CURSO DE CIÊNCIA DA COMPUTAÇÃO – TCC	
( ) PRÉ-PROJETO (X) PROJETO	ANO/SEMESTRE: 2023/1

OTIMIZAÇÃO EM PROCESSAMENTO DE GRAFOS UTILIZANDO PARALELISMO EM GPU E RUST

Igor Christofer Eisenhut

Prof. Aurélio Faustino Hoppe – Orientador

1 INTRODUÇÃO

De acordo com Hariri, Fredericks e Bowers (2019), o surgimento de novos recursos tecnológicos, como a internet das coisas, redes sociais e cidades inteligentes, resultou em uma crescente geração de dados. Estima-se que apenas em 2018 a quantidade de dados trafegados na internet diariamente ultrapassaram 2,5 quintilhões de bytes. Ainda segundo os autores, o processamento e armazenamento desses enormes conjuntos de dados ganhou, em 2011, o nome de *big data*, que, além de descrever o grande volume de dados, também compreende os crescentes números no que se refere à variedade, velocidade, valor e veracidade dos dados. Nesse contexto, a variedade se refere às inúmeras maneiras com que os dados de *big data* podem ser apresentados: (i) dados com uma estrutura conhecida e que podem ser facilmente armazenados em um banco de dados relacional; (ii) dados semiestruturados, que possuem elementos conhecidos, mas que podem apresentar variações de estrutura; (iii) dados não estruturados, como conteúdos multimídia, que não apresentam padronização alguma (HARIRI; FREDERICKS; BOWERS, 2019).

Santos *et al*. (2022) ressaltam que nas últimas décadas houve muitos esforços para transformar dados brutos em informações relevantes, o que resultou na organização de grandes estruturas de grafos compostas por nós e arestas. Nesses grafos, os nós representam entidades e as arestas representam os relacionamentos entre elas. Na área da saúde, por exemplo, a organização via grafos tem várias aplicações, tais como a representação de interações moleculares, vias de sinalização celular, comorbidades de doenças e sistemas de saúde (LI; HUANG; ZITNIK, 2022). Além da medicina, as estruturas de grafos também são utilizadas no meio eletrônico para representar relacionamentos e interações entre pessoas em uma rede social (JIANG *et al*., 2021), no meio linguístico para representação da estrutura sintática e semântica de objetos em uma sentença (RIBEIRO *et al*., 2020), na robótica para representação de redes de comunicação entre robôs (LI *et al*., 2020), entre outros.

Ainda segundo Santos *et al*. (2022), utilizando grafos, torna-se possível uma série de análises e a extração do conhecimento inerente ao conjunto de dados. Porém, de acordo com Dafir, Lamari e Slaoui (2020), os algoritmos convencionais não são capazes de lidar com grandes quantidades de dados de maneira eficiente. Para contornar esse problema, uma solução encontrada é o processamento na Graphics Processing Unit (GPU), que oferece alta capacidade de processamento em paralelo e aceleração de algoritmos intensivos em cálculos de ponto flutuante. Porém, as GPUs possuem uma capacidade de memória limitada, requerendo uma abordagem complexa para seu gerenciamento (DAFIR; LAMARI; SLAOUI, 2020).

Yamato (2021) destaca que a maioria das aplicações que realizam processamento em GPU são escritas nas linguagens de programação C/C++. Ou seja, a escolha da linguagem de programação, neste caso, também pode influenciar na eficiência do processamento de grandes massas de dados. Nesse sentido, a linguagem Rust tem se destacado como uma opção interessante para a programação de sistemas que lidam com grandes volumes de dados. Segundo Bugden e Alahmar (2022), Rust é uma linguagem de programação que oferece um alto desempenho e segurança de memória, além de ser projetada para suportar concorrência e paralelismo.

Viitanen (2020) ressalta que a combinação de grafos, GPU e Rust pode trazer muitas vantagens em relação ao processamento de grandes massas de dados. Os grafos podem ser usados para modelar dados complexos, a GPU pode executar várias tarefas simultaneamente, enquanto Rust pode garantir a segurança e eficiência do código, reduzindo o tempo de desenvolvimento e permitindo o controle preciso do *hardware* utilizado no processamento. Sendo assim, a partir deste contexto, este trabalho possui a seguinte pergunta de pesquisa: o paralelismo em GPU juntamente com a utilização da linguagem de programação Rust podem otimizar o processamento de grandes grafos?

1.1 OBJETIVOS

O objetivo deste trabalho é desenvolver uma ferramenta que otimize o processamento de grandes grafos por meio da utilização do paralelismo em GPU e da linguagem de programação Rust.

Os objetivos específicos são:

a) identificar as limitações de desempenho no processamento de grandes grafos utilizando técnicas convencionais por meio da análise de métricas como tempo de execução e consumo de memória;

b) analisar a utilização de paralelismo em GPU no processamento de grandes grafos, por meio da arquitetura, tecnologias e bibliotecas disponíveis para programação em GPU;
c) avaliar o desempenho da linguagem de programação Rust no processamento de grandes grafos, por meio da implementação de algoritmos em Rust, da comparação do desempenho com outras linguagens de programação utilizadas para processamento de grafos (CUDA, OpenCL e Rust GPU) e da análise de métricas como tempo de execução e consumo de memória.

2 TRABALHOS CORRELATOS

Nesta seção serão apresentados os trabalhos cujo tema se relaciona ao objetivo deste projeto. A subseção 2.1 aborda o trabalho de Zarebavani *et al*. (2019), no qual foram desenvolvidas duas abordagens para execução em GPU do algoritmo PC. A subseção 2.2 apresenta o *framework* híbrido desenvolvido por Wang *et al.* (2019) para a obtenção do caminho mais curto em grafos utilizando a GPU. E, a subseção 2.3 descreve o Pangolin, um *framework* extensível para mineração de padrões em grafos utilizando a CPU e a GPU (CHEN *et al*., 2020).

2.1 CUPC: CUDA-BASED PARALLEL PC ALGORITHM FOR CAUSAL STRUCTURE LEARNING ON GPU

Zarebavani *et al.* (2019) abordam a inexistência de uma implementação completa do algoritmo PC em GPU. Segundo os autores, a implementação existente e mais avançada até então, denominada Parallel-PC, implementa o paralelismo em GPU de forma parcial e não pode ser considerada uma solução completa para *datasets* complexos. Os autores propõem o CUDA-Accelerated PC Algorithm (cuPC), uma implementação do algoritmo PC que realiza de forma completa o paralelismo em GPU.

