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

OTHO JOSÉ SIRTOLI MARCONDES

ESTUDO SOBRE FERRAMENTAS PARA PROGRAMAÇÃO HÍBRIDA EM ARQUITETURAS COM CPU E GPU

CAXIAS DO SUL

2024

OTHO JOSÉ SIRTOLI MARCONDES

ESTUDO SOBRE FERRAMENTAS PARA PROGRAMAÇÃO HÍBRIDA EM ARQUITETURAS COM CPU E GPU

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

Orientador: André Luis Martinotto

CAXIAS DO SUL

2024

OTHO JOSÉ SIRTOLI MARCONDES

ESTUDO SOBRE FERRAMENTAS PARA PROGRAMAÇÃO HÍBRIDA EM ARQUITETURAS COM CPU E GPU

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

Aprovado em 00/00/2021

BANCA EXAMINADORA

_____

André Luis Martinotto
Universidade de Caxias do Sul - UCS

_____

Daniel Luis Notari
Universidade de Caxias do Sul - UCS

_____

Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

RESUMO

Arquiteturas com processadores de múltiplos núcleos e com GPUs estão se tornando cada vez mais comuns nos computadores atuais. Essas arquiteturas trazem complicações no sentido de programação, pois diferentes ferramentas devem ser utilizadas para cada tipo de arquitetura. Nesse contexto, surgiram ferramentas com o objetivo de facilitar o desenvolvimento de programas paralelos para as arquiteturas com CPUs e GPUs, como por exemplo, o StarPU, OpenACC e OmpSs. Após um estudo comparativo, optou-se por avaliar o desempenho do OpenACC, devido ao fato desse ser uma ferramenta mais consolidada no mercado, tendo apoio de empresas como Nvidia, AMD e HP. O OpenACC foi avaliado a partir da paralelização de 4 aplicações: multiplicação de matrizes, método de Jacobi, busca em largura em grafos e *radix sort*. A seleção baseou-se no fato dessas aplicações serem comumente utilizadas em ferramentas de *benchmarks* que avaliam o desempenho de arquiteturas com CPUs e GPUs. Para essa avaliação, foram desenvolvidas versões sequenciais das aplicações; implementações paralelizadas para CPUs utilizando OpenMP; implementações desenvolvidas para GPUs utilizando CUDA; e aplicações paralelizadas para arquiteturas híbridas utilizando o OpenACC. Todas as implementações foram comparadas considerando o tempo de execução. As implementações utilizando OpenACC tiveram resultados satisfatórios, obtendo reduções consideráveis nos tempos de execução das aplicações. O OpenACC mostrou-se uma alternativa eficiente, pois permite aproveitar de forma satisfatória os recursos das GPUs sem a necessidade de alterações significativas no código fonte, como é frequentemente exigido em CUDA.

Palavras-chave: Paralelização, Arquiteturas Híbridas, Unidade Central de Processamento (CPU), Unidade de Processamento Gráfico (GPU)

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

3D Tridimensional

AMD *Advanced Micro Devices*

BSC *Barcelona Supercomputing Center*

CEO *Chief Executive Officer*

CPU *Central Processing Unit*

CUDA *Compute Unified Device Architecture*

DSM *Distributed Shared Memory*

GB *Gigabyte*

GCC *GNU Compiler Collection*

GHz *Gigahertz*

GPU *Graphics Processing Unit*

HDD *Hard Disk Drive*

HP *Hewlett-Packard*

HPC *High-Performance Computing*

KB *Kilobyte*

lws *Local Work-stealing*

MB *Megabyte*

Mflops *Millions of Floating-Point Operations Per Second*

MHz *Megahertz*

MIMD *Multiple Instruction, Multiple Data*

MISD *Multiple Instruction, Single Data*

MWIPS *Mega Whetstone instructions per second*

OpenACC *Open Accelerators*

OpenCL *Open Computing Language*

OpenMP *Open Multi-Processing*

RAM *Random Access Memory*

SHOC *Scalable HeterOgeneous Computing*

SIMD *Single Instruction, Multiple Data*

SISD *Single Instruction, Single Data*

TB *Terabyte*

ws *Work-stealing*

SUMÁRIO

1 INTRODUÇÃO

Apesar dos avanços na capacidade computacional, há uma crescente demanda por poder de processamento, principalmente por aplicações que manipulam uma grande quantidade de dados e/ou executam um elevado número de operações. Essas aplicações abrangem diversas áreas do conhecimento, tais como engenharias, química, matemática, física, bioinformática, entre outras (FOSTER, 1995; XU *et al.*, 2007; STERLING; BRODOWICZ; ANDERSON, 2017). Formas de diminuir o tempo de execução dessas aplicações são amplamente discutidas, sendo a programação paralela uma das mais utilizadas (KIRK; HWU, 2013). Essa baseia-se na divisão das tarefas entre os núcleos de processamento disponíveis, que trabalham em conjunto, para reduzir o tempo total de execução da aplicação (STALLINGS, 2010).

Dentre as arquiteturas paralelas existentes destacam-se os computadores com múltiplos núcleos, visto que essa é uma das arquiteturas mais comuns nos computadores atuais. Essa arquitetura se caracteriza pela presença de dois ou mais núcleos de processamento, o que possibilita a execução de múltiplas instruções simultaneamente (DABAH *et al.*, 2018). Além disso, outra arquitetura que vem recebendo destaque nos últimos anos são as GPUs (*Graphics Processing Unit*). De fato, essas se fazem cada mais presentes nos computadores atuais devido à popularização dos jogos eletrônicos e das aplicações 3D.

No entanto, para explorar de forma eficiente os recursos de *hardware* em arquiteturas híbridas é necessário dividir de forma adequada o processamento entre os núcleos da CPU (*Central Processing Unit*) e da GPU (SONG *et al.*, 2020). Existem ferramentas que se propõem a facilitar o desenvolvimento de aplicações para essas arquiteturas híbridas, simplificando por exemplo, a transferência de dados entre a memória dos dispositivos. Entre essas destacam-se o StarPU (AUGONNET; THIBAULT; NAMYST, 2010), o OpenACC (OpenACC.org, 2022a) e o OmpSs (DURAN; CORBALAN; MARTORELL, 2011).

Neste trabalho foi realizado um estudo sobre ferramentas para paralelização de aplicações em arquiteturas híbridas. Com base neste estudo, foi selecionada a ferramenta utilizada para a avaliação. Além disso, definiu-se um conjunto de aplicações para os testes. A definição das aplicações foi realizada a partir de uma análise de ferramentas de *benchmarks* para CPUs e GPUs, optando-se pela utilização das aplicações mais comuns na avaliação de desempenho nestes tipos de arquiteturas.

A análise de desempenho das aplicações foi realizada a partir de uma comparação do tempo total de execução. De fato, os tempos de execução obtidos com a ferramenta de programação híbrida foram comparados aos tempos obtidos pela: aplicação sequencial (utilizando um único núcleo de processamento); implementações paralelizadas para CPUs utilizando OpenMP; implementações desenvolvidas para GPUs utilizando CUDA.

1.1 OBJETIVOS

O principal objetivo deste trabalho consistiu na avaliação de ferramentas de programação paralela para arquiteturas híbridas com CPUs e GPUs. Para que esse objetivo fosse atingido os seguintes objetivos específicos foram realizados:

- Definir a ferramenta de programação hibrída a ser utilizada;
- Definir um conjunto de aplicações a serem utilizadas para realização dos testes;
- Paralelizar as aplicações escolhidas;
- Realizar testes e avaliar os resultados obtidos.

1.2 ESTRUTURA DO TRABALHO

O trabalho está estruturado da seguinte forma:

- No Capítulo 1 foi realizada uma breve introdução do trabalho, apresentando sua motivação e objetivos;
- No Capítulo 2 é realizada uma breve apresentação sobre arquiteturas paralelas e sobre ferramentas de programação para arquiteturas híbridas com CPUs e GPUs. Neste capítulo, é apresentada ainda a ferramenta escolhida para o desenvolvimento deste trabalho.
- No Capítulo 3 é realizada uma apresentação sobre *benchmarks* para CPUs e GPUs. Além disso, são descritas as aplicações que foram paralelizadas neste trabalho.
- No Capítulo 4 são descritas as implementações realizadas e suas respectivas paralelizações.
- No Capítulo 5 são apresentados os testes e resultados obtidos.
- No Capítulo 6 são apresentadas as considerações finais e sugestões de trabalhos futuros.

2 PROGRAMAÇÃO EM ARQUITETURAS HÍBRIDAS

A previsão feita por Gordon Earle Moore, um dos cofundadores da Intel, defendia que a densidade de transistores em uma CPU dobraria a cada 24 meses. Em decorrência disso, o poder computacional também dobraria neste mesmo período de tempo (MOORE, 1965). Essa previsão ficou conhecida como a Lei de Moore e nos últimos anos tem sido desafiada pela dificuldade crescente no processo de fabricação dos processadores. Em 2019, Jensen Huang (CEO da Nvidia) declarou que não seria possível manter esse ritmo de crescimento num futuro próximo (WITKOWSKI, 2022). Essa dificuldade é oriunda principalmente da complexidade em se desenvolver CPUs cada vez mais rápidas, devido principalmente à limitação de fabricação, materiais e *design* (MOHAMED, 2020).

Uma das alternativas para contornar esses problema é a utilização de arquiteturas paralelas, nas quais a carga de trabalho é dividida entre múltiplos processadores, que executam as tarefas de forma simultânea, reduzindo o tempo total de execução da aplicação (STALLINGS, 2010). Existem diferentes tipos de arquiteturas paralelas, que podem ser classificadas de acordo com a taxonomia de Flynn (FLYNN, 1972).

2.1 TAXONOMIA DE FLYNN

A taxonomia de Flynn foi criada em 1966 por Michael J. Flynn e divide as arquiteturas a partir do número de fluxos de instruções e de dados (FLYNN, 1972). De acordo com a taxonomia de Flynn os sistemas computacionais podem ser divididos em quatro categorias:

- *SISD (Single Instruction, Single Data)*: são arquiteturas que possuem um único núcleo de processamento, executando um único fluxo de instruções sobre um único fluxo de dados.
- *SIMD (Single Instruction, Multiple Data)*: possuem um único fluxo de instruções que é executado sobre múltiplos fluxos de dados. Os principais representantes dessa categoria são os processadores vetoriais e matriciais, sendo as GPUs um exemplo dessa categoria.
- *MISD (Multiple Instruction, Single Data)*: possuem múltiplos fluxos de instruções que são executados sobre um único fluxo de dados. Não existem representantes para essa categoria.
- *MIMD (Multiple Instruction, Multiple Data)*: possuem vários núcleos de processamento, executando simultaneamente múltiplas instruções sobre dados diferentes. Os principais representantes dessa categoria são os multiprocessadores, computadores com processadores *multicore* e os multicomputadores.

