ANÁLISE DE DESEMPENHO DOS ALGORITMOS COUNTING SORT E DE BUSCA EM PROFUNDIDADE (BFS) PARALELIZADOS COM CUDA EM GPU

PERFORMANCE ANALYSIS OF COUNTING SORT AND BREADTH-FIRST SEARCH (BFS) ALGORITHMS PARALLELIZED WITH CUDA ON GPU

Mateus da Silva Vieira*
Diego Rocha Lima**

RESUMO

Este trabalho investiga o impacto da paralelização de algoritmos utilizando a plataforma CUDA em ambiente GPU. Com a crescente demanda por desempenho computacional, a programação paralela torna-se essencial, especialmente em arquiteturas manycore projetadas para executar milhares de threads simultaneamente. No entanto, nem todos os algoritmos se beneficiam dessa abordagem. O objetivo deste trabalho é identificar quais características estruturais determinam o ganho de desempenho na GPU. Para isso, foram implementadas e comparadas versões sequenciais (CPU) e paralelas (GPU) de dois algoritmos com paradigmas opostos: o Counting Sort, representando o paralelismo de dados e acesso regular à memória, e a Busca em Profundidade (DFS), caracterizada por forte dependência sequencial e acesso irregular. Os resultados experimentais demonstraram que o Counting Sort obteve ganhos expressivos, atingindo um speedup de até 6,03 vezes para entradas de 1 milhão de elementos. Em contrapartida, o DFS sofreu severa degradação de desempenho na GPU, apresentando tempos de execução até 100 vezes superiores à CPU em cenários de estresse. O trabalho conclui que a paralelização em CUDA é altamente eficaz para algoritmos com independência de dados, mas contraproducente para lógicas estritamente seriais, oferecendo uma base comparativa para auxiliar na decisão de arquitetura de software.

Palavras-chaves: paralelização, CUDA, Counting Sort, DFS, programação paralela, desempenho computacional.

ABSTRACT

This work investigates the impact of parallelizing algorithms using the CUDA platform in a GPU environment. With the growing demand for computational performance, parallel programming has become essential, especially in manycore architectures designed to execute thousands of threads simultaneously. However, not all algorithms benefit from this approach. The objective of this work is to identify which structural characteristics determine performance gains on the GPU.

* Graduando em Ciência da Computação, Instituto Federal de Educação, Ciência e Tecnologia do Ceará, Aracati, CE, Brasil. E-mail: mmateusinfo@gmail.com
** Doutor em Ciência da Computação, docente do Instituto Federal de Educação, Ciência e Tecnologia do Ceará, Aracati, CE, Brasil. E-mail: diego.rocha@ifce.edu.br

To this end, sequential (CPU) and parallel (GPU) versions of two algorithms with opposing paradigms were implemented and compared: Counting Sort, representing data parallelism and regular memory access, and Depth-First Search (DFS), characterized by strong sequential dependency and irregular access. The experimental results showed that Counting Sort achieved significant gains, reaching a speedup of up to 6.03 times for inputs of 1 million elements. In contrast, DFS suffered severe performance degradation on the GPU, with execution times up to 100 times higher than the CPU in stress scenarios. The study concludes that parallelization in CUDA is highly effective for data-independent algorithms but counterproductive for strictly serial logic, providing a comparative basis to aid in software architecture decisions.

1 INTRODUÇÃO

O avanço da computação moderna tem demandado soluções cada vez mais eficientes para o processamento de grandes volumes de dados e para a execução de aplicações com alto custo computacional. Para ilustrar a magnitude desse cenário, estimativas apontavam que, até 2025, cerca de 463 exabytes de dados seriam gerados diariamente ao redor do globo Desjardins (2019). A programação paralela surgiu como uma estratégia fundamental para o aproveitamento pleno dos recursos de hardware disponíveis, especialmente com a popularização de arquiteturas multicores e o crescente uso de unidades de processamento gráfico (GPUs) em aplicações de propósito geral.

Diferentemente das unidades centrais de processamento (CPUs), que são otimizadas para tarefas sequenciais com alta performance por núcleo, as GPUs são arquiteturas voltadas à execução massiva de milhares de threads simultaneamente. Essa característica vai torná-los adequados para os algoritmos de alto grau de paralelismo, abrindo espaço para ganhos significativos de desempenho em áreas de simulações científicas, inteligência artificial, processamento de imagens e aprendizado de máquina.

No entanto, nem todo algoritmo vai se beneficiar da execução paralela. Em muitos casos, o custo de sincronização, a estrutura dos dados, a necessidade de comunicação entre threads ou o padrão de acesso à memória podem anular os ganhos esperados, tornando a paralelização ineficaz ou até contraproducente. Isso reforça a importância de se investigar quais algoritmos realmente são mais vantajosos de se paralelizar, considerando tanto o seu comportamento computacional quanto a arquitetura de execução. Essa realidade impõe uma problemática central: *quais algoritmos realmente valem a pena ser paralelizados?*

Este trabalho busca investigar, por meio de experimentos empíricos, quais algoritmos irão se beneficiar da paralelização. Por meio da implementação e comparação entre versões sequenciais e paralelas de diferentes algoritmos clássicos, serão analisadas métricas como tempo de execução, speedup e eficiência, com foco em entender os fatores que influenciam diretamente os ganhos reais de desempenho.

Diversos estudos na literatura abordam essa questão. Ribeiro, Sêmeler e Dias (2024), por exemplo, compararam as versões sequenciais e paralelas dos algoritmos de Números Primos

e Monte Carlo utilizando OpenMP e MPI, e concluíram que apenas o primeiro apresentou ganhos expressivos de desempenho. Já Nogueira et al. (2024) analisaram algoritmos recursivos implementados em OpenMP, CUDA e CUDA Dynamic Parallelism (DP), demonstrando que a escolha de modelo de execução e do algoritmo impactam diretamente nos resultados obtidos. Khalilov e Timoveev (2021), por sua vez, compararam os modelos CUDA, OpenACC e OpenMP na GPU Tesla V100 e observaram que, apesar da maior complexidade de implementação, o CUDA apresentou desempenho superior em aplicações reais.

Assim, a relevância deste trabalho está em fornecer um estudo comparativo que ajude a orientar escolhas mais eficazes na programação paralela com CUDA, contribuindo para o uso mais consciente de recursos computacionais e para o desenvolvimento de soluções mais eficientes em ambientes heterogêneos.

Este trabalho está organizado da seguinte forma: o Capítulo 2 apresenta o referencial teórico, abordando os conceitos de programação paralela, arquitetura de GPU e o modelo de programação CUDA. O capítulo 3 descreve os trabalhos relacionados. No capítulo 4, é apresentada a metodologia utilizada. Por fim, os experimentos e resultados obtidos são discutidos no capítulo 5.

2 FUNDAMENTAÇÃO TEORICA

2.1 Programação Paralela

A programação paralela é um paradigma de desenvolvimento de software que busca executar múltiplas operações ou tarefas simultaneamente, aproveitando o uso de múltiplos processadores ou núcleos de processamento. O objetivo principal é reduzir o tempo de execução de algoritmos e aumentar a eficiência computacional, especialmente em aplicações que demandam alto poder de processamento.

A popularização da programação paralela (PP) impulsionada pelo aumento da performance computacional com o uso de aceleradores, coprocessadores multicore e manycore, além de clusters, impactou tanto a indústria quanto os métodos de ensino. No entanto, a complexidade envolvida na codificação paralela representa um desafio maior em comparação à programação tradicional sequencial, exigindo novas estratégias para ensinar. (PEREZ et al., 2023)

2.2 Unidade de Processamento Gráfico

A unidade de processamento gráfico (GPU) tem muitos núcleos pequenos e mais especializados. Esses núcleos oferecem desempenho sólido, trabalhando em conjunto e dividindo tarefas de processamento entre muitos núcleos simultaneamente (ou em paralelo). A GPU destaca-se em tarefas altamente paralelas, como renderização de visuais durante a jogabilidade, manipulação de dados de vídeo durante a criação de conteúdo e resultados de computação em cargas de trabalho de IA intensivas.