Segundo Zarebavani *et al.* (2019), o núcleo da aprendizagem de estrutura causal baseia-se na execução de testes de Independência Condicional (IC) sobre cada aresta (`Vi,Vj`) da rede bayesiana contra um segundo grupo de vértices `S`. Caso o resultado do teste seja positivo, os vértices `Vi` e `Vj` são condicionalmente independentes e a aresta (`Vi,Vj`) que liga ambos é removida, sendo executados em níveis consecutivos. Dessa forma, os autores propuseram dois algoritmos utilizando a *Application Programming Interface* (API) CUDA para GPUs da Nvidia. As duas variações do algoritmo proposto, chamado CUDA-Accelerated PC Algorithm (cuPC), são o cuPC-E e o cuPC-S, desenvolvidos na linguagem de programação C. O cuPC-E baseia-se em dois níveis de paralelismo: (i) processando todas as arestas do nível em paralelo e, (ii) para cada aresta, executando os testes IC paralelamente em um número pré-determinado de *threads*. Caso uma aresta seja removida, o algoritmo ignora os demais testes a serem executados a partir desta aresta. De maneira semelhante, o cuPC-S baseia-se nos mesmos princípios do cuPC-E com a adição de que a matriz pseudo-inversa de cada conjunto `S`, necessária para realizar os testes IC, é computada uma única vez. Assim, de acordo com os autores, ao direcionar todos os testes IC que dependem de um mesmo conjunto `S` da *thread*, é possível evitar cálculos redundantes e acelerar o processamento ao compartilhar a matriz pseudo-inversa entre os testes.

Zarebavani *et al.* (2019) destacam que apesar das diferentes abordagens dos dois algoritmos, os cálculos do nível 0, onde `S` é um conjunto vazio, são executados por uma mesma rotina compartilhada. Quando `S` for igual a vazio significa que não existe uma matriz pseudo-inversa a ser compartilhada entre os testes IC, anulando os ganhos obtidos pela estratégia implementada no cuPC-S. Da mesma forma, a estratégia implementada no cuPC-E também não se justifica com `S` sendo um conjunto vazio, pois é necessário executar apenas um teste para cada aresta da rede, os quais são executados de forma independente em *threads* separadas.

Segundo Zarebavani *et al.* (2019), foram realizados testes comparando o cuPC contra 3 algoritmos existentes. Ou seja, duas implementações seriais do algoritmo PC-stable, que não fazem uso da paralelização. Neste caso, a implementação original do algoritmo escrita em R denominada Stable e a implementação mais recente denominada Stable.fast, escrita em C. A terceira implementação utilizada foi a Parallel-PC, que possibilita a paralelização do processamento. Os testes foram executados em uma máquina com processador Intel Xeon de 8 núcleos rodando a 2,5 GHz e uma placa de vídeo Nvidia GTX 1080. Além disso, também utilizou-se o sistema operacional Ubuntu 16.04, o compilador gcc na versão 5.4 e a API CUDA na versão 9.2. Os algoritmos que não permitiam a paralelização foram executados em um único núcleo da CPU e os demais em 8 núcleos. De acordo com os autores, nas validações, foram utilizados 6 *datasets* diferentes: NCI-60, MCC, BR-51, S.cerevisiae, S.aureus e DREAM5-Insilico, sobre os quais os algoritmos deveriam encontrar a estrutura causal.

A partir dos testes, Zarebavani *et al.* (2019) concluíram que as duas implementações propostas do cuPC foram mais performáticas que as existentes. Além disso, dentre os resultados obtidos, o processamento do *dataset* DREAM5-Insilico, no algoritmo sequencial Stable, levou 265.360 segundos (cerca de 3 dias) para ser completado, enquanto no cuPC-E levou 48,08 segundos e no cuPC-S 4,09 segundos. A Figura 1 apresenta de forma detalhada os tempos de execução de cada algoritmo/*dataset*. Pode-se perceber que nas implementações propostas e existentes, o processamento foi em média 1.296 vezes mais rápido utilizando o cuPC-S e 525 vezes mais rápido utilizando o cuPC-E.

Figura 1 – Resultados obtidos com os testes

			NCI-60	MCC	BR-51	S.cerevisiae	S.aureus	DREAM5-Insilico	
Runtime (sec.)	Stable (R, Single Core)	T1	646	3,522	3,118	10,568	11,324	265,360 (~3 days)	
	Parallel-PC (R, 8 Cores)	T2	102	570	549	2,847	1,920	39,880 (~11 hours)	
	Stable.fast (C, Single Core)	T3	74	510	491	5,567	3,359	41,668	
	cuPC-E	T4	0.44	0.85	1.15	7.99	4.21	48.08	
	cuPC-S	T5	0.39	0.44	0.56	4.76	1.64	4.09	
Speedup Ratio	Parallel-PC	T1/T2	6.3	6.2	5.7	3.7	5.9	6.7	Mean = 5.6
	cuPC-E	T3/T4	171	600	425	697	799	867	Mean = 525
	cuPC-S	T3/T5	193	1,157	868	1,170	2,052	10,178	Mean = 1,296

Fonte: Zarebavani *et al.* (2019).

Zarebavani *et al*. (2019) também realizaram testes de escalabilidade, aumentando gradativamente a quantidade de vértices, a quantidade de amostras obtidas da rede e a densidade do grafo. Os testes foram realizados de forma independente e demonstraram que ao incrementar qualquer uma das três variáveis, o tempo de processamento aumenta linearmente para o cuPC-E e o cuPC-S, enquanto a implementação Stable.fast falhou em produzir resultados após 48 horas mesmo em grafos menores. Por fim, os autores introduzem o conceito de bloco, isto é, estruturas que compõem o *kernel* da GPU e agrupam as *threads*, e concluem que o bom desempenho dos algoritmos ocorre por conta da configuração referente ao número de blocos (β) e arestas por bloco (γ) processadas simultaneamente, no caso do cuPC-E, e ao número de blocos (δ) e *threads* por bloco (θ) no caso do cuPC-S. Nos cenários observados, as configurações mais adequadas foram $\beta = 2$ e $\gamma = 32$ para o cuPC-E e $\delta = 2$ e $\theta = 64$ para o cuPC-S. Isso ocorre pois, conforme constatado por Zarebavani *et al.* (2019), em grafos mais densos a quantidade de testes IC necessária se eleva, beneficiando o maior nível de paralelização, ao passo que em grafos mais esparsos a quantidade necessária de testes diminui, resultando em validações desnecessárias e menor performance caso haja um elevado grau de paralelismo.