Atualmente, a maioria dos computadores possuem processadores com múltiplos núcleos de processamento (arquiteturas *multicore*). Além disso, devido à popularização dos jogos eletrônicos e do uso de aplicações 3D, tais computadores possuem ainda uma GPU para auxiliar no processamento gráfico. Portanto, nos computadores atuais é comum a presença simultânea das arquiteturas do tipo MIMD e SIMD.

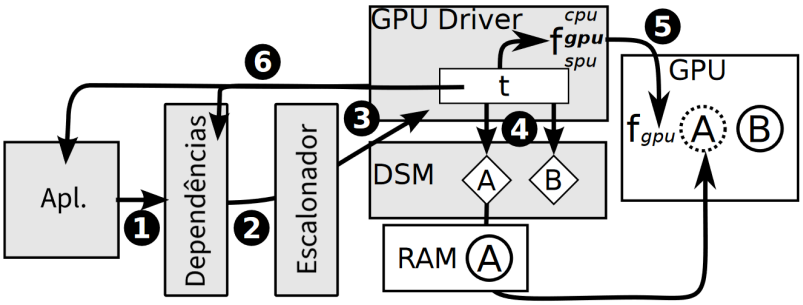
Os sistemas que utilizam mais de uma arquitetura apresentam complicações no que se refere ao desenvolvimento de aplicações paralelas, sendo necessário o uso de diferentes ferramentas, uma para cada tipo de arquitetura. Por exemplo, para a exploração do paralelismo em processadores *multicore*, frequentemente é utilizado o padrão OpenMP (CHAPMAN; JOST; PAS, 2007). Já para a programação de GPUs da Nvidia é utilizada a biblioteca CUDA (NVIDIA Corporation, 2010). Além disso, quando há a utilização de GPUs, torna-se necessário que a transferência de dados entre a memória principal e a memória da GPU seja realizada explicitamente pelo programador. Por fim, para a compilação de programas para GPUs, geralmente são utilizados compiladores específicos, o que dificulta a transição das tarefas entre a CPU e a GPU (AUGONNET; THIBAULT; NAMYST, 2010). Devido a essas dificuldades, foram criadas ferramentas com o intuito de facilitar a programação para essas arquiteturas híbridas. Algumas dessas ferramentas são o StarPU, OpenACC e OmpSs, descritas nas próximas seções.

2.2 STARPU

O StarPU é um sistema de execução que provê suporte para arquiteturas *multicore* com GPUs. Esse foi desenvolvido por Cédric Augonnet em sua Tese de Doutorado na Universidade de Bordeaux (AUGONNET, 2011). O StarPU, além de prover uma visão unificada dos recursos computacionais, também se encarrega da execução das tarefas nos diferentes tipos de arquiteturas (AUGONNET *et al.*, 2011). O StarPU possui dois princípios básicos de funcionamento. O primeiro deles é que as tarefas devem apresentar diferentes implementações, sendo uma para cada tipo de arquitetura. O segundo consiste em um recurso que facilita a transferência de dados entre as unidades de processamento (AUGONNET; THIBAULT; NAMYST, 2010).

Na Figura 1 tem-se o fluxo de execução de uma aplicação no StarPU. Esse baseia-se em um escalonador dinâmico que é responsável por distribuir as tarefas às unidades de processamento adequadas. Para a transferência de dados entre as diferentes unidades de processamento, o StarPU utiliza-se de um mecanismo chamado de DSM (*Memory Management*). Esse permite que múltiplas cópias dos dados sejam armazenados em diferentes unidades de processamento, desde que não sejam modificados. A transferência dos dados é feita de forma automática no momento que as tarefas são alocadas para a execução (AUGONNET; THIBAULT; NAMYST, 2010).

Figura 1 – StarPU: Execução de uma tarefa



Fonte: Adaptado de (AUGONNET; THIBAULT; NAMYST, 2010)

2.2.1 Desenvolvimento em StarPU

Como pode ser observado no Algoritmo 1, um programa desenvolvido em StarPU deve incluir a biblioteca `starpu.h` (linha 1). Além disso, é utilizada a função `starpu_init()` (linha 4), responsável por inicializar o ambiente StarPU. Através dessa é possível definir as configurações iniciais de execução, como por exemplo, a política de agendamento e o número de unidades de processamento que serão utilizados. Caso o valor utilizado seja igual a `NULL`, será utilizado um número de unidades definido automaticamente pelo StarPU e a política de escalonamento padrão (Seção 2.2.3). Para encerrar o StarPU, deve ser utilizada a função `starpu_shutdown()` (linha 6).

Algoritmo 1 – StarPU: Estrutura básica de um programa

```
1 #include <starpu.h>
2
3 int main(){
4     starpu_init(NULL);
5     //Tarefas a serem executadas
6     starpu_shutdown();
7     return 0;
8 }
```

Fonte: O Autor (2024)

2.2.2 Implementação das tarefas (Codelet)

Para a utilização do StarPU torna-se necessário que o usuário tenha conhecimento específico sobre cada uma das arquiteturas. Para tanto, é desenvolvido um *kernel* computacional para cada uma das arquiteturas disponíveis. Isso é feito através de um descritor de funções, chamado *codelet*. Um *codelet* é uma estrutura (`struct starpu_codelet`) que contém as diferentes implementações de um *kernel*, além dos dados sobre os quais o *kernel* será executado.

No Algoritmo 2 tem-se um exemplo de um *codelet* que soma uma constante a um vetor. Inicialmente, é necessário informar em que arquitetura o *codelet* será executado. No caso do exemplo, ele será executado exclusivamente em uma CPU (linha 13). Na linha 14 é informada a função que será executada em cada uma das arquiteturas, onde é definida uma única função, visto que o *codelet* será executado somente em CPUs. O campo `nbuffers` (linha 15) representa quantos *buffers* de dados serão utilizados no *codelet*. Está sendo atribuído o valor 1 pois será utilizado apenas um vetor. O campo `modes` (linha 11) indica os modos de acesso aos dados, podendo assumir os valores: `STARPU_R` (apenas leitura), `STARPU_W` (apenas escrita) ou `STARPU_RW` (leitura e escrita). No exemplo é utilizado o valor `STARPU_RW`, pois o vetor será utilizado em operações de leitura e escrita. A função `STARPU_VECTOR_GET_NX` tem como retorno o número de elementos do vetor e `STARPU_VECTOR_GET_PTR` retorna o endereço inicial do vetor (*pointer*).

Algoritmo 2 – StarPU: *Codelet*

```
1  void cpu_func(void *buffers[], void *cl_arg)
2  {
3      unsigned i;
4      float *factor = cl_arg;
5      unsigned n = STARPU_VECTOR_GET_NX(buffers[0]);
6      float *val = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
7      for (i = 0; i < n; i++)
8          val[i] *= *factor;
9  }
10
11 struct starpu_codelet cl =
12 {
13     .where = STARPU_CPU,
14     .cpu_funcs = { cpu_func },
15     .nbuffers = 1,
16     .modes = {STARPU_RW}
17 };
```

Fonte: Adaptado de (TEAM, 2022)

No Algoritmo 3 tem-se o código da aplicação principal, onde na linha 8 é chamada a função de inicialização do StarPU (`starpu_init`). Na linha 10 tem-se a chamada da função `starpu_vector_data_register`, responsável por registrar os dados que serão utilizados nas operações paralelas. Essa função possui como parâmetros o vetor que será utilizado e o local onde o mesmo está armazendo. Neste caso, o valor `STARPU_MAIN_RAM` indica que o vetor está originalmente armazenado na memória principal. Na linha 14 é criada a tarefa, que posteriormente é submetida na linha 21. O valor 1 (linha 15) indica que a tarefa será síncrona (qualquer valor maior que 0 indica que a tarefa será síncrona). Na linha 23 o vetor é liberado e, por fim, na linha 25 o ambiente StarPU é finalizado através da função `starpu_shutdown`.

Algoritmo 3 – StarPU: Programa Principal

```
1  int main(int argc, char **argv)
2  {
3      float vector[NX];
4      unsigned i;
5      for (i = 0; i < NX; i++)
6          vector[i] = 1.0f;
7
8      starpu_init(NULL);
9      starpu_data_handle_t vector_handle;
10     starpu_vector_data_register(&vector_handle, STARPU_MAIN_RAM,
11     (uintptr_t)vector, NX, sizeof(vector[0]));
12     float factor = 3.14;
13
14     struct starpu_task *task = starpu_task_create();
15     task->synchronous = 1;
16     task->cl = &cl;
17     task->handles[0] = vector_handle
18     task->cl_arg = &factor;
19     task->cl_arg_size = sizeof(factor);
20
21     starpu_task_submit(task);
22
23     starpu_data_unregister(vector_handle);
24
25     starpu_shutdown();
26     return 0;
27 }
```

Fonte: (TEAM, 2022)

Como pode ser observado no Algoritmo 4, é possivel também fornecer uma implementação específica para GPUs. Na linha 3, é definido um *kernel* CUDA (`vector_mult_cuda`). O *kernel* deve ser compilado utilizando um compilador com suporte à CUDA, como por exemplo o compilador *nvcc* da Nvidia (NVIDIA Corporation, 2024a). Na linha 21, é feita a criação do *kernel*, que possui como argumentos o número de blocos a serem criados (`nblocks`), o número de *threads* por bloco (`threads_per_block`), o tamanho do vetor (`n`), o vetor (`val`) e a constante (`fact`).

Algoritmo 4 – StarPU: CUDA

```
1  #include <starpu.h>
2
3  static __global__ void vector_mult_cuda(unsigned n, float *val,
4                                          float factor)
5  {
6          unsigned i =  blockIdx.x*blockDim.x + threadIdx.x;
7
8          if (i < n)
9                  val[i] *= factor;
10 }
11
12 extern "C" void scal_cuda_func(void *buffers[], void *_args)
13 {
14         float *factor = (float *)_args;
15         unsigned n = STARPU_VECTOR_GET_NX(buffers[0])
16         float *val = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
17
18         unsigned threads_per_block = 64;
19         unsigned nblocks = (n + threads_per_block-1) / threads_per_block;
20
21         vector_mult_cuda<<<nblocks,threads_per_block,0,
22                            starpu_cuda_get_local_stream()>>>(n, val,
23                                                              *factor);
24         cudaError_t status = cudaGetLastError();
25         if (status != cudaSuccess) STARPU_CUDA_REPORT_ERROR(status);
26 }
```

Fonte: (TEAM, 2022)

No Algoritmo 5 tem-se o *codelet* para a execução do *kernel* em uma GPU. Na linha 5 é adicionada a implementação CUDA, utilizando o campo `cuda_funcs`. A execução do *kernel* em uma GPU é definida através do campo `where`.