Apesar de compartilharem características como a capacidade de processamento de dados, CPUs e GPUs possuem arquiteturas distintas e são otimizadas para finalidades diferentes. A CPU é projetada para lidar com tarefas de forma sequencial e de baixa latência, com poucos núcleos de alto desempenho voltados para a execução rápida de tarefas individuais. Já a GPU, originalmente desenvolvida para a aceleração gráfica, evoluiu para um processador altamente paralelizável, capaz de executar milhares de operações simultâneas, sendo amplamente utilizada em aplicações como a renderização 3D, inteligência artificial e computação científica. Atualmente, essas duas unidades são complementares, e os melhores resultados computacionais são obtidos com o uso conjunto, aplicando a ferramenta mais adequada para cada tipo de tarefa. Intel Corporation (2023).

2.3 CUDA

CUDA é uma tecnologia desenvolvida pela NVIDIA que oferece uma plataforma de computação paralela e um modelo de programação. Seu propósito é permitir que aplicações sejam executadas de forma mais rápida ao explorar o poder de processamento das GPUs. Essa abordagem não é usada apenas em ambientes acadêmicos e de pesquisa, mas também ganhou espaço em aplicações comerciais e industriais.

Por meio do CUDA, os programadores podem desenvolver códigos em linguagens como C, C++, Fortran, Python e MATLAB, utilizando recursos específicos adicionados a essas linguagens. Esses recursos incluem palavras-chave e comandos que facilitam o controle da execução paralela diretamente na GPU.

2.3.1 Modelo de programação

O CUDA é uma plataforma para programação heterogênea. Seu modelo de programação assume um sistema constituído por diferentes tipos de processadores: *host* (por exemplo, CPU) e *device* (ex.: GPUs), cada um com sua própria memória. Por padrão, o *host* não pode acessar nenhum endereço de memória do dispositivo e vice-versa. Todos os dados são transferidos entre eles por meio do barramento de comunicação, usando chamadas explícitas de cópia de memória. Na versão do CUDA 6.0, a NVIDIA introduziu a Memória Unificada, que permite que a GPU e CPU compartilhem os mesmos endereços virtualmente, isentando os programadores do uso de chamadas explícitas de memória NVIDIA Corporation (2024).

Os códigos compilados no host são diferentes dos compilados para os dispositivos; portanto, é preciso definir de forma explícita quais funções ou variáveis são feitas para GPU e quais são feitas para CPU, de modo que o compilador CUDA possa gerar o código de máquina para o alvo correto. O modelo de programação divide o código em duas partes: (1) *código do host* e (2) *código do dispositivo*. Essa divisão é feita por meio de marcadores especiais que são adicionados antes da declaração de funções e variáveis. Por exemplo, o qualificador `__global__` é utilizado para as funções que serão compiladas para arquitetura da GPU, mas podem ser chamadas pela CPU. Diferentemente das funções como o qualificador `__device__`,

que são compiladas para a GPU, mas podem ser chamadas apenas em funções que executam na GPU. O papel do *host* é controlar o contexto e o fluxo de execução principal da aplicação, enquanto a GPU atua como um coprocessador, cujas tarefas são enviadas pela CPU por meio da chamada de *kernels*. Um *kernel* é um código (ou uma função) que executa na GPU. Na GPU, a execução de *kernels* se difere das chamadas tradicionais na linguagem C. Por padrão, a chamada de um *kernel* ocorre de forma assíncrona, ou seja, quando ele é lançado, o controle da execução retorna imediatamente ao *host*, sem esperar que a GPU conclua sua tarefa (CHENG; GROSSMAN; MCKERCHER, 2014). Além disso, os kernels são invocados utilizando uma sintaxe específica com colchetes (<<<...>>>), que tem a função de informar à GPU como organizar e distribuir as *threads* que executarão o código no dispositivo.

Em CUDA, uma *thread* é uma entidade abstrata que representa a execução de um *kernel*. Quando um *kernel* é lançado para execução, são criados diferentes *threads* que executam o código desse *kernel*, sendo atribuída a cada *thread* uma indexação única SANDERS Jason e KANDROT (2010). As *threads* são agrupadas em *Blocos de Threads* e os blocos são agrupados em um *Grid*. A arquitetura das GPUs é desenvolvida para suportar a execução de milhares de *threads* simultaneamente e, com base nessa característica, o CUDA oferece um modelo de programação escalável e hierárquico, que facilita o aproveitamento eficiente desse paralelismo massivo.

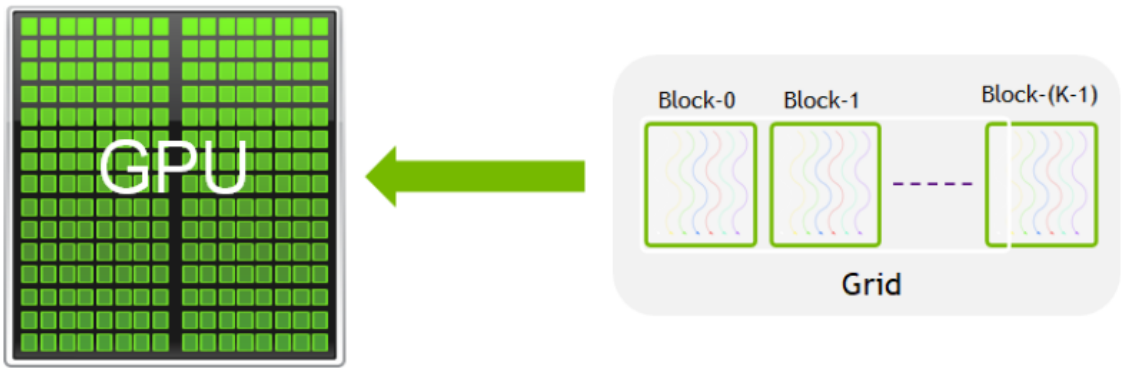


Figura 1 – os *kernels* são subdivididos em blocos. fonte: NVIDIA (2023)

2.3.2 Hierarquia de Memória da GPU

Um dos principais gargalos na computação de alto desempenho é a latência de memória. Para mitigar esse problema, a arquitetura CUDA expõe uma hierarquia de memória que oferece diferentes compromissos entre capacidade e velocidade. O uso eficiente dessa hierarquia é frequentemente o fator determinante entre um algoritmo rápido e um lento Gupta (2020).

- **Memória Global (Global Memory):** É a memória principal da placa de vídeo (VRAM). Possui a maior capacidade (vários gigabytes), mas a maior latência (centenas de ciclos de clock) e menor largura de banda relativa. O acesso a esta memória deve ser coalescido, ou

seja, threads vizinhas devem ler endereços de memória vizinhos para que a transação seja eficiente.

- **Memória Compartilhada (*Shared Memory*):** É uma memória on-chip extremamente rápida, localizada dentro de cada Multiprocessador (SM)[1]. Diferente do *cache* L1/L2 tradicional (que é gerenciado pelo hardware), a memória compartilhada é gerenciada explicitamente pelo programador. Ela permite que threads de um mesmo bloco troquem dados sem precisar acessar a lenta memória global.
- **Registradores (*Registers*):** São as memórias mais rápidas disponíveis, privadas para cada thread. O número de registradores utilizados por um *kernel* pode limitar quantas *threads* podem estar ativas simultaneamente (*occupancy*).
- **Memória Local e Constante:** A memória local é uma abstração para dados que não cabem nos registradores (armazenada fisicamente na memória global), enquanto a memória constante é otimizada para dados que não mudam durante a execução do *kernel*, possuindo *caches* dedicados.

