UNIVERSIDADE FEDERAL DO PAMPA

Angelo Gaspar Diniz Nogueira

Um Estudo da Extensão de Paralelismo Dinâmico Fornecido por CUDA em Aplicações Recursivas

Alegrete
2023

Angelo Gaspar Diniz Nogueira

Um Estudo da Extensão de Paralelismo Dinâmico Fornecido por CUDA em Aplicações Recursivas

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Marcelo Caggiani Luizelli

Coorientador: Prof. Dr. Arthur Francisco Lorenzon

Alegrete
2023

Angelo Gaspar Diniz Nogueira

Um Estudo da Extensão de Paralelismo Dinâmico Fornecido por CUDA em Aplicações Recursivas

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 7 de julho de 2023
Banca examinadora:

Prof. Dr. Marcelo Caggiani Luizelli
Orientador
UNIPAMPA

Prof. Dr. Arthur Francisco Lorenzon
Coorientador
UFRGS

Prof. Dr. Adriano Quilião de Oliveira
UFSM

Prof. Dr. Fábio Diniz Rossir
IFFAR

Assinado eletronicamente por MARCELO CAGGIANI LUIZELLI, PROFESSOR DO MAGISTERIO SUPERIOR, em 10/07/2023, às 09:36, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por Fábio Diniz Rossi, Usuário Externo, em 10/07/2023, às 09:37, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por Adriano Quilião de Oliveira, Usuário Externo, em 10/07/2023, às 09:45, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por Arthur Francisco Lorenzon, Usuário Externo, em 11/07/2023, às 10:04, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir& id_orgao_acesso_externo=0, informando o código verificador 1170691 e o código CRC 4545F630.

RESUMO

A demanda por desempenho de aplicações nos domínios de computação gráfica, aprendizado de máquina, computação de alto desempenho, etc. vêm crescendo ao longo dos anos. Para atender à esta demanda, empresas destes domínios frequentemente tendem a atualizar seus recursos computacionais, como por exemplo, processadores, quantidade de *clusters*, memória, etc. No entanto, esta atualização incremental pode ter um impacto negativo nos custos relacionados a energia. Portanto, uma das alternativas para a melhoria do desempenho de aplicações sem um impacto negativo no consumo de energia de sistemas de alto desempenho é a exploração do paralelismo no nível de *threads*. O paralelismo neste nível pode ser explorado em arquiteturas homogêneas e heterogêneas, através do uso de modelos de programação que tendem a tornar a tarefa do programador mais simples. Um exemplo de tais modelos é o CUDA (*Compute Unified Device Architecture*), um modelo de programação paralela para ambientes heterogêneos compostos de CPUs (*Central Processing Units*) e GPUs -(*Graphics Processing Units*). Nesse sentido, CUDA permite a execução de trechos ou segmentos de uma aplicação na GPU, através de funções chamadas de *kernels*. Por padrão, as chamadas de *kernel* são realizadas na CPU, e como tal, existe um custo associado à comunicação entre CPU e GPU. Dessa forma, os possíveis ganhos de desempenho em algoritmos recursivos são limitadas, já que estes requerem diversas chamadas de *kernel* durante sua execução, devido a recursão. Para resolver esta limitação CUDA fornece ao programador a extensão paralelismo dinâmico, que permite que chamadas de *kernels* sejam realizadas diretamente na GPU. Dessa forma, evitando ou minimizando o custo mencionado. Nesse sentido, este trabalho de conclusão de curso tem como objetivo avaliar o desempenho e consumo de energia da extensão de paralelismo dinâmico do CUDA em algoritmos recursivos.

Palavras-chave: CUDA, Computação Heterogênea, Paralelismo Dinâmico.

ABSTRACT

The demand for application performance in the domains of graphical computation, machine learning, high-performance computing, etc. has been increasing over the years. To meet this demand, enterprises of such domains frequently tend to upgrade their computational resources, such as memory, processor, number of clusters, etc. However, this incremental upgrade may lead to a negative impact on costs related to energy. Therefore, one of the alternatives to increase the performance of applications without an increase in energy consumption of high-performance systems is the exploration of parallelism on the thread level. This level of parallelism can be explored in both homogeneous and heterogeneous architectures. Through the use of programming models that tend to make the programmer's task simpler. One example of such a model is CUDA- (Compute Unified Device Architecture), a parallel programming model for heterogeneous environments composed of CPUs -(Central Processing Units) and GPUs -(Graphics Processing Units). In that sense, CUDA allows for the execution of parts or short segments of an application on the GPU. Through the use and creation of functions called kernels. By default, kernel calls need to be made on the CPU, and as such, there exists a cost associated with the communication between the CPU and GPU. In that sense, possible performance gains of recursive algorithms are limited. Given that, they require multiple kernel calls during their execution due to recursion. Besides that limitation, there also exists the necessity to use specialized kernels, to address the recursions of sub-problems of the graph. To solve this limitation, CUDA offers the programmer the extension dynamic parallelism, which allows kernel calls to be made directly on the GPU. This way, avoiding or minimizing the aforementioned cost. In this sense, this course conclusion work aims to evaluate the performance and energy consumption of the dynamic parallelism extension of CUDA in recursive algorithms.

Key-words:Dynamic parallelism; Parallelism in GPUs; OpenMP.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

API *Application Programming Interface*

BFS *breadth first search*

CPU *Central Processing Units*

CUDA *Compute Unified Device Architecture*

DP *Dynamic Parallelism*

GPU *Graphics Processing Units*

MIMD *Multiple Instruction, Multiple Data*

MISD *Multiple Instruction, Single Data*

SIMD *Single Instruction, Multiple Data*

SIMT *Single Instruction, Multiple Threads*

SISD *Single Instruction, Single Data*

SSSP *Single-Source Shortest Path*

VLIW *Very Long Instruction Word*

SUMÁRIO

1 INTRODUÇÃO

A demanda por desempenho de aplicações nos domínios de computação gráfica, aprendizado de máquina vem crescendo ao longo dos anos. Para atender a esta demanda, empresas destes domínios frequentemente tendem a atualizar seus recursos computacionais, como, por exemplo, processadores, quantidade de *clusters*, memória, etc. No entanto, esta atualização incremental pode ter um impacto negativo em custos relacionados à energia, já que mais recursos computacionais significam maior consumo de energia. Portanto, uma das alternativas para a melhoraria do desempenho de aplicações sem um significativo aumento ou sem aumento em alguns casos, na métrica de consumo de energia de sistemas computacionais de alta desempenho, é a exploração da computação paralela. Em suma, conforme (BARNEY et al., 2010), o conceito de computação paralela refere-se ao processamento simultâneo de operações ou cálculos em uma máquina computacional, através da divisão do problema em instruções computáveis. Além disso, as razões para a aplicação da computação paralela vão além da melhoria do desempenho de aplicações. Estas também incluem melhorias na eficiência energética energia conforme (JIN et al., 2017) e (FALCÃO, 1996).

A computação paralela pode ser explorada nos níveis de dados e/ou tarefas de uma aplicação e de várias formas pelo hardware. Uma das formas mais comumente utilizadas de paralelismo de hardware é o paralelismo no nível de *threads*. O paralelismo neste nível pode ocorrer em arquiteturas homogêneas, onde os núcleos de processamento ou unidades computacionais são semelhantes; Como heterogêneas, onde os sistemas computacionais são compostos pela interconexão de mais de um tipo de núcleo de processamento ou unidades computacionais (GAO; ZHANG, 2016). Estes núcleos ou unidades computacionais, geralmente possuem capacidades especializadas para o processamento de dados. Adicionalmente, existem diversos exemplos de arquiteturas heterogêneas. Exemplos incluem: *clusters* computacionais, computadores de proposito geral, celulares, etc. Ademais, arquiteturas GPU-CPU são constituídas da combinação dos recursos computacionais de uma/várias CPU(s) e uma/várias GPU(s).

Ao explorar o paralelismo, as arquiteturas heterogêneas e homogêneas possuem particularidades próprias. No caso de arquiteturas homogêneas, a exploração do paralelismo no nível de *threads* pode ocorrer através do uso de APIs (*Application Programming Interface*), onde a troca de dados acontece via memória compartilhada. Exemplos incluem: OpenMP (*Open Multi-Processing*) e Pthreads (*POSIX threads*) (ORACLE, 2022). Por outro lado, a exploração do paralelismo em arquiteturas GPU-CPU pode ocorrer através do uso de inúmeras interfaces para programação. Exemplos destas interfaces incluem: OpenACC, Matlab, DirectCompute, etc. Assim, algumas das mais utilizadas são: CUDA e OpenCL. Enquanto CUDA é definida como uma plataforma e modelo de programação que utiliza GPUs (NVIDIA, 2022), OpenCL é definida como um padrão de programação paralela de proposito geral em GPUs (KHORNOS GROUP, 2011). No entanto, este o

foco deste trabalho é o modelo de programação CUDA.

O modelo de programação CUDA consiste da utilização dos núcleos de processamento presentes nas GPUs NVIDIA para a computação paralela de trechos e/ou partes de uma aplicação originada da CPU através da criação e execução de funções na GPU (chamadas de *kernels*). Por padrão, as chamadas de funções de *kernel* precisam ser realizadas na CPU. E como tal, existe um custo associado com a comunicação entre GPU e CPU. Nesse sentido, possíveis ganhos de desempenho de algoritmos recursivos são limitados uma vez que exigem múltiplas chamadas de *kernel* durante sua execução devido a recursão. Exemplos de tais algoritmos incluem: algoritmos de ordenação; algoritmos de processamento de grafos; algoritmos de processamento de estruturas hierárquicas, etc.

Portanto, para resolver as limitações listadas acima, CUDA oferece ao programador a extensão paralelismo dinâmico (*dynamic parallelism* — DP). Esta extensão permite a criação de funções recursivas, onde *kernels* realizam as chamadas de seus subproblema no fluxo de execução da GPU, conforme (NVIDIA, 2022). Isto é, um *kernel* pode realizar chamadas de *kernel* diretamente na GPU sem a necessidade do retorno para CPU. Além disso, a utilização do paralelismo dinâmico também possibilita ao programador a especificação da quantidade de recursos computacionais necessários para a execução de subproblemas na GPU. Portanto, os fatores mencionados reduzem à necessidade de comunicação entre a GPU e CPU. Desta forma, eliminando as principais razões do sobrecusto mencionado anteriormente.

1.1 Objetivos

Considerando o texto elaborado acima, este trabalho visa avaliar o desempenho da extensão CUDA paralelismo dinâmico em algoritmos recursivos amplamente utilizados pela comunidade acadêmica. Afim de relacionar quais são os custos e benefícios que a respectiva extensão traz as métricas de tempo de execução de aplicações e consumo de energia para o modelo de programação CUDA. Para tanto, foi realizada uma comparação entre a extensão CUDA, paralelismo dinâmico e a API OpenMP considerando as métricas mencionadas anteriormente. Para atingir este objetivo, os seguintes objetivos específicos foram definidos:

- Estudar a extensão CUDA paralelismo dinâmico.
- Implementar os algoritmos de ordenação (*mergesort* e *quicksort*) e de processamento de grafos (*Dijkstra* e *breadth-first search*) nas seguintes variações:
  - Versão padrão de CUDA.
  - Versão CUDA com o uso de DP.
  - Versão para CPU usando OpenMP.

- Coletar os dados dos *benchmarks* utilizando *nsight*, *perf* e as ferramentas nativas do Linux como coletores de métricas.
- Avaliar as vantagens e custos relacionados ao uso de paralelismo dinâmico, em relação às métricas estabelecidas.

1.2 Estrutura do Texto

Este trabalho de conclusão de curso está estruturado da seguinte forma: o Capítulo 2 contem o referencial teórico utilizado para formulação deste trabalho, assim como uma breve introdução a API OpenMP e CUDA. Em sequência, o Capítulo 3 contem trabalhos anteriores relacionados à análise, implementação e/ou comparação da extensão CUDA paralelismo dinâmico. Subsequentemente, encontra-se o capítulo 4, que contem a descrição dos *benchmarks* implementados para realizar os experimentos. Além disso, este também descreve o ambiente de execução utilizado e a metodologia utilizada para a realização dos experimentos. E o Capítulo 5, onde estão descritos os resultados encontrados através da realização dos experimentos descritos no capítulo anterior e um relato das dificuldades encontradas durante a formulação deste trabalho. E por fim, o Capítulo 6, onde se encontram as conclusões finais obtidas a partir dos resultados encontrados.

2 REFERENCIAL TEÓRICO

Neste capítulo, estão presentes os principais conceitos relacionados à computação paralela em arquiteturas homogêneas e heterogêneas. Além disso, é dada uma breve introdução ao modelo de programação CUDA, a extensão paralelismo dinâmico e a API OpenMP.

2.1 Computação Paralela e Arquiteturas Paralelas

O paralelismo pode ser explorado em diversos níveis de aplicação e de várias formas pelo hardware. Nesse sentido, conforme (HENNESSY, 2012), o paralelismo em aplicações pode ser explorado no nível de dados (*data-level parallelism* – DLP) e/ou no nível de tarefas (*task-level parallelism* – TLP). Nesse sentido, quando DLP é explorado, vários itens de dados podem ser operados simultaneamente. Exemplos de aplicações onde DLP pode ser explorado incluem: multiplicação de matrizes, produto matricial, somas vetoriais, etc. Em contraste, quando o paralelismo é explorado no nível de tarefas, tarefas podem ser operadas independentemente e em paralelo. Exemplos de aplicações onde TLP é aplicado incluem: *pipelining*, banco de dados, sistemas operacionais, etc.

Independente do tipo de paralelismo fornecido pela aplicação, é possível explorar o paralelismo em quatro maneiras distintas através do hardware:

- *Instruction-Level Parallelism* (ILP): neste nível o paralelismo é explorado sobre o nível de dados de instruções. Para a realização de tal, são utilizadas as técnicas de *pipelining* e execução preemptiva. Adicionalmente, exemplos de arquiteturas que utilizam este nível de paralelismo incluem: processadores superescalares e arquiteturas VLIW (Very long instruction word).

- *Thread-Level Parallelism* (TLP): neste nível múltiplas *threads* exploram paralelismo sobre o nível de dados ou tarefas. Por esta razão, diversas aplicações podem explorar o paralelismo no nível de *threads*. Exemplos incluem: algoritmos de busca, aplicações *desktop*, consultas em banco de dados, etc. No entanto, para o uso deste nível de paralelismo há a necessidade de um modelo de hardware que permita a comunicação entre múltiplas *threads* executando concorrentemente. Nesse sentido, exemplos de arquiteturas que possuem tal modelo de *hardware* e podem explorar este nível de paralelismo incluem: uni-processadores e multiprocessadores.

- *Request-Level Parallelism* (RLP): neste nível o paralelismo é explorado sobre tarefas que são pela maior parte independente umas das outras. Estas tarefas representam um conjunto de uma ou mais requisições, criadas pelo sistema operacional ou pelo usuário. E requisições representam comandos que devem ser executados. Adicionalmente, exemplos de arquiteturas que fazem uso deste nível de paralelismo nas suas aplicações incluem servidores e *clusters* computacionais.