2.2 SEP-GRAPH: FINDING SHORTEST EXECUTION PATHS FOR GRAPH PROCESSING UNDER A HYBRID FRAMEWORK ON GPU

De acordo com Wang *et al.* (2019), a performance do processamento paralelo de grafos é determinada por três fatores principais: (i) o modo de execução (síncrono ou assíncrono); (ii) o método de comunicação (*pull* ou *push*); e (iii) a maneira de travessia do grafo (orientado a dados ou orientado a topologia). Porém, os *frameworks* de percorrimento de grafos existentes utilizam apenas um dos fatores de maneira fixa para realizar o processamento. Wang *et al.* (2019) também visavam explorar duas questões fundamentais: (i) é possível obter a causa raiz que leva a uma variação no tempo de execução de acordo com o modo de execução, comunicação e travessia do grafo? E, com base nessa causa raiz, (ii) é possível criar um *framework* leve o suficiente para realizar a troca da metodologia em tempo de execução de forma viável? Com base nisso, os autores estudaram as principais propriedades dos tipos de algoritmos iterativos e de travessia e verificaram que as melhores combinações para algoritmos iterativos são: execução síncrona, método de comunicação *pull* e travessia orientada a dados; e execução assíncrona, método de comunicação *push* e com a orientação da travessia sendo indiferente. Para algoritmos de travessia, as combinações que mais beneficiarão a execução na GPU são aquelas que envolvem processamento assíncrono e percorrimento orientado a dados.

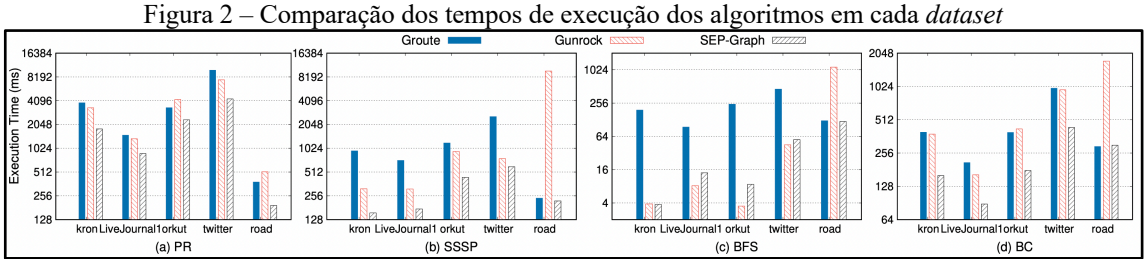
Wang *et al.* (2019) desenvolveram o *framework* SEP-Graph utilizando a linguagem C/C++ para a execução de algoritmos de grafos de forma híbrida, baseando-se em um sistema com duas partes principais: (i) o motor de percorrimento de grafos que é executado na GPU e o (ii) controlador que é executado na *Central Processing Unit* (CPU). O controlador coleta métricas de execução, como por exemplo, o número de vértices ativos e o total acumulado dos graus de entrada e saída dos vértices ativos. E, a partir dessas métricas, realiza a troca da configuração do motor em tempo de execução.

Segundo Wang *et al.* (2019), foram utilizados os *datasets* road_usa, kron_g500-logn21, soc-LiveJournal1, soc-orkut e soc-twitter-21. O road_usa é um grafo esparso com grande diâmetro e os demais são grafos cujo grau de entrada e saída dos vértices obedece a distribuição da lei de força. Como algoritmos iterativos para testes, foram utilizados os o PageRank (PR) e o Single-Source Shortest Path (SSSP). Quanto aos algoritmos de travessia foram utilizados o Breadth-First Search (BFS) e o Betweenness Centrality (BC). Os testes foram conduzidos executando os algoritmos em diferentes GPUs e realizando uma comparação dos tempos de execução do SEP-Graph com os *frameworks* Groute e Gunrock. Segundo os autores, a maior diferença entre ambos é que o Groute utiliza o modo de execução assíncrono e o Gunrock síncrono, sendo que para cada algoritmo suportado, as configurações referentes ao método de comunicação e travessia são fixas.

De acordo com Wang *et al.* (2019), o SEP-Graph apresentou um ganho de performance de até 2,9x no algoritmo PR, até 39,4x no algoritmo SSSP, até 45,8x no algoritmo BFS e de até 5,8x no algoritmo BC. Por mais que, de modo geral, o SEP-Graph tenha se sobressaído em relação aos demais, existem casos em que a performance dele foi inferior, como na execução do BFS onde o Gunrock utiliza a mesma abordagem que o SEP-Graph mas com otimizações adicionais, atingindo uma performance 2,4x melhor. Nesse caso, segundo Wang *et al.* (2019), a

validação da melhor abordagem a cada mudança nos parâmetros do grafo atuou como um gargalo para o SEP-Graph. A Figura 2 apresenta os tempos em cada *dataset*.

Figura 2 – Comparação dos tempos de execução dos algoritmos em cada *dataset*



Fonte: Wang *et al.* (2019).

Por fim, Wang *et al*. (2019) concluem que a exploração de heurísticas para determinar as melhores estratégias de processamento são benéficas para o desempenho dos algoritmos. Através disso, o SEP-Graph conseguiu apresentar tempos de execução significativamente menores em relação aos *frameworks* existentes. Porém, o tempo de avaliação das heurísticas para determinar as estratégias de processamento não é desprezível e pode representar um ponto de gargalo quando o sistema proposto é comparado à sistemas que já apresentam a maneira de processamento mais otimizada para determinado cenário.

2.3 PANGOLIN: AN EFFICIENT AND FLEXIBLE GRAPH MINING SYSTEM ON CPU AND GPU

Segundo Chen *et al.* (2020), os *frameworks* de Mineração de Padrões em Grafos (MPG) possuem soluções genéricas e não são performáticos o suficiente para serem utilizados em cenários comerciais. Eles focam principalmente em disponibilizar abstrações que facilitam a programação. As aplicações desenvolvidas, utilizando um determinado algoritmo, apresentam um ganho enorme de performance, porém são implementações complexas, pois precisam gerenciar o paralelismo, sincronização e gerenciamento de memória. A partir deste cenário, Chen *et al.* (2020) desenvolveram um *framework* genérico para facilitar a implementação de código de domínios específicos a fim de se equiparar à performance obtida em aplicações de cunho específico.