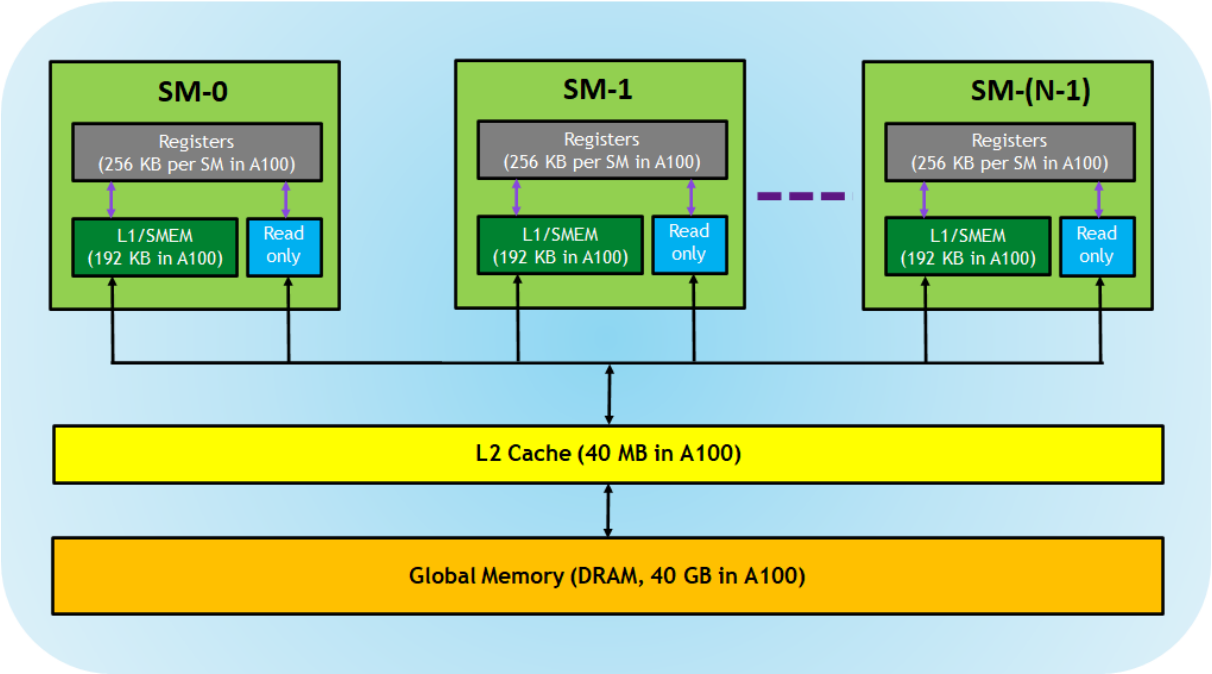


Figura 2 – Hierarquia de memória em GPUs. fonte: NVIDIA (2023)

2.4 Padrões de Paralelismo e Algoritmos

Nem todos os problemas computacionais são adequados para a arquitetura massivamente paralela das GPUs. A análise da estrutura do algoritmo e do padrão de acesso aos dados permite

[1] Multiprocessador (*Streaming Multiprocessor*): Unidade fundamental de processamento na arquitetura da GPU, responsável por gerir e executar os blocos de . Cada SM é composto por diversos núcleos de execução, um conjunto de registradores privados e memória compartilhada de alta velocidade.

classificar quais aplicações terão ganho de desempenho (*speedup*) e quais sofrerão degradação (*slowdown*).

2.4.1 Paralelismo de Dados

O paralelismo de dados representa o cenário ideal para a arquitetura de GPUs, caracterizando-se pela capacidade de distribuir os dados de entrada entre múltiplos processadores para que sejam operados de forma independente. A premissa fundamental desse padrão é a ausência de dependência entre as operações, em que o cálculo de um elemento não necessita do resultado do elemento vizinho ou anterior. Essa característica permite que milhares de *threads* executem a mesma instrução simultaneamente sobre diferentes partes dos dados, maximizando o *throughput* do dispositivo e garantindo uma alta taxa de ocupação dos núcleos de processamento.

Algoritmos que se enquadram nessa categoria geralmente acessam a memória de forma regular e previsível. Quando múltiplas *threads* leem ou escrevem em endereços de memória contíguos simultaneamente, o hardware da GPU consegue agrupar essas transações em uma única operação de acesso, fenômeno conhecido como acesso coalescido (*coalesced access*). Devido a essa eficiência no uso do barramento de memória e à independência das tarefas, essas aplicações tendem a apresentar uma escalabilidade linear, em que o desempenho melhora proporcionalmente ao aumento do número de núcleos disponíveis na GPU.

2.4.2 Dependência de Dados e Acesso Irregular

Em contrapartida, algoritmos que apresentam fortes dependências de dados impõem desafios severos ao modelo SIMT (Single Instruction, Multiple Threads) das GPUs. A dependência sequencial ocorre quando a execução de uma etapa está estritamente condicionada ao resultado da etapa imediatamente anterior, o que impede a execução simultânea. Algoritmos baseados em estruturas recursivas, pilhas ou que exigem uma ordem estrita de visitação forçam a serialização da execução. Nesses casos, a capacidade de processamento paralelo da GPU é prejudicada, pois a maioria das threads precisa aguardar enquanto uma única *thread* resolve a dependência atual, deixando potencialmente milhares de núcleos ociosos.

Além da serialização lógica, outro fator degradante é o acesso irregular à memória, conhecido como divergência de memória (*memory divergence*). Em estruturas de dados dinâmicas ou baseadas em ponteiros e indireção, os dados frequentemente estão armazenados em endereços de memória distantes e não contíguos. Isso obriga a GPU a realizar múltiplas transações de memória para buscar dados dispersos, impedindo o agrupamento de requisições. Como resultado, a memória global opera com eficiência mínima, criando um gargalo de desempenho que muitas vezes torna a execução na GPU mais lenta do que em arquiteturas de CPU, que possuem grandes caches para mitigar esse tipo de latência.

3 TRABALHOS RELACIONADOS

Ribeiro, Sêmeler e Dias (2024) avaliou o comportamento de dois algoritmos, o algoritmo de números primos e o algortimo de Monte Carlo, e os implementou de forma sequencial e paralela utilizando as bibliotecas OpenMP e MPI. Os experimentos foram conduzidos em ambientes de baixo custo, como o Raspberry Pi 4 e o ClusterPi, com o objetivo de analisar o impacto da paralelização sobre o tempo de execução e o consumo de recursos computacionais.

Os resultados indicaram que o algoritmo de números primos apresentou ganhos expressivos de desempenho com a paralelização, atingindo *speedups* de até 2,94x com o OpenMP e 1,44x com MPI. Por outro lado, o algoritmo de Monte Carlo só demonstrou melhora significativa quando foi submetido a cargas de trabalho maiores, evidenciando que nem todo algoritmo se beneficia igualmente da execução paralela. O estudo também apontou que o *overhead* introduzido pelas chamadas de comunicação (no caso do MPI) ou pela criação de múltiplas *threads* (no OpenMP) pode anular os ganhos esperados, especialmente em tarefas de curta duração ou com baixa complexidade computacional. Veja Figura 3 e 4.

Iteração	Intervalo [1, 1.000.000]			Intervalo [1, 10.000.000]			Intervalo [1, 100.000.000]		
	Serial	OpenMP (4 *cores*)	MPI (5 nós)	Serial	OpenMP (4 *cores*)	MPI (5 nós)	Serial	OpenMP (4 *cores*)	MPI (5 nós)
1	1,158	0,395	0,905	29,551	10,137	15,180	779,229	265,409	366,445
2	1,164	0,400	0,832	29,429	9,955	14,509	779,943	265,922	326,440
3	1,226	0,437	0,991	29,503	10,080	17,554	779,151	271,152	307,113
4	1,160	0,398	1,022	29,411	10,152	14,668	779,392	272,546	306,472
5	1,231	0,395	0,888	30,469	10,055	16,419	799,451	273,054	305,390
Média	**1,190**	**0,405**	**0,928**	**29,673**	**10,076**	**15,666**	**783,450**	**269,620**	**322,372**
Speedup	--	**2,93**	**1,28**	--	**2,94**	**1,89**	–	**2,91**	**2,43**

Figura 3 – Tempo de execução do algoritmo de números primos.

Iteração	1.000.000 de pontos			5.000.000 de pontos			10.000.000 de pontos		
	Serial	OpenMP (4 *cores*)	MPI (5 nós)	Serial	OpenMP (4 *cores*)	MPI (5 nós)	Serial	OpenMP (4 *cores*)	MPI (5 nós)
1	0,320	0,175	1,140	1,315	0,900	3,420	2,537	1,650	3,771
2	0,267	0,185	2,000	1,252	0,829	3,360	2,550	1,774	3,034
3	0,267	0,179	1,600	1,371	0,855	3,360	2,498	1,718	3,370
4	0,253	0,194	2,050	1,316	0,874	3,560	2,516	1,764	3,185
5	0,254	0,174	2,240	1,377	0,901	3,810	2,544	1,779	4,632
Média	**0,272**	**0,181**	**1,806**	**1,326**	**0,872**	**3,503**	**2,529**	**1,737**	**3,598**
Speedup	--	**1,5**	**0,2**	--	**1,52**	**0,4**	--	**1,46**	**0,7**

Figura 4 – Tempo de execução do algoritmo de Monte Carlo.

Esse trabalho reforça a importância de avaliar as características convencionais de cada algoritmo, como independência entre operações, custo de comunicação e escalabilidade, antes de decidir por sua paralelização. A análise cuidadosa desses fatores é fundamental para que a aplicação de técnicas paralelas não apenas produza ganhos de desempenho, mas também mantenha a viabilidade e eficiência do sistema como um todo.