- *Vector Architectures and Graphic Processor Units* (GPUs): o paralelismo é explorado no nível de dados, através da execução de uma única instrução por vez. De forma que esta instrução é aplicada sobre diferentes iterações de um conjunto de dados paralelamente. No entanto, para tal é necessário que todas as iterações da instrução sobre o conjunto de dados sejam independentes. Nesse sentido, operações vetoriais e matriciais são exemplos claros de aplicações que podem ser exploradas, neste nível de paralelismo. Adicionalmente, arquiteturas heterogêneas são exemplos de implementações que utilizam este nível de paralelismo.

Conforme a taxonomia descrita por (FLYNN, 1966), os sistemas computacionais paralelos podem ser classificados em 4 categorias com base na quantidade de fluxos de instruções e dados que estes podem executar concorrentemente. A Figura 1 ilustra a quantidade de fluxos de cada classificação da taxonomia de Flynn, conforme descrito abaixo.

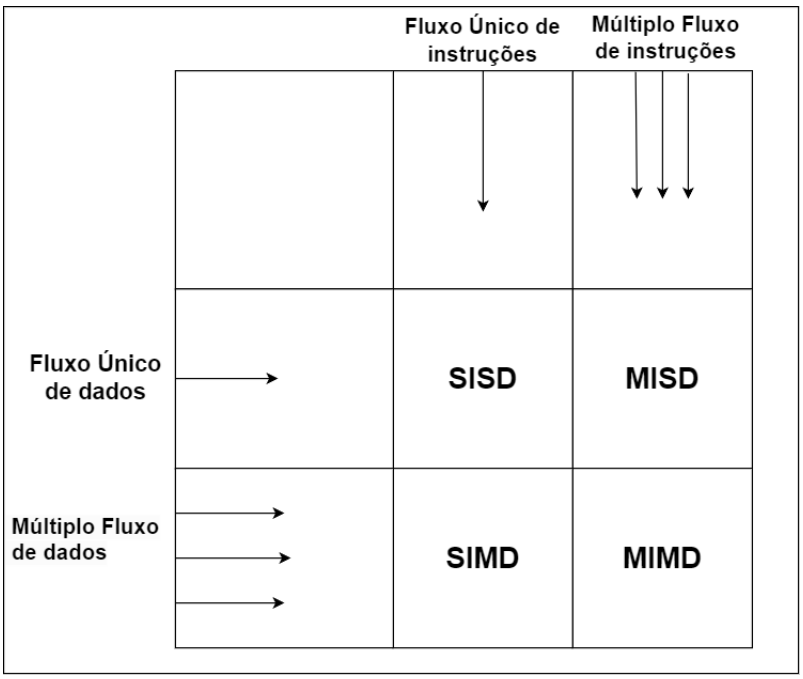


Figura 1 – Classificação de Flynn

Fonte: do autor

- *Single Instruction, Single Data* (SISD): sistemas computacionais nesta categoria executam um único fluxo de instruções sobre um único fluxo de dados. Desta forma, explorando o paralelismo no nível de instruções. Adicionalmente, o uni-processador é um dos exemplos comerciais desta categoria.

- *Single instruction, Multiple data* (SIMD): sistemas computacionais nesta categoria utilizam diversos processadores para a execução de um único fluxo de instruções sobre vários fluxos de dados. Ou seja, cada processador itera a mesma instrução

sobre um subconjuntos de dados que lhe foi atribuído. Dessa forma, explorando o paralelismo no nível de dados. Adicionalmente, o controlador de processos é responsável pelo envio e busca de instruções que devem ser executadas por estes processadores. Após o envio e busca, estas instruções são armazenadas na memória de instruções de cada processador. Ademais, exemplos comerciais desta categoria incluem: arquiteturas vetoriais, GPUs e extensões de multimídia para conjuntos padrões de instruções.

- *Multiple instruction, Single Data* (MISD): sistemas computacionais nesta categoria executam múltiplos fluxos de instruções sobre um único fluxo de dados. De acordo com (HENNESSY, 2014), no momento não existe um modelo comercial desta categoria, e o mais proximo desta é o processador de *streams*.

- *Multiple instruction, Multiple data* (MIMD): sistemas computacionais nesta categoria executam múltiplos fluxos de instruções sobre múltiplos fluxos de dados, através do uso de múltiplos processadores. Isto é, cada processador busca seu próprio conjunto de instruções e atua sobre seu próprio conjunto de dados. Nesse sentido, o paralelismo é explorado no nível de tarefas ao realizar a divisão do programa em tarefas para cada processador. Além disso, sistemas desta categoria também possuem a capacidade de explorar o paralelismo no nível de dados. Similarmente aos sistemas na categoria SIMD. Contudo, há a necessidade de uma granularidade computacional maior para a eficiência tal. Devido aos custos da sincronização implícita em MIMD. Adicionalmente, além dos níveis de paralelismo mencionados anteriormente, sistemas nesta categoria também possuem a capacidade de explorar o paralelismo no nível de *threads*. Exemplos comerciais de arquiteturas desta categoria incluem *clusters* e computadores de escala *warehouse*.

Além das categorias descritas na classificação de Flynn, categorias adicionais foram criadas após a sua publicação em 1966. Um exemplo destas é o *single instruction, multiple thread* (SIMT), que engloba as GPUs. Em um SIMT, múltiplas *threads* executam um fluxo de instruções sobre vários fluxos de dados. De acordo com (NVIDIA, 2022), a categoria SIMT é semelhante a SIMD na forma que uma instrução controla múltiplos elementos sendo processados em suas organizações vetoriais. A principal diferença entre estas é que organizações vetoriais em SIMD expõem a largura de SIMD para o software. Em comparação as instruções de SIMT, especificam a execução e comportamento de *branching* de *thread* singular.

2.2 Modelos de Programação Paralela

De acordo com (OSHANA; KRAELING, 2019), um modelo de programação é definido como a junção de linguagens e bibliotecas que criam uma visão abstrata de uma

máquina. Desta forma, modelos de programação agem como um intermediário entre as instruções do código de um programador e as instruções de código de máquina. Assim, facilitando o processo de programação ao usuário. Nesta Seção encontra-se uma explicação dos conceitos de modelos de programação e uma introdução a CUDA e OpenMP.

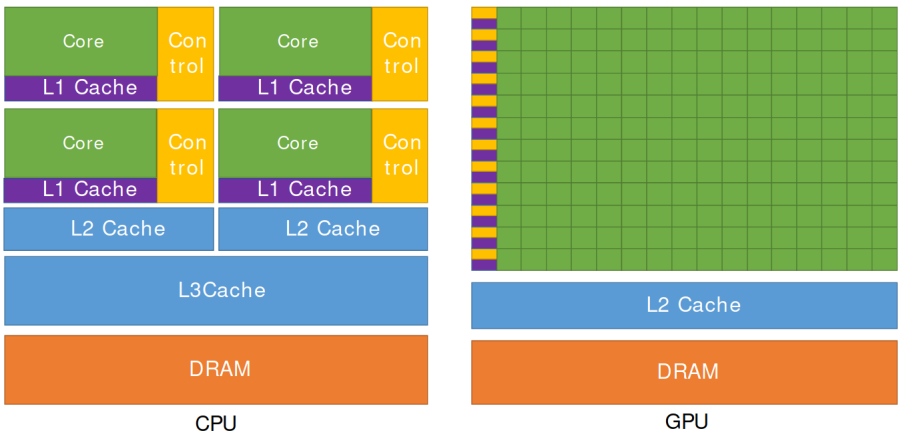
2.2.1 CUDA

De acordo com (NVIDIA, 2022), CUDA é uma plataforma paralela de propósito geral e modelo de programação, que utiliza o motor paralelo em GPUs NVIDIA para a resolução de problemas computacionais complexos. Isto é, *cores* (ou CUDAS *cores*) como são referidos comumente nas GPUs NVIDIA, são utilizados para a resolução de problemas computacionais. Ademais, à execução de um programa no modelo CUDA geralmente segue quatro etapas na sua execução: 1. Alocação; 2. Transferência; 3. Execução; 4. Sincronização.

2.2.2 Arquitetura Genérica de GPU

A Figura 2 representa uma arquitetura heterogênea genérica composta por um processador *multicore* e uma GPU. O processador *multicore* contém quatro núcleos de processamento, onde cada um destes cores possui uma memória *cache* L1 privada. Em adição, todos os núcleos de processamento compartilham o mesmo terceiro nível de memória *cache*. O propósito deste nível de memória é permitir a comunicação entre *threads* que executam em distintos núcleos. Em contraste, a GPU genérica possui particularidades que a diferenciam da organização do processador *multicore*. Por exemplo, a GPU é composta por *streaming multiprocessors* (SMs) que contém os CUDA *cores*. Onde cada SM possui sua própria memória *cache* L1 privada. Além disso, a memória *cache* L2 é utilizada como uma memória global para os CUDA cores.

Figura 2 – Arquitetura genérica de CUDA

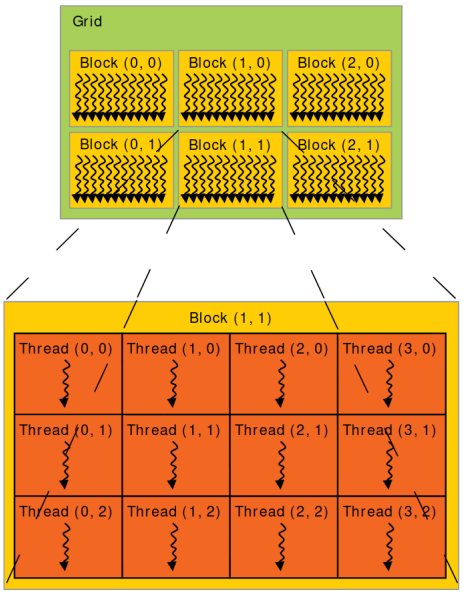


Fonte: NVIDIA (2022, p.2)

Na GPU a execução e manejamento de *threads* diferem da CPU. Este processo é realizada através do uso de CUDA *cores*, SMs e *warp schedulers*. De tal forma que CUDA *cores* realizam a execução de *threads*. E SMs são responsáveis pela execução de um ou mais conjuntos lógicos de *threads*, denominados blocos. Ademais, uma sequência consecutivas de 32 *threads* de um mesmo bloco é denominada *warp*. Assim, *Warp schedulers* são responsáveis pelo escalonamento e troca de contexto entre conjuntos de *warps* em execução.

Conforme demonstrado na Figura 3, as GPUs NVIDIA utilizam um agrupamento logico de *threads* para a execução de *kernels*. Composta pelos níveis de *threads*, blocos e *grids*. Onde *threads* executam concorrentemente um *kernel* na GPU. E blocos representam conjunto independentes de até 1024 *threads* organizados em uma, duas ou três dimensões. Similarmente à *threads*, blocos são agrupados e contidos em organizações de até 1024 elementos de uma, duas ou três dimensões, denominadas *grids*. Ademais, em relação a execução desta organização lógica, o número de blocos X *threads*, representa o número de execuções concorrentes de um *kernel* em um *grid*. Além disso, a execução de blocos em uma *grid*, podem ser executados fora de ordem, paralelamente ou serialmente.

Figura 3 – Hierarquia lógica de CUDA

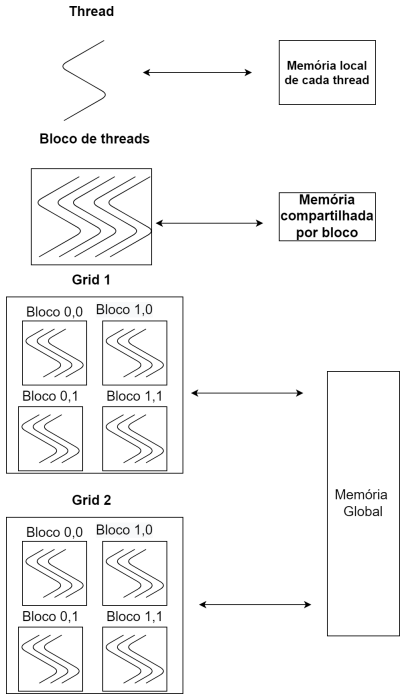


Fonte: NVIDIA (2022, p.11)

A Figura 4 demonstra a organização da hierarquia de memória em GPUs NVIDIA. Onde os níveis mais altos da hierarquia possuem o menor tempo de acesso. E os níveis mais baixos possuem maior tempo de acesso comparativamente. No nível mais alto (próximo dos *cores*), estão as memórias locais privadas de cada *thread*. Estas memórias são utilizadas por cada *thread* para o armazenamento dos dados que esta ira executar. No nível abaixo deste, está a memória compartilhada entre as *threads* de um bloco. Este

nível de memória permite uma comunicação rápida entre *threads* de um mesmo bloco. No nível mais baixo desta hierarquia está a memória global, compartilhada por todas as *threads* e utilizada para as funções de alocação de memória. Além das memórias citadas, conforme (NVIDIA, 2022), existem outros dois espaços adicionais de memória utilizados apenas para leitura: constantes e de textura. O propósito da memória constante é facilitar o acesso de *threads* às informações escritas pela CPU. Em contraste, a memória de textura é utilizada para acessos de memória que exibem localidade espacial.

Figura 4 – Hierarquia de memória CUDA



Fonte: autor com base na Figura 5 de NVIDIA (2022)

Como mencionado anteriormente a execução de um programa CUDA geralmente segue quatro etapas: 1. Alocação, 2. Transferência, 3. Execução, 4. Sincronização. Em maiores detalhes estas são efetuadas da seguinte maneira:

- **Alocação**: nesta etapa é feita a alocação de memória global da GPU para variáveis que serão utilizadas na execução de *kernels*. Através da chamada de uma ou mais das seguintes funções na CPU: ***cudaMalloc***, ***cudaMallocPitch***, ***cudaMalloc3D***, ***cudaMallocArray*** e ***cudaMalloc3DArray***. Ademais, estas funções recebem como argumentos um ponteiro de memória e a quantidade de memória em bytes que deve ser alocado ao ponteiro na GPU. Alternativamente, também é possível alocar uma variável na GPU com a cláusula ***__device__*** ***__managed__***.
- **Transferência**: nesta etapa é realizada a transferência de dados de variáveis locais da CPU para variáveis previamente alocadas na GPU. Esta transferência é realizada

através das funções: ***cudaMencpy***, ***cudaMencpy2D***, ***cudaMemcpy2DArrayToArray***, ***cudaMemcpy3D***, etc. Alternativamente, também é possível utilizar a função ***cudaMallocManaged*** para a alocação. Nesse caso variáveis são automaticamente manejadas pelo sistema de memória unificada CPU-GPU. Assim, removendo a necessidade de funções de transferência de memória.

- **Execução**: esta etapa representa a execução da aplicação na GPU. A execução é realizada através da chamada de funções na CPU denominadas *kernels*. Estes *kernels* são declarados com a cláusula ***__global__*** e devem possuir retorno nulo. Adicionalmente, devem ser invocados com os campos $<<<>>>$, estes campos especificam argumentos de execução da GPU. Os dois primeiros campos são obrigatórios e representam o número de blocos e *threads* respectivamente. Em sequência, o terceiro e quarto argumentos são opcionais, e estes representam a quantidade de memória compartilhada em bytes alocada dinamicamente para cada bloco e a *stream* associado ao *kernel*, respectivamente. Ademais, *streams* representam fluxos de execução de *kernels* na GPU. De forma que cada fluxo é associado a uma *grid*. No entanto, em casos onde o quarto argumento não é especificado, o *kernel* será executado na *NULL stream*. A *NULL stream* é única para cada GPU e serve como a *stream* padrão. Adicionalmente, é possível executar diversos fluxos de execução diferentes ao criar diversos *streams*. Para isso é utilizada a função ***cudaStreamCreate***. Além disso, para a destruição de *streams* é utilizada a função ***cudaStreamDestroy***.