Chen *et al.* (2020), denominaram o *framework* de Pangolin, ao qual permite a execução de 4 algoritmos: Contagem de Triângulos (CT), Busca de Cliques (BC), *Motif Counting* (MC) e Mineração de Subgrafo Frequente (MSF). Segundo os autores, o Pangolin, que é desenvolvido na linguagem de programação C++, implementa uma série de otimizações a fim de melhorar o processamento dos algoritmos sem comprometer sua flexibilidade. Dessa forma, a API do *framework* permite a extensão das funções toAdd, toExtend, getPattern, getSupport, Aggregate e toDiscard, possibilitando a customização de soluções específicas para cada algoritmo, visando otimizar seus tempos de execução. Além disso, segundo Chen *et al.* (2020), o Pangolin pode executar o processamento MPG tanto na CPU, quanto na GPU e, diferentemente dos *frameworks* existentes como o RStream, que salva os subgrafos resultantes em disco, realiza o processamento do modelo em memória, melhorando o desempenho.

Chen *et al.* (2020) compararam o Pangolim com os *frameworks* Arabesque, RStream, Fractal e com outros que apresentam implementações específicas para cada algoritmo. Todos possuem apenas a capacidade de processamento na CPU. Para realizar os testes, foram utilizados os *datasets* Mico, Patents, Youtube, ProteinDB, LiveJournal, Orkut, Twitter e Gsh-2015, os quais apresentam diferentes números de vértices, arestas e grau médio dos vértices. Estes números variam de 100 mil vértices no Mico, a 988.490.691 vértices no Gsh-2015, entre 2.160.312 arestas no Micro a 51.381.410.236 arestas no Gsh-2015 e grau médio dos vértices entre 8 e 76 no ProteinDB e Orkut, respectivamente. Segundo os autores, os experimentos foram realizados em uma máquina com CPU Intel Xeon Gold 5120 com *clock* de 2,2 GHz, 4 sockets de 14 núcleos cada, 190 GB de memória RAM e um SSD de 3 TB. Chen *et al.* (2020) destacam que nos experimentos envolvendo processamento em GPU foram utilizadas máquinas com placas de vídeo (Nvidia GTX 1080Ti com 11 GB de memória e Nvidia V100 com 32 GB de memória) e a API CUDA na versão 9.0.

De acordo com Chen *et al.* (2020), os resultados obtidos da execução em CPU em comparação com os *frameworks* genéricos de MPG foram muito favoráveis ao Pangolin, sendo 49, 88 e 80 vezes mais rápido que o Arabesque, RStream e Fractal, respectivamente. No entanto, segundo os atores, ao executar os algoritmos implementados pelo Pangolim na GPU, foram obtidos tempos de execução, em média, 15 vezes melhores em relação aos resultados obtidos em CPU. Contudo, Chen *et al.* (2020) destacam que na comparação com os algoritmos específicos, os resultados foram diversos. Enquanto para alguns algoritmos a execução em CPU, no Pangolin, foram 20 vezes mais lentas, outras execuções em GPU foram 290 vezes mais rápidas.

Chen *et al.* (2020) salientam que, embora o tempo de processamento não tenha sido melhor em relação aos algoritmos específicos, todas as aplicações otimizadas de maneira específica demandam um esforço de programação maior ao que é necessário para desenvolver no Pangolin. Além disso, os algoritmos possuem em média 4 vezes mais linhas de código. Inclusive, muitos são tão específicos que se limitam a resolver apenas um determinado escopo do algoritmo, como é o caso do algoritmo MC cujas aplicações específicas apenas realizam

buscas por subgrafos que possuem 3 ou 4 vértices, enquanto o Pangolin consegue realizar a mesma busca considerando qualquer número de vértices.

Por fim, Chen *et al.* (2020) concluem que, apesar de não obter uma performance superior às aplicações específicas, o Pangolin apresenta um enorme ganho de performance em comparação aos *frameworks* genéricos de MPG. Além disso, também abstrai grande parte das complexidades envolvidas através da possibilidade de extensão de suas funcionalidades.

3 PROPOSTA

Nesta seção será detalhada a proposta do trabalho: sua justificativa, requisitos funcionais e não funcionais, objetivos específicos, suas metodologias e o cronograma de desenvolvimento da aplicação.

3.1 JUSTIFICATIVA

O Quadro 1 apresenta uma comparação dos três trabalhos correlatos selecionados, destacando suas principais características. A disposição do quadro é a seguinte: cada linha representa uma característica distinta e cada coluna um trabalho correlato.

Quadro 1 – Comparativo dos trabalhos correlatos

Trabalhos Correlatos / Características	Zarebavani *et al.* (2019)	Wang *et al.* (2019)	Chen *et al.* (2020)
API para programação na GPU	CUDA 9.2	CUDA 9.1	CUDA 9.0
necessidade de pré-configuração da ferramenta	Sim	Não	Não
modo de execução, comunicação e travessia	Fixo	Híbrido	Fixo
linguagem de programação utilizada	C	C/C++	C++
hardware utilizado para processamento	GPU	GPU	CPU/GPU
algoritmo(s) utilizado(s)	PC	PR, SSSP, BFS e BC	CT, BC, MC e MSF
base de dados	NCI-60, MCC, BR-51, S.cerevisiae, S.aureus e DREAM5-Insilico	road_usa, kron_g500-logn21, soc-LiveJournal1, soc-orkut e soc-twitter-21	Mico, Patents, Youtube, ProteinDB, LiveJournal, Orkut, Twitter e Gsh-2015
ganho de performance	Até 1.296x	Até 45,8x	Até 290x

Fonte: elaborado pelo autor.

Zarebavani *et al.* (2019) exploraram a criação de dois algoritmos com estratégias distintas para resolução do algoritmo PC em GPU, o cuPC-E e o cuPC-S. Os autores realizaram uma comparação entre diversas implementações deste algoritmo e constataram que ambas as soluções tiveram um desempenho superior às demais devido ao paralelismo obtido na GPU. Zarebavani *et al.* (2019) ressaltam que o bom desempenho dos algoritmos se deve grande parte à pré-configuração referente à quantidade de *threads* a serem executadas paralelamente, uma vez que a topologia do grafo processado influencia na efetividade do modelo.

Wang *et al.* (2019) propuseram um *framework* de processamento de grafos em GPU com tipos de execução híbridas quanto ao modo de execução, estratégia de comunicação, e travessia com base em heurísticas obtidas do processamento. Foram utilizados algoritmos iterativos e de travessia a fim de se testar a eficiência do modelo em processamentos que demandam abordagens diferentes para se obter o melhor tempo de processamento. Segundo os autores, o comparativo foi realizado contra ferramentas que executavam os mesmos algoritmos através de uma abordagem fixa. Além disso, Wang *et al.* (2019) demonstraram que a troca adaptativa das estratégias durante o processamento é benéfica. No geral, o SEP-Graph apresenta um desempenho inferior apenas ao Gunrock no processamento da BFS, ao qual utiliza a mesma abordagem, porém sem a sobrecarga da checagem das heurísticas.