A escolha do modelo de programação paralela ideal pode influenciar significativamente o desempenho de aplicações científicas, principalmente quando se utiliza hardware especializado como as GPUs. O trabalho de Khalilov e Timoveev (2021) contribui para essa discussão ao comparar, sob as mesmas condições experimentais, três modelos de programação paralela: Cuda, OpenACC e OpenMP. Todas foram testadas e uma GPU NVIDIA Tesla V100.

Os autores realizaram uma análise detalhada da performance dos modelos em diferentes contextos computacionais: multiplicação de matrizes (função sgemm), operações com acesso regular à memória (via o *benchmark* BabelStream) e aplicações reais como Cloverleaf e LULESH. Os resultados indicaram que o modelo de mais alto nível (OpenACC e OpenMP), especialmente em situações que exigem maior otimização de recursos de *hardware*, como uso eficiente da memória compartilhada e controle de execução de *threads*.

No *benchmark* BabelStream, por exemplo, CUDA atingiu até 91% da largura de banda máxima de memória, enquanto OpenACC e OpenMP ficaram cerca de 3% a 7% abaixo. Em aplicações reais, as diferenças foram ainda mais expressivas: no caso da simulação LULESH, a versão do CUDA foi até 100 vezes mais rápida do que a versão equivalente em OpenACC em tamanhos pequenos, e cerca de 30 vezes em tamanhos maiores.

Essa análise evidencia que, embora modelos de mais alto nível facilitem o desenvolvimento (por meio de diretivas como #pragma), eles podem não explorar totalmente o potencial da arquitetura paralela da GPU. A depender da complexidade do algoritmo e da estrutura dos dados, o overhead gerado por compiladores e abstrações pode limitar os ganhos de desempenho.

No contexto de otimização combinatória, Dorneles (2021) investigou o impacto da granularidade do paralelismo em Algoritmos Genéticos utilizando CUDA. O autor comparou duas abordagens: o paralelismo por indivíduo (grão grosso, que foca na execução de unidades de trabalho robustas e independentes) e o paralelismo por dimensão (grão fino, foca na divisão extrema do problema em componentes mínimos). Os experimentos demonstraram que a abordagem de grão fino escala de forma superior, permitindo um uso mais eficiente dos recursos da GPU. Esse resultado corrobora a premissa de que algoritmos que permitem a decomposição dos dados em tarefas minúsculas e cooperativas tendem a obter melhores *speedups* do que aqueles baseados em tarefas complexas e independentes.

Ferraz et al. (2022) destacam os desafios de implementar a Busca em Profundidade (DFS) em arquiteturas de GPU. O estudo aponta que a natureza irregular do acesso à memória e a divergência de execução entre *threads* tornam a paralelização direta do DFS ineficiente em comparação com abordagens sequenciais ou baseadas em BFS. Embora os autores proponham técnicas avançadas de otimização (*warp-centric*), o trabalho serve como base para compreender as barreiras arquiteturais que limitam algoritmos baseados em recursão e dependência de dados em ambientes SIMT.

O trabalho de Nogueira et al. (2024) investigou o desempenho e o consumo energético de quatro algoritmos recursivos, neste caso, os algoritmos de Mergesort, Quicksort, Breadth-First Search (BFS) e Single-Source Shortest Path (SSSP). Todos os algoritmos foram implementados em três APIs diferentes: OpenMP, CUDA, CUDA Dynamic Parallelism (DP).

O estudo avaliou o comportamento desses algoritmos sob diferentes volumes de entrada, analisando o impacto da paralelização tanto em tempo de execução quanto em consumo de energia (figuras 5 e 6). Os resultados indicaram que o CUDA DP pode oferecer vantagens significativas, especialmente para algoritmos como o Mergesort, que apresentaram melhorias de até 23x no tempo de execução e 7x no consumo de energia em comparação com suas versões em OpenMP e CUDA. Por outro lado, o desempenho de algoritmos como o Quicksort com CUDA DP foi inferior às demais alternativas em cenários em que há maior profundidade recursiva, devido às limitações da extensão DP, como o limite de chamadas recursivas e o custo das chamadas de *kernel*.

Além disso, o estudo evidenciou que o comportamento dos algoritmos e a natureza dos dados processados influenciam diretamente os ganhos da paralelização. No caso de algoritmos de busca em grafos, como o BFS e o SSSP, o desempenho variou conforme o grau de conectividade e o balanciamento de carga de trabalho. A implementação CUDA DP, por exemplo, apresentou ganhos marginais em relação ao CUDA no BFS, mas foi menos eficiente no SSSP devido à falta de balanceamento nas chamadas recursivas.

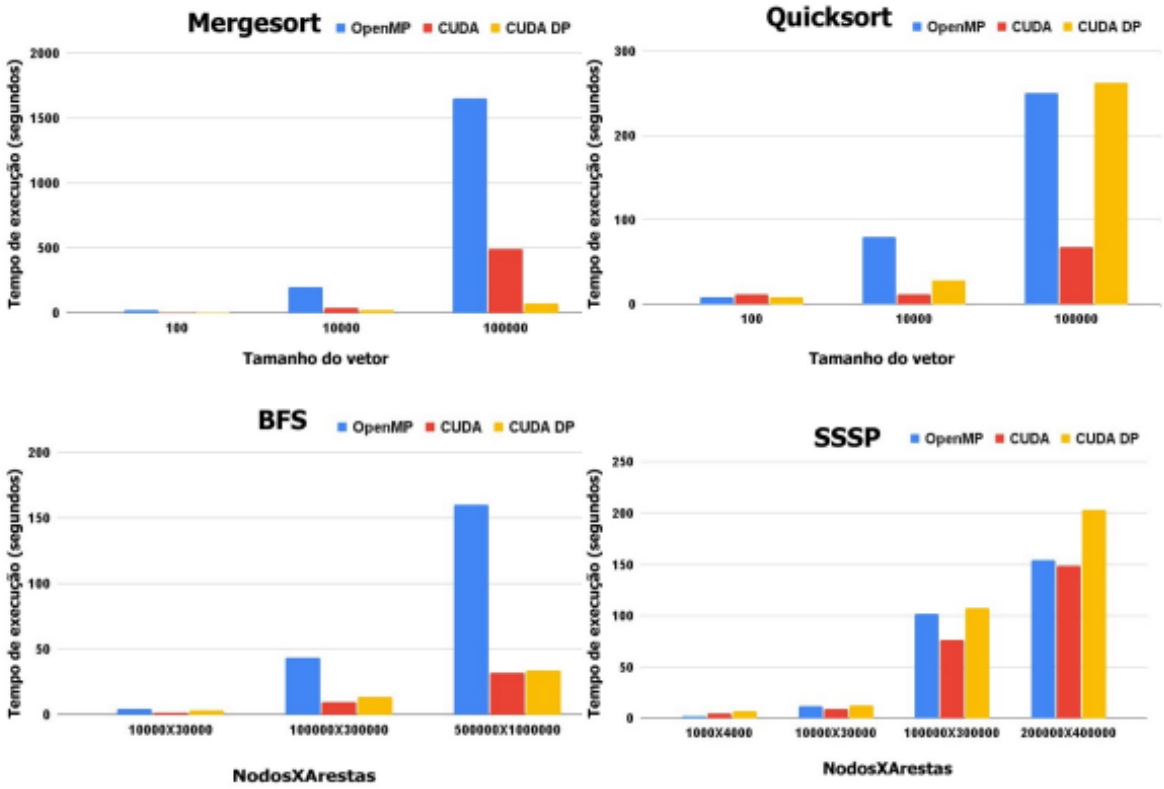


Figura 5 – Tempo de execução em segundos dos experimentos realizados, separados por algoritmo e tamanho das entradas utilizadas..