- **Sincronização**: esta etapa representa o retorno dos dados processados na GPU para a CPU. Para isto, é necessária uma barreira de sincronização na forma da função *cudaDeviceSynchronize* para garantir que a CPU aguarde a conclusão da execução na GPU. Além de uma chamada de transferência de dados entre GPU e CPU. Adicionalmente, em casos onde é necessária uma barreira de sincronização entre *threads* internas a um bloco, a função ***__syncthreads*** pode ser utilizada.

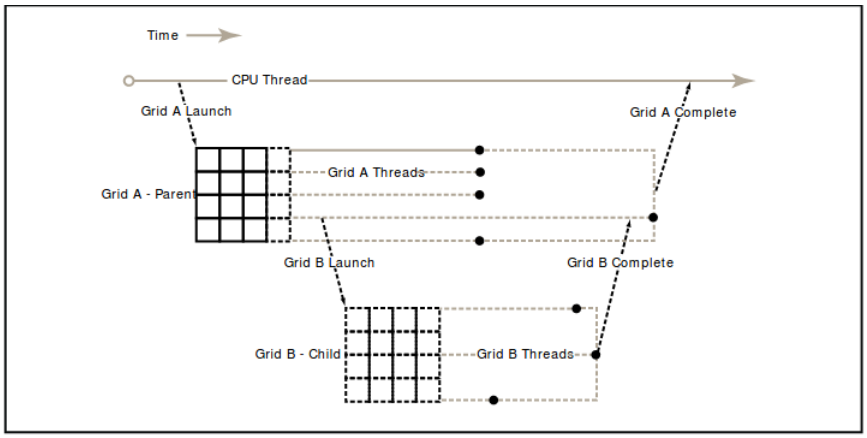
2.2.3 Paralelismo Dinâmico

De acordo com (NVIDIA, 2022), paralelismo dinâmico é uma extensão do modelo de programação CUDA. Que permite a criação e sincronização de novos *kernels* CUDA diretamente na GPU. Assim, eliminando à necessidade do retorno para a CPU para a realização de chamadas de *kernels* para problemas recursivos. No entanto, há uma limitação máxima de 24 no nível de profundidade de chamadas recursivas de *kernels*. Além disso, esta extensão está disponível apenas para dispositivos NVIDIA com capacidade computacional 3,5 ou maior em relação a classificação da NVIDIA. Em suma, a capacidade computacional de um dispositivo é a representação dos recursos suportados pela GPU.

As chamadas de criação de *kernel* dentro da GPU com paralelismo dinâmico são assíncronas e podem ser feitas no nível de *threads*, blocos ou *grids*. Ademais, quando é realizada uma chamada de *kernel* na GPU com a extensão do paralelismo dinâmico, é criada uma *grid* "filha". E nesse sentido, a *grid, thread* ou bloco que realizou a sua invocação é considerada o "pai". A execução de um pai não é considerada completa até que a execução de todos seus filhos sejam finalizadas. Consequentemente, a execução de um bloco também não é considerada concluída até que todas as chamadas de *kernels*, realizadas pelas suas *threads* estejam completas. Além disto, *grids* filhas criadas por elementos de um bloco são visíveis para todos os elementos do mesmo bloco. Adicionalmente, pais e filhos compartilham a memória global, memória constante e a memória compartilhada. No entanto, cada filho possui uma memória local e compartilhada.

A Figura 5 exemplifica as etapas da execução de uma *grid* filha. Nas etapas demonstradas é possível garantir a coerência de memória entre filho e pai apenas na criação e término de uma *grid* filha.

Figura 5 – Exemplo de criação de *grid* filha



Fonte: NVIDIA (2022, p.255)

2.3 OpenMP

De acordo com (OPENMP ARCHITECTURE REVIEW BOARD, 20221), a API OpenMP utiliza um modelo paralelo de execução *fork-join*. Em um modelo *fork-join*, o termo *fork* refere-se a divisão de um problema em sub-problemas menores paralelizáveis. Por outro lado, o termo *join* refere-se à junção das soluções obtidas dos subproblemas para obter a solução para o problema original.

Um programa OpenMP começa sua execução sequencialmente com apenas uma *thread*, denominada *thread* mestre. No entanto, ao encontrar uma região paralela (*pragma omp parallel* ou *pragma omp for*) a *thread* mestre realiza um *fork*. Este *fork* consiste da criação de um time composto por zero ou mais *threads*. Onde cada integrante deste time possui um identificador que possibilita a sua referência. Dessa forma, a execução de

uma área paralela é realiza através da atribuição de tarefas, criadas a partir do código da região paralela para integrantes do time. Após a conclusão da execução da região paralela é realizado um *join* entre as *threads* do time e a *thread* mestre. Através desse processo OpenMP pode explorar de paralelismo no nível de dados e tarefas, em ambas arquiteturas MIMD e SIMD.

A organização de memória entre *threads* é feita utilizando um modelo compartilhado, onde cada *thread* possui a permissão de escrita e leitura sobre a memória compartilhada. Além da memória compartilhada cada *thread* possui uma memória privada na forma de registradores, cache ou armazenamento local. Adicionalmente, a memória privada não possui garantia de coerência com a memória compartilhada. A Figura 6 ilustra esta organização.

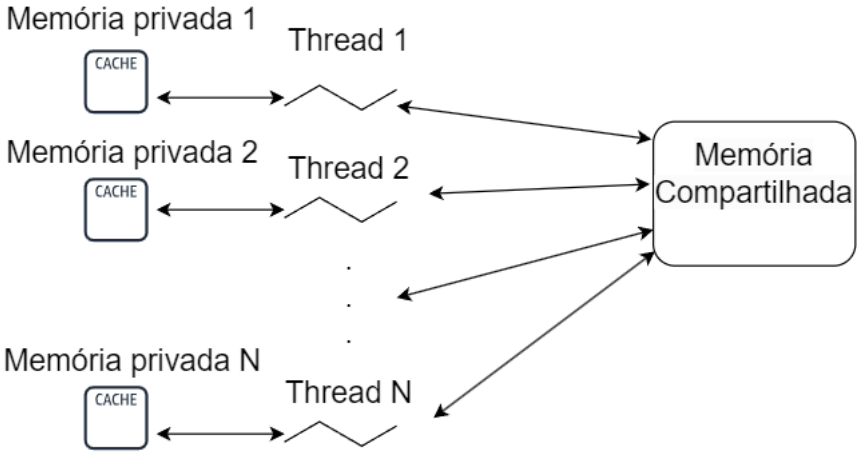


Figura 6 – Modelo de memória OpenMP

Fonte: do autor

Adicionalmente, variáveis na API OpenMP também são dividas em privadas (*private*) ou compartilhadas (*shared*). Onde cada *thread* possui uma cópia local de cada variável privada. No entanto, no caso das variáveis compartilhadas estas são referenciadas de tal forma que todas as *threads* acessam a mesma posição de memória. De tal forma que cada variável compartilhada possui uma posição para ser referenciada na memória compartilhada.

3 TRABALHOS RELACIONADOS

Neste capítulo são apresentados os trabalhos que realizam análises, implementações e/ou comparações de algoritmos implementados com a extensão paralelismo dinâmico. Nesse sentido, os trabalhos descritos a seguir, exploraram ou realizam avaliações de custos e/ou benefícios do uso do paralelismo dinâmico em CUDA são discutidos a seguir.

Em Wang e Yalamanchili (WANG; YALAMANCHILI, 2014), é realizada uma avaliação da execução de aplicações com dados não estruturados implementadas utilizando CUDA DP. A motivação deste trabalho parte do pressuposto que aplicações com dados não-estruturados, formam *pockets* (bolsos) dinâmicos do paralelismo estruturado, que podem ser explorados pelas chamadas recursivas do paralelismo dinâmico. Os resultados demonstram que o uso da extensão paralelismo dinâmico atingiu uma aceleração de até 2.73x, quando comparada com implementações que não usam a extensão. Adicionalmente, é especificado que o *overhead* causado por criações de *kernels*, pode limitar e reduzir os ganhos de desempenho com uso da extensão. Conforme os autores, este *overhead* está associado ao tempo de envio de parâmetros e chamadas internas de funções CUDA. Como por exemplo *cudaGetParameterBuffer* e *cudaLaunchDevice*, além da criação e despacho dos *kernels* filhos.

Em Zhang et al.(ZHANG et al., 2015), é realizada uma implementação dos algoritmos para grafos BFS (*breadth-first search*) e SSSP (*single-source shortest paths*) com paralelismo dinâmico. Visando demonstrar a aplicabilidade do paralelismo dinâmico para grafos. Os resultados encontrados demonstram que a implementação do algoritmo BFS teve um desempenho superior em grafos de escala menores ou iguais a 19. Quando comparada com as soluções encontradas em *Graph 500*. Porém, houve uma piora nos resultados da implementação em relação a grafos de redes de estrada. Quando comparado à quase todas as implementações *Lonestar* e do *Graph 500*. Os autores justificam esta piora, como a falta de graus de adjacência suficientemente grandes, para a exploração eficiente do paralelismo dinâmico. Por outro lado, os resultados de desempenho do algoritmo SSSP em grafos de redes de estrada foram superados por apenas algumas das soluções *Lonestar*.

Em Jarzabek e Czarnul (JARZĄBEK; CZARNUL, 2017), é realizada uma análise do desempenho do paralelismo dinâmico e *unified memory*, através do uso de uma comparação de desempenho com os seguintes algoritmos: conjectura de Goldbach, transferência de calor em um plano 2D e integração numérica adaptativa. Os resultados demonstram que o uso do paralelismo dinâmico melhorou o desempenho do algoritmo de transferência de calor em casos de teste com poucas fontes de calor. No entanto, em casos com um número significativamente maior de fontes de calor ocorreu uma piora. A razão para esta piora, conforme os autores é o *overhead* causado pela quantidade de chamadas de *kernel*. De forma que cada fonte de calor representa uma chamada de *kernel*. Por outro lado, os resultados da integração numérica e conjectura de Goldbach demonstram uma piora para à maioria dos casos. Novamente foi citando um *overhead* gerado pelas chamadas de

kernels e divergência de execução entre *threads*. Conclui-se que paralelismo dinâmico traz grandes benefícios para algoritmos recursivos que usam dados hierárquicos e pioras para casos de divergência entre *threads* e demasiadas chamadas de *kernel*.

Em ALIAGA et al. (ALIAGA et al., 2016), é realizada uma implementação do conjugado de gradiente com paralelismo dinâmico, visando reduzir o consumo de energia e tempo de execução. Os resultados demonstram que a implementação com paralelismo dinâmico obteve melhoras em relação às métricas de tempos de execução e consumo de energia em relação à implementação anterior dos autores deste problema.

Em Dimarco e Taufer (DIMARCO; TAUFER, 2013), é realizada uma análise dos ganhos de desempenho do paralelismo dinâmico em *clustering*, através da comparação de implementações dos algoritmos *k-means* e divisão de *cluster* hierárquicos. Os resultados demonstram que implementação CUDA com paralelismo dinâmico obteve um melhor desempenho em relação à divisão de *clusters* e pior com o algoritmo *k-means*. Em comparação com as implementações CUDA que não usam o paralelismo dinâmico. A razão atribuída para a piora do desempenho no algoritmo *k-means* é a necessidade de sincronização de trabalho.

Em Alandoli et al. (ALANDOLI et al., 2016) é realizada a implementação de diversas versões de um algoritmo de detecção de comunidades em redes sociais com GPUs, com o propósito da melhoria de desempenho do algoritmo através da paralelização. Adicionalmente, também foi realizada uma comparação de desempenho entre as implementações GPUs para determinar qual obteve à melhor desempenho. As Implementações utilizadas são: sequencial, hibrida GPU-CPU e CUDA com paralelismo dinâmico. Os resultados deste trabalho demonstram que às implementações hibrida e de paralelismo dinâmico obtiveram um *speedup* de 1.92x e 1.86x respectivamente. Em relação ao desempenho da implementação sequencial, sobre um *dataset* composto por 115 nodos e 613 vértices não direcionados. Adicionalmente, quando comparadas com a implementação sequencial utilizando sub-redes de um *dataset* maior, a implementação hibrida obteve um *speedup* de: 7.1x, 7.86x, 7.9x e 8.29x para as sub-redes de tamanhos de 1000, 2000, 3000 e 4039 respectivamente. Em comparação, os algoritmos de paralelismo dinâmico obtiveram um *speedup* de 2.82X, 3.64X, 3.90X e 4.45X. Para as sub-redes de tamanho 1000, 2000, 3000 e 4039 respectivamente. Nesse sentido, as razões, atribuídas para a discrepância entres os valores de *speedup* das implementações GPU. Foram as seguintes: dependência entre funções e a carga de trabalho demasiada na *thread* inicial do *kernel* pai. Pois esta dependência causa longos tempos de processamento para o paralelismo dinâmico. Em comparação, a implementação hibrida não sofre esta penalidade devido à estas funções serem processadas na CPU.

Em Plauth et al. (PLAUTH et al., 2015), é realizado um estudo de caso sobre o uso do paralelismo dinâmico CUDA em algoritmos de árvore. Através da análise e comparação entre implementações do algoritmo *N-QUEENS*. A motivação deste estudo parte

do fato que CUDA padrão não é adequada para cargas de trabalhos irregulares como árvores. Para a formulação, dos resultados, foram comparadas as seguintes implementações: CUDA paralelismo dinâmico; implementação CPU; implementação base GPU. Ademais, os resultados demonstram que as implementações de paralelismo dinâmico tiveram resultados piores em quase todos os casos em comparação com as outras implementações. As razões atribuídas para está piora foram o *overhead* criado pelas chamadas aninhadas recursivas de *kernel* e alocação dinâmica de memória.

Em Dong e Yuan (DONG; WANG; YUAN, 2013), é realizado um estudo para verificar o quão eficientemente é possível paralelizar a técnica de clusterização B.I.R.C.H (*Balanced Iterative Reducing and Clustering using Hierarchies*) com a extensão CUDA paralelismo dinâmico. A motivação deste trabalho parte do potencial de paralelização de *datasets* de larga escala. Além da possível redução significativa de tempo necessário para à conclusão da execução destes algoritmos. Os resultados da execução dos 6 *datasets* selecionados para a avaliação, indicam um *speedup* máximo de 154,29 no *dataset* KDD CUP 2012 (149, 639, 105 amostras, 12 atributos). Adicionalmente, *datasets* maiores tenderam a obter resultados melhores. A razão atribuída para tal é a redução do tempo ocupado na construção das árvores e o aumento no tempo do processo de absorção.