Chen *et al.* (2020) abordam a problemática da ineficiência de *frameworks* genéricos de processamento de grafos frente a implementações específicas, que costumam ser muito mais complexas. Apesar de ter uma abordagem genérica, possibilitam a extensão de suas funcionalidades e a incorporação de determinado algoritmo para o *framework* visando agilizar o seu processamento. Os algoritmos testados foram o CT, BC, MC e MSF. A partir deles, Chen *et al.* (2020) comprovaram que a extensão traz ganho de performance em relação aos *frameworks* genéricos. No entanto, inferior ao desempenho de implementações específicas.

Zarebavani *et al.* (2019) abordam duas técnicas para o processamento paralelo massivo de grafos em GPU, a paralelização dos testes IC e o compartilhamento de informações relevantes entre os núcleos de processamento.

Wang *et al.* (2019) focam no processamento híbrido com configurações variáveis em tempo de execução. Por fim, Chen *et al.* (2020) não desenvolvem uma nova estratégia de processamento, mas exploram o processamento em GPU dando importância à facilidade em implementar algoritmos otimizados através do *framework*. Quanto a linguagem de programação utilizada, todos os modelos foram implementados em C ou C++.

Diante deste contexto, é possível observar que o processamento de grafos em GPU é uma área em constante exploração no meio científico, havendo vários ramos de pesquisa envolvendo a utilização deste *hardware* em tarefas que se beneficiam do processamento paralelo. Além disso, também pode-se constatar que nos algoritmos em que é utilizado GPU, os resultados foram significativamente melhores em comparação com o processamento serial ou em paralelo na CPU. Porém, torna-se visível a complexidade no que se refere ao desenvolvimento de algoritmos, no qual o programador precisa lidar com gerenciamento de memória, concorrência e sincronismo de forma manual enquanto coordena a utilização da GPU. As linguagens de programação C e C++, por mais que seus ecossistemas sejam os mais desenvolvidos no que se refere ao processamento em GPU e sejam adotadas pela maioria dos trabalhos, exigem um grande conhecimento de programação de baixo nível e dificultam a implementação de programas que utilizam este *hardware*. Porém, por outro lado, os *frameworks* que facilitam o desenvolvimento expondo APIs mais simples e genéricas, tendem a ter um desempenho significativamente pior que os algoritmos desenvolvidos de maneira específica, que são mais complexos devido aos fatores anteriormente citados. Com o exposto, o presente trabalho propõe a utilização da linguagem Rust para facilitar o gerenciamento de memória, concorrência e sincronismo no que se refere ao desenvolvimento de um algoritmo de processamento em GPU de cunho específico, deixando o programador menos propício a cometer erros de desenvolvimento e, ao mesmo tempo, buscando ganho de performance em relação aos *frameworks* genéricos. A relevância técnica deste trabalho reside no fato de que o processamento de grandes grafos é uma tarefa complexa e que requer muitos recursos computacionais. O uso de técnicas convencionais de processamento pode levar a um tempo de execução muito longo e ao consumo excessivo de memória, o que pode ser problemático para aplicativos que exigem processamento em tempo real ou interativo. A utilização de paralelismo em GPU e da linguagem de programação Rust pode oferecer uma abordagem mais eficiente para o processamento de grandes grafos, reduzindo o tempo de execução e o consumo de memória. Do ponto de vista social, este trabalho pode ter impacto em várias áreas que dependem do processamento de grandes grafos. Por exemplo, na área de saúde, o processamento de grafos pode ser usado para identificar interações entre proteínas e desenvolver novas terapias. Na área de logística e transporte, o processamento de grafos pode ser utilizado para otimizar rotas de transporte e reduzir custos e emissões de carbono.

3.2 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O trabalho proposto deverá apresentar os seguintes Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF):

a) permitir a leitura de arquivos de entrada contendo a definição do grafo a ser processado (RF);
b) realizar o processamento de algoritmos comuns, como busca em largura e profundidade, caminho mínimo, entre outros (RF);
c) realizar o processamento paralelo em GPU (RF);
d) adaptar-se à topologia do grafo sem necessidade de pré-configuração (RF);
e) disponibilizar métricas de execução após a execução de cada teste (como tempo de execução e consumo de memória) (RF);
f) utilizar a API CUDA para programação em GPU (RNF);
g) ser capaz de lidar com dados de entrada e saída em formatos comuns de grafos, como GML e GraphML (RNF);
h) utilizar a linguagem de programação Rust para o desenvolvimento da aplicação (RNF);
i) ser capaz de processar grafos com milhões de vértices e arestas de forma eficiente, sem comprometer o desempenho (RNF).

3.3 METODOLOGIA

O trabalho será desenvolvido observando as seguintes etapas:

a) levantamento bibliográfico: pesquisar e estudar sobre grafos, paralelismo em GPU, linguagem Rust e trabalhos correlatos;
b) levantamento de *frameworks*: pesquisar por *frameworks* que processam grandes grafos de forma convencional via CPU;
c) estudo da linguagem Rust: compreender as principais funcionalidades e bibliotecas da linguagem Rust para a execução de algoritmos em GPU utilizando a API CUDA;
d) pesquisa e escolha de métricas: pesquisar e definir as métricas que serão utilizadas para avaliar e comparar o desempenho da ferramenta em relação aos *frameworks* encontrados no item (b), as configurações e conjuntos de dados, tais como tempo de execução, consumo de memória, utilização da CPU e GPU, e acurácia;

e) configuração do ambiente: preparo da máquina local para executar os testes e garantir que a API CUDA esteja configurada corretamente para a respectiva GPU;
f) desenvolvimento da ferramenta: a partir dos itens (c), (d) e (e) realizar a implementação de algoritmos comuns de grafos através da linguagem Rust e com processamento paralelo em GPU através da API CUDA;
g) levantamento de *datasets*: pesquisar por *datasets* a serem utilizados nos testes;
h) execução dos *frameworks* existentes: instalação e execução local dos *frameworks* identificados no item (b), verificando suas principais funcionalidades e características;
i) execução dos testes e comparações: execução da ferramenta desenvolvida e dos *frameworks* existentes (item h) sobre os *datasets* selecionados (item g) para obter as métricas (item d) de cada algoritmo e realizar as comparações;
j) avaliação do desempenho da linguagem Rust: a partir do (i) avaliar o desempenho da linguagem em comparação aos *frameworks* do item (h);
k) análise da otimização do paralelismo em GPU: a partir do (i) analisar se paralelismo em GPU otimizou o processamento de grandes grafos em comparação aos *frameworks* do item (h).