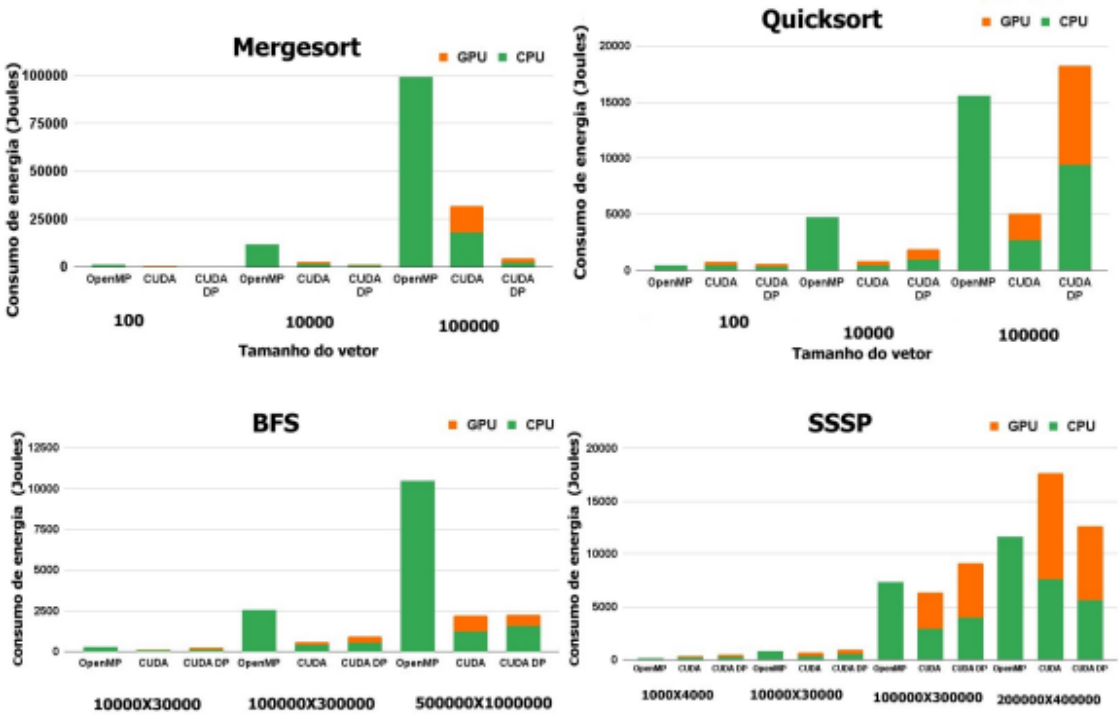


Figura 6 – Apresenta os resultados da soma da métrica de consumo de energia em joules da CPU e GPU.

Tabela 1 – Comparação entre trabalhos relacionados e o presente estudo.

Autor(es)	Algoritmos Analisados	Modelo/Paralelismo	Métricas	Conclusões Principais
Ribeiro et al. (2024)	Números Primos, Monte Carlo	Paralelismo de dados (OpenMP, MPI) em CPU	Tempo, Speedup	Algoritmos com baixa carga computacional não se beneficiam significativamente da paralelização em *clusters* de baixo custo.
Khalilov e Timoveev (2021)	MatMul, BabelStream, Cloverleaf, LULESH	Comparação entre CUDA (baixo nível) vs. OpenACC/OpenMP (alto nível) em GPU	Tempo, Banda de Memória	CUDA apresenta desempenho superior em aplicações reais complexas, apesar da maior dificuldade de implementação em comparação a diretivas.
Nogueira (2023)	Quicksort, Mergesort, BFS, SSSP	CUDA *Dynamic Parallelism* (recursão na GPU) vs. OpenMP	Tempo, Energia	O paralelismo dinâmico é eficaz para cargas balanceadas (Mergesort), mas ineficiente para recursão desbalanceada (Quicksort/DFS).
Dorneles (2021)	Algoritmos Genéticos	Paralelismo por Indivíduo (grosso) vs. por Dimensão (fino)	Tempo, Speedup	O paralelismo de grão fino (por dimensão/dado) escala melhor na GPU, aproveitando a arquitetura SIMT de forma mais eficiente.
Ferraz et al. (2022)	Mineração de Padrões em Grafos	Estratégias *Warp-centric* e *DFS-wide*	Tempo, Eficiência	O DFS tradicional sofre com acesso irregular à memória e divergência; técnicas avançadas de cooperação de *warp* são necessárias para mitigação.
(Este Trabalho)	Counting Sort, DFS	Comparação CPU vs. GPU (CUDA) com foco em estrutura do algoritmo	Tempo, Speedup, Eficácia	Confirma que a GPU (modelo SIMT) é ideal para paralelismo de dados massivo (*Counting Sort*), mas ineficaz para lógica estritamente sequencial/recursiva (*DFS*) sem reestruturação profunda.

Fonte: Elaborada pelo autor.

4 METODOLOGIA

A metodologia adotada neste trabalho foi estruturada de forma sistemática para garantir a robustez da análise de desempenho. Conforme ilustrado no fluxograma da Figura 7, o processo de investigação inicia-se com a seleção de algoritmos com características opostas e segue um ciclo rigoroso de implementação, definição experimental e execução. O estudo consiste em implementar, paralelizar e comparar o desempenho de quatro versões de algoritmos, em que cada cenário foi validado por meio de 30 (trinta) execuções independentes.

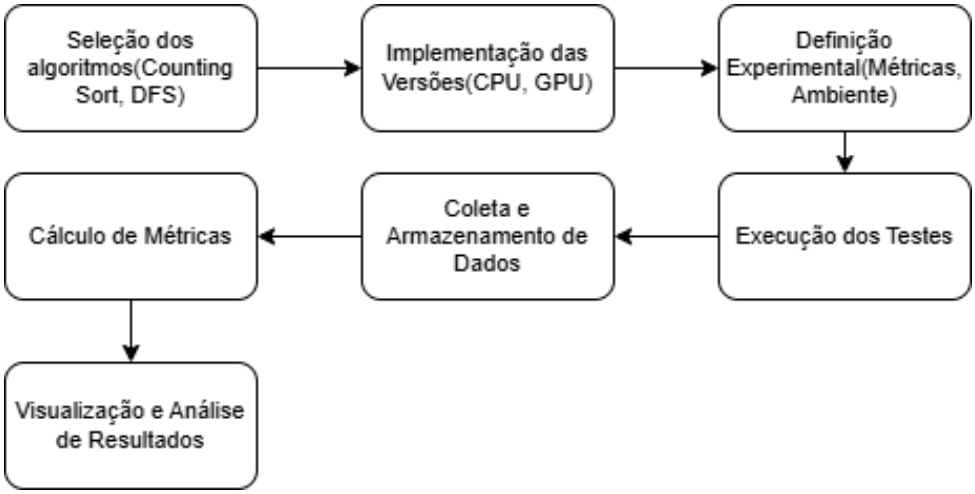


Figura 7 – Fluxograma da metodologia.

- Counting Sort Sequencial (CPU)
- Counting Sort Paralelo em CUDA (GPU)
- DFS Sequencial (CPU)
- DFS Paralelo em CUDA (GPU — *kernel* único com pilha na GPU)

Esses dois algoritmos foram escolhidos por representarem padrões computacionais distintos:

- o Counting Sort apresenta paralelismo de dados e acesso regular à memória, favorecendo a paralelização;
- o DFS possui dependências e acesso irregular, tornando a paralelização mais complexa.

A comparação permite analisar quando e por que a execução paralela em CUDA produz ganhos, ou não, dependendo da estrutura do algoritmo.

4.1 Ambiente de Execução

Os experimentos serão conduzidos em uma máquina heterogênea com arquitetura x64 e as seguintes especificações de hardware e software :

- Microsoft Windows 11 Pro
- Processador 11th Intel(R) Core(TM) i-7-1165G7 2.80GHz.
- 16GB de memoria ram
- GPU NVIDIA Geforce MX450.
- CUDA TOOLKIT 12.8.
- Compilador NVCC (NVIDIA CUDA Compiler).

4.2 Coleta de Métricas e Ferramentas

As seguintes ferramentas foram utilizadas:

- cudaEvent_t: Para tempo de execução GPU
- std::chrono: Para tempo CPU
- nvidia-smi: Para monitoramento da GPU

As métricas analisadas incluem:

- **Tempo de Execução (ms)**
- **Speedup =** Tsequencial / Tparelelo
- **Eficiência =** Speedup / Nº de threads

Essas métricas foram aplicadas tanto ao Counting Sort quanto ao DFS, permitindo avaliar o grau de paralelismo aproveitado por cada algoritmo.

4.3 Algoritmos Avaliados[2]

4.3.1 Counting Sort (CPU)

A versão sequencial do Counting Sort segue a estrutura clássica do algoritmo:

- Etapa de contagem: cada elemento da entrada é percorrido uma única vez.
- Cálculo das posições de saída: o vetor de frequências é transformado em um prefixo acumulado, indicando a posição inicial de cada chave no vetor ordenado.
- Redistribuição dos elementos: os valores da entrada são percorridos novamente e colocados diretamente em suas posições finais.

