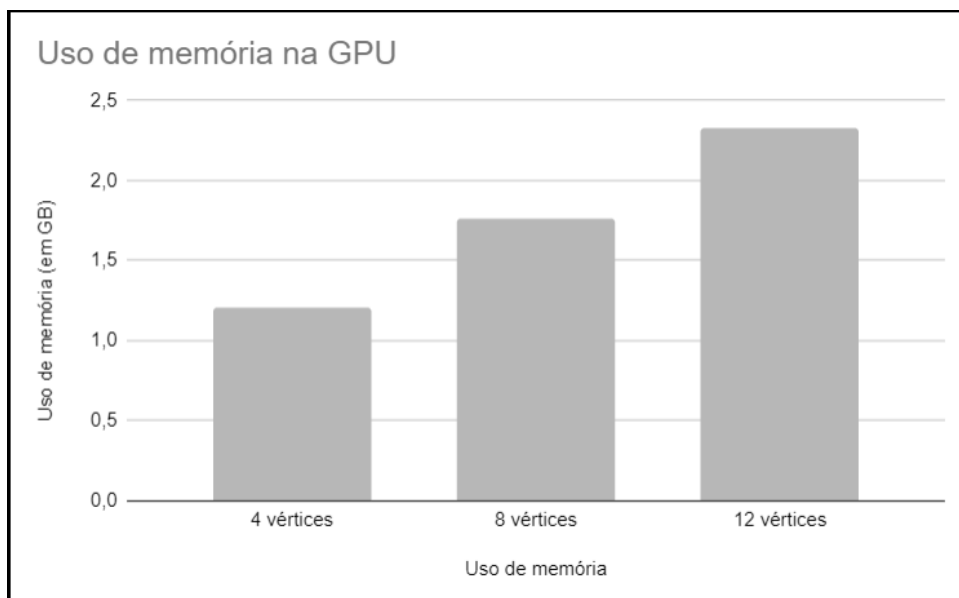
Figura 37 – Tempo de Execução do Método VF2



A queda de desempenho da versão paralelizada em CUDA com o aumento no número de vértices, deve-se ao fato que implementação do VF2, baseia-se em um algoritmo de busca em profundidade (DFS), sendo que este é comumente implementado utilizando recursão, como é o caso neste trabalho. Algoritmos recursivos utilizam uma estrutura de pilha para o armazenamento das informações de iterações anteriores. No caso da programação em GPU, a pilha é armazenada na memória local GPU, que apresenta uma tamanho limitado. Em problemas maiores (níveis maiores de recursão), quando essa memória é totalmente utilizada, é realizada uma transferência para a memória global da GPU, que apresenta uma maior latência de acesso e, consequentemente, uma perda de desempenho da aplicação.

Na Figura 38 pode-se observar o uso de memória global da GPU em função do tamanho dos grafos. No caso de programas utilizando GPU, o tamanho da pilha de execução deve ser definido explicitamente pelo programador no início da execução. Desta forma, na Figura 38 está sendo considerado somente a quantidade de memória global utilizada para a pilha de execução, ou seja, não está sendo considerado o espaço de memória consumido para o armazenamento dos grafos. Como pode ser observado, a quantidade utilizada pela pilha de execução aumenta com o aumento no número de vértices dos grafos.

Figura 38 – Uso de memória na GPU



5 CONSIDERAÇÕES FINAIS

O problema de *graph matching* é um problema complexo e amplamente discutido na literatura devido a sua importância e utilização na representação da informação. De fato, esse tem sido utilizado nas mais diversas áreas, como por exemplo, reconhecimento de imagens, de padrões, biologia, entre outras.

A forma mais simples para resolver o problema *graph matching* seria testar todas as combinações possíveis, porém este tipo de abordagem tem custo fatorial ($n!$), sendo impraticável para grafos com um grande número de vértices e arestas. Desta forma, as heurísticas visam reduzir o espaço de busca, limitando as possíveis combinações, porém não apresentam uma garantia de encontrar a solução ótima do problema.

Dentre as heurísticas disponíveis para resolver o problema de *graph matching* destacam-se o algoritmo de Ullmann, o VF2 e Nauty. Porém, o algoritmo de Ullmann apresenta tempos de execução muito elevados para grafos grandes (acima de 700 vértices). Já o VF2 e Nauty apresentam tempos aceitáveis e na mesma ordem de grandeza. Desta forma, optou-se pela utilização do método VF2, uma vez que esse pode ser utilizado para encontrar o isomorfismo em grafos e em subgrafos, ao contrário do Nauty que só pode ser utilizado para encontrar o isomorfismo de grafos.