As etapas serão realizadas nos períodos relacionados no Quadro 2.

Quadro 2 – Cronograma de atividades a serem desenvolvidas

	2023									
	jul.		ago.		set.		out.		nov.	
etapas / quinzenas	1	2	1	2	1	2	1	2	1	2
levantamento bibliográfico										
levantamento dos *frameworks*										
estudo da linguagem Rust										
pesquisa e escolha de métricas										
configuração do ambiente										
desenvolvimento da ferramenta										
levantamento de *datasets*										
execução dos *frameworks* existentes										
execução dos testes e comparações										
avaliação do desempenho da linguagem Rust										
análise da otimização do paralelismo em GPU										

Fonte: elaborado pelo autor.

4 REVISÃO BIBLIOGRÁFICA

Nesta seção serão apresentados os principais conceitos que fundamentam o estudo proposto. A subseção 4.1 apresenta a definição de grafos e seus principais algoritmos. Na subseção 4.2 aborda-se processamento em paralelo com foco na GPU. Por fim, a subseção 4.3 discorre sobre a linguagem de programação Rust.

4.1 GRAFOS E SEUS ALGORITMOS

Bacciu *et al*. (2020) define grafos como uma forma de representar dados através de uma estrutura composta por nós de informações unidos por arestas que denotam os relacionamentos entre elas. Ainda segundo Bacciu *et al*. (2020), grafos podem representar inúmeros produtos provenientes de processos naturais e artificiais, como propriedades químicas, a força de ligações moleculares e até mesmo os relacionamentos entre pessoas em uma rede social. Hogan *et al*. (2021) reforça que, além das representações mais triviais, essa estrutura de nós e arestas pode ser utilizada também para representar conceitos mais complexos de um determinado domínio, agregando uma grande quantidade de conhecimento de forma que seja possível executar algoritmos para se obter *insights* e resolver questões referentes ao domínio sendo representado nessa estrutura de dados.

De acordo com Needham e Hodler (2019) e Ore (1990), a primeira utilização documentada de grafos foi em 1736 na resolução do problema "As Sete Pontes de Königsberg", onde o matemático suíço Leonhard Euler utilizou uma representação em grafos para comprovar que não era possível visitar todas as quatro áreas da cidade de Königsberg cruzando uma única vez as pontes que as conectavam. Segundo Ore (1990), a teoria dos grafos, como é denominada sua área de estudo, por originar-se em quebra-cabeças lógicos voltados ao entretenimento, não ganhou muita importância no início de seu desenvolvimento. Com o desenvolvimento da matemática e suas aplicações, a teoria dos grafos ganhou notoriedade e já a partir do século XIX era utilizada na representação de circuitos elétricos e diagramas moleculares. Atualmente, a teoria dos grafos é uma ferramenta natural para diversas áreas de estudo.

Referente aos algoritmos de processamento em grafos, Sedgewick (2001) ressalta que eles permitem a obtenção de propriedades a partir da visitação de cada um de seus nós e arestas. Needham e Hodler (2019) complementam defendendo que "os algoritmos de processamento em grafos provêm as melhores abordagens para

processamento de dados interrelacionados, uma vez que a lógica e a matemática que os compõem foram especialmente desenvolvidas para operar sobre relacionamentos". Entre os tipos de algoritmos de processamento em grafos estão os algoritmos de busca, destacando-se a busca em profundidade (Depth-First Search – DFS) e a busca em largura (Breadth-First Search – BFS) e algoritmos de caminhamento mínimo, como o algoritmo de Dijkstra.

Segundo Corneil e Krueger (2008), a DFS consiste em uma função recursiva que progride na exploração do grafo o mais profundamente possível, para apenas então retornar e visitar uma nova ramificação do grafo. Sedgewick (2001) complementa que internamente a DFS utiliza uma pilha *Last In, First Out* (LIFO) para definir a ordem dos vértices a serem visitados, isto é, o vértice mais recente descoberto é também o primeiro a ser expandido.

Ao contrário da DFS, a BFS utiliza uma fila *First In, First Out* (FIFO) para determinar a ordem de visitação dos vértices, onde o primeiro vértice mais antigo descoberto é o primeiro a ser expandido (SEDGEWICK, 2001). Isso implica que o funcionamento da BFS se baseia em expandir todos os vértices acessíveis a partir de um determinado nó para só então avançar um nível e realizar o mesmo processo recursivamente em um próximo nó. Sedgewick (2001) destaca que, por conta dessa característica, a BFS é mais indicada para identificar o caminho mais curto entre dois vértices.

Segundo Chen (2003), o algoritmo de Dijkstra, proposto por Edsger Dijkstra em 1959, encontra o menor caminho de um vértice inicial para os demais vértices do grafo, sendo que o custo de cada aresta não deve ser negativo. Ainda de acordo com Chen (2003), Dijkstra funciona partindo de um vértice inicial e percorre o grafo em uma ordem de custo crescente das arestas, mantendo uma lista de vértices visitados, cujo custo a partir do vértice inicial já foi computado, e uma lista de vértices não visitados. A cada iteração, o vértice não visitado com o menor custo é expandido e o custo total de caminhamento até ele desde o vértice inicial é computado, com esse processo se repetindo até não haver mais vértices inexplorados.

4.2 PARALELIZAÇÃO EM GPU

Lalwani *et al*. (2019) salienta que, por conta dos problemas de pesquisa atualmente serem naturalmente complexos e ao advento da *big data*, muitos desses algoritmos precisaram passar por otimizações que envolveram, entre outras melhorias, a utilização de processamento em paralelo para compensar o alto custo computacional demandado por esse novo contexto. Lalwani *et al*. (2019) descreve a paralelização como "o uso simultâneo de múltiplos recursos computacionais a fim de solucionar um problema computacional quebrando-o em partes discretas", sendo que tal divisão do processamento pode ser realizada através do uso de uma CPU com múltiplos núcleos, ou através de GPUs.

Segundo Dally, Keckler e Kirk (2021), as GPUs foram introduzidas no mercado em 1999 na forma da NVIDIA GeForce 256, que combinava computações de vértices para transformações e iluminação com computações de fragmentos. Estes chips surgiram como um *hardware* dedicado para o processamento de gráficos em tempo real, como a transformação de perspectiva, que apresenta alta demanda de cálculos aritméticos envolvendo pontos flutuantes.