Com base nestes trabalhos é possível notar que o perfil de aplicações que se encaixam na extensão paralelismo dinâmico, são as compostas de dados semi-estruturados ou organizados hierarquicamente. Exemplos incluem aplicações que operam sobre grafos, *meshes* e árvores. Adicionalmente, vários autores relataram custos associados ao uso do paralelismo dinâmico, como as chamadas recursivas de *kernel*, a criação e destruição de *streams* e a divergência de execução entre *threads*.

Considerando os custos descritos acima, diferentes trabalhos focaram no desenvolvimento de estratégias para eliminar ou minimizar tais custos. Em Tang et al. (TANG et al., 2017), é proposta uma solução na forma de SPAWN um *framework* que busca para reduzir os custos por meio de técnicas de *threshold* e monitoramento da GPU. Os resultados encontrados demonstram que SPAWM obteve uma aceleração de 69% e 57% quando comparado com implementações sem uso do paralelismo dinâmico e com o uso paralelismo dinâmico respectivamente.

Em Olabi et al. (OLABI et al., 2022), é proposto um *framework* para compiladores que através do uso de *thresholding* e agregação de *kernels* visa reduzir estes custos. Para demonstrar seu desempenho foram utilizados vários *benchmarks* de grafos. Os resultados demonstram uma melhora de desempenho de 43.0x em relação a aplicações que usam paralelismo dinâmico. 8.7x em relação a aplicações que não usam a extensão. E 3.6x em relação a aplicações que utilizam a extensão com agregação como proposta em trabalhos anteriores.

Em Pessoa et al. (PESSOA et al., 2018), é realizada uma implementação de um algoritmo de *backtracking*, com a estratégia de divisão e conquista. Este algoritmo realiza

o cálculo da exigência de memória, para chamadas de *kernel*, antes da sua realização. Dessa forma, é realizada a alocação de apenas a quantidade de recursos necessários. Os resultados encontrados demonstram que quando utilizado um número otimizado de blocos e *threads* sobre problemas de certo tamanho a implementação pode ter um fator 25x de melhora de desempenho quando comparados a implementações anteriores.

Em Hajj et al. (HAJJ et al., 2016), foram propostas diversas técnicas de complicação que buscam melhorar o desempenho de aplicações que utilizam o paralelismo dinâmico. Estas técnicas incluem: a agregação de *kernels* e a promoção de chamadas de *kernels*. Adicionalmente, o processo de promoção funciona de tal maneira que chamadas que foram promovidas sejam executadas com prioridade pelo compilador. Os resultados encontrados demonstram uma aceleração média de 6.5x sobre implementações padrão do paralelismo dinâmico. Em adição, estas técnicas permitem a criação de *kernel* que não seriam possíveis em outros casos sem o seu uso, causando uma melhora no rendimento da média geométrica de 30x.

Em Yang e Zoug (YANG; ZHOU, 2014), é proposta CUDA-NP um *framework* que propõem otimizar *kernels* e eliminar *overheads* de *loops* paralelos. Através do uso de estruturas de controle para a realização de mudanças no número de *threads*. Em conjunto com o uso da divisão eficiente de *loops* entre *threads* e otimização de código. Os resultados encontrados demonstram um *speedup* de 1.36x até 6.69x quando comparado com versões modificadas dos *benchmarks*: *MarchingCubes, Libor, Lud, Leukocyte, Streamcluster, Computational Fluid Dynamics, BucketSort e Nearest Neighbor*.

Em Ozen & Ayguade & Labarta (OZEN; AYGUADE; LABARTA, 2015), é realizada a proposta de uma implementação no compilador MACC. Que propõem combinar as capacidades paralelas do paralelismo dinâmico com a API OpenMP. Visando estender a desempenho paralela do modelo OpenMP. Para isto, ocorrências de cláusulas aninhadas de times e condicionais são tratadas como a criação de uma liga de *threads*. E o compilador por sua vez as transformaria em uma chamada de *kernel*. Além disso, também são utilizadas barreiras implícitas no final do construtor de times e um mecanismo de alocação prévia de memória do compilador. Dessa forma facilitando a comunicação entre *kernels* pais e filhos. Para a avaliação dos resultados foram utilizados *kernels* de multiplicação esparsa de matrizes, *breadth-first search* e a divisão e conquista de Mandel-Brot. Os resultados encontrados demonstram um *speedup* de até 40x sobre versões que não utilizam o paralelismo dinâmico.

A tabela abaixo demonstra as contribuições de cada trabalho citado e deste. Com base nos algoritmos, implementações e métricas utilizadas.

Tabela 1 – Contribuições de trabalhos Citados

Fonte: do autor

Trabalho	Métricas analisadas	Algoritmos	Comparação	Contribuições
Do autor	Tempo Consumo de energia	SSSP BFS Mergesort Quicksort	CUDA OpenMP CUDA DP	Análise e Comparação de DP com cuda OpenMP
(WANG; YALAMANCHILI, 2014)	Tempo Desempenho Comportamento de cache Uso de memória	Adaptive Mesh Refinement Barnes Hut Tree BFS SSSP Product Recommendation Relational Join Graph Coloring Regular Expression Match	CUDA CUDA DP	Análise dos benefícios e custos de DP
(ZHANG et al., 2015)	Tempo	BFS e SSSP	Implementações LoneStar e Graph 500 CUDA DP	Demonstrar aplicabilidade de DP em grafos
(JARZĄBEK; CZARNUL, 2017)	Tempo Consumo de energia	GoldBatch Transferência de calor em plano 2D Integração numérica	CUDA CUDA DP	Análise da desempenho de DP e Unified Memory
(ALIAGA et al., 2016)	Tempo Consumo de energia	Conjugado de gradiente	CUDA CUDA DP	Demonstração da aplicabilidade de DP para conjugado do gradiente
(DIMARCO; TAUFER, 2013)	Tempo	K-means Divisão de clusters	CUDA CUDA DP	Análise dos ganhos de DP em clustering
(ALANDOLI et al., 2016)	Tempo Aceleração	Detecção de comunidades em redes sociais	GPU-CPU Sequencial CUDA DP	Demonstração da aplicabilidade de DP para detecção de comunidades em redes sociais
(PLAUTH et al., 2015)	Tempo	N-QUEENS	CUDA CUDA DP	Análise do uso de DP para algoritmos de processamento de árvores
(DONG; WANG; YUAN, 2013)	Tempo	Balanced Iterative Reducing Clustesring	CUDA CUDA DP	Estudo da eficiência do uso de DP em B.I.R.C.H
(TANG et al., 2017)	Tempo Utilização de memória compartilhada Utilização de unidades computacionais da GPU Distribuição de carga de trabalho Uso de memória	BFS SSSP Relational Join Multiplicação de matrizes Alinhamento Sequencial Adaptative Mesh refinement Mandelbros	CUDA DP	Framework SPAWN
(OLABI et al., 2022)	Tempo Aceleração	BFS Bezier Tesselation Minimum Spanning Tree SSSP Contagem de triângulos Survey propagation	CUDA DP CUDA	Proposta de um framework de thresholding e agregação para DP. Visando melhorar o desempenho.
(PESSOA et al., 2018)	Tempo Aceleração Influência das métricas de diferentes conFigurações de recursos da GPU na execução	Backtracking	OpenMP CUDA CUDA DP	Implementação de um algoritmo de backtracking, que realiza o cálculo de memória necessária para subsequentes chamadas de kernel
(HAJJ et al., 2016)	Instruções por segundo Tempo Aceleração	BFS Barnes Hut Tree Graph Coloring Minimun Spanning Tree SSSP Bezier Tesselation Survey propagation Rotulamento de componentes conectados	CUDA DP Sequencial	Proposta de diversas técnicas para a melhora do desempenho de DP
(YANG; ZHOU, 2014)	Tempo Acesso de memória Aceleração	MarchingCubes Streamcluster MatchingCubes Bucket Sort Computação de fluidos dinâmicos Nearest Neighbor	CUDA CUDA DP	Proposta de um framework para otimizar kernels e eliminar overheads de loops
(OZEN; AYGUADE; LABARTA, 2015)	Tempo Aceleração	Computational Fluid Dynamics, BucketSort Nearest Neighbor	CUDA CUDA DP	Proposta da implementação de um compilador que combina OpenMP e DP

4 METODOLOGIA

Neste capítulo encontra-se a descrição da metodologia utilizada para atingir os objetivos estabelecidos no sub-seção 1.1.

4.1 Benchmarks

Nesta subseção encontram-se as descrições das implementações de cada *benchmark*, assim como os seus pseudocódigos. Adicionalmente, a implementação sequencial de todos os algoritmos possui a mesma estrutura das suas implementações OpenMP, com a única diferença sendo o uso de cláusulas paralelas.

4.1.1 Djikstra/SSSP

Djikstra/SSSP, trata-se de um algoritmo de processamento de grafos, cujo objetivo é encontrar o caminho com custo mínimo, que percorra todos os vértices de um grafo. Este processo começa através da escolha de um vértice como o ponto inicial de busca. E subsequente estimativa da distância entre o ponto inicial e os outros vértices. Inicialmente, é atribuída uma distância infinita a estimativa de todos os vértices, exceto para o ponto inicial, cuja estimativa é sempre 0. Ademais, geralmente, durante a execução os vértices com arestas de possuem menor o custo total são priorizadas durante a travessia. Ou seja, é escolhido o vértice cuja aresta possua menor peso de travessia entre si e o vértice sendo processado, para ser iterada na próxima etapa, ao invés de escolher aquele com a menor estimativa global atual. Nesse sentido, conforme as arestas do grafo são percorridas durante a execução do algoritmo. Suas estimativas são ajustadas para refletir a menor distância encontrada para alcançar cada vértice. Dessa forma, o algoritmo é considerado concluído apenas quando todos os vértices forem visitados e não ocorram mais mudanças nas estimativas.

- Implementação Geral:

  Todas as implementações realizam a representação do grafo através do uso de diversos conjuntos de vetores, de forma que cada um destes vetores representa um dos aspectos da conexão entre vértices, estes sendo: custo de arestas; vértice de origem de arestas; destino de arestas. Portanto, a solução do problema é realizada através da atribuição de posições destes conjuntos vetoriais para *threads*, de forma que, cada *thread* realiza o cálculo dos valores do seu conjunto de posições. E caso seja necessário, realiza ajustes nos valores das estimativas atuais. Ademais, o processo descrito é iterado sobre um laço de repetição, com a condição de parada de que não ocorram mais mudanças nos valores das distâncias mínimas dos vértices.

- Implementação CUDA:

O pseudocódigo 1 ilustra a implementação CUDA, nesta é realizada a solução através do uso de um laço de repetição na CPU, que realiza chamadas do *kernel* de solução. De forma que o resultado da execução do *kernel* determina o termino do laço. Em suma, este *kernel* de solução realiza a atribuição do conjunto de posições vetoriais com base no ID de *threads* (linhas 2-8). E realiza a comparação entre os valores do caminho atual do conjunto vetorial com o valor da estimativa atual (linhas 9-12).

Algorithm 1: CUDA

Data: inicio(numEdges(número de arestas); dist(vetor que contem a distancia miníma de nodos); finished(variável utilizada para sinalizar o fim do *loop* de chamadas de *kernel* na CPU); preNode(vetor utilizado para determinar o predecessor de nodos)

```
1:  ThreadId=Id da thread atual ;
2:  StartId= ThreadId*numEdges ;
3:  EndId= ThreadId+1*numEdges;
4:  verificação para ver os valores estão foras das dimensões do vetor
5:  for i=StartId; i<EndId;i=i+1 do
6:      Origem = Origem da aresta[i];
7:      Destino = Destino da aresta[i];
8:      Custo = Custo da aresta[i];
9:      if dist[Origem] + Custo < dist[Destino] then
10:         dist[Destino] = dist[Origem] + Custo ;
11:         preNode[end] = Origem;
12:         finished = false;
13:     end if
14: end for
```

[1]

- Implementação OpenMP:

  O pseudocódigo 2 representa a implementação OpenMP, onde a solução do problema é realizada por meio de uma função, que consiste da criação de uma área paralela em um laço de repetição (linhas 2-5). Onde é realizada a atribuição (linhas 8-9) e cálculo (linhas 14–20) do conjunto de posições vetoriais para *threads*. Ademais, caso alguma *thread* realize mudanças na estimativa de um vértice, de forma que cada *thread* tem um conjunto atribuído para si. Ademais, a variável *finished* responsável pelo término do laço tera o valor falso atribuída a si (linha 20). Assim garantindo mais uma iteração do laço de repetição.

[1] Implementação baseada na implementação GPU do repositório <https://github.com/lixi-zhou/SSSP_On_CUDA>

Algorithm 2: OpenMP

```
1:  while !finished do
2:    finished = true;
3:    numIteration++;
4:    #pragma omp parallel
5:    #pragma omp parallel threadId = Id da thread;
6:    int numThreads = Total de threads;
7:    int numEdgesPerThread = numero de arestas / numThreads + 1;
8:    inicio = threadId * numEdgesPerThread;
9:    fim= (threadId + 1) * numEdgesPerThread;
10:   if  inicio = numEdges; then
11:     inicio > numEdges
12:   else if fim = numEdges; then
13:     fim > numEdges
14:     for  i=inicio; i<fim; i++ do
15:       Origem = Origem da aresta[i];
16:       Destino = Destino da aresta[i];
17:       Custo = Custo da aresta[i];
18:       if Custo = Custo da aresta[i]; then
19:         dist[Destino] = dist[Origem] + Custo;
20:         finished = false;
21:       end if
22:     end for
23:   end if
24: end while
```

[2]

- Implementação CUDA DP:

  O pseudocódigo 3 representa a implementação CUDA DP, esta similarmente à implementação CUDA padrão realiza à atribuição de posições através do ID de *threads* (linhas 2 e 3). E faz o uso de um laço de repetição para iterar sobre o seu *kernel* de solução. No entanto, estas diferem na forma que tratam mudanças nos valores das estimativas de vértices. Na implementação CUDA padrão, caso isso ocorra só haverá mais uma iteração do laço. Porém, quando isto ocorre na implementação CUDA DP, é realizada a chamada de um *kernel* secundário através da *thread* 0 (linha 15 – 16). Este *kernel* secundário é praticamente idêntico ao da solução. No entanto, este não realiza subsequentes chamadas recursivas. Nesse sentido, o objetivo do uso deste *kernel* secundário, é diminuir o número de comunicações entre GPU e CPU.

[2] Implementação baseada na implementação CPU do repositório <https://github.com/lixi-zhou/SSSP_On_CUDA>

Algorithm 3: CUDA DP

Data: inicio(numEdges(número de arestas); dist(vetor que contem a distancia miníma de nodos); finished(variável utilizada para sinalizar o fim do *loop* de chamadas de *kernel* na CPU); preNode(vetor utilizado para determinar o predecessor de nodos)

```
1: ThreadId=Id da thread atual ;
2: StartId= ThreadId*numEdges ;
3: EndId= ThreadId+1*numEdges;
4: verificação para ver os valores estão foras das dimensões do vetor;
5: for i=StartId; i<EndId;i=i+1 do
6:    Origem = Origem da aresta[i];
7:    Destino = Destino da aresta[i];
8:    Custo = Custo da aresta[i];
9:    if dist[Origem] + Custo < dist[Destino] then
10:      dist[Destino] = dist[Origem] + Custo ;
11:      preNode[end] = Origem;
         finished = false;
12:   end if
13: end for
14: sincronização entre Threads ;
15: if finished ==false e ThreadId==0 then
16:   Realizar a chamada do kernel secundário que possui a mesma função, mas não é
      recursivo ;
17: end if
```

[3]

4.1.2 Quicksort

Quicksort é um algoritmo de ordenação que utiliza uma estratégia de divisão e conquista para chegar a solução. Neste sentido, o processo de divisão e conquista ocorre através da escolha de um elemento do vetor como pivô. Para utilizar este pivô para dividir o conjunto atual ao meio, em dois subconjuntos. Um com valores maiores e outro com valores menores que o pivô. Este processo é repetido até restarem apenas subconjuntos de até 3 elementos. Nesse sentido, os subconjuntos são recombinados e dessa forma é realizada a etapa de conquista.