Algoritmo 5 – StarPU: *Codelet* com CUDA

```
1
2  struct starpu_codelet cl =
3  {
4      .where = STARPU_CPU | STARPU_CUDA,
5      .cpu_funcs = { cpu_func },
6      .cuda_funcs = { scal_cuda_func },
7      .nbuffers = 1,
8      .modes = {STARPU_RW}
9  };
```

Fonte: Adaptado de (TEAM, 2022)

2.2.3 Políticas de Agendamento das Tarefas

As tarefas podem ser executadas utilizando diferentes politicas de agendamento, definidas através da função `starpu_init()`, ou ainda, através de uma variável de ambiente (`STARPU_SCHED`). Cada tarefa a ser executada é adicionada em uma fila, sendo que a escolha da tarefa a ser executada depende da política de agendamento. A utilização das políticas podem impactar no desempenho da aplicação, distribuindo as tarefas de forma mais eficiente entre os recursos de processamento (chamados de *workers*), ou ainda, diminuindo a necessidade de transferência dos dados. A política de agendamento padrão é a *lws*, que é descrita a seguir junto com as demais políticas disponíveis:

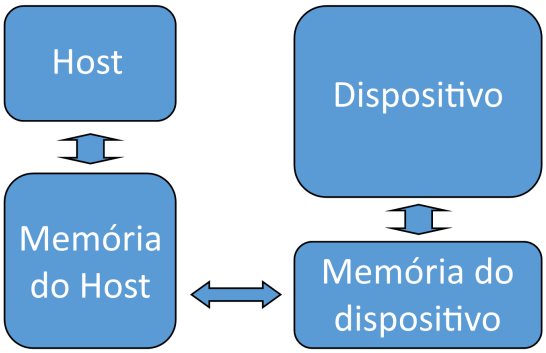
- *lws*: cada *worker* possui sua própria fila de tarefas. Porém, no momento que um *worker* fica ocioso, esse busca uma tarefa de um *worker* vizinho.
- *ws*: cada *worker* possui sua própria fila de tarefas, porém quando um *worker* fica ocioso, esse busca uma tarefa do *worker* com o maior número de tarefas na fila.
- *prio*: utiliza uma fila de tarefas centralizada, sendo que as tarefas são organizadas de acordo com a prioridade. Nesse caso, o escalonador atribui as tarefas levando em conta, primeiramente, aquelas de maior prioridade.
- *heteroprio*: para cada tarefa é definida uma prioridade e o tipo de unidade de processamento em que essa deve ser executada. Os *workers* retiram a tarefa de maior prioridade para o seu tipo de dispositivo.
- *eager*: utiliza uma única fila de tarefas, onde cada *worker* obtém as tarefas a serem executadas.
- *random*: o escalonador utilizada uma fila para cada *worker*, distribuindo as tarefas aleatoriamente, considerando o poder de processamento de cada *worker*.

2.3 OPENACC

OpenACC foi desenvolvido em conjunto pelas empresas Cray, CAPS, Nvidia e PGI para simplificar a programação paralela de sistemas heterogêneos com CPUs e GPUs. Esse possibilita, por meio de diretivas de compilação, informar as regiões que podem ser paralelizadas em GPUs ou CPUs. O OpenACC apresenta implementações para as linguagens de programação C, C++ e Fortran (OpenACC.org, 2022b).

O modelo de execução do OpenACC se baseia na execução da aplicação em um *host*, que pode ter uma GPU e/ou CPU com múltiplos núcleos como aceleradores. A *thread* do *host* é responsável pela distribuição das regiões de computação intensiva entre os aceleradores. A transferência dos dados entre a memória principal e a GPU pode ser realizada de acordo com diretivas definidas pelo programador (OpenACC.org, 2022a). A Figura 2 mostra uma representação do modelo utilizado pelo OpenACC.

Figura 2 – OpenACC: Modelo de aceleração



Fonte: Adaptado e traduzido de (OpenACC.org, 2022b)

Para a identificação das regiões que serão executadas nos aceleradores são utilizadas diretivas de compilação. Na linguagem C ou C++, essas regiões são especificadas através da diretiva `pragma`, disponibilizada pela própria linguagem. A diretiva `kernels` identifica um trecho de código que pode ser paralelizado, sendo que o compilador será o responsável pela paralelização e pela definição da estratégia a ser utilizada. A região não será paralelizada caso o compilador não consiga identificar uma independência entre os dados. Por isso, o uso dessa diretiva é recomendada somente para iniciantes. A síntaxe da diretiva `kernels` pode ser observado no Algoritmo 6. As chaves indicam que os dois laços de repetição devem ser paralelizados.

Algoritmo 6 – OpenACC: Diretiva *kernels*

```
1  #pragma acc kernels
2  {
3      for (i=0; i<N; i++)
4      {
5          y[i] = 0.0f;
6          x[i] = (float)(i+1);
7      }
8
9      for (i=0; i<N; i++)
10     {
11         y[i] = 2.0f * x[i] + y[i];
12     }
13 }
```

Fonte: (OpenACC.org, 2022b)

A diretiva `parallel` identifica uma região de código que será paralelizada. Quando utilizada em conjunto com a diretiva `loop`, o compilador irá gerar uma versão paralela do laço para um acelerador. No Algoritmo 7, tem-se um exemplo de utlização da diretiva `parallel` em conjunto com a diretiva `loop`. Diferentemente, da diretiva `kernels`, cada região deve ser identificada com uma diretiva, cabendo ao programador identificar o paralelismo. Assim, a principal diferença entre as diretivas `kernels` e `parallel loop` é que no caso da diretiva `kernels` o compilador se encarrega de analisar se é viável e seguro paralelizar o código. Ou seja, no caso da diretiva `kernels`, o código corre o risco de não ser paralelizado, mesmo que o programador identifique que a paralelização é segura, o que pode resultar em perda de desempenho (OpenACC.org, 2022b).

Algoritmo 7 – OpenACC: Diretiva *Parallel*

```
1  #pragma acc parallel loop
2  for (i=0; i<N; i++)
3  {
4      y[i] = 0.0f;
5      x[i] = (float)(i+1);
6  }
7  #pragma acc parallel loop
8  for (i=0; i<N; i++)
9  {
10     y[i] = 2.0f * x[i] + y[i];
11 }
```

Fonte: (OpenACC.org, 2022b)

Existem ainda diretivas que podem ser utilizadas para a definição dos aceleradores a serem utilizados. A cláusula `self` especifica que a região será executada no dispositivo *host*. Isso permite que sejam utilizados tanto o acelerador quanto o *host* para paralelizar as aplicações. Como pode ser visto no Algoritmo 8, a primeira iteração seria paralelizada no próprio *host* e a segunda iteração seria executada em um acelerador, definido na compilação do programa.

Algoritmo 8 – OpenACC: Cláusula *self*

```
1  #pragma acc parallel loop self
2  for (i=0; i<N; i++){
3      y[i] = 0.0f;
4      x[i] = (float)(i+1);
5  }
6  #pragma acc parallel loop
7  for (i=0; i<N; i++){
8      y[i] = 2.0f * x[i] + y[i];
9  }
```

Fonte: O autor(2024)

Além disso, a função `acc_set_device_type()` pode ser utilizada para definir o acelerador a ser utilizado. Como pode ser observado no Algoritmo 9, na linha 1 é definido que será utilizado uma GPU da Nvidia. Na linha 9, é definido que a região será paralelizada no *host*.

Algoritmo 9 – OpenACC: Definição do Dispositivo

```
1  acc_set_device_type(acc_device_nvidia);
2  #pragma acc parallel loop
3  for (i=0; i<N; i++)
4  {
5      y[i] = 0.0f;
6      x[i] = (float)(i+1);
7  }
8
9  acc_set_device_type(acc_device_host);
10 #pragma acc parallel loop
11 for (i=0; i<N; i++)
12 {
13     y[i] = 2.0f * x[i] + y[i];
14 }
```

Fonte: O autor(2024)

Para um melhor desempenho é importante se atentar ao uso da localidade dos dados, visto que transferências desnecessárias podem causar um perda significativa de desempenho. A diretiva `data` busca facilitar o compartilhamento de dados entre múltiplas regiões paralelas. Além disso, o OpenACC disponibiliza também cláusulas que possibilitam definir quando os dados serão criados e copiados para um dispositivo (OpenACC.org, 2022b). Essas cláusulas podem ser utilizadas em conjunto com as diretivas `data` e as diretivas de computação (`parallel` ou `kernels`):

- `copy`: cria um espaço na memória do dispositivo, copiando os dados no início da região paralela. Ao final da região, as variáveis são copiadas para o *host* e o espaço de memória é liberado.
- `copyin`: cria um espaço na memória do dispositivo e os dados são copiados no início da região paralela. Ao final o espaço de memória é liberado, porém não é realizada uma cópia dos dados para o *host*.
- `copyout`: cria um espaço na memória do dispositivo, porém esse espaço não é inicializado. Ao final os dados são copiados para o *host* e o espaço de memória é liberado.
- `create`: cria um espaço para as variáveis e o libera ao final, porém não são realizadas cópias.
- `present`: indica que os dados já estão presentes no dispositivo, não sendo necessária nenhuma cópia.

- `deviceptr`: indica que as variáveis utilizam a memória do dispositivo, que não é administrada pelo OpenACC. Essa diretiva é utilizada quando o OpenACC é utilizado em conjunto com outras ferramentas.

No Algoritmo 10 é possível ver um exemplo de uso da diretiva `data` e das cláusulas de manipulação de dados. Na linha 1, a diretiva `data` inclui duas regiões paralelas (linha 3 até linha 10), indicando que os vetores `x` e `y` podem ser utilizados entre as duas regiões, não necessitando de cópias. Nesse caso o vetor `x` é criado no dispositivo através da cláusula `create`, não sendo copiado para o *host* ao final da região paralela. Já o vetor `y` será criado no dispositivo e os dados gerados serão copiados para o *host* no final da região paralela.