[2] códigos disponiveis em: https://github.com/TeusVieira/Cuda/blob/main/CUDA

Algoritmo 1 Counting Sort Sequencial (CPU)

Entrada: Vetor de entrada $A[0..N-1]$, Valor máximo K
Saída: Vetor ordenado $B[0..N-1]$

Inicializar $Count[0..K-1] \leftarrow 0$
▷ 1. Histograma
para $i \leftarrow 0$ **até** $N-1$ **faça**
    $Count[A[i]] \leftarrow Count[A[i]] + 1$
fim para
▷ 2. Soma de Prefixos
para $i \leftarrow 1$ **até** $K-1$ **faça**
    $Count[i] \leftarrow Count[i] + Count[i-1]$
fim para
▷ 3. Ordenação (Scatter)
para $i \leftarrow N-1$ **até** 0 **faça** ▷ Ordem inversa para estabilidade
    $Chave \leftarrow A[i]$
    $Pos \leftarrow Count[Chave] - 1$
    $B[Pos] \leftarrow Chave$
    $Count[Chave] \leftarrow Count[Chave] - 1$
fim para

4.3.2 Counting Sort(GPU)

A versão paralela do Counting Sort adapta a lógica sequencial para a arquitetura de GPU, dividindo o processamento em três etapas coordenadas. Inicialmente, realiza-se a construção paralela do histograma, em que a entrada é particionada entre múltiplas *threads* que acumulam as frequências das chaves de forma concorrente mediante sincronização. Em seguida, aplica-se uma operação de prefix-sum paralelo (varredura) para transformar essas frequências em um prefixo acumulado, determinando as posições iniciais de cada chave de maneira distribuída entre as unidades de processamento. Por fim, executa-se a redistribuição paralela, na qual as *threads* percorrem novamente a entrada para posicionar os elementos no vetor de saída, calculando os índices de escrita com base nos prefixos acumulados.

Algoritmo 2 Counting Sort Paralelo (Kernel CUDA)

Entrada: Vetor $d\_in[0..N-1]$ na GPU
Saída: Vetor $d\_out[0..N-1]$ na GPU

Variáveis Globais: d_hist, $d\_offsets$
procedure KERNELHISTOGRAMA
    $idx \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
    se $idx < N$ **então**
        $chave \leftarrow d\_in[idx]$
        atomicAdd$(\&d\_hist[chave], 1)$
    fim se
fim procedure
▷ No Host: Executar Thrust::ExclusiveScan(d_hist) $\rightarrow d\_offsets$
procedure KERNELSCATTER
    $idx \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
    se $idx < N$ **então**
        $chave \leftarrow d\_in[idx]$
        $pos \leftarrow$ **atomicAdd**$(\&d\_offsets[chave], 1)$
        $d\_out[pos] \leftarrow chave$
    fim se
fim procedure

4.3.3 DFS(CPU)

A Implementação sequencial da busca em profundidade utiliza uma estrutura de pilha para armazenar os vértices que serão explorados. A cada iteração:

- Um vértice é removido da pilha.
- O vértice é registrado na ordem de visita.
- Todos os seus vizinhos não visitados são adicionados à pilha.
- Esse processo se repete até que não haja vértices pendentes.

Como o grafo utilizado é representado por listas de adjacência, o algoritmo percorre todos os vértices e suas arestas de forma determinística.

Algoritmo 3 Busca em Profundidade (DFS) Sequencial

Entrada: Grafo G (Lista de Adjacência), Nó Inicial S
Saída: Vetor de ordem de visitação R

```
1: Criar pilha P
2: Empilhar(P, S)
3: Marcar S como visitado
4: enquanto P não está vazia faça
5:     U ← Desempilhar(P)
6:     Adicionar U em R
7:     para cada vizinho V de U faça
8:         se V não foi visitado então
9:             Marcar V como visitado
10:            Empilhar(P, V)
11:        fim se
12:    fim para
13: fim enquanto
```

4.3.4 DFS(GPU)

Embora o DFS seja naturalmente sequencial, uma versão experimental foi implementada na GPU para avaliar o impacto da arquitetura paralela em algoritmos com forte dependência de ordem.

A adaptação utilizou uma abordagem em que:

- Toda a lógica do DFS é executada dentro de um único kernel.
- Uma pilha é mantida e manipulada diretamente na GPU.
- Os vértices são visitados seguindo a mesma ordem da versão sequencial;
- O grafo é armazenado em estruturas lineares compactadas, adequadas ao modelo de memória da GPU.

Algoritmo 4 DFS Serializado em GPU (Kernel Único)

Entrada: Grafo CSR ($Offsets, Edges$), Nó Inicial S
Entrada: Pilha $Stack$ (Memória Global), Vetor $Visited$
Saída: Contador de nós visitados d_count

```
1:  procedure DFSKERNEL
2:      tid ← threadIdx.x
3:      bid ← blockIdx.x
4:      se tid == 0 e bid == 0 então                    ▷ Execução Serial (1 Thread)
5:          Topo ← −1
6:          Stack[++Topo] ← S                           ▷ Empilha nó inicial
7:          Visited[S] ← True
8:          LCount ← 0
9:          enquanto Topo > −1 faça
10:             U ← Stack[Topo]                         ▷ Peek (Espiar topo)
11:             Achou ← False
12:             Ini ← Offsets[U]
13:             Fim ← Offsets[U + 1]
14:             para i ← Ini até Fim faça               ▷ Acesso Coalescido
15:                 V ← Edges[i]
16:                 se não Visited[V] então
17:                     Visited[V] ← True
18:                     Stack[++Topo] ← V
19:                     d_result[LCount++] ← V
20:                     Achou ← True
21:                     break                           ▷ Aprofunda na busca
22:                 fim se
23:             fim para
24:             se não Achou então
25:                 Topo ← Topo − 1                     ▷ Pop (Backtrack)
26:             fim se
27:         fim enquanto
28:         ∗d_count ← LCount
29:     fim se
30: fim procedure
```

4.4 Análise de dados

Os dados obtidos nos experimentos foram organizados em planilhas e representados graficamente para facilitar a análise comparativa. Serão produzidos gráficos de tempo de execução, speedup e eficiência, além de tabelas com os resultados consolidados por algoritmo e tamanho de entrada.

5 RESULTADOS E DISCUSSÕES

Nesta seção, são apresentados os resultados experimentais obtidos na execução dos algoritmos Counting Sort e DFS. Os testes foram realizados conforme a metodologia descrita, utilizando 30 execuções para cada algoritmo a fim de garantir a relevância estatística dos dados.

O ambiente de testes compreendeu uma CPU Intel Core i7 de 11ª geração e uma GPU NVIDIA GeForce MX450.

Para facilitar a interpretação dos dados, a coluna 'Eficácia' nas tabelas classifica o desempenho da GPU baseado no *Speedup*[3] (S) obtido, considerando a configuração de execução com 896 *threads* por bloco. Esta métrica, definida na metodologia, categoriza os resultados como '*Slowdown*/Inviável' quando a paralelização massiva introduz latência superior ao ganho de processamento ($S < 1$); 'Ganho Moderado' para acelerações entre 1 e 3 vezes; e 'Ganho Expressivo/Máximo' para cenários em que a GPU satura eficientemente as threads, superando a CPU em mais de 3 vezes.

5.1 Análise de Desempenho: Counting Sort

O algoritmo Counting Sort foi avaliado comparando-se sua implementação sequencial (CPU) com a versão paralelizada em CUDA (GPU). Foram realizados testes escalonados com vetores variando de 10^5 (100 mil) a 10^8 (100 milhões) de elementos. A Tabela 2 apresenta os tempos médios e o *speedup* obtido.