- Implementação Geral:

  Todas as implementações realizam o mesmo processo para a ordenação de seus subconjuntos. De forma que o vetor é percorrido por dois ponteiros, que apontam para o início e o fim do subconjunto. E durante a execução estes são usados para comparar elementos com o pivô. De forma que o endereço do ponteiro do início e incrementado e o do fim decrementado. Além disso, caso necessário estes são utilizados para

[3] Implementação baseada na implementação GPU do repositório <https://github.com/lixi-zhou/SSSP_On_CUDA>

modificar a posição do elemento entre si. No entanto, cada implementação possui uma abordagem diferente em relação à solução do problema.

- Implementação CUDA:

  O pseudocódigo 4 representa a implementação CUDA padrão, nesta é realizada a solução iterativamente com o uso de vetores auxiliares. De forma que é utilizado um laço de repetição onde os vetores auxiliares demarcam as posições que cada *thread* deve processar em cada iteração. Assim, os valores destes vetores são determinados através da execução dos *kernels quicksort* e *arrangeQueueAndPivot*. Onde o *kernel quicksort* é responsável pela comparação de elementos com pivôs, troca de posições e determinação das dimensões dos subconjuntos da próxima iteração. Além disso, em casos onde uma *thread* que esta executando este *kernel*, e opera sobre um dos elementos selecionados como pivô, este elemento será salvo no vetor pivot_arr. E posteriormente tratado no segundo *kernel* (linhas 4 e 9). Nesse sentido, o *kernel arrangeQueueAndPivot* é responsável pela seleção de novos pivôs, dimensionamento dos subconjuntos e rearranjo de pivôs salvos (linhas 6 e 11). Ademais, o algoritmo é considerado terminado apenas quando haja 0 ou 1 elementos para serem organizados na próxima iteração. Adicionalmente, após o fim cada iteração, são alternados os vetores com seus auxiliares(linhas 2 e 8).

Algorithm 4: CUDA

Result: Vetor organizado

Data: list1(vetor original); list2(vetor auxiliar); q(vetor de índices); q_copy(vetor de índices auxiliar); pivot_arr(vetor para armazenar pivôs); size(tamanho)

```
1:  while !(*done) do
2:    *done = true;
3:    if count%2 == 0 then
4:      quickSort«<numBlocks, blockSize»>(size, list1, list2, q, q_copy, pivot_arr);
5:      cudaDeviceSynchronize();
6:      arrangeQueueAndPivot«<numBlocks, blockSize»>(q2, q, size, list2, pivot_arr,
        q_copy2, q_copy, done);
7:      cudaDeviceSynchronize();
8:    else
9:      quickSort«<numBlocks, blockSize»>(size, list2, list1, q2, q_copy2, pivot_arr);
10:     checkcudaError(cudaDeviceSynchronize());
11:     arrangeQueueAndPivot«<numBlocks, blockSize»>(q, q2, size, list1, pivot_arr,
        q_copy, q_copy2, done);
12:     cudaDeviceSynchronize();
13:   end if
14:   count++;
15: end while
```

[4]

- Implementação OpenMP:

  O pseudocódigo 5 representa a implementação OpenMP, é utilizada uma abordagem *Top-Down* com o uso de chamadas recursivas. Em relação à etapa de divisão é utilizado o algoritmo de organização *insertion-sort*, para criar um conjunto com valores maiores que o pivô e outro menor em cada iteração (linhas 19 e 23). Em sequencia é realizada uma chamada recursiva para cada conjunto e o processo se repete.

[4] Baseada na implementação do repositório <https://github.com/gongminaaa/GPU_Parallel_Computing>

[5] Baseada na implementação do repositório <https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features/cdpSimpleQuicksort>

Algorithm 5: OpenMP

Result: Vetor organizado

Data: inicio(posição vetorial inicial desta iteração); fim (posição final desta iteração); data(vetor)

```
1: lptr= data+inicio ;
2: rptr = data+fim ;
3: pivô = posição do vetor (inicio+fim)/2 nesta iteração;
4: while lprtr<=rptr do
5:    lval = *lptr;
6:    rval = *rptr;
7:    while lval < pivot do
8:       Percorre o vetor
9:    end while
10:   while rval > pivot do
11:      Percorre o vetor
12:   end while
13:   if lptr <= rptr then
14:      Swap ;
15:   end if
16: end while
17: if inicio <rptr-data then
18:    #pragma omp task
19:    quicksort(data, inicio, nptr-data) ;
20: end if
21: if lptr-data < right then
22:    #pragma omp task
23:    quicksort(data, lptr-data, fim) ;
24: end if
5
```

- Implementação CUDA DP:

  O pseudocódigo 6 representa a implementação CUDA DP, neste é realizado um processo similar ao da implementação OpenMP para chegar a solução. Isto é, ambas utilizam uma abordagem *Top-Down* e chamadas recursivas (linhas 5-11). No entanto, devido à limitação de 24 na profundidade de chamadas recursivas, é necessário o uso de um algoritmo de ordenação adicional, para casos onde a profundidade de chamadas recursivas é maior ou igual 24. Além disso, também é necessário seu uso para casos onde a carga computacional não é suficientemente grande para justificar o uso de uma chamada recursiva. Nesse sentido, é utilizado o algoritmo *selection-sort* para tais casos(linhas 1-3). Adicionalmente, a nossa implementação difere da original através do uso de memória unificada e uso de múltiplas *streams*.

Algorithm 6: CUDA DP

Result: Vetor organizado

Data: inicio(posição vetorial inicial desta iteração); fim (posição final desta iteração); data(vetor); Profundidade(Profundidade da recursão)

```
1: if Profundidade >= 23 || inicio-fim <= 32 then
2:    selection_sort(data, left, right);
3:    return;
4: else
5:    Mesmo processo de organização ;
6:    if inicio <rptr-data then
7:       quicksort(data, inicio, nptr-data, profundidade+1) ;
8:    end if
9:    if lptr-data < right then
10:      quicksort(data, lptr-data, fim, profundidade+1) ;
11:   end if
12: end if
```

[6]

4.1.3 Mergesort

Mergesort é um algoritmo de ordenação que utiliza uma estratégia de divisão e conquista para solucionar o problema, semelhantemente ao algoritmo *Quicksort*. No entanto, não são utilizados pivôs como no *Mergesort*. Em vez disso, é apenas realizada a divisão continua do tamanho problema. Desta forma criando subconjuntos de elementos do vetor original. Até restarem apenas subconjuntos compostos por até 2 elementos. Após este processo, é realizada a etapa de conquista onde cada subconjuntos é ordenado. E em sequência, estes são combinados com subconjuntos que pertencem a mesma vizinhança nas posições do vetor original. E então este processo da etapa de conquista se repete até a formação do vetor original ordenado.

[6] Baseada na implementação do repositório <https://github.com/NVIDIA/CUDA-samples/tree/master/Samples/3_CUDA_Features/cdpSimpleQuicksort>

- Implementação Geral:

  Todas as implementações realizam a atribuição de conjuntos de posições do vetor para *threads*. Isto é, cada *thread* processa um subconjunto do vetor. No entanto, cada uma das implementações possui uma abordagem diferente para a solução do problema.

- Implementação CUDA:

  O pseudocódigo 7 representa a implementação CUDA, neste é utilizada uma abordagem iterativa para a etapa de divisão do problema, através do uso de um laço de repetição na CPU para determinar quantas posições do vetor, cada *thread* deve processar na iteração atual (linhas 1-2). Ou seja, o tamanho dos conjuntos atribuídos a *threads* vai crescendo a cada iteração. Nesse sentido, a etapa de conquista é realiza através do uso *threads* na GPU. Onde cada *thread* realiza a ordenação das posições do seu subconjunto (linhas 3-14). Adicionalmente, a nossa implementação difere da original através do uso de um número maior de *threads* em duas dimensões em vez de três, e o uso de memória unificada.

Algorithm 7: CUDA

Result: Vetor organizado
Data: source(vetor original desorganizado); dest(vetor resultante do algoritmo); size(tamanho do vetor); nThreads(número de threads)

```
1:  for largura = 2; largura < (size « 1); largura «= 1 do
2:    corte = size / ((nThreads) * largura) + 1;
3:    idx= Id da thread;
4:    inicio = largura*idx*corte ;
5:    for i = 0; i< cortes; i++ do
6:
7:      if inicio >= size then
8:        sair do for ;
9:      end if
10:     meio = mínimo entre (inicio + largura/2, size);
11:     fim = mínimo entre (inicio + largura, size);
12:     Insertion_Sort(source, dest, inicio, meio, fim);
13:     inicio= inicio+largura;
14:   end for
15: end for
```

- Implementação OpenMP:

  O pseudocódigo 8 representa a implementação OpenMP, neste é utilizada uma abordagem *BottomUp* para a solução do problema. Através do uso de *tasks* sobre chamadas recursivas continuas (linhas 2-6), até que o vetor seja divido em subconjunto

[7] baseada na implementação do repositório https://github.com/kevin-albert/cudamergesort

de um elemento. Dessa forma realizando a etapa de divisão. Em contraste, a etapa de conquista é realizada através do uso do algoritmo de ordenação *selection_sort* (linha 8).

Algorithm 8: OpenMP

Result: Vetor organizado

Data: inicio(posição vetorial inicial desta iteração); fim (posição final desta iteração); vetor(vetor desorganizado),tmp(vetor auxiliar)

```
1: if inicio<fim then
2:    separar=(inicio+fim)/2;
3:    #pragma omp task
4:    mergesort(vetor, inicio, separar, tmp);
5:    #pragma omp task
6:    mergesort(vetor, separar+1, fim, tmp);
7:    #pragma omp taskwait
8:    selection_sort(vetor, inicio, fim, separar, tmp);
9: end if
```

- Implementação CUDA DP:

  O pseudocódigo 9 representa a implementação CUDA DP, neste o processo de divisão é realizado similarmente a implementação CUDA padrão. Ambas, utilizam laços de repetições para determinar o número de posições que devem ser processados em uma iteração. (linhas 1-7). No entanto, estas diferem na etapa de conquista em casos onde o tamanho do subproblema seja suficientemente grande. Na implementação DP é realizada uma sub-divisão, através da chamada de um segundo *kernel*. Onde é realizada à ordenação via uma busca binaria (linhas 11–20). Senão, o processo de ordenação é o mesmo da implementação CUDA padrão.

Algorithm 9: CUDA DP

Result: Vetor organizado

Data: Size(tamanho do vetor); arr(vetor); aux(vetor auxiliar)

Loop na CPU

1: **for** tamanho_atual = 1 ;
tamanho_atual < size ;
tamanho_atual= tamanho_atual*2 **do**
2: largura= currentSize*2 ;
3: num_sorts = (size + largura - 1) / largura ;
4: Kernel na GPU
5: idx= Id da thread ;
6: inicio = largura*idx ;
7: Verificação se os índices estão fora das dimensões dos vetores
8: meio = inicio+tamanho_atual-1 ;
9: fim = mínimo entre(inicio +largura -1, size -1);
10: num_Threads = fim - inicio + 1 ;
11: **if** n_Threads > 16384 && n_Threads<1024*1024 **then**
12: numThreadsPerBlock = 1024 ;
13: Chamada de um kernel secundário para realizar um *binary search*
14: idx2 = Id da thread no kernel secundário;
15: ninicio = meio - inicio + 1;
16: nfim = fim - meio ;
17: nTot = ninicio + nfim ;
18: função busca o id correto para a *thread* e realiza *binary search*
19: arrIndex = getIndex(&aux[inicio], idx2, nfim, nTot) ;
20: arr[inicio + arrIndex] = aux[inicio + idx2] ;
21: **end if**
22: **end for**

[8]

4.1.4 Bread First Search

Bread First Search é um algoritmo de processamento de grafos árvores. Seu objetivo consiste da realização de uma busca de largura em busca de um elemento. Ou seja, é realizada uma busca a partir da raiz do grafo que percorre todo o grafo. Através da travessia das adjacências de vértices. De tal forma que cada iteração processa todas as adjacências do vértice sendo processado. E na seguinte iteração, as adjacências dos vértices adjacentes da iteração anterior são percorridas. Até que a busca chegue nas folhas do

[8] baseada na implementação do repositório https://github.com/JoeyOhman/GPUMergeSort

grafo ou encontre o vértice que esta sendo buscado. No entanto, a nossa implementação visa percorrer todos os vértices. E atribuir a estes a menor distância entre si e a raiz.

- Implementação Geral:

  Todas as implementações utilizam uma abordagem Top-Down em relação à travessia do grafo. De forma que a travessia é realizada através do uso de filas ou *queues*. Para demarcar quais vértices devem ser processados em subsequentes iterações. Nesse sentido, a inserção de elementos nestas estruturas é realizada após o processamento de um vértice. De forma que as adjacências do vértice processado são inseridas nestas. Adicionalmente, em relação à representação do grafo, é utilizada uma estrutura de dados grafo, que contem conjuntos de vetores. Sendo estes: adjacências; vértice inicial; número de adjacências de cada vértice.

- Implementação CUDA:

  O pseudocódigo 10 representa a implementação CUDA padrão, a sua execução é inicializada através da realização de uma estimativa da distância de todos os vértices do grafo até a raiz. Tal que é atribuído um valor infinito a distância de todos os vértices, exceto a raiz. Após esta etapa de inicialização, é realizado um laço de repetição na CPU, responsável pela chamada do *kernel* de solução (linhas 1 e 13–19). Em suma, o *kernel* de solução, consiste da utilização de um vetor para simular a fila de nodos. E através desta fila realiza o processamento das adjacências dos vértices inseridos (linhas 5-9). De tal forma que todas as adjacências com menor distância que a estimava são adicionadas a fila (linha 6-9). Nesse sentido, inicialmente o único elemento desta fila é o nodo raiz, já que este possui distância 0. Adicionalmente, após o termino da execução de uma iteração do *kernel*, é verificado na CPU se a fila está vazia. E em casos positivos, quais vértices devem ser processados na próxima iteração (linhas 13–19).