Algoritmo 10 – OpenACC: Manipulação de Dados

```
1  #pragma acc data create(x[0:N]) copyout(y[0:N])
2  {
3      #pragma acc parallel loop
4      for (i=0; i<N; i++)
5      {
6          y[i] = 0.0f;
7          x[i] = (float)(i+1);
8      }
9
10     #pragma acc parallel loop
11     for (i=0; i<N; i++)
12     {
13         y[i] = 2.0f * x[i] + y[i];
14     }
15 }
```

Fonte: (OpenACC.org, 2022b)

2.4 OMPSS

O OmpSs é um modelo de programação desenvolvido pelo *Barcelona Supercomputing Center* (BSC) para o desenvolvimento de aplicações em arquiteturas heterôgeneas com CPUs e GPUs. A forma de funcionamento é similar ao padrão OpenACC, também sendo composta por um conjunto de diretivas e funções (Barcelona Supercomputing Center, 2019).

O OmpSs utiliza um modelo de execução no qual o paralelismo é implícito desde o início do programa. Ou seja, ao iniciar uma aplicação OmpSs é criado um conjunto de *workers*, que executarão as tarefas existentes em uma fila. É importante destacar que o programa principal também é uma tarefa que será adicionada à lista de tarefas (Barcelona Supercomputing Center, 2019). Ao encontrar a declaração de uma tarefa, essa é instanciada e incluída na fila. As tarefas poderão ser executadas imediamente, considerando as restrições de escalonamento, ou de acordo da disponibilidade dos *workers* (Barcelona Supercomputing Center, 2019).

Uma tarefa é especificada pela diretiva `task`, conforme pode ser observado no Algoritmo 11. Essa diretiva indica que o código será inicializado como uma tarefa. Por exemplo, na chamada da função `do_computation` (linha 11) será criada uma nova tarefa (diretiva `task` na linha 10). Já a execução da função `do_computation` (linha 8), será realizada de forma convencional, visto que essa não está atrelada a nenhuma diretiva `task`. Além disso, é possível utilizar a diretiva `task` na declaração de funções, fazendo com que a cada chamada dessa função uma nova tarefa seja criada (linhas 3 e 4).

Algoritmo 11 – OmpSs: Declaração de *Tasks*

```
1  void do_computation(float a);
2
3  #pragma omp task
4  void do_computation_task(float a);
5
6  float x = 0.0;
7  int main() {
8      do_computation(x);
9
10     #pragma omp task
11     do_computation(x);
12
13     do_computation_task(x);
14     return 0;
15 }
```

Fonte: Adaptado de (Barcelona Supercomputing Center, 2019)

No início de uma tarefa é verificado se essa tarefa tem dependências de dados de tarefas anteriores. Caso seja encontrada uma dependência, a tarefa só irá ser agendada para execução quando as tarefas prévias forem concluídas. Além disso, o OmpSs possibilita também especificar pontos de sincronização entre as tarefas por meio da diretiva `taskwait`. Essa diretiva pode ser utilizada em conjunto com as diretivas de dependências de dados. Para a identificação de uma dependência de dados entre as tarefas podem ser utilizadas as cláusulas:

- `in`: se a tarefa possui um valor de entrada identificado com a cláusula `in`, essa tarefa não será executada enquanto as tarefas previamente criadas com as cláusulas `out`, `inout`, `commutative` ou `concurrent`, aplicadas a esse mesmo valor, não forem concluídas.
- `out`: se a tarefa possui um valor de saída identificado com a cláusula `out`, essa tarefa não será executada enquanto as tarefas previamente criadas com as cláusulas `in`, `out`, `inout`, `concurrent` ou `commutative` não forem concluídas.
- `inout`: se uma tarefa possui uma cláusula `inout`, aplica-se um tratamento idêntico, considerando-se o valor como se estivesse simultaneamente associado às cláusulas `in` e `out`. Ou seja, as semânticas das cláusulas `in` e `out` são aplicadas.

- `concurrent`: é uma versão alternativa do `inout`, sendo dependente das cláusulas `in`, `out`, `inout`, `commutative`, porém as dependências de outras cláusulas do tipo `concurrent` não são consideradas, ou seja, essas tarefas podem ser executadas concorrentemente.

- `commutative`: a tarefa é adicionada a um conjunto de tarefas comutativas após tarefas anteriores que contenham uma cláusula `in`, `out`, `inout`, `concurrent` para esse mesmo valor tenham sido concluídas. Após, qualquer tarefa desse conjunto de tarefas comutativas pode ser escolhida para a execução. No entanto, essas tarefas só podem ser executada uma por vez.

No Algoritmo 12 são apresentados alguns exemplos de utilização das cláusulas de dependências de dados. Nas linhas 3 e 4, tem-se que a variável `x` tem uma dependência de entrada e saída (`inout`). Como não existe nenhuma tarefa prévia com dependência de `x` a tarefa não terá dependências para ser executada. Nas linhas 6 e 7 tem-se que a tarefa especificada possui uma dependência de entrada para a variável `x` e de entrada e saída para a variável `y`. Ou seja, essa tarefa só será executada após a conclusão da primeira tarefa devido à presença da cláusula `out` para `x`. O uso da diretiva `taskwait` pode ser visualizado na linha 9, onde pode ser observada uma dependência de entrada para a variável `x`. Neste caso, a execução dessa tarefa só será liberada após a conclusão da primeira tarefa, possibilitando a execução simultânea desta tarefa e da segunda tarefa. Por fim, tem-se um último ponto de sincronização na linha 12, aguardando a conclusão de todas as tarefas anteriores.

Algoritmo 12 – OmpSs: Definição de Dependências

```
1  int main() {
2      int x = 0, y = 2;
3      #pragma omp task inout(x)
4      x++;
5
6      #pragma omp task in(x) inout(y)
7      y -= x;
8
9      #pragma omp taskwait in(x)
10     assert(x == 1);
11
12     #pragma omp taskwait
13     assert(x == y);
14 }
```

Fonte: (Barcelona Supercomputing Center, 2019)

Os dispositivos possuem memórias próprias. Desta forma, é necessário garantir que os dados utilizados pelas tarefas estão disponíveis nas memórias dos diferentes dispositivos. Para tanto, o OmpSs possui um único espaço de endereçamento, com acesso de todos os dispositivos. O gerenciamento desse espaço é feito através das diretivas:

- `copy_in`: os dados especificados são disponibilizados no espaço de endereçamento comum. Neste caso, os dados serão utilizados apenas como leitura.
- `copy_out`: os dados serão gerados no espaço de endereçamento comum.
- `copy_inout`: os dados são disponibilizados no espaço de endereçamento comum. Além disso, os dados poderão ser atualizados.
- `copy_deps`: utiliza as cláusulas de dependência de dados (`in, out, inout`) em conjunto com as cláusulas `copy_`.

No Algoritmo 13 tem-se um exemplo da utilização da cláusula `copy_inout`. Neste caso, a tarefa só vai ser executada quando os dados estiverem disponíveis no espaço de endereçamento comum. A diretiva `taskwait` (linha 7) corresponde a um ponto de sincronização, que além de garantir a execução de tarefas prévias, também sincroniza os dados das tarefas em espaços memória diferentes. Assim, a execução da tarefa principal só irá continuar quando a tarefa definida na linha 5 terminar e os dados forem sincronizados.

Algoritmo 13 – OmpSs: Movimentação de Dados

```
1  float y[128];
2  int main () {
3      #pragma omp target device(cuda)
4      #pragma omp task copy_inout(y)
5      do_computation_GPU(y);
6
7      #pragma omp taskwait
8
9      float value1 = y[64];
10     return 0;
11 }
```

Fonte: Adaptado de (Barcelona Supercomputing Center, 2019)

A especificação do dispositivo onde a tarefa será executada pode ser realizada através da diretiva `target`. Ela pode ser utilizada tanto na declaração de tarefas como na implementação de funções que serão executadas como tarefas. No Algoritmo 14 é possível observar a utilização da diretiva `target`. Na linha 2 é indicado que a tarefa será executada em uma CPU (cláusula `device(smp)`). Já na linha 6 é especificado que a tarefa será executada em uma GPU CUDA (cláusula `device(cuda)`).

Algoritmo 14 – OmpSs: Diretiva *Target*

```
1  int main () {
2      #pragma omp target device(smp)
3      #pragma omp task inout(x) copy_deps
4      do_computation_CPU(x);
5
6      #pragma omp target device(cuda)
7      #pragma omp task inout(y) copy_deps
8      do_computation_GPU(y);
9
10     #pragma omp taskwait
11     return 0;
12 }
```

Fonte: (Barcelona Supercomputing Center, 2019)

2.5 DEFINIÇÃO DA FERRAMENTA DE PROGRAMAÇÃO HÍBRIDA

Para o desenvolvimento deste trabalho optou-se pela utilização do padrão OpenACC, pois esse é uma ferramenta mais consolidada. De fato, o desenvolvimento dessa ferramenta teve apoio de grandes empresas, como por exemplo, a Nvidia, AMD e HP, entre outras. Optou-se por não utilizar o OmpSs, pois essa ferramenta apresentou sua última atualização no ano de 2019, além de apresentar pouca documentação. A falta de suporte é um critério importante, pois a ausência de atualizações e correções pode ser um impeditivo para o desenvolvimento do trabalho, especialmente em caso de erros. A utilização do StarPU foi descartada pois neste caso é necessário desenvolver *kernels* específicos para cada uma das arquiteturas. Neste caso, o programador necessita conhecer as particularidades de programação em cada uma das arquiteturas. Essa abstração era um dos requisitos para a escolha da ferramenta.

Não foram encontrados na literatura trabalhos que comparem o desempenho do OpenACC com as demais ferramentas de programação híbrida. De fato, só foram encontrados trabalhos comparando o desempenho do StarPU com o OmpSs. No trabalho desenvolvido por Scogland *et al.* (2015), o StarPU e o OmpSs foram comparados na paralelização dos métodos

GEMM [1], K-means [2] e Helmholtz [3], sendo que o StarPU apresentou melhor desempenho na paralelização dos métodos GEMM e do K-means. Da mesma forma, no trabalho desenvolvido por Chen, Huo e Agrawal (2014) o StarPU também apresentou melhor desempenho na paralelização dos métodos Kmeans, Jacobi e Moldyn [4].

[1] Um *kernel* de modelagem molecular
[2] Método de agrupamento iterativo
[3] Código de diferenças discretas finitas que utiliza o método de Jacobi para a resolução da equação de Helmholtz
[4] Dinâmica molecular (Moldyn) é uma aplicação que visa simular a interação e movimento de móleculas em dado período de tempo em um espaço 3D

3 APLICAÇÕES A SEREM PARALELIZADAS

Neste capítulo, são apresentadas as aplicações que foram paralelizadas objetivando avaliar o desempenho do OpenACC. Essas aplicações foram definidas a partir da análise de *benchmarks*, visto que esses são frequentemente utilizados para comparar o desempenho de sistemas paralelos. Os *benchmarks* se baseiam em definir uma carga de trabalho que represente como o sistema irá comportar em sua utilização real (VOKOLOS; WEYUKER, 1998).