A implementação do VF2 foi realizada de forma explorar o paralelismo em arquiteturas do tipo SIMD (*Single Instruction Multiple Data*), mais precisamente para ser executada em GPUs (Unidade Gráfica de Processamento), que encontram-se presentes em um grande número de computadores pessoais. Para o desenvolvimento foi utilizada a plataforma CUDA (CUDA, 2020), que foi desenvolvida pela NVIDIA e é uma plataforma de computação de propósito geral para GPUs.

A partir de testes realizados observou-se que a versão paralela em GPU, apresenta um tempo de execução inferior a versão desenvolvida para CPU. Porém, o desempenho da versão paralela cai com o aumento no número de vértices dos grafos. Essa queda de desempenho deve-se ao fato da implementação desenvolvida neste trabalho ser recursiva. No caso de grafos menores a pilha de execução pode ser quase que completamente armazenada na memória local GPU. Já em caso de grafos maiores, tem-se níveis maiores de recursão, fazendo que a memória local seja totalmente utilizada, sendo necessário a transferência da pilha de execução para memória global da GPU, que apresenta um tempo maior de acesso. Desta forma, o frequente acesso à memória principal da GPU provocou uma significativa redução no desempenho da versão paralelizada para GPU.

5.1 SUGESTÃO DE TRABALHOS FUTUROS

Como trabalhos futuros sugere-se:

- Comparação dos resultados obtidos utilizando abordagens paralelas tradicionais, como por exemplo, a biblioteca de *threads* OpenMP.
- Implementação do algoritmo VF2 utilizando uma abordagem iterativa.
- Paralelização do algoritmo *Nauty* utilizando CUDA.

REFERÊNCIAS

ABD-EL-BARR, M.; EL-REWINI, H. **Fundamentals of computer organization and architecture**. [S.l.]: John Wiley & Sons, 2005. v. 38.

AMORIM, P. H. J.; ARAÚJO, F. R. da S. Gpu: A matriz de processadores. p. 1–5, 2011.

BASU, S. **PARALLEL AND DISTRIBUTED COMPUTING : ARCHITECTURES AND ALGORITHMS**. PHI Learning, 2016. (Eastern economy edition). ISBN 9788120352124. Disponível em: <https://books.google.com.br/books?id=5t8PDAAAQBAJ>.

BENGOETXEA, E. **The graph matching problem**. Tese (Doutorado) — Ecole Nationale Supérieure des Télécommunications, Paris, France, 2002.

CARLETTI, V. **Exact and Inexact Methods for Graph Similarity in Structural Pattern Recognition PhD thesis of Vincenzo Carletti.** Tese (Doutorado), 2016.

CHAPEL, S. 2016. (Acessado: 13.06.2020). Disponível em: <https://commons.wikimedia.org/wiki/File:Injection.svg>.

COMBA, J. L. D. **Backtracking**. 2018. Acesso: 31.05.2020. Disponível em: <http://www.inf.ufrgs.br/~comba/inf1056-files/class07.pdf>.

CORDELLA, L. P. *et al.* An improved algorithm for matching large graphs. In: **In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen**. [S.l.: s.n.], 2001. p. 149–159.

CUDA. 2020. (Acessado: 19.04.2020). Disponível em: <https://developer.nvidia.com/cuda-zone>.

DAHLBERG, C. **Speeding up matrix computation kernels by sharing vector coprocessor among multiple cores on chip**. Dissertação (Mestrado) — Jönköping University, JTH, Computer and Electrical Engineering, 2012.

ESPARRACHIARI, S.; GOMES, V. H. P. Um tutorial sobre gpus. São Paulo, Brazil, 2008.

FOGGIA, P.; SANSONE, C.; VENTO, M. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In: **CoRR**. [S.l.: s.n.], 2001. p. 176–187.

____. A performance comparison of five algorithms for graph isomorphism. In: **Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition**. [S.l.: s.n.], 2001. p. 188–199.

HARTKE, S.; RADCLIFFE, A. Mckay's canonical graph labeling algorithm. **Contemporary Mathematics book series**, v. 479, 02 2013.