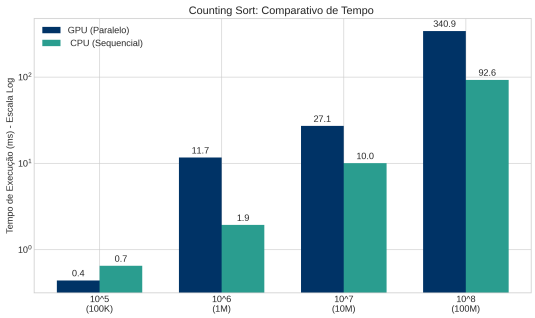
Tabela 2 – Comparação de desempenho e *speedup* entre CPU e GPU

Tamanho da Entrada (N)	**Tempo CPU (ms)**	**Tempo GPU (ms)**	**Speedup**	**Eficácia**
100.000 (10^5)	0,44	0,65	0,67x	Slowdown
1.000.000 (10^6)	11,66	1,93	6,03x	Ganho Máximo
10.000.000 (10^7)	27,08	10,04	2,70x	Ganho Moderado
100.000.000 (10^8)	340,85	92,58	3,68x	Ganho Consistente

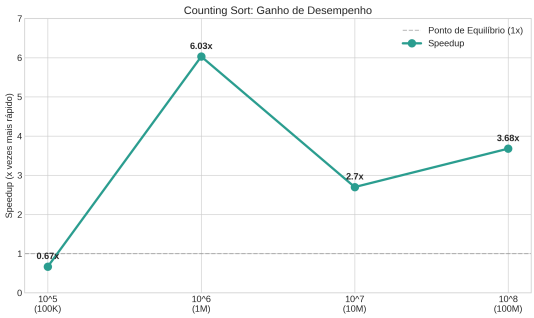
As figuras 8a e 8b demonstram a escalabilidade do Counting Sort na GPU. Inicialmente ineficiente para 10^5 elementos (*speedup* de 0,67x) devido ao *overhead* de inicialização, o desempenho atinge seu pico com 10^6 elementos (ganho de 6,03x). Mesmo sob carga máxima de 10^8, a GPU mantém vantagem significativa. O ganho estabiliza-se em 3,68x para grandes entradas, indicando saturação da largura de banda, mas sustentando superioridade sobre a execução sequencial.

[3] O *speedup* (S) é a métrica utilizada para quantificar o ganho relativo de desempenho obtido pela paralelização. Neste trabalho, é calculado pela razão $S = T_{seq}/T_{par}$, onde T_{seq} representa o tempo de execução do algoritmo sequencial (CPU) e T_{par} o tempo de execução do algoritmo paralelo (GPU). Um valor de $S > 1$ indica aceleração (ganho de desempenho), enquanto um valor entre 0 e 1 indica desaceleração (*slowdown*).

Figura 8 – (a) Comparativo das medias de tempo do algoritmo de counting sort em CPU x GPU e (b) Speedup obtido



(a) Tempos de execução GPU x CPU.



(b) Speedup obtido pelo algoritmo em GPU.

5.1.1 Discussão

Os resultados evidenciam o comportamento típico de algoritmos com alto grau de paralelismo de dados. Para entradas pequenas ($N = 10^5$), a execução na GPU apresentou desempenho inferior à CPU (*speedup* de 0,67x) devido ao *overhead* de transferência de dados e inicialização dos *kernels*.

O cenário se inverte a partir de 10^6 elementos, em que a GPU atingiu um *speedup* expressivo de 6 vezes. Nesse ponto, a quantidade de dados é suficiente para saturar os núcleos de processamento, mascarando a latência de memória. Para cargas massivas (10^8 elementos), a GPU manteve-se cerca de 3,6 vezes mais rápida que a CPU, demonstrando robustez e escalabilidade.

Os resultados positivos obtidos com o Counting Sort alinham-se às observações de Khan et al. (2011), que, ao analisarem algoritmos de ordenação paralela, identificaram que o desempenho na GPU é maximizado quando se minimiza a sincronização e se explora a independência de dados. Assim como o Bitonic Sort analisado pelos autores, o Counting Sort aqui implementado beneficia-se da arquitetura massivamente paralela para superar as limitações de frequência de *clock* da CPU por meio do *throughput* elevado.

5.2 Análise de Desempenho: Busca em Profundidade(DFS)

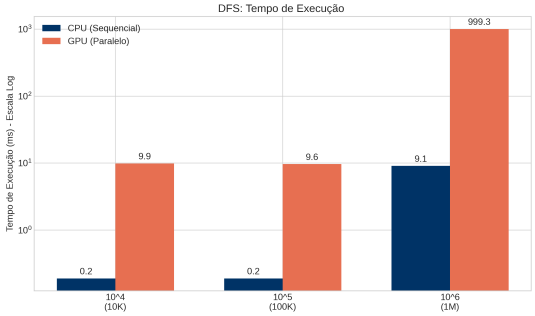
Para avaliar o impacto da arquitetura paralela em algoritmos com dependência de dados, o DFS foi testado em grafos com tamanho variando de 10.000 a 1.000.000 de vértices.

Tabela 3 – Análise de desempenho: Cenários de ineficiência da GPU

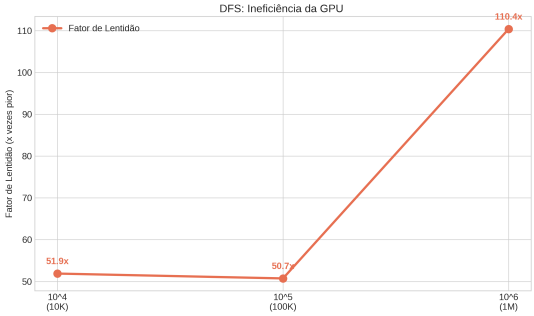
Nº de Vértices	Tempo CPU (ms)	Tempo GPU (ms)	Speedup	Eficácia
10.000 (10^4)	0,19	9,86	0,02x	Inviável
100.000 (10^5)	0,19	9,64	0,02x	Inviável
1.000.000 (10^6)	9,05	999,32	0,009x	Pior Caso

A comparação CPU-GPU (figura 9a) no DFS revela a ineficiência da arquitetura paralela para lógicas sequenciais. A GPU sofre um slowdown severo: para 10^6 vértices, leva 999,32 ms contra apenas 9,05 ms da CPU. O desempenho degrada-se com o aumento da complexidade, conforme se observa na figura 9b, atingindo um speedup ínfimo de 0,009x. Essa queda deve-se à serialização da pilha e ao acesso irregular à memória, incompatíveis com o modelo SIMT, tornando o custo da GPU proibitivo frente à CPU.

Figura 9 – (a) comparativo do tempo de execução em sequencial e paralelo do DFS e (b) Speedup obtido pelo DFS.



(a) Tempo de execução GPU x CPU.



(b) Slowdown obtido pelo algoritmo em GPU.

5.2.1 Discussão

Os resultados do DFS demonstram uma degradação severa de desempenho na GPU em todos os cenários. Para grafos de até 10^5 vértices, o tempo na GPU estacionou em 9,8 ms, indicando que o custo dominante foi o *overhead* de lançamento do *kernel*, enquanto a CPU executou a tarefa quase instantaneamente (0,19 ms).

O cenário mais crítico ocorreu no teste de estresse com 1 milhão de vértices. Enquanto a CPU escalou eficientemente (9,05 ms), o tempo na GPU subiu para quase 1 segundo (999,32 ms), resultando em uma execução mais de 100 vezes mais lenta. Esse comportamento confirma que a serialização forçada pela pilha do DFS e o acesso irregular à memória anulam qualquer benefício da GPU, tornando a CPU a arquitetura preferencial para esta classe de algoritmos.

6 CONCLUSÃO

Este trabalho investigou o impacto da paralelização de algoritmos utilizando a plataforma CUDA em ambiente GPU, com o objetivo principal de identificar quais características tornam um algoritmo apto ou inapto para este modelo de computação. Por meio da implementação e comparação de desempenho entre o Counting Sort (representando o paralelismo de dados) e a Busca em Profundidade - DFS (representando dependência de dados), foi possível estabelecer uma base comparativa sólida para orientar a escolha de estratégias de paralelização.

Os experimentos demonstraram que a arquitetura da GPU oferece ganhos de desempenho expressivos, mas estritamente condicionados à estrutura do algoritmo. O Counting Sort paralelo validou a eficiência do modelo SIMT (*Single Instruction, Multiple Threads*) para tarefas com

independência de dados e acesso regular à memória. Nos testes com 1 milhão de elementos, a versão em GPU obteve um *speedup* de 6,03 vezes em relação à CPU, mantendo-se consistente mesmo em cargas massivas de 100 milhões de registros.

Em contrapartida, os resultados obtidos com o algoritmo DFS confirmaram as limitações teóricas da plataforma para problemas com forte dependência sequencial. A necessidade de manter uma ordem estrita de visitação (pilha) e o padrão de acesso irregular à memória (grafo) resultaram em uma degradação severa de desempenho. No cenário de estresse com 1 milhão de vértices, a execução na GPU foi mais de 100 vezes mais lenta que na CPU. Este comportamento alinha-se aos achados de Cardoso e Santiago (2013) na análise de algoritmos NP-Completos, como o Clique Máximo. Os autores identificaram que a abordagem de recursão direta (*backtracking*) em GPU é ineficiente devido ao consumo de memória e à divergência de *threads*, concluindo que o ganho de desempenho só é viável mediante reestruturações profundas, como a divisão em subgrafos independentes. Essa correlação valida a nossa observação de que o *overhead* de gerenciamento de *threads* e a latência de memória global anulam qualquer benefício do paralelismo massivo para esta classe de problemas sem uma mudança de paradigma algorítmico.