Sesin e Bolbakov (2021) citam que, ao contrário das CPUs, as GPUs são especializadas no processamento em paralelo de grande quantidade de dados, o que é justificado pelos milhares de núcleos físicos de processamento presentes neste *hardware* e pela memória de vídeo que é unificada à placa. Crow (2004) soma argumentando que por conta do modelo de processamento das GPUs apresentar baixo consumo de memória, os gargalos comumente presentes nos processamentos em CPU são praticamente eliminados, possibilitando o uso de todos os transistores da placa e aumentando sua performance computacional.

Contudo, as primeiras implementações de uma esteira de processamento gráfico não permitiam que os dados fossem manipulados após serem enviados à GPU e sua funcionalidade exata era apenas determinada pelo conjunto de APIs utilizado para interação com a placa, como o DirectX e o OpenGL. Por conta disso, as esteiras de processamento eram chamadas de "esteiras de função fixa" e não permitiam a programação para que novos e mais sofisticados efeitos visuais fossem alcançados (CROW, 2004). A crescente demanda apresentada pelos jogos de computadores pessoais resultou no desenvolvimento da GeForce 3 em 2001 e na GeForce FX em 2002, que apresentavam computações de vértice e fragmento programáveis, respectivamente (DALLY; KECKLER; KIRK, 2021). Segundo Crow (2004), através dessa evolução, os desenvolvedores puderam sobrescrever as seções da esteira de processamento gráfico da GPU com trechos de código próprios, podendo assim manipular os dados após a entrada no *hardware*.

A alta performance envolvendo o cálculo de pontos flutuantes, a alta capacidade de processamento em paralelo disponibilizada pelas GPUs e a capacidade de programação despertou o interesse da comunidade científica, que faz uso deste *hardware* para os mais variados cenários de pesquisa (DALLY; KECKLER; KIRK, 2021). Dally, Keckler e Kirk (2021) apontam que o frequente uso das GPUs em projetos de pesquisa gerou inúmeros feedbacks da comunidade científica, levando à criação de funcionalidades específicas para o processamento de alta performance. Com isso, atualmente as GPUs impulsionam aplicações que variam desde

supercomputadores até carros autônomos. Dally, Keckler e Kirk (2021) vão além e constatam que os grandes avanços alcançados nas áreas que fazem uso do *deep learning*, por exemplo, só foram possíveis graças ao desenvolvimento das GPUs e sua elevada performance.

4.3 LINGUAGEM DE PROGRAMAÇÃO RUST

Segundo Bugden e Alahmar (2022), a linguagem de programação Rust teve seu início em 2006 como um projeto pessoal de Graydon Hoare, colaborador da Mozilla. Inicialmente, a linguagem foi desenvolvida visando a segurança na manipulação de memória, mas os bons resultados também incluíram a performance como foco de seu desenvolvimento, com seu potencial resultando no patrocínio da Mozilla a partir de 2010. Seu primeiro compilador, em versão pré-alfa, foi lançado em janeiro de 2012 (BUGDEN; ALAHMAR, 2022).

A linguagem oferece grande suporte para programação de baixo nível e permite que companhias desenvolvam sistemas embarcados, motores de busca, criptomoedas ou sistemas operacionais (KLABNIK; NICHOLS, 2023). Com isso, Klabnik e Nichols (2023) citam que o principal recurso apresentado por Rust é seu sistema de gerenciamento de memória denominado Ownership, através do qual é garantida a segurança no desenvolvimento da aplicação. Nele, cada valor em memória possui uma variável proprietária, que é a única com a possibilidade de acessar ou modificar seu valor. Esta propriedade pode ser transferida ou emprestada de maneira mutável ou imutável a outra variável através de referências.

Segundo Viitanen (2020), o sistema Ownership estabelece duas regras para a manipulação de referências a fim de garantir a segurança na manipulação de memória e evitar a existência de ponteiros inválidos: (1) um valor não pode ter um escopo menor do que a variável que a referencia e (2) quando um valor em memória é referenciado de forma mutável, não pode haver outra variável o referenciando no mesmo escopo. Uma violação da segunda regra do Ownership pode ser observada na Figura 3, onde a utilização da variável `borrowed_numbers`, que é uma referência imutável para `numbers`, pela macro `println!` estende seu escopo até a linha 9, compartilhando seu escopo com a variável `mutably_borrowed_numbers`, que é uma referência mutável para `numbers`. Por conta disso, o trecho de código apresentado na Figura 3 não compilará (VIITANEN, 2020). Caso não houvesse a utilização da macro `println!` na linha 9, Rust conseguiria identificar que a variável `borrowed_numbers` não é utilizada após a linha 4 e encerraria seu escopo nesta linha, permitindo a compilação.

Figura 3 – Trecho de código Rust demonstrando uma violação da segunda regra do Ownership

```
1  fn main() {
2      // This will not compile
3      let mut numbers = vec![1, 2, 3];
4      let borrowed_numbers = &numbers;
5      let mutably_borrowed_numbers = &mut numbers;
6
7      // immutable borrow used after a mutable borrow
8      // yields a compilation error
9      println!("{:?}", borrowed_numbers);
10 }
```

Fonte: Viitanen (2020).

Viitanen (2020) ressalta que Rust, embora similar às demais linguagens de programação, apresenta algumas características que podem dificultar sua adoção até mesmo por desenvolvedores experientes, como o fato de não ser orientada a objetos e não permitir herança entre classes. Além disso, o autor também cita que seu sistema de gerenciamento de memória é mais rígido do que o de linguagens como C++ e pode ser limitante para determinadas implementações, mas que a própria linguagem oferece recursos para flexibilizar as validações e permitir que o desenvolvimento seja feito de maneira mais livre.

De acordo com Klabnik e Nichols (2023), a validação das regras inerentes à utilização da memória ocorre em tempo de compilação e não afeta o desempenho da aplicação durante sua execução, possibilitando que problemas como a concorrência em rotinas paralelas, por exemplo, sejam facilmente gerenciados sem renunciar a alta performance. Por conta desses fatores, a linguagem Rust vem ganhando popularidade entre desenvolvedores, sendo adotada em diversos setores do mercado atualmente (BUGDEN; ALAHMAR, 2022), com Viitanen (2020) defendendo que Rust é a linguagem mais indicada para aplicações que demandam segurança e performance.

REFERÊNCIAS

BACCIU, Davide *et al*. A gentle introduction to deep learning for graphs. **Neural Networks**, [S.l.], v. 129, p. 203-221, jun. 2020. Elsevier BV.

BUGDEN, William; ALAHMAR, Ayman. Rust: the programming language for safety and performance. **Arxiv**, [S.l.], 2022. ArXiv. http://dx.doi.org/10.48550/ARXIV.2206.05503. Disponível em: https://arxiv.org/abs/2206.05503. Acesso em: 24 abr. 2023.