HOPCROFT, J. E.; WONG, J. K. Linear time algorithm for isomorphism of planar graphs (preliminary report). In: **Proceedings of the Sixth Annual ACM Symposium on Theory of Computing**. New York, NY, USA: Association for Computing Machinery, 1974. (STOC '74), p. 172–184. ISBN 9781450374231. Disponível em: <https://doi.org/10.1145/800119.803896>.

HWU, W.-M. W.; KIRK, D. B. **PROGRAMANDO PARA PROCESSADORES PARALELOS**. ELSEVIER EDITORA, 2011. ISBN 9788535241884. Disponível em: <https://books.google.com.br/books?id=tZe5D674mp4C>.

JüTTNER, A.; MADARASI, P. Vf2++—an improved subgraph isomorphism algorithm. **Discrete Applied Mathematics**, v. 242, p. 69 – 81, 2018. ISSN 0166-218X. Computational Advances in Combinatorial Optimization. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0166218X18300829>.

KINDRATENKO, V. **Numerical Computations with GPUs**. Springer International Publishing, 2014. ISBN 9783319065489. Disponível em: <https://books.google.com.br/books?id=CbH0AwAAQBAJ>.

KöBLER UWE SCHöNING, J. T. a. J. **The Graph Isomorphism Problem: Its Structural Complexity**. 1. ed. [S.l.]: Birkhäuser Basel, 1993. (Progress in Theoretical Computer Science). ISBN 978-1-4612-6712-6,978-1-4612-0333-9.

LÓPEZ-PRESA, J. L.; ANTA, A. F. Fast algorithm for graph isomorphism testing. In: VAHRENHOLD, J. (Ed.). **Experimental Algorithms**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 221–232. ISBN 978-3-642-02011-7.

MAKSOV, A.; LI, Y.; BUTLER, R. Subgraph isomorphism. 2015. (Acessado: 26.04.2020). Disponível em: <http://web.eecs.utk.edu/~cphill25/cs594_spring2015_projects/subgraph_isomorphism.pdf>.

MCKAY, B. D.; PIPERNO, A. **Practical graph isomorphism, II**. 2013.

____. **nauty and Traces User's Guide**. 2019. (Acessado: 16.04.2020). Disponível em: <http://pallini.di.uniroma1.it/nug27.pdf>.

Moore, G. E. Cramming more components onto integrated circuits, reprinted from electronics. **IEEE Solid-State Circuits Society Newsletter**, v. 38, p. 114, 1965.

MOTTIN, D.; LAZARIDOU, K. Querying graphs. 2016. (Acessado: 26.04.2020). Disponível em: <https://hpi.de/fileadmin/user_upload/fachgebiete/mueller/courses/graphmining/GraphMIning-03-Querying-Graphs.pdf>.

OMNISCI. **What is a GPU and Why Do I Care? A Businessperson's Guide**. 2017. Acesso: 04.04.2020. Disponível em: <https://https://www.omnisci.com/blog/what-is-a-gpu-and-why-do-i-care-a-businesspersons-guide-2>.

OPENCL. 2020. (Acessado: 14.06.2020). Disponível em: <https://www.khronos.org/opencl/>.

PEARL, J. Heuristics: Intelligent search strategies for computer problem solving. 1 1984.

RIESEN, K.; JIANG, X.; BUNKE, H. Exact and inexact graph matching: Methodology and applications. In: **Managing and Mining Graph Data**. Springer US, 2010. p. 217–247. Disponível em: <https://doi.org/10.1007%2F978-1-4419-6045-0_7>.

RODRIGUES, D. B. **Teoria Espectral e o Problema de Isomorfismo de Grafos Regulares**. 84 p. — UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO, Vitória, 2011.

STALLINGS, W. **Arquitetura e organização de computadores**. [S.l.]: Editora Pearson, 2017. v. 10. ISBN 9788543020532.

SUMSI, M. F. **Theory and Algorithms on the Median Graph. Application to Graph-bases Classification and Clustering**. Tese (Doutorado) — Universitat Autònoma de Barcelona, Bellaterra, Spain, 2008.

TABAK, P. **DISTANCE BASED CANONICAL LABELING ALGORITHMS WITH APPLICATIONS TO GRAPH MATCHING**. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, 2020.

ULLMANN, J. R. An algorithm for subgraph isomorphism. **J. ACM**, Association for Computing Machinery, New York, NY, USA, v. 23, n. 1, p. 31–42, jan. 1976. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/321921.321925>.