Conclui-se, portanto, que a decisão de portar um código para CUDA não deve basear-se apenas na complexidade computacional, mas primariamente na análise dos padrões de paralelismo e acesso à memória. Algoritmos baseados em processamento de vetores e matrizes com acesso coalescido são candidatos ideais, enquanto algoritmos recursivos ou com dependências estritas (como o DFS iterativo) devem permanecer na CPU, onde se beneficiam de *caches* maiores e frequências de *clock* mais altas.

6.1 Trabalhos futuros

Para a sustentabilidade em HPC (High Performance Computation), trabalhos futuros devem priorizar métricas granulares (como FLOPS/Watt ou Soluções/Joule) em detrimento do consumo total. Além disso, o monitoramento de potência em tempo real permitiria identificar desperdícios em fases específicas da execução, viabilizando otimizações direcionadas como o ajuste dinâmico de voltagem e frequência (DVFS).

A generalização dos resultados obtidos requer a validação em um espectro mais amplo de *hardware*. Seria valioso estender os experimentos para arquiteturas de GPU mais recentes da NVIDIA (como as microarquiteturas Ampere ou Hopper) para verificar se melhorias no *hardware* de escalonamento mitigaram os custos do Paralelismo Dinâmico. Adicionalmente, explorar a portabilidade dos algoritmos para GPUs de outros fabricantes (AMD e Intel), por meio de APIs abertas como OpenCL ou SYCL, permitiria isolar se as limitações encontradas são inerentes ao modelo SIMT geral ou específicas da implementação CUDA.

Propõe-se a criação de um guia de melhores práticas que sintetize os achados empíricos deste estudo. Ao estabelecer critérios claros baseados em recursão e acesso à memória, o guia auxiliaria na seleção da arquitetura ideal (CPU, híbrida ou GPU), prevenindo implementações

ineficientes em que o paralelismo não compensa o esforço.

REFERÊNCIAS

CARDOSO, E. A.; SANTIAGO, R. de. Análise comparativa de algoritmos np-completo executados em cpu e gpu utilizando cuda. **Anais do Computer on the Beach**, v. 4, p. 79–87, 2013.

CHENG, J.; GROSSMAN, M.; MCKERCHER, T. **Professional CUDA C Programming**. Indianapolis: John Wiley & Sons, 2014. Acesso em: 15 abr. 2025. ISBN 9781118739273.

DESJARDINS, J. **How Much Data is Generated Each Day?** 2019. Disponível em: <https://www.visualcapitalist.com/how-much-data-is-generated-each-day/>.

DORNELES, L. d. L. Comparando diferentes implementações paralelas de algoritmos genéticos em gpus com cuda. 2021.

FERRAZ, S. et al. Efficient strategies for graph pattern mining algorithms on gpus. In: IEEE. **2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. [S.l.], 2022. p. 110–119.

GUPTA, P. **CUDA Refresher: The CUDA Programming Model**. 2020. Disponível em: <https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/>.

Intel Corporation. **CPU versus GPU: qual é a diferença?** 2023. Acesso em: 17 abr. 2025. Disponível em: <https://www.intel.com.br/content/www/br/pt/products/docs/processors/cpu-vs-gpu.html>.

KHALILOV, M.; TIMOVEEV, A. Performance analysis of cuda, openacc and openmp programming models on tesla v100 gpu. **Journal of Physics: Conference Series**, IOP Publishing, v. 1740, n. 1, p. 012056, jan 2021. Disponível em: <https://dx.doi.org/10.1088/1742-6596/1740/1/012056>.

KHAN, F. G. et al. Analysis of fast parallel sorting algorithms for gpu architectures'. In: **2011 Frontiers of Information Technology**. [S.l.: s.n.], 2011. p. 173–178.

NOGUEIRA, A. et al. Análise de desempenho e consumo energético de aplicações recursivas em ambientes openmp, cuda e cuda dp. In: **Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho**. Porto Alegre, RS, Brasil: SBC, 2024. p. 264–275. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/sscad/article/view/31003>.

NVIDIA. **CUDA Refresher: CUDA Programming Model**. 2023. Acesso em: 9 dez. 2025. Disponível em: <https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/>.

NVIDIA Corporation. **CUDA C++ Programming Guide**. Santa Clara, CA, 2024. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Acesso em: 15 abr. 2025.

PEREZ, F. P. et al. **Ensino de Programação Paralela na Educação Básica: Uma Revisão Sistemática da Literatura, Análise Bibliométrica e Agenda para Estudos Futuros**. [S.l.]: SBC, 2023. 19–26 p.

RIBEIRO, G.; SêMELER, L.; DIAS, W. Avaliação de desempenho dos algoritmos de números primos e monte carlo em ambientes hpc. In: **Anais Estendidos do XXV Simpósio em Sistemas Computacionais de Alto Desempenho**. Porto Alegre, RS, Brasil: SBC, 2024. p. 65–72. ISSN

0000-0000. Disponível em: <https://proceedings-sol.sbc.org.br/index.php/sscad_estendido/article/view/30970>.

SANDERS JASON E KANDROT, E. **CUDA by Example: An Introduction to General-Purpose GPU Programming**. 1. ed. Upper Saddle River, NJ: Addison-Wesley Professional, 2010. Disponível em: <https://edoras.sdsu.edu/~mthomas/docs/cuda/cuda_by_example.book.pdf>. Acesso em: 17 abr. 2025. ISBN 9780131387683.

APÊNDICE A - Instalando o CUDA

1 INSTALAÇÃO DO AMBIENTE CUDA

Para o desenvolvimento e execução de programas em GPU, foi utilizado CUDA Toolkit, disponibilizado pela NVIDIA. O toolkit inclui um compilador NVCC, bibliotecas, amostras e ferramentas de depuração e análise de desempenho. A instalação pode ser realizada tanto em sistemas Windows quanto em sistemas Linux.

- **Placa de vídeo compatível com CUDA**
- **Sistema operacional:** Windows 10/11, Ubuntu, Debian ou Fedora
- **Compilador C/C++ (GCC ou MSVC)**
- **Driver NVIDA atualizado**

2 PASSOS DE INSTALAÇÃO NO WINDOWS

- Acesse o site oficial da Nvidia: https://developer.nvidia.com/cuda-downloads
- Escolha a versão compatível com seu sistema operacional e driver.
- Baixe e execute o instalador.
- Marque a opção "Express Installation"para instalar o Toolkit e os drivers.
- Após a instalação, no terminal, verifique o sucesso com o comando:

```
1    nvcc -version
```

ou, no PowerShell:

```
1    nvcc -V
```

o comando deve retornar a versão instalada do CUDA Toolkit

3 INSTALAÇÃO DO LINUX(UBUNTU)

1. Atualize os pacotes do sistema:

```
1    sudo apt update && sudo apt upgrade
```

2. Adicione o repositório da NVIDIA e instale o Toolkit:

```
1    sudo apt install nvidia-cuda-toolkit
```

3. Confirme a instalação:

```
1    nvcc --version
```

4 ESTRUTURA DE PROGRAMAS EM CUDA.

Um programa CUDA geralmente é composto por duas partes:

- **Código do Host (CPU): controla a execução e envia tarefas para a GPU.**
- **Código do Dispositivo (GPU): executa funções paralelas chamadas kernels.**

Exemplo simples de programa CUDA em C++:

```
1   #include <stdio.h>
2
3   __global__ void helloFromGpu(){
4   printf("Ola do thread %d\\n", threadIdx.x);
5   }
6
7   int main(){
8   hello<<<1,4>>>();
9   cudaDeviceSynchronize();
10  return 0;
11  }
```

5 COMPILAÇÃO E EXECUÇÃO.

Após salvar o código em um arquivo, por exemplo, "helloWorld.cu", compile e execute com os comandos:

```
1   nvcc helloWorld.cu -o helloWorld
2   ./helloWorld
```

O programa exibirá a mensagem de "Hello World"e o thread utilizado. Esse exemplo serve para verificar