Os *benchmarks* podem ser divididos em duas categorias: sintéticos e de aplicação. Os *benchmarks* sintéticos se caracterizam por realizar funções básicas de computação, sendo criados artificialmente com o objetivo de se assemelhar a um perfil de execução (HUGUE, 1998). Esse tipo de *benchmark* frequentemente é utilizado para avaliar a capacidade de um componente em específico, como por exemplo, discos rígidos, placas de vídeos e processadores. Os *benchmarks* de aplicação se referem a aplicações reais que executam funções comuns a um determinado segmento de problemas. Normalmente, são utilizados para avaliar o desempenho geral de um sistema computacional (HUGUE, 1998).

Devido ao aumento da disponibilidade de sistemas que utilizam arquiteturas híbridas com multiprocessadores e/ou GPUs, tornou-se necessário criar *benchmarks* específicos que pudessem medir o desempenho desses sistemas. Na próxima seção, são apresentados alguns dos principais *benchmarks* existentes, além de *benchmarks* exclusivos para arquiteturas com GPUs e CPUs.

3.1 ANÁLISE DOS BENCHMARKS

Os *benchmarks* mais conhecidos e utilizados para a avaliação de sistemas com múltiplas CPUs e/ou GPUs são:

- *Whetstone*: é um *benchmark* sintético, que foi desenvolvido no *National Physical Laboratory* (NPL), no Reino Unido, por Harold Curnow e Brian Wichmann. Esse é dividido em diversos módulos, que são utilizados para diferentes avaliações, incluindo, aritmética de inteiros e de ponto flutuante, chamadas de funções, entre outros. As principais *aplicações* utilizadas são o cálculo de seno, cosseno, raiz quadrada, exponenciações, além de ramificações condicionais. Os resultados desse *benchmark* são apresentados utilizando uma unidade própria chamada de *Mega Whetstone instructions per second* (MWIPS) (WEICKER, 1990; LONGBOTTOM, 2005).

- *Linpack*: é um *benchmark* de aplicação desenvolvido em Fortran por Jack Dongarra, Jim Bunch, Cleve Moler e Gilbert W. Stewart. Esse objetiva avaliar o desempenho de sistemas considerando operações de ponto flutuante, mais especificamente, na solução de sistemas de equações lineares densos. O Linpack apresenta os resultados utilizando como unidade *Millions of Floating-Point Operations Per Second* (Mflops) (WEICKER, 1990; DONGARRA; LUSZCZEK; PETITET, 2003).

- *Dhrystone*: é um *benchmark* sintético desenvolvido por Reinhold P. Weicker. Esse objetiva avaliar o desempenho de um sistema considerando operações aritméticas com inteiros. Entre os métodos implementados estão principalmente operações de manipulação de *strings*. O desempenho do sistema é apresentado utilizando uma unidade própria chamada *Dhrystones per second* (WEICKER, 1990).

- *Sysbench*: é um *benchmark* originalmente implementado por Peter Zaitsev. Esse é uma ferramenta modular, multiplataforma e *multi-threaded*, que objetiva avaliar CPUs, sistema de memória, sistemas de E/S e servidores de bancos de dados. Seu funcionamento baseia-se na criação de *threads* que executam requisições de forma paralela. Entre os métodos implementados estão: verificação de primalidade, *loops* de incrementação, alocação de memória, leitura e escrita em memória, entre outros (KOPYTOV, 2012).

- *Scalable HeterOgeneous Computing* (SHOC): é um conjunto de *benchmarks* para arquiteturas híbridas, desenvolvido em conjunto pelo *Oak Ridge National Laboratory* (ORNL) e *Georgia Institute of Technology*. Esses *benchmarks* possuem como objetivo avaliar sistemas computacionais com GPUs e CPUs *multicores*. Os *benchmarks* possuem implementações em OpenCL e CUDA. Entre os métodos implementados estão: transformada rápida de Fourier, redução de soma, soma de prefixo, multiplicação de matrizes, método de Jacobi e ordenação por *radix sort* (DANALIS *et al.*, 2010).

- *Rodinia*: é um conjunto de *benchmarks* para arquiteturas híbridas, originalmente desenvolvido na *University of Virginia*. Os *benchmarks* possuem implementações para GPUs e CPUs *multicores*, utilizando OpenMP e CUDA. Entre as aplicações implementadas estão: *K-means*, *Needleman-Wunsch*, *Back Propagation* e busca em largura em grafos (BFS - *Breadth First Search*) (CHE *et al.*, 2009).

- *Parboil*: é um conjunto de *benchmarks* para arquiteturas híbridas, desenvolvido na *University of Illinois*. As implementações do *benchmark* foram desenvolvidas utilizando OpenCL e CUDA. O *Parboil* apresenta implementações de diversos métodos iterativos, multiplicação de matrizes, busca em largura em grafos, entre outras (STRATTON *et al.*, 2012).

3.2 DEFINIÇÃO DAS APLICAÇÕES A SEREM PARALELIZADAS

Para avaliar o OpenACC foram escolhidas 4 aplicações de diferentes domínios. Foram escolhidas aplicações que pudessem avaliar principalmente operações com ponto flutuante e inteiros, visto que são operações comuns em aplicações que demandam alto desempenho. Além disso, a escolha considerou aplicações que não possuem uma implementação muito complexa, visto que desejava-se desenvolver versões paralelas utilizando diferentes ferramentas. As aplicações escolhidas foram: multiplicação de matrizes, método de Jacobi, busca em largura em grafos e o método de ordenação *radix sort*.

Optou-se pela implementação do método de multiplicação de matrizes pois esse é tradicionalmente utilizado em *benchmarks* para ambientes de arquiteturas híbridas. De fato, esse é implementado nos *benchmarks* SHOC e *Parboil*. Além disso, ele é frequentemente utilizado na literatura para testes e comparação de ferramentas paralelas (YANG; HUANG; LIN, 2011; KHALILOV; TIMOVEEV, 2021; LI; SHIH, 2018).

Outro método que frequentemente é utilizado em *benchmarks* para ambientes híbridos é o método de Jacobi. Esse método também é implementado nos *benchmarks* SHOC e Parboil. O método de Jacobi também é muito utilizado em trabalhos que objetivam avaliar ferramentas paralelas (MEMETI *et al.*, 2017; LI; SHIH, 2018).

A terceira aplicação escolhida foi a de busca em largura em grafos. A escolha deu-se pois esse é implementado nos *benchmarks* para arquiteturas híbridas *Rodinia* e *Parboil*. Da mesma forma, esse é um método utilizado na literatura para a comparação de ferramentas paralelas (MEMETI *et al.*, 2017; LI; SHIH, 2018).

Por fim, optou-se pela escolha de uma aplicação que utiliza-se de operações com números inteiros. Neste caso, a aplicação escolhida foi o método de ordenação *radix sort*. Esse foi o método escolhido pois ele encontra-se implementado na ferramenta de *benchmark* para arquiteturas híbridas SHOC.

3.2.1 Multiplicação de matrizes

A multiplicação de matrizes é um dos problemas fundamentais da álgebra linear, tendo importância como operação por si mesma, mas também devido a problemas similares que podem ser reduzidos a ela (COHN *et al.*, 2005). O problema pode ser representado por duas matrizes A e B em que a quantidade de colunas de A é igual à quantidade de linhas de B. O produto entre A e B gerará uma matriz C que contém o número de linhas da matriz A e o número de colunas da matriz B (CORMEN *et al.*, 2022). O produto matricial de duas matrizes é definido a partir da Equação 3.1.

$$C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}. \tag{3.1}$$

No Algoritmo 15 tem-se o pseudocódigo da operação de multiplicação de matrizes. Os valores de entrada são três matrizes A, B e C. As matrizes são percorridas, realizando a operação de multiplicação de cada elemento das linhas de A com os elementos das colunas de B (linhas 4 a 11).

Algoritmo 15 – Pseudocódigo da Multiplicação de Matrizes

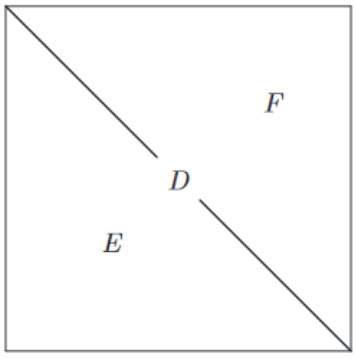
```
1  entrada: matrizes A, B e C
2  saida: matriz C com o resultado final
3
4  para i = 0 ate i < n faca:
5      para j = 0 ate j < m faca:
6          C[i][j] = 0
7          para k = 0 ate k < o faca:
8              C[i][j] += A[i][k] * B[k][j]
9          fim
10     fim
11 fim
```

Fonte: Adaptado de (CORMEN *et al.*, 2022)

3.2.2 Método de Jacobi

O método de Jacobi é um método iterativo para a solução de um sistema de equações lineares $Ax = b$, sendo A a matriz de coeficientes, x o vetor de incógnitas e b o vetor de termos independentes. A matriz de coeficientes A pode ser decomposta em $A = D + R$ (Figura 3), onde D é a diagonal da matriz de coeficientes. A matriz R é a soma da parte inferior E e da parte superior F (SAAD, 2003).

Figura 3 – Particionamento da matriz A



Fonte: Adaptado de (SAAD, 2003)

O sistema de equações lineares pode ser reescrito através da Equação 3.2.

$$Dx = b - (R)x \tag{3.2}$$

O método de Jacobi é um método iterativo que obtém o valor do vetor x (lado esquerdo da Equação 3.2) utilizando o valor de x da iteração anterior (lado direito Equação 3.2). Ou seja, a partir de suposição inicial $x^{(0)}$ é gerada uma sequência de vetores x que buscam convergir até a solução (BURDEN; FAIRES, 1997). Assim, a Equação 3.2 pode ser reescrita como

$$x^{k+1} = D^{-1}(b - Rx^k) \tag{3.3}$$

ou, equivalentemente

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[b_i - \sum_{j \neq i} a_{ij} x_j^k \right] \text{ para } i = 1, 2, \ldots, n. \tag{3.4}$$

No Algoritmo 16 tem-se o pseudocódigo do Método de Jacobi. Os valores de entrada são a matriz de coeficientes `A`, vetor de incógnitas `x`, o vetor de termos independentes `b` e o número total de equações do sistema linear. Além disso, tem-se os valores de tolerância `TOL` e o número máximo de iterações `N`. A cada iteração é calculado um novo valor para o vetor de incógnitas `x` (linha 18). Após, é calculada a norma entre o valor antigo e o novo valor do vetor `x` (linha 21). Se o valor da norma for menor do que `TOL`, o método termina. Caso, contrário uma nova iteração é realizada. Esse procedimento é realizado até a tolerância ser atingida ou o número máximo de iterações seja atingido.