CHEN, Jing-Chao. Dijkstra's Shortest Path Algorithm. **Journal of Formalized Mathematics**, [S.l.], v. 15, mar. 2003.

CHEN, Xuhao *et al*. Pangolin: an efficient and flexible graph mining system on cpu and gpu. **Proceedings Of The Vldb Endowment**, [S.l.], v. 13, n. 8, p. 1190-1205, abr. 2020. Association for Computing Machinery (ACM).

CROW, Thomas S.. **Evolution of the Grapical Processing Unit**. 2004. 53 f. Monografia (Mestrado) – Curso de Computer Science, University of Nevada, Reno, 2004.

CORNEIL, Derek G.; KRUEGER, Richard M.. A Unified View of Graph Searching. **Siam Journal On Discrete Mathematics**, [S.L.], v. 22, n. 4, p. 1259-1276, jan. 2008. Society for Industrial & Applied Mathematics (SIAM). http://dx.doi.org/10.1137/050623498.

DAFIR, Zineb; LAMARI, Yasmine; SLAOUI, Said Chah. A survey on parallel clustering algorithms for Big Data. **Artificial Intelligence Review**, [S.l.], v. 54, n. 4, p. 2411-2443, 6 out. 2020. Springer Science and Business Media LLC.

DALLY, William J.; KECKLER, Stephen W.; KIRK, David B.. Evolution of the Graphics Processing Unit (GPU). **Ieee Micro**, [S.l.], v. 41, n. 6, p. 42-51, 1 nov. 2021. Institute of Electrical and Electronics Engineers (IEEE).

HARIRI, Reihaneh H.; FREDERICKS, Erik M.; BOWERS, Kate M.. Uncertainty in big data analytics: survey, opportunities, and challenges. **Journal Of Big Data**, [S.l.], v. 6, n. 1, 4 jun. 2019. Springer Science and Business Media LLC.

HOGAN, Aidan *et al*. Knowledge Graphs. **Acm Computing Surveys**, [S.l.], v. 54, n. 4, p. 1-37, 2 jul. 2021. Association for Computing Machinery (ACM).

JIANG, Yanbin *et al*. Enhancing social recommendation via two-level graph attentional networks. **Neurocomputing**, [S.l.], v. 449, n. 999, p. 71-84, ago. 2021. Elsevier BV.

KLABNIK, Steve; NICHOLS, Carol. **The Rust Programming Language**. 2023. Disponível em: https://doc.rust-lang.org/stable/book. Acesso em: 06 maio 2023;

LALWANI, Soniya *et al*. A Survey on Parallel Particle Swarm Optimization Algorithms. **Arabian Journal For Science And Engineering**, [S.l.], v. 44, n. 4, p. 2899-2923, 8 jan. 2019. Springer Science and Business Media LLC.

LI, Michelle M.; HUANG, Kexin; ZITNIK, Marinka. Graph representation learning in biomedicine and healthcare. **Nature Biomedical Engineering**, [S.l.], v. 6, n. 12, p. 1353-1369, 31 out. 2022. Springer Science and Business Media LLC.

LI, Qingbiao *et al*. Graph Neural Networks for Decentralized Multi-Robot Path Planning. **2020 Ieee/Rsj International Conference On Intelligent Robots And Systems (Iros)**, [S.l.], 24 out. 2020. IEEE. http://dx.doi.org/10.1109/iros45743.2020.9341668. Disponível em: https://ieeexplore.ieee.org/document/9341668. Acesso em: 23 abr. 2023.

NEEDHAM, Mark; HODLER, Amy E.. **Graph Algorithms**: practical examples in apache spark and neo4j. [S.l.]: O'Reilly Media, 2019.

RIBEIRO, Leonardo F. R *et al*. Investigating Pretrained Language Models for Graph-to-Text Generation. **Arxiv**, [S.l.], 2020. ArXiv. http://dx.doi.org/10.48550/ARXIV.2007.08426. Disponível em: https://arxiv.org/abs/2007.08426. Acesso em: 23 abr. 2023.

SANTOS, Alberto *et al*. A knowledge graph to interpret clinical proteomics data. **Nature Biotechnology**, [S.l.], v. 40, n. 5, p. 692-702, 31 jan. 2022. Springer Science and Business Media LLC.

SEDGEWICK, Robert. **Algorithms in C, Part 5**: graph algorithms. [S.l.]: Pearson Education, 2001.

SESIN, I. Yu.; BOLBAKOV, R. G.. Comparative analysis of software optimization methods in context of branch predication on GPUs. **Russian Technological Journal**, [S.l.], v. 9, n. 6, p. 7-15, 2 dez. 2021. RTU MIREA.

ORE, Oystein. **Graphs and Their Uses**. 2. ed. Washington: The Mathematical Association of America, 1990.

VIITANEN, Rasmus. **Evaluating Memory Models for Graph-Like Data Structures in the Rust Programming Language**: performance and usabiliy. 2020. 58 f. Monografia (Mestrado) - Curso de Computer Engineering, Department Of Computer And Information Science, Linköping University, Linköping, 2020.

WANG, Hao; *et al*. SEP-graph: finding shortest execution paths for graph processing under a hybrid framework on gpu. **Proceedings Of The 24Th Symposium On Principles And Practice Of Parallel Programming**, [S.l.], p. 38-52, 16 fev. 2019. ACM. Disponível em: https://dl.acm.org/doi/10.1145/3293883.3295733. Acesso em: 13 fev. 2023.

YAMATO, Yoji. Study and evaluation of automatic GPU offloading method from various language applications. **International Journal Of Parallel, Emergent And Distributed Systems**, [S.l.], v. 37, n. 1, p. 22-39, 6 set. 2021. Informa UK Limited.

ZAREBAVANI, Behrooz *et al*. CuPC: cuda-based parallel pc algorithm for causal structure learning on gpu. **Ieee Transactions On Parallel And Distributed Systems**, [S.l.], v. 31, n. 3, p. 530-542, 23 set. 2019. Institute of Electrical and Electronics Engineers (IEEE).

PROJETO: OBSERVAÇÕES – PROFESSOR ORIENTADOR

Observações do orientador em relação a itens não atendidos do pré-projeto:

Referente ao item (j) “avaliação do desempenho da linguagem Rust” da metodologia, incluímos essa etapa propositalmente pois a intenção é verificar a viabilidade da linguagem Rust para o contexto de processamento de grafos em GPU. A avaliação da ferramenta desenvolvida é composta por este item e o item (k).