Algorithm 10: CUDA

Result: Menor distancia entre a raiz da árvore e seu nodos

Data: distancia(vetor utiliza para armazenar a distancia); nivel(altura atual da árvore); lista_adj(lista de adjcencias); edgesOffset(vetor que guarda a posição inicial das adjacências de nodos no vetor lista_adj); edgesSize(vetor utilizado para aguardar o número de adjacências de cada nodo); grafo; d_fila_atual(vetor que serve como fila dos nodos a serem processados nesta iteração); d_proxima_fila(vetor que serve como fila dos nodos a serem processados na proxima iteração)

```
1:  while queueSize ! =0 do
2:    thid = Id da thread ;
3:    if thid < queueSize then
4:      u = d_fila_atual[thid];
5:      for i = posição inicial das adj de u no vetor de adj;
        i < posição final das adj de u no vetor de adj ;
        i++ do
6:        int v = Lista_adj[i];
7:        if d_distance[v] == INT_MAX && d_distancia=mínimo entre
          (d_distance[v], level + 1) == INT_MAX then
8:          position = nextQueueSize, 1;
9:          d_proxima_fila[position] = v;
10:       end if
11:     end for
12:   end if
13:   Retorno para GPU
14:   cudaDeviceSynchronize() ;
15:   nivel=nivel+1;
16:   queueSize=nextQueueSize;
17:   nextQueueSize=0;
18:   Realiza troca entre filas
19:   swap(d_fila_atual,d_proxima_fila);
20: end while
```

9

[9] baseado na implementação GPU no repositório https://github.com/rafalk342/bfs-cuda

- Implementação OpenMP:

  O pseudocódigo 11 representa a implementação OpenMP, neste a solução é realizada através do uso de um laço de repetição contendo uma área paralela. Onde cada iteração realiza a atribuição de um conjunto vetorial para *threads*. De forma que cada *thread* realiza o processamento de um conjunto de adjacências (linhas 9-13). E caso ocorra alguma mudança na estimativa da distância de um vértice, é realizada outra iteração do laço (linhas 17–18 e 3). Nesse sentido, o algoritmo só é considerado finalizado quando não ocorrem mais mudanças nas estimativas.

Algorithm 11: OpenMP

Result: Menor distancia entre a raiz da árvore e seu nodos

Data: distancia(vetor utiliza para armazenar a distancia); nivel(altura atual da árvore); lista_adj(lista de adjcencias); edgesOffset(vetor que guarda a posição inicial das adjacências de nodos no vetor lista_adj); edgesSize(vetor utilizado para aguardar o número de adjacências de cada nodo); grafo

```
1:  changed=1 ;
2:  while changed==1 do
3:    valueChange = 0;
4:    # pragma omp parallel for shared(changed, valueChange)
5:    for j=0; j<num_vertices; j++ do
6:      changed = 0;
7:      if distancia[j]==nivel then
8:        for  i = posição inicial das adj de j no vetor de adj;
          i < posição final das adj de j no vetor de adj ;
          i++ do
9:          v = Lista_adj[i];
10:         if nivel + 1 < distancia[v] then
11:           distancia[v] = nivel + 1;
12:           valueChange = 1;
13:         end if
14:       end for
15:     end if
16:     if valueChange then
17:       changed = valueChange;
18:     end if
19:   end for
20:   nivel++;
21: end while
```

[10]

[10] baseado na implementação CPU no repositório https://github.com/rafalk342/bfs-cuda

- Implementação CUDA DP:

  O pseudocódigo 12 representa a implementação CUDA DP, nesta a solução é realizada de uma forma similar a implementação CUDA. Tal que, é utilizado um laço de repetição na CPU (linha 1 e 25–28), que chama um *kernel* na GPU (linhas 5-24). Onde são atribuídos vértices a *threads* e estás *threads* realizam o processamento de adjacências de nodos (linhas 7-13). No entanto, estas diferem em casos onde ocorrem mudanças nas estimativas de vértices que possuam alguma adjacência. Na implementação CUDA DP é realizada a chamada de um *kernel* secundário que analisa as adjacências do vértice que sofreu a modificação (linhas 14–16).

[11] baseado na implementação do repositório <https://github.com/NPSLab/show-me-dynamic-parallelism/tree/master/apps/bfs-rec>

Algorithm 12: CUDA DP

Result: Menor distancia entre a raiz da árvore e seu nodos

Data: distancia(vetor utiliza para armazenar a distancia); nivel(altura atual da árvore),lista_adj(lista de adjcencias); edgesOffset(vetor que guarda a posição inicial das adjacências de nodos no vetor lista_adj); edgesSize(vetor utilizado para aguardar o número de adjacências de cada nodo); grafo

```
1: Loop na CPU
2: while changed do
3:    changed = 0;
4:    kernel da GPU
5:    thid = Id da thread;
6:    valueChange = 0;
7:    if thid < N && d_distance[thid] <= level then
8:       int count=0;
9:       for i = posição inicial das adj de thid no vetor de adj;
         i < posição final das adj de thid no vetor de adj ;
         i++ do
10:         v = Lista_adj[i];
11:         if d_distance[u] + 1< d_distance[v] then
12:            d_distance[v] = d_distance[u] + 1;
13:            d_parent[v] = i;
14:            if Número de adj de V>0 then
15:               O kernel dois é responsável pelo processamento das adjacências do nodo
                  adjacente ao nodo original
16:               kernel2«<....»>(....);
17:            end if
18:            valueChange = 1;
19:         end if
20:      end for
21:   end if
22:   if valueChange then
23:      changed = valueChange ;
24:   end if
25:   retorno para CPU
26:   cudaDeviceSynchronize();
27:   nivel=nivel+2;
28: end while
11
```

4.2 Ambiente de execução

Nesta subseção encontra-se a descrição do ambiente de execução utilizado para a condução dos experimentos. Além das ferramentas utilizadas para a coleta dos resultados e os métodos utilizados para a validação de tais.

Os experimentos foram realizados em uma máquina heterogênea com arquitetura x86_64 64-bit com as seguintes especificações:

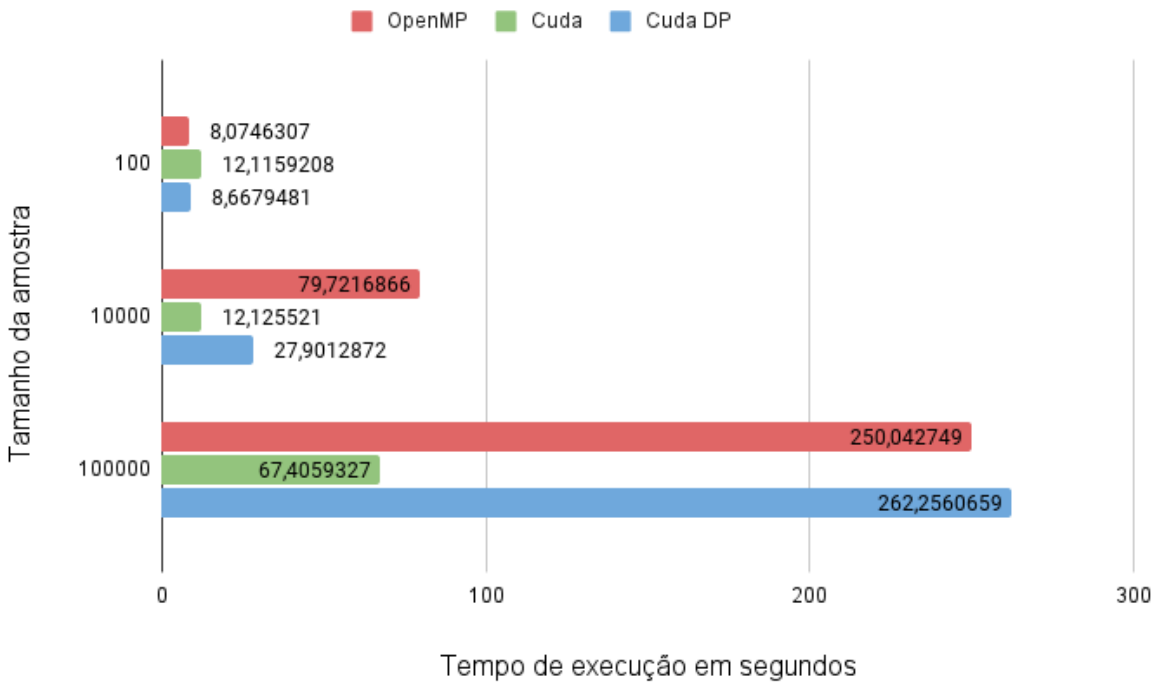
- Memória: 50 GB de memória.
- Processador: *AMD Ryzen 9 3900X 12-Core* com 2 *threads* por *core*
- GPU: GeForce GTX 1050 com 869 CUDA *cores*.
- Sistema Operacional: Ubuntu 18.04 do Linux

Para a coleta das métricas estabelecidas, foram realizadas 10 iterações de 10000 execuções em sequencia de cada *benchmark*. De forma, que cada iteração totaliza a soma do tempo utilizado para as 10000 execuções de cada *benchmark*. Ademais, a partir destas 10 iterações foi calculada a média utilizada como métrica para os resultados na Seção 5. Ademais, o tempo total de cada uma das execuções foi obtida através do cálculo da diferença entre o tempo coletado da função *gettimeofday()* (FREE SOFTWARE FOUNDATION, 2021), antes do início e após o termino da execução. Por outro lado, para a obtenção da métrica do consumo de energia na CPU foi utilizada a ferramenta *perf* (FREE SOFTWARE FOUNDATION, 2022), com o monitoramento da métrica *energy-pkg* da ferramenta duranta as execuções. Adicionalmente, foi utilizada a ferramenta *nvidia-smi*, para a medição do consumo de energia na GPU, através do monitoramento da energia utilizada entre intervalos de 1000ms durante a execução de cada implementação. Ademais, todos os resultados das execuções foram comprovados através do uso dos resultados de suas versões sequências no caso dos *benchamarks* *Dijkstra* e *breadth-first search*. E através da verificação da ordenação dos vetores no caso dos *benchamarks* *mergesort* e *quicksort*. .

5 RESULTADOS

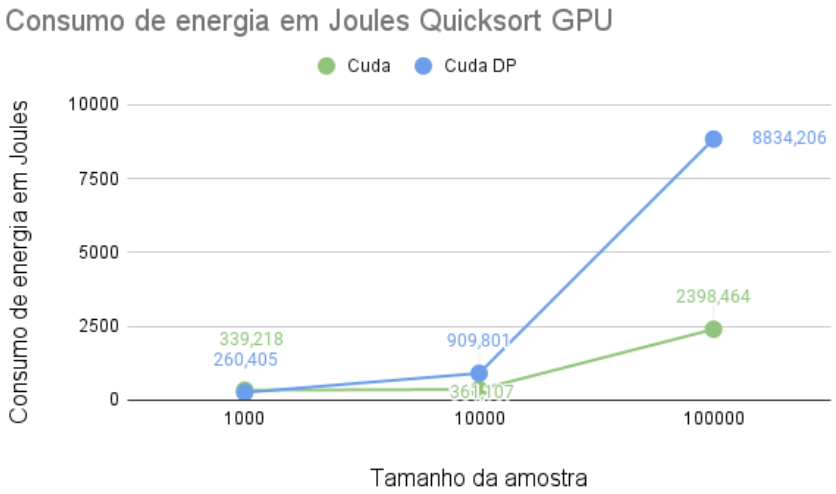
A Figura 7, demonstra o resultado da média entre 10 iterações de 10000 execuções de cada uma das 3 implementações do *benchmark Quicksort*. É observado um padrão esperado para a implementação DP. Onde o seu tempo de execução é o maior para amostras de maior tamanho e relativamente menor com amostras de tamanho menor em comparação as outras implementações. A razão para tal é a limitação da profundidade de chamadas recursivas na implementação DP, necessitando o uso da função *selection sort* mais vezes em problemas maiores. Dessa forma criando o tempo elevado observado na figura. Ademais, a implementação CUDA possui o menor tempo em duas das amostras consideradas. A razão para isto é devido à carga de trabalho entre as *threads* esteja relativamente balanceada, assim permitindo eficiência em amostras maiores. Adicionalmente, a implementação OpenMP também possui um aumento significativo de tempo em relação a amostras maiores. Similarmente a implementação DP, no entanto, esta não possui um tempo de execução tão grande em relação a maior amostra.

Figura 7 – Média de tempo entre execuções das implementações *Quicksort*



Fonte: do autor

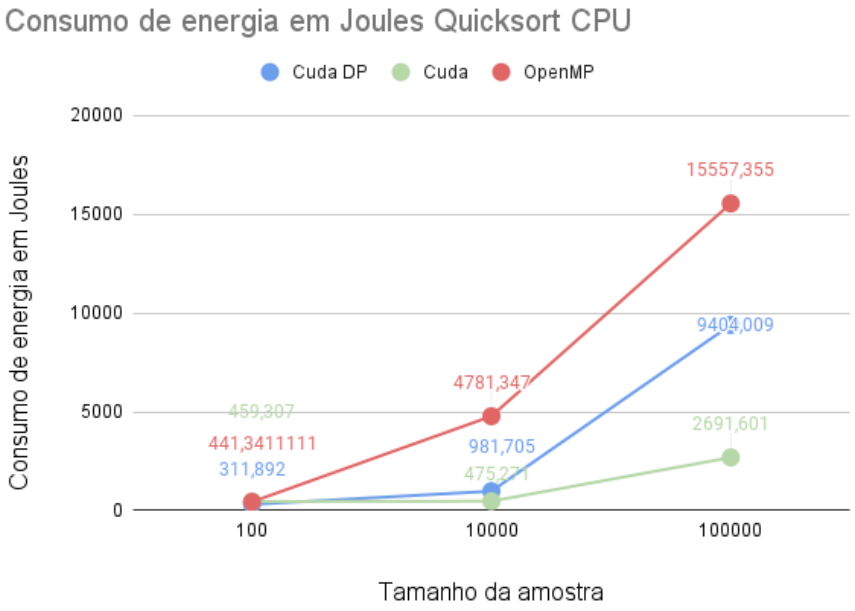
Figura 8 – Média do consumo de energia da GPU em Joules das implementações CUDA *Quicksort*



Fonte: do autor

A Figura 8, demonstra o de consumo de energia da GPU nas implementações CUDA *Quicksort*. Observar-se que a implementação CUDA DP com a menor amostra possui o menor consumo de energia. No entanto, como é esperado, quando há uma diferença maior entre os tempos de execução das implementações, foi observada uma diferença significativa entre o consumo de energia das implementações CUDA.