Algoritmo 16 – Pseudocódigo do Método de Jacobi

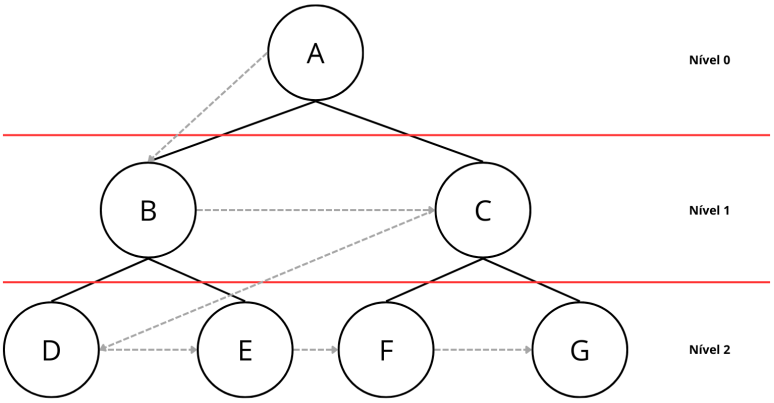
```
1  entrada: matriz A, vetor de termos independentes b,  numero de equacoes n,
2           vetor de incognitas x, tolerancia TOL, numero maximo de iteracoes N
3  saida: o resultado aproximado de x
4  k := 0
5  x := inicializa um vetor de zeros
6  novo_x := inicializa um vetor de zeros
7  enquanto k < N
8      para i = 0, ate i < n faca:
9          soma = 0
10         para j = 0, ate j < n faca:
11             se i != j faca:
12                 soma += A[i,j] * x[j]
13             fim
14         fim
15         novo_x[i] = (b[i] - soma)/A[i,i]
16     fim
17     se  abs((norma_vetor(x) - norma_vetor(novo_x) < TOL)) faca:
18         retorna novo_x
19     fim
20     x := novo_x
21     k++
22 fim
```

Fonte: O Autor (2024)

3.2.3 Busca em largura

A busca em largura (*BFS – Breadth-First Search*) é um algoritmo de busca em grafos. O problema pode ser representado por um grafo $G = (V, E)$, sendo V um conjunto de vértices e E um conjunto de arestas. O algoritmo de BFS tem como entrada um grafo $G(V, E)$ e um vértice raiz r. Esse explora as arestas de G com o intuito de descobrir os vértices que podem ser alcançados a partir de r. Uma representação visual do BFS pode ser vista na Figura 4. Considerando o vértice raiz como A, o algoritmo inicialmente irá percorrer todos os vértices vizinhos que estão no nível 1. Após, para todos os vértices do nível 1 (nesse caso os vértices B e C), são percorridos todos os vértices vizinhos do nível 2. Esse processo é realizado até que todos os níveis do grafo forem percorridos.

Figura 4 – Busca em Largura em Grafos



Fonte: O Autor (2024)

No Algoritmo 17, pode ser visto um pseudocódigo para o algoritmo de BFS. Os valores de entrada são um grafo `G(V,E)` e o vértice raiz `r`. O vetor de níveis dos vértices, `vniv`, é inicializado com todas as posições atribuídas ao valor -1, indicando a ausência de nível (linha 5). Apenas o vértice raiz é marcado como nível 0 (linha 6). Após isso, a fila de vértices do nível atual `Q` é inicializada com os vértices vizinhos de `r` (linha 7). A fila de vizinhos é percorrida, inicializando uma fila vazia (`Qprox`), que representa os vértices a serem explorados no próximo nível (linha 10). Para cada vértice de `Q` são buscados seus vizinhos e caso ainda não tiverem sido visitados, terão seu nível atualizado e adicionados na fila `Qprox` (linhas 13 a 18). Quando todos os vértices do nível forem explorados, a fila de vértices `Q` é atualizada com os vértices de `Qprox` e o nível é incrementado (linhas 20 e 21). Esse procedimento é repetido até que não existam mais vértices a serem explorados.

Algoritmo 17 – Pseudocódigo da Busca em Largura em Grafos

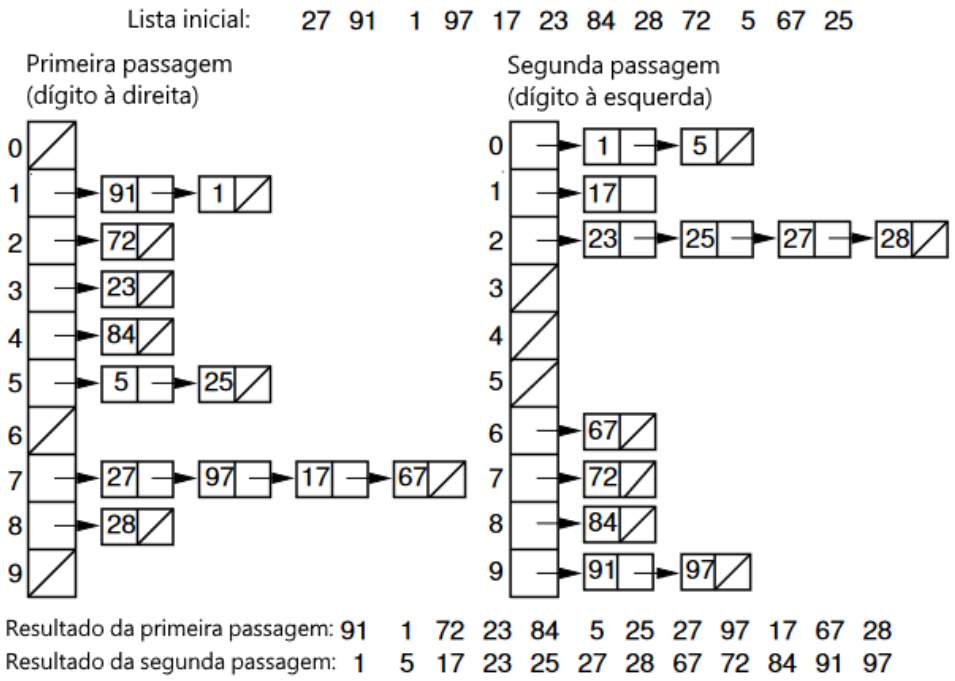
```
1  entrada: um grafo G(V,E), raiz r
2  saida:
3
4  nivel := 0
5  vniv[V] := inicializa o vetor de vertices visitados com -1
6  vniv[r] := 0
7  Q := {r} //fila contendo os vertices a serem exploradas no nivel atual
8
9  enquanto Q nao estiver vazio faca:
10     Qprox := {} /*inicializa fila com vertices
11                    a serem explorados no proximo nivel*/
12     para cada v em Q faca:
13         para cada n em E[v] faca: // vizinhos do vertice atual
14             se vniv[n] == -1 faca:
15                 vniv[n] = vniv[v] + 1
16                 Qprox := Qprox + {n}
17             fim
18         fim
19     fim
20     Q := Qprox
21     nivel++
22 fim
```

Fonte: Adaptado de (SCARPAZZA; VILLA; PETRINI, 2008)

3.2.4 Radix sort

O *radix sort* é um método de ordenação usado para classificar itens que são identificados por chaves únicas. Esse baseia-se na ordenação dos valores considerando os dígitos individualmente. Por exemplo, como pode ser observado na Figura 5, o método inicialmente ordena os valores considerando apenas os dígitos menos significativos, ou seja, os dígitos referentes à unidade. Após, o vetor resultante da primeira passagem, é ordenado considerando os dígitos das dezenas. Assim, o vetor resultado da segunda passagem é o vetor final ordenado, visto que neste exemplo os valores do vetor possuem apenas dois dígitos.

Figura 5 – Método de Ordenação *Radix Sort*



Fonte: Adaptado e traduzido de (SHAFFER, 1997)

No Algoritmo 18 tem-se o pseudocódigo do método *radix sort*. Os valores de entrada são o vetor `A`, com os números a serem ordenados, e um valor `n` que corresponde ao número de elementos de A. Primeiramente, é obtido o maior valor de A, sendo armazenado na variável `m`. A partir do valor de `m` será obtido o número máximo de dígitos dos valores existentes no vetor. Após, tem-se uma iteração partindo do dígito menos significativo até o número de dígitos de `m` (linha 5). Na linha 6 é inicializado um vetor com 10 posições (vetor `balde`), onde cada posição corresponde a um dígito específico. Já na linha 7 é criado um vetor de saída para armazenar os valores ordenados. Entre as linhas 9 e 11 são contadas as ocorrências de cada dígito. Entre as linhas 12 a 14, os valores do vetor `balde` são ajustados de forma a calcular a posição final de cada valor no vetor de saída. Por fim, o vetor de saída é montado (da linha 15 a 17) e copiado para o vetor `A` (da linha 18 a 20). Este procedimento será repetido até que todos os dígitos sejam considerados na ordenação.

Algoritmo 18 – Pseudocódigo do Método Radix Sort

```
1  entrada: vetor A, n numero de elementos de A
2
3  m = maior numero de A
4
5  para exp = 1, ate m / exp > 0, incremente exp *= 10
6      balde[10] = inicializa um vetor com zeros
7      saida[n] = vetor com o mesmo numero de elementos de A
8
9      para i = 0, i < n faca:
10         balde[(A[i] / exp) % 10]++
11     fim
12     para i = 1, i < 10 faca:
13         balde[i] += balde[i-1]
14     fim
15     para i = n - 1, i >= 0, decremente i-1 faca:
16         saida[--balde[(A[i] / exp) % 10]] = A[i]
17     fim
18     para i = 0, i < n faca:
19        A[i] = saida[i]
20     fim
21
22 fim
```

Fonte: Adaptado de (SHAFFER, 1997)

4 IMPLEMENTAÇÕES DESENVOLVIDAS

O OpenACC possui suporte às linguagens C/C++ e Fortran. Neste trabalho, as implementações foram desenvolvidas utilizando a linguagem de programação C. Para a compilação foi utilizado o compilador *nvc* da Nvidia (NVIDIA Corporation, 2024b), que possui suporte nativo para o OpenACC. O compilador *nvc* está incluído no pacote Nvidia HPC, com otimizações voltadas para as GPUs da marca. Outra opção seria o uso do compilador GCC (GOUGH; STALLMAN, 2004). Testes foram realizados com o compilador GCC, no entanto esse não oferece o mesmo nível de integração e otimização para as GPUs da Nvidia. Além disso, não foi possível configurar o compilador GCC para utilização do OpenACC simultaneamente em CPUs e GPUs.

As implementações foram paralelizadas para CPUs com múltiplos núcleos utilizando a biblioteca OpenMP (CHAPMAN; JOST; PAS, 2007). Optou-se pela utilização do OpenMP, pois esse é um padrão amplamente utilizado para a paralelização em ambientes multiprocessados. A utilização do OpenMP, assim como do OpenACC, é feita através do uso de diretivas de compilação, o que facilita a paralelização de forma incremental sem a necessidade de mudanças significativas no código fonte. Outra alternativa seria o uso da biblioteca *Pthreads* (NICHOLS; BUTTLAR; FARRELL, 1996), entretanto o uso dessa biblioteca requer um maior gerenciamento sobre as *threads* e mudanças significativas no código. Por fim, o desempenho de ambas as bibliotecas é similar (SWAHN, 2016).

A paralelização para GPUs foi realizada utilizando CUDA (FARBER, 2011), visto que esse é o ambiente comumente empregado em GPUs da Nvidia, fabricante da placa utilizada neste trabalho. Outra alternativa seria a utilização do OpenCL (MUNSHI, 2009), no entanto esse foi desenvolvido para dar suporte a diferentes tipos de dispositivos e fabricantes. Assim, o desenvolvimento é mais complexo e as implementações podem apresentar um desempenho inferior quando comparadas às implementações em CUDA. Por exemplo, no trabalho desenvolvido por Karimi, Dickson e Hamze (2010) pode ser observado que o CUDA obteve melhor desempenho quando comparado ao OpenCL.

4.1 PARALELIZAÇÃO DA MULTIPLICAÇÃO DE MATRIZES

Esta seção oferece uma breve descrição da paralelização da operação de multiplicação de matrizes, destacando as funções e diretivas utilizadas com as bibliotecas OpenMP, CUDA e OpenACC.

4.1.1 Paralelização da Multiplicação de Matrizes com OpenMP

Para a paralelização da operação de multiplicação de matrizes em computadores multiprocessados e com processadores *multicore*, foi utilizada a biblioteca OpenMP. Neste caso, foi utilizada a diretiva `parallel for`, que realiza a paralelização do laço mais externo entre as *threads* disponíveis (linha 4 do Algoritmo 19). Além disso, a cláusula `private` foi utilizada para que cada *thread* possua uma cópia das variáveis `j` e `k` de forma a evitar problemas de concorrência (PACHECO, 2011). Destaca-se que a diretiva `parallel for` priva de forma automática a variável de controle do laço mais externo (variável `i`).

Algoritmo 19 – Multiplicação de matrizes: Paralelização com OpenMP

```
1  void mat_mul(double * A, double * B, double * C, int colsB,
2               int rowCol, int rowsA){
3      int i, j, k;
4      #pragma omp parallel for private (j, k)
5      for (i = 0; i < rowsA; i++){
6          for (j = 0; j < colsB; j++){
7              C[i*colsB+j] = 0;
8              for (k = 0; k < rowCol; k++){
9                  C[i*colsB+j] += A[i*rowCol+k] * B[k*colsB+j];
10             }
11         }
12     }
13 }
```

Fonte: O autor (2024)

4.1.2 Paralelização da Multiplicação de Matrizes com CUDA

Para a paralelização para GPU foi utilizado o ambiente de programação CUDA. Conforme pode ser observado no Algoritmo 20, as matrizes são alocadas na memória da GPU através da função `cudaMalloc` e transferidas através da função `cudaMemcpy`. O parâmetro `cudaMemcpyHostToDevice` indica que a transferência é realizada da memória do *host* para a memória da GPU (*device*).

Algoritmo 20 – Multiplicação de Matrizes: Transferência das matrizes para a memória da GPU

```
1  cudaMalloc(&d_A, rowsA * rowCol * sizeof(double));
2  cudaMalloc(&d_B, rowCol * colsB * sizeof(double));
3  cudaMalloc(&d_C, rowsA * colsB * sizeof(double));
4
5  cudaMemcpy(d_A, A, rowsA * rowCol * sizeof(double), cudaMemcpyHostToDevice);
6  cudaMemcpy(d_B, B, rowCol * colsB * sizeof(double), cudaMemcpyHostToDevice);
```

Fonte: O autor (2024)

O Algoritmo 21 apresenta a implementação do *kernel* para a multiplicação de matrizes em CUDA. No *kernel*, as *threads* são organizadas em blocos bidimensionais. Os índices das linhas e das colunas são determinados pelo identificador do bloco (`blockIdx`), pela quantidade de *threads* por bloco (`blockDim`) e pelo identificador da thread (`threadIdx`). Dessa forma, cada *thread* calcula um elemento específico da matriz C.

Algoritmo 21 – Multiplicação de matrizes: *Kernel* CUDA

```
1  __global__ void matmul(double *A, double *B, double *C, int colsB,
2                         int rowCol, int rowsA){
3      int i = blockIdx.y * blockDim.y + threadIdx.y;
4      int j = blockIdx.x * blockDim.x + threadIdx.x;
5      int k;
6
7      if (i < rowsA && j < colsB){
8          C[i*colsB+j] = 0;
9          for (k = 0; k < rowCol; k++){
10             C[i*colsB+j] += A[i*rowCol+k] * B[k*colsB+j];
11         }
12     }
13 }
```

Fonte: O autor (2024)

No Algoritmo 22, é possível observar a chamada da função *kernel*. Após a execução, a matriz resultante C é transferida por meio da função `cudaMemcpy`. Neste caso, o parâmetro `cudaMemcpyDeviceToHost` indica que a matriz é transferida da memória da GPU para a memória principal do *host*. Por fim, as matrizes são desalocadas da memória da GPU através da função `cudaFree`.

Algoritmo 22 – Multiplicação de Matrizes: Chamada da função *kernel*

```
1  int blockSize = 32;
2  dim3 threadsPerBlock(blockSize, blockSize);
3  dim3 numBlocks((rowsA + blockSize -1) / blockSize,
4                 (colsB + blockSize -1) / blockSize);
5
6  matmul<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, colsB, rowCol, rowsA);
7
8  cudaMemcpy(C, d_C, rowsA * colsB * sizeof(double), cudaMemcpyDeviceToHost);
9
10 cudaFree(d_A);
11 cudaFree(d_B);
12 cudaFree(d_C);
```

Fonte: O autor (2024)

4.1.3 Paralelização da Multiplicação de Matrizes com OpenACC

No Algoritmo 23 tem-se a paralelização da multiplicação de matrizes utilizando OpenACC. Neste caso, é utilizada a diretiva `acc parallel loop`, que realiza a paralelização do laço mais externo (linha 6). O OpenACC priva as variáveis de controle dos laços de forma automática. Para garantir que os dados das matrizes `A` e `B` sejam copiados no início da região paralela para a memória da GPU (ou outro acelerador) é utilizada a cláusula `copyin` (linha 4 a linha 5). Em relação à matriz `C`, é necessário apenas alocar o espaço no início da região paralela e copiá-la para a memória do *host* ao final, sendo portanto utilizada a cláusula `copyout`.

Algoritmo 23 – Multiplicação de matrizes: Paralelização com OpenACC

```
1  void mat_mul(double * A, double * B, double * C, int colsB,
2               int rowCol, int rowsA){
3      int i, j, k;
4      #pragma acc data copyin(A[0:rowCol*rowsA],
5                              B[0:rowCol*colsB]) copyout(C[0:rowsA*colsB])
6      #pragma acc parallel loop
7      for (i = 0; i < rowsA; i++){
8          for (j = 0; j < colsB; j++){
9              C[i*colsB+j] = 0;
10             for (k = 0; k < rowCol; k++){
11                 C[i*colsB+j] += A[i*rowCol+k] * B[k*colsB+j];
12             }
13         }
14     }
15 }
```

Fonte: O autor (2024)

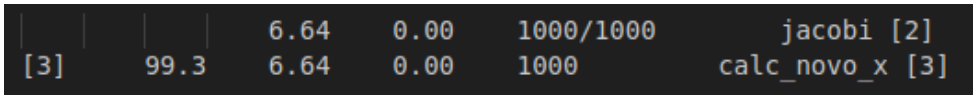
4.2 PARALELIZAÇÃO DO MÉTODO DE JACOBI

Neste seção, é realizada uma descrição da paralelização do método de Jacobi. Nela são apresentadas as funções e diretivas utilizadas nas implementações com OpenMP, CUDA e OpenACC.

4.2.1 Paralelização do Método de Jacobi com OpenMP

O método de Jacobi atualiza o vetor de incógnitas `x` a cada iteração. Neste caso, optou-se pela paralelização do processo de cálculo do novo `x`, visto que essa é a operação mais custosa do método. De fato, como pode ser visto na Figura 6, esse procedimento corresponde a 99,3% do tempo de execução do método.

Figura 6 – Perfilamento do Método de Jacobi



Fonte: O autor (2024).

No Algoritmo 24 tem-se a paralelização do método de Jacobi utilizando OpenMP. Como pode ser observado, na linha 8 tem-se a paralelização do cálculo do novo vetor `x`, através da diretiva `omp parallel for`. A cláusula `private` foi utilizada para que cada *thread* possua uma cópia para a variável `j`. A diretiva `parallel for` priva de forma automática a variável `i`, utilizada como variável de controle do laço mais externo. Após a conclusão da região paralela, é calculada de forma sequencial a diferença entre a norma do vetor `x` da iteração anterior e a norma do vetor `x` atual (linha 17). O método é encerrado caso a diferença seja menor que o valor de tolerância `TOL` ou o número máximo de iterações `N` seja atingido.

Algoritmo 24 – Método de Jacobi: Paralelização com OpenMP

```
1  void jacobi (double *A, double *b, int n, double *x){
2      int k = 0;
3      double * novo_x = (double *) calloc(n, sizeof(double));
4
5      int i,j;
6
7      while (k < N){
8          #pragma omp parallel for private (j)
9          for (i = 0; i < n; i++){
10             double soma = 0;
11             for (j = 0; j < n; j++){
12                 if (i != j) soma += A[i*n+j] * x[j];
13             }
14             novo_x[i] = (b[i] - soma)/A[i*n+i];
15         }
16
17         if (fabs((norma_vetor(x, n) - norma_vetor(novo_x, n))) < TOL){
18             for (i = 0; i < n; i++) x[i] = novo_x[i];
19             return;
20         }
21
22         for (i = 0; i < n; i++) x[i] = novo_x[i];
23
24         k++;
25     }
26 }
```

Fonte: O autor (2024)

4.2.2 Paralelização do Método de Jacobi com CUDA

Como pode ser observado no Algoritmo 25, inicialmente a matriz de coeficientes, o vetor de incógnitas e o vetor solução são alocados na memória da GPU através da função `cudaMalloc`. Após, os mesmos são transferidos para a memória da GPU através da função `cudaMemcpy`. O vetor `x` e o vetor `novo_x` foram inicializados com todos os elementos zerados.

Algoritmo 25 – Método de Jacobi: Transferência dos dados para a memória da GPU

```
1 cudaMalloc(&d_b, n * sizeof(double));
2 cudaMalloc(&d_A, n * n * sizeof(double));
3 cudaMalloc(&d_x, n * sizeof(double));
4 cudaMalloc(&d_novo_x, n * sizeof(double));
5 cudaMemcpy(d_b, b, n * sizeof(double), cudaMemcpyHostToDevice);
6 cudaMemcpy(d_A, A, n * n * sizeof(double), cudaMemcpyHostToDevice);
7 cudaMemcpy(d_x, x, n * sizeof(double), cudaMemcpyHostToDevice);
8 cudaMemcpy(d_novo_x, novo_x, n * sizeof(double), cudaMemcpyHostToDevice);
```

Fonte: O autor (2024)

No Algoritmo 26 tem-se a função *kernel* do método de Jacobi em CUDA. Neste caso, as *threads* são identificadas utilizando blocos unidimensionais. A linha do sistema de equações a ser processada por cada *thread* é calculada através de um combinação dos valores do bloco (`blockIdx`), da quantidade de *threads* por bloco (`blockDim`) e pelo identificador da *thread* (`threadIdx`). Neste caso, cada *thread* é responsável pelo cálculo de uma posição específica do novo vetor `x` (variável `i`). Além disso, o vetor `x` é armazenado na memória compartilhada da GPU para diminuir o tempo de acesso (variável `shared_x`). A função `__syncthreads()` (linha 6) é utilizada para sincronizar as *threads* e garantir que todas elas atualizaram a variável `shared_x`.

Algoritmo 26 – Método de Jacobi: *Kernel* CUDA

```
1  __global__ void jacobi(double *A, double *b, int n, double *x
2                         , double *novo_x){
3      int j, i =  blockIdx.x * blockDim.x + threadIdx.x;
4      extern __shared__ double shared_x[];
5      for (j = threadIdx.x; j < n; j += blockDim.x) shared_x[j] = x[j];
6      __syncthreads();
7      if (i < n){
8          double soma = 0.0;
9          for (j = 0; j < n; j++)
10             if (i != j) soma += A[i*n+j] * shared_x[j];
11         novo_x[i] = (b[i] - soma)/A[i*n+i];
12     }
13 }
```

Fonte: O autor (2024)

No Algoritmo 27 tem-se a chamada da função *kernel* do método de Jacobi, sendo que o número de blocos é calculado de acordo com o número de equações do sistema, definido pelo valor da variável `n`. A cada iteração, é verificada a convergência do método. Essa convergência é calculada de forma sequencial na CPU, sendo utilizada a função `cudaMemcpy` para realizar as transferências de dados entre a memória da GPU e CPU (linha 6 e 7). Ao final de cada iteração, o vetor `x` tem seu valor substituído pelo valor de `novo_x`, como pode ser observado na linha 14. Ao final do método, o vetor `x` é copiado da memória da GPU para a CPU, utilizando a função `cudaMemcpy`. Além disso, as estruturas são desalocadas da memória da GPU por meio da função `cudaFree`.

Algoritmo 27 – Método de Jacobi: Chamada da função *kernel*

```
1  int threadsPerBlock = 128;
2  int numBlocks = (n + threadsPerBlock - 1) / threadsPerBlock;
3  for (k = 0; k < N; k++){
4      jacobi<<<numBlocks, threadsPerBlock, n*sizeof(double)>>>(d_A, d_b, n
5                                                              , d_x, d_novo_x);
6      cudaMemcpy(x, d_x, n * sizeof(double), cudaMemcpyDeviceToHost);
7      cudaMemcpy(novo_x,d_novo_x,n * sizeof(double),cudaMemcpyDeviceToHost);
8
9      if (fabs((norma_vetor(x, n) - norma_vetor(novo_x, n))) < TOL){
10         cudaMemcpy(d_x, d_novo_x, n * sizeof(double),
11                    cudaMemcpyDeviceToDevice);
12         break;
13     }
14     cudaMemcpy(d_x, d_novo_x, n * sizeof(double), cudaMemcpyDeviceToDevice);
15 }
16 cudaMemcpy(x, d_x, n * sizeof(double), cudaMemcpyDeviceToHost);
17
18 cudaFree(d_A);
19 cudaFree(d_b);
20 cudaFree(d_x);
21 cudaFree(d_novo_x);
```

Fonte: O autor (2024)

4.2.3 Paralelização do Método de Jacobi com OpenACC

No Algoritmo 28 tem-se a paralelização do método de Jacobi utilizando OpenACC. A matriz de coeficientes `A`, o vetor de incógnitas `x` e o vetor de termos independentes `b` são copiadas da memória da CPU para a memória da GPU utilizando a cláusula `copyin`. Ao final da região paralela, o valor de `novo_x` é copiado da GPU para a CPU, utilizando a cláusula `copyout`. Essa cópia é realizada para que seja possível executar o cálculo da convergência pela CPU (linha 19). Como pode ser observado, nas linhas 10 a 17 tem-se a paralelização do cálculo do novo vetor `x`, através da diretiva `acc parallel loop`. O OpenACC priva

as variáveis de controle dos laços de forma automática. Após o término da região paralela, é calculada a diferença entre a norma do vetor `x` da iteração anterior e a norma do vetor `x` atual (linha 19). O método é encerrado caso a diferença seja menor que o valor de tolerância `TOL`. Essa verificação de convergência é realizada de forma sequencial pela CPU.

Algoritmo 28 – Método de Jacobi: Paralelização com OpenACC

```
1  void jacobi (double *A, double *b, int n, double *x){
2      int k = 0;
3      double * novo_x = (double *) calloc(n, sizeof(double));
4
5      int i,j;
6      #pragma acc data copyin(A[0:n*n], b[0:n])
7      {
8          while (k < N){
9              #pragma acc data copyout(novo_x[0:n]) copyin(x[0:n])
10             #pragma acc parallel loop
11             for (i = 0; i < n; i++){
12                 double soma = 0;
13                 for (j = 0; j < n; j++){
14                     if (i != j) soma += A[i*n+j] * x[j];
15                 }
16                 novo_x[i] = (b[i] - soma)/A[i*n+i];
17             }
18
19             if (fabs((norma_vetor(x, n) - norma_vetor(novo_x, n))) < TOL){
20                 for (i = 0; i < n; i++) x[i] = novo_x[i];
21                 break;
22             }
23
24             for (i = 0; i < n; i++) x[i] = novo_x[i];
25             k++;
26         }
27     }
28 }
```

Fonte: O autor (2024)

4.3 PARALELIZAÇÃO DO BFS

Nesta seção, é descrita a paralelização do algoritmo BFS, apresentando as funções e diretivas utilizados nas versões com OpenMP, CUDA e OpenACC.

4.3.1 Paralelização do BFS com OpenMP

Para a paralelização do BFS foi utilizada uma abordagem baseada no trabalho de Yasui e Fujisawa (2015). Neste, para cada vértice do nível atual, é realizada uma busca em paralelo pelos vértices vizinhos. Como pode ser observado no Algoritmo 29, a função `bfs` recebe como parâmetros o vértice inicial `r`, o grafo e o número de vértices desse (valor `n`). Inicialmente, é adicionado o vértice inicial r na fila de vértices `Q` (linha 3) e o vetor de níveis é inicializado (linhas 5 a 10). Após, é verificado se ainda existem vértices a serem buscados no próximo nível (linha 12). Em caso positivo, essa busca é realiza em paralelo através da diretiva `parallel for`. A variável `j` é privada de forma automática. A diretiva `private` é utilizada para que cada *thread* tenha uma cópia da variável `i`. Se uma *thread* identificar a existência de um vértice vizinho que ainda não foi visitado (linha 19), esse é adicionado à próxima posição da fila `Q` (linhas 20 a 23) e seu nível é atribuído no vetor de níveis `vniv` (linha 24). A diretiva `atomic capture`, é utilizada para evitar problemas de concorrência no incremento da variável `pf`, responsável por armazenar o tamanho da fila `Q`. Ao final de região paralela, o número de vértices do nível atual (variável `n _vert_nivel_atual`) é adicionado à variável `po` (linha 28), responsável por armazenar o número de vértices já percorridos. Neste caso, não é necessário nenhum tipo de proteção, visto que o incremento da variável `po` encontra-se fora da região paralela.

Algoritmo 29 – BFS: Paralelização com OpenMP

```
1  void bfs(int r, int n, int * grafo) {
2      int * Q = (int * ) malloc(n * sizeof(int));
3      Q[0] = r;
4      int po = 0, pf = 1;
5      int * vniv = (int * ) malloc(n * sizeof(int));
6      int i;
7
8      for (i = 0; i < n; i++) vniv[i] = -1;
9
10     vniv[r] = 0;
11
12     while (po != pf) {
13         int j, n_vert_nivel_atual = pf - po;
14
15         #pragma omp parallel for private(i)
16         for (j = po; j < po + n_vert_nivel_atual; j++) {
17             int v = Q[j];
18             for (i = 0; i < n; i++) {
19                 if (grafo[v * n + i] != 0 && vniv[i] == -1) {
20                     #pragma omp atomic capture
21                     int old_pf = pf++;
22
23                     Q[old_pf] = i;
24                     vniv[i] = vniv[v] + 1;
25                 }
```

```
26                  }
27              }
28              po += n_vert_nivel_atual;
29          }
30          free(Q);
31          free(vniv);
32      }
```

Fonte: O autor (2024)

4.3.2 Paralelização do BFS com CUDA

Como pode ser observado na Algoritmo 30, as variáveis `grafo` (matriz de adjacências), `Q` (fila de vértices a serem procurados), `nivel` (vetor que armazena o nível de cada vértice) e a variável `pf` (que armazena o número de vértices a serem buscados), são alocadas na memória da GPU através da função `cudaMalloc`. Após, os valores dessas variáveis são transferidas da memória principal para a memória da GPU, através da função `cudaMemcpy`.

Algoritmo 30 – BFS: Alocação e transferência dos dados para a memória da GPU

```
1  cudaMalloc(&d_grafo, n * n * sizeof(int));
2  cudaMalloc(&d_Q, n * sizeof(int));
3  cudaMalloc(&d_vniv, n * sizeof(int));
4  cudaMalloc(&d_pf, sizeof(