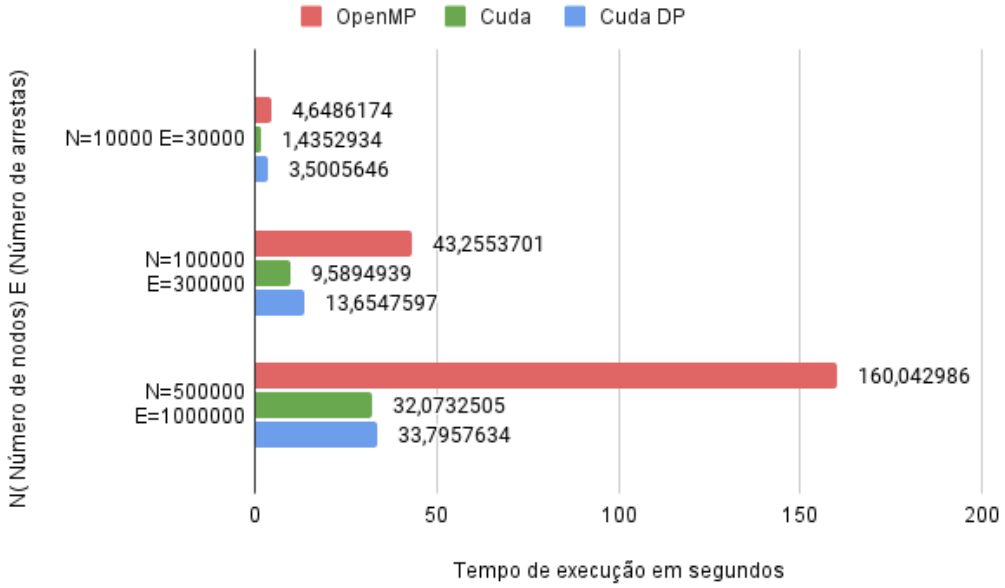
Figura 9 – Média do consumo de energia da CPU em Joules das implementações CUDA *Quicksort*



Fonte: do autor

A Figura 9 demonstra o consumo de energia na CPU das implementações *quick-sort*. Como esperado, as implementações CUDA possuem um menor consumo de energia comparado a OpenMP. Devido a OpenMP ser processada inteiramente na CPU. Adicionalmente, é observado o mesmo padrão da Figura 9 entre as implementações CUDA.

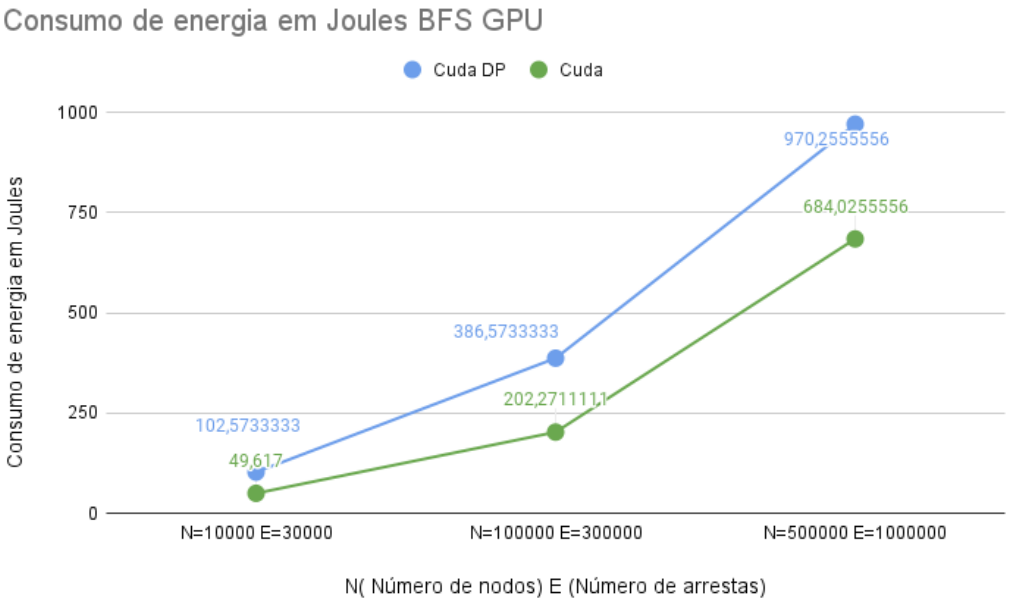
Figura 10 – Média entre execuções das implementações BFS



Fonte: do autor

A Figura 10, demonstra o resultado da média entre 10 iterações de 10000 execuções do *benchmark* em sequencia, de cada uma das 3 implementações do *benchmark BFS*. Neste é observado um tempo de execução elevada na implementação OpenMP em relação a outras implementações. A razão para tal é o processo de atribuição de vértices. Onde independente da necessidade, *threads* processarão suas adjacências. Isto é, mesmo que já tenha sido calculada a altura de um nodo e não tenha sido encontrado outro caminho, este ainda será processado. Em contraste, aparenta haver padrão entre as implementações CUDA, onde a diferença entre o tempo de execução destas aparenta diminuir quanto maior o tamanho do problema. É provável que a razão para tal é que existem adjacências suficientes para justificar a chamada do segundo *kernel* em amostras maiores.

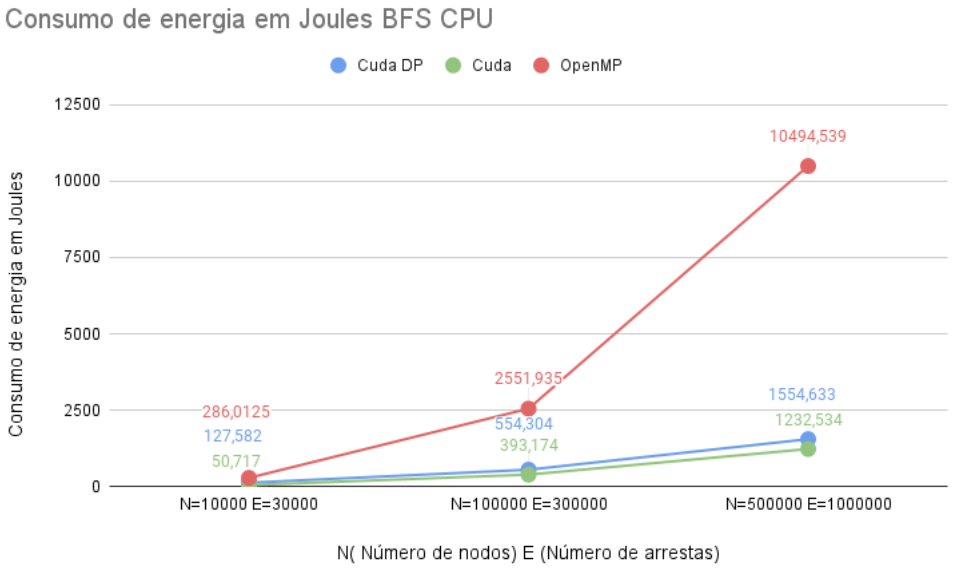
Figura 11 – Média do consumo de energia da GPU em Joules das implementações CUDA BFS



Fonte: do autor

A Figura 11, demonstra à média de consumo de energia na GPU das implementações CUDA BFS. Onde é observado que apesar de ambas possuírem tempos de execução comparáveis, há um aumento significativo no consumo de energia na implementação CUDA DP, a razão para tal é o uso do segundo *kernel*.

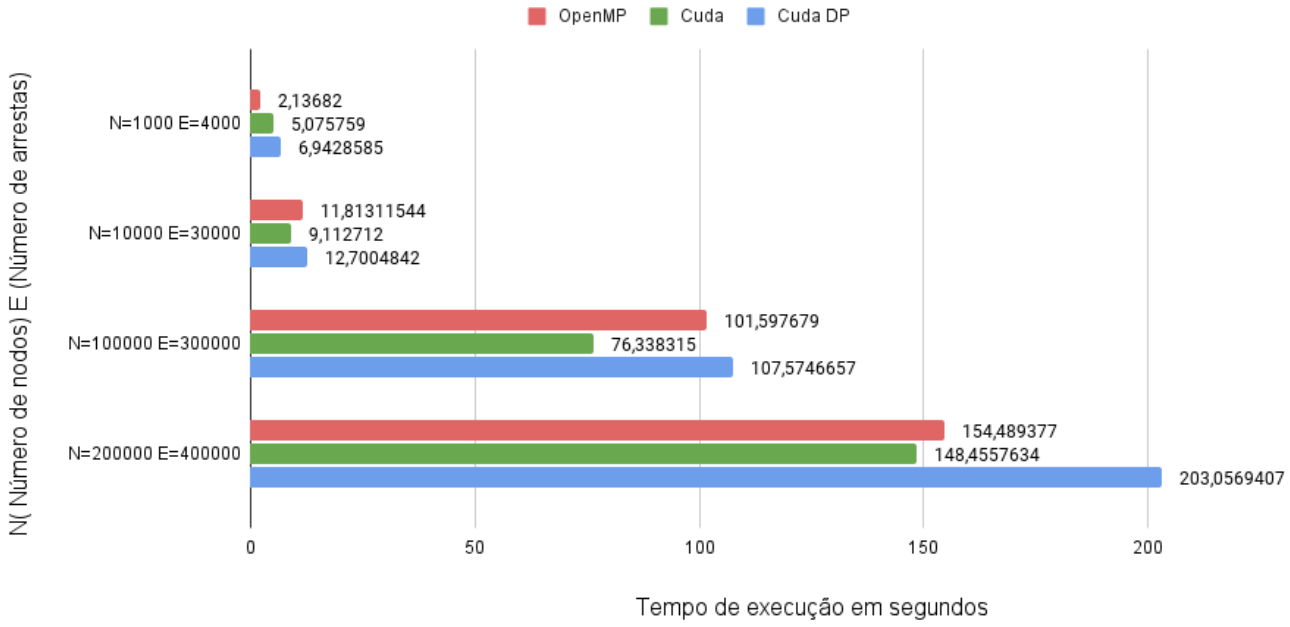
Figura 12 – Média do consumo de energia da CPU em Joules das implementações CUDA BFS



Fonte: do autor

A Figura 12, demonstra o consumo de energia na CPU durante a execução das implementações BFS. Novamente, é observado um comportamento esperado entras as implementações CUDA e OpenMP. Adicionalmente, há uma diferença entre o consumo de energia das implementações CUDA como observado na Figura 12.

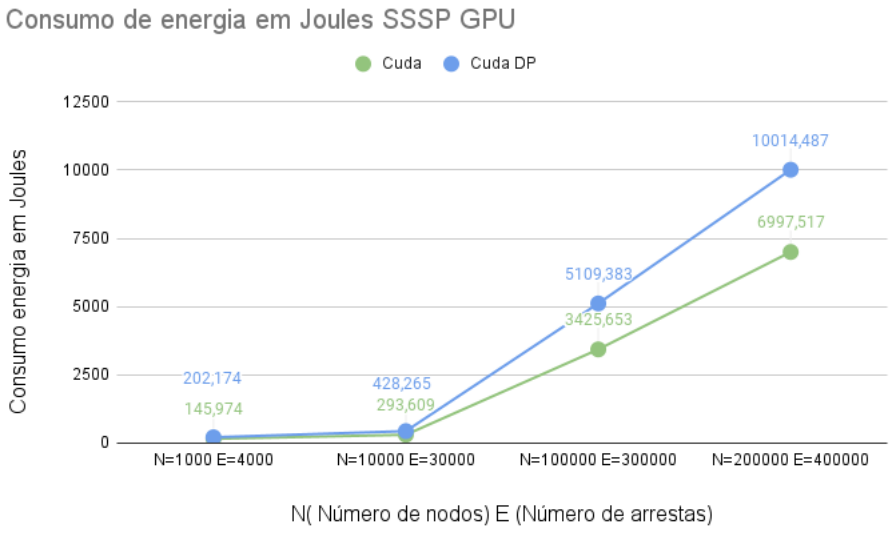
Figura 13 – Média entre 10000 execuções das implementações SSSP



Fonte: Do autor

A Figura 13, demonstra o resultado da média entre 10 execuções de 1000 iterações de cada uma das 3 implementações do *benchmark Djikstra/SSSP*. Onde é observado um tempo de execução elevado na implementação CUDA DP, a razão para tal é a necessidade de sincronização entre *threads*, após a chamada do *kernel* secundário. Adicionalmente, é observada um desempenho superior nas implementações CUDA e OpenMP em relação a DP. Ademais, na implementação DP é provável que à condição de chamada recursiva baseada na existência de adjacências tenha criado divergência entre a carga de trabalho entre *threads* ou a repetição do processamento desnecessário. Já que, nem todos os nodos possuem o mesmo número de adjacências. E essa analise pode ser desnecessária em alguns casos, no entanto, devido à natureza da consistência de memória em DP, não é possível prever este fato durante a execução.

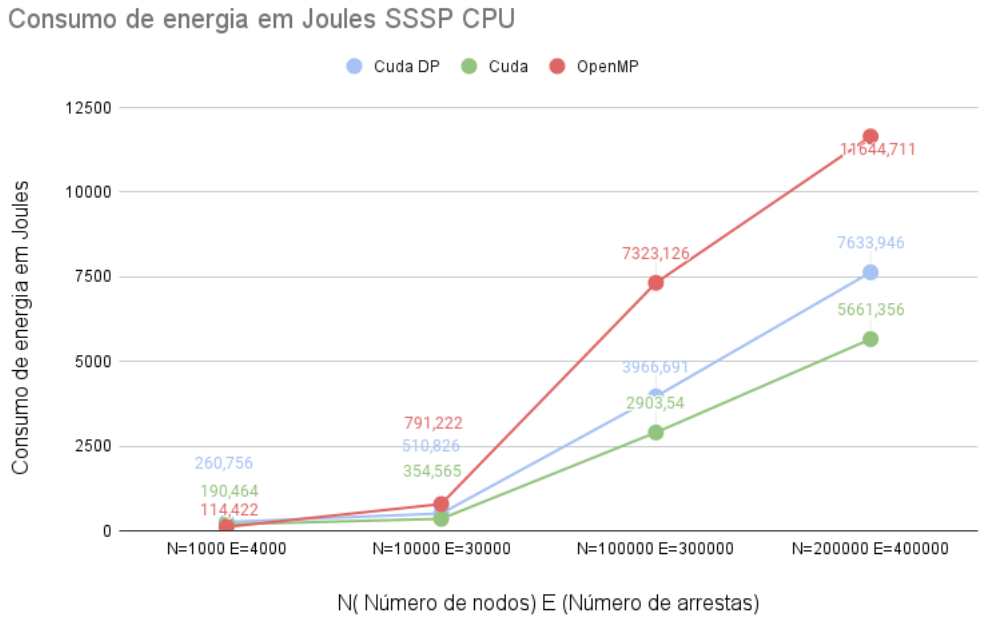
Figura 14 – Média do consumo de energia da GPU em Joules das implementações CUDA SSSP



Fonte: Do autor

A Figura 14, demonstra a média de consumo de energia na GPU durante a execução das implementações CUDA *Djikstra*. É observado que mesmo em etapas onde seus tempos eram relativamente próximos, a implementação CUDA DP possui um consumo de energia maior, a razão para tal e o segundo *kernel*.

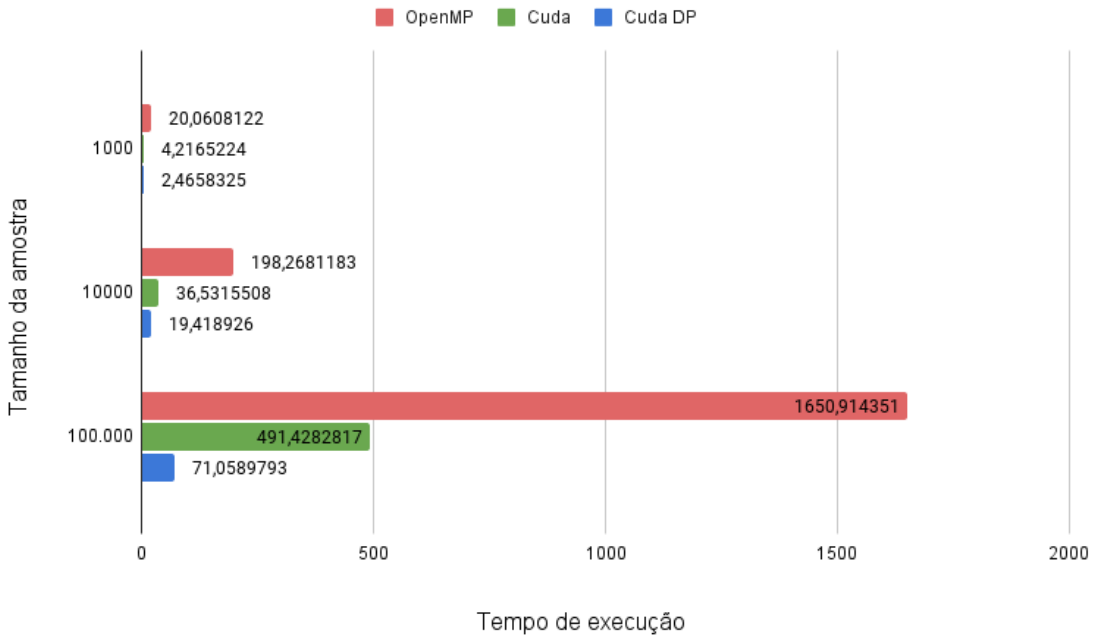
Figura 15 – Média do consumo de energia da CPU em Joules das implementações CUDA SSSP



Fonte: Do autor

A Figura 15, demonstra o consumo de energia da CPU das implementações SSSP. Onde como esperado, a implementação OpenMP possui o maior consumo por ser processada inteiramente na CPU. Após esta, a implementação DP possui o segundo maior e a CUDA padrão o menor consumo.

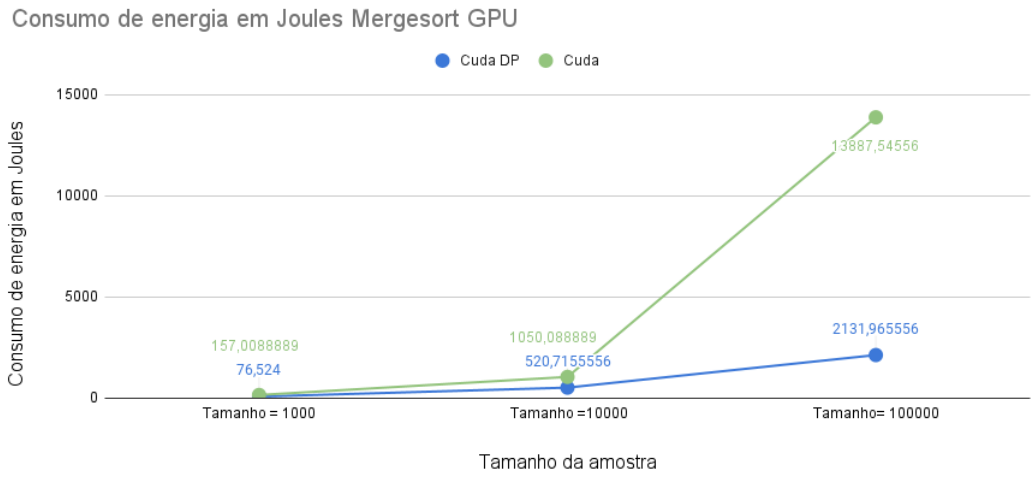
Figura 16 – Média entre 10000 execuções das implementações Mergesort



Fonte: do autor

A Figura 16, demonstra o resultado da média entre 10 iterações de 10000 execuções do *benchmark* em sequencia, de cada uma das 3 implementações do *benchmark mergesort*. Nesta figura é observado um de tempo de execução elevado na implementação OpenMP em comparação a outras implementações. A razão para tal é que foi atribuída uma carga de trabalho desbalanceada para as *thread*. Em contraste, o desempenho superior do DP é devido ao uso de uma carga de trabalho balanceada entre *threads* e suficientemente grande para justificar o uso do segundo *kernel* de busca binaria. Ademais, a implementação CUDA padrão possui um desempenho mediano em relação as outras implementações.

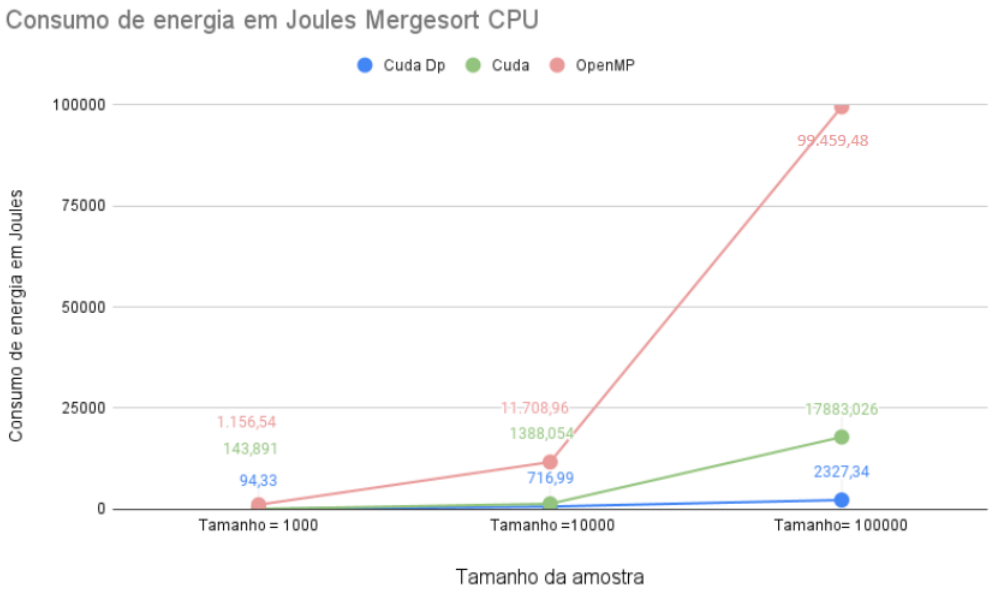
Figura 17 – Média do consumo de energia da GPU em Joules das implementações CUDA Mergesort



Fonte: do autor

A Figura 17 demonstra a média de consumo de energia da GPU das implementações CUDA *Mergesort* Como esperado a implementação DP possui um menor consumo de energia devido ao seu tempo de execução comparativamente menor a implementação CUDA.

Figura 18 – Média do consumo de energia da CPU em Joules das implementações CUDA Mergesort



Fonte: do autor

A Figura 18, demonstra o consumo de energia na CPU das implementações *mergesort*. Assim como observado na Figura 17, a implementação DP possui um menor consumo

de energia em todos os tamanhos de amostras, devido ao menor tempo de execução. Além disso, como esperado, a implementação com maior consumo de energia foi a OpenMP. Ademais, a implementação com o segundo menor consumo foi a CUDA padrão.

5.1 Dificuldades Encontradas

Nesta Seção encontra-se uma breve descrição sobre as dificuldades encontradas durante o desenvolvimento deste trabalho.

Além das limitações previamente listadas pelo autor, como a profundidade de chamadas recursivas e os custos inerentes do uso de DP, foram encontradas outras características e limitações durante o desenvolvimento deste trabalho que dificultaram a condução dos experimentos. Estas incluem: 1.limitações de memória em relação à quantidade total de chamadas recursivas; 2. dificuldades na implementação de *benchmarks* devido ao modelo de memória entre pai e filho; 3. Dificuldade da atribuição de um carga de trabalho adequada entre *threads*.

Elaborando no primeiro ponto, devido à quantidade limitada de memória reservada para chamadas recursivas pendentes, algumas das soluções consideradas anteriormente para os *benchmarks*, foram modificadas ou descartadas. Pois estas estouravam este limite, e este fato causava um comportamento imprevisível durante a execução de tais. Já que, não era possível inserir estas na fila de chamadas pendentes e estas eram descartadas. Adicionalmente, em relação ao segundo ponto, a inconsistência de memória entre pai e filho dificultou a implementação de soluções para algoritmos tradicionalmente *Bottom-Up* como o *mergesort*. E soluções que envolviam uso de filas ou listas, como o BFS, pois não era possível garantir a consistência de elementos acessados, durante a execução do pai ou filho. E por fim, em relação ao terceiro ponto, algumas das abordagens de uso de paralelismo dinâmico consideradas não justificam seu uso. Por razão do *speedup* providenciado não superava os custos inerentes de DP. Devido à carga de trabalho mal otimizada ou mal distribuído entre *threads*. Ou seja, é necessária uma análise do trabalho computacional sendo atribuído a *kernels* e como este é subdividido entre *threads*.

6 CONSIDERAÇÕES FINAIS

Neste trabalho foram exemplificados os custos e benefícios do uso do paralelismo dinâmico para o modelo de programação CUDA. Através da realização de uma comparação entre CUDA, CUDA DP, e OpenMP. Assim, é possível afirmar que CUDA paralelismo dinâmico é uma extensão útil para a redução de tempo de execução e consumo de energia. E seu desempenho se destaca quando utilizada com cargas de trabalho balanceada e suficientemente grande em aplicações que utilizam dados semi-estruturados ou organizados hierarquicamente. Assim como foi demonstrado em relação aos resultados das implementações *Mergesort* e BFS. No entanto, este possui particularidades que dificultam seu uso para algumas abordagens de solução de problemas. Como problemas onde recursões possuem dependências entre si, soluções *Bottom-Up* e soluções que requerem profundidade demasiada em suas recursões. Além disso, nem todas abordagens recursivas trazem bons resultados para o desempenho de aplicações. No entanto, programadores experientes que possuem um bom entendimento das características da extensão e de CUDA conseguem contornar estas dificuldades.

REFERÊNCIAS

ALANDOLI, M. et al. Using dynamic parallelism to speed up clustering-based community detection in social networks. In: **2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)**. [S.l.: s.n.], 2016. p. 240–245. Citado na página 23.

ALIAGA, J. et al. Harnessing cuda dynamic parallelism for the solution of sparse linear systems. **Parallel Computing: On the Road to Exascale**, IOS Press, v. 27, p. 217, 2016. Citado na página 23.

BARNEY, B. et al. Introduction to parallel computing. **Lawrence Livermore National Laboratory**, v. 6, n. 13, p. 10, 2010. Citado na página 10.

DIMARCO, J.; TAUFER, M. Performance impact of dynamic parallelism on different clustering algorithms. In: SPIE. **Modeling and Simulation for Defense Systems and Applications VIII**. [S.l.], 2013. v. 8752, p. 97–104. Citado na página 23.

DONG, J.; WANG, F.; YUAN, B. Accelerating birch for clustering large scale streaming data using cuda dynamic parallelism. In: SPRINGER. **International Conference on Intelligent Data Engineering and Automated Learning**. [S.l.], 2013. p. 409–416. Citado na página 24.

FALCÃO, D. M. High performance computing in power system applications. In: SPRINGER. **International Conference on Vector and Parallel Processing**. [S.l.], 1996. p. 1–23. Citado na página 10.

FLYNN, M. Very high-speed computing systems. **Proceedings of the IEEE**, v. 54, n. 12, p. 1901–1909, 1966. Citado na página 14.

FREE SOFTWARE FOUNDATION. **gettimeofday(2) — Linux manual page**. [S.l.], 2021. Disponível em: <https://man7.org/linux/man-pages/man2/settimeofday.2.html>. Citado na página 43.

FREE SOFTWARE FOUNDATION. **perf(1) — Linux manual page**. [S.l.], 2022. Disponível em: <https://man7.org/linux/man-pages/man1/perf.1.html#OPTIONS>. Citado na página 43.

GAO, Y.; ZHANG, P. A survey of homogeneous and heterogeneous system architectures in high performance computing. In: **2016 IEEE International Conference on Smart Cloud (SmartCloud)**. [S.l.: s.n.], 2016. p. 170–175. Citado na página 10.

HAJJ, I. E. et al. Klap: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In: IEEE. **2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.], 2016. p. 1–12. Citado na página 25.

HENNESSY, D. A. P. J. L. **Computer Architecture a Quantitative Approach**. [S.l.]: Morgan kaufmann, 2012. Citado na página 13.

HENNESSY, D. A. P. J. L. **Computer Organization and Design The hardware/Software Interface**. [S.l.]: Morgan kaufmann, 2014. Citado na página 15.

JARZĄBEK, Ł.; CZARNUL, P. Performance evaluation of unified memory and dynamic parallelism for selected parallel cuda applications. **The Journal of Supercomputing**, Springer, v. 73, n. 12, p. 5378–5401, 2017. Citado na página 22.

JIN, C. et al. A survey on software methods to improve the energy efficiency of parallel computing. **The International Journal of High Performance Computing Applications**, Sage Publications Sage UK: London, England, v. 31, n. 6, p. 517–549, 2017. Citado na página 10.

KHORNOS GROUP. **OpenCL Reference Pages**. [S.l.], 2011. Disponível em: <https://registry.khronos.org/OpenCL/sdk/1.2/docs/man/xhtml/>. Citado na página 10.

NVIDIA. **CUDA C++ Programming Guide**. [S.l.], 2022. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Citado 8 vezes nas páginas 10, 11, 15, 16, 17, 18, 19 e 20.

OLABI, M. G. et al. A compiler framework for optimizing dynamic parallelism on gpus. In: IEEE. **2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)**. [S.l.], 2022. p. 1–13. Citado na página 24.

OPENMP ARCHITECTURE REVIEW BOARD. **OpenMP Application ProgrammingInterface**. [S.l.], 20221. Disponível em: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf>. Citado na página 20.

ORACLE. **Multithreaded Programming Guide**. [S.l.], 2022. Disponível em: <https://docs.oracle.com/cd/E26502_01/pdf/E35303.pdf>. Citado na página 10.

OSHANA, R.; KRAELING, M. **Software engineering for embedded systems: Methods, practical techniques, and applications**. [S.l.]: Newnes, 2019. Citado na página 15.

OZEN, G.; AYGUADE, E.; LABARTA, J. Exploring dynamic parallelism in openmp. In: **Proceedings of the Second Workshop on Accelerator Programming Using Directives**. New York, NY, USA: Association for Computing Machinery, 2015. (WACCPD '15). ISBN 9781450340144. Citado na página 25.

PESSOA, T. C. et al. Gpu-accelerated backtracking using cuda dynamic parallelism. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 30, n. 9, p. e4374, 2018. Citado na página 24.

PLAUTH, M. et al. Using dynamic parallelism for fine-grained, irregular workloads: A case study of the n-queens problem. In: **2015 Third International Symposium on Computing and Networking (CANDAR)**. [S.l.: s.n.], 2015. p. 404–407. Citado na página 23.

TANG, X. et al. Controlled kernel launch for dynamic parallelism in gpus. In: IEEE. **2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.], 2017. p. 649–660. Citado na página 24.

WANG, J.; YALAMANCHILI, S. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In: IEEE. **2014 IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.], 2014. p. 51–60. Citado na página 22.

YANG, Y.; ZHOU, H. Cuda-np: Realizing nested thread-level parallelism in gpgpu applications. In: **Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: Association for Computing Machinery, 2014. (PPoPP '14), p. 93–106. ISBN 9781450326568. Citado na página 25.

ZHANG, P. et al. Dynamic parallelism for simple and efficient gpu graph algorithms. In: **Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms**. [S.l.: s.n.], 2015. p. 1–4. Citado na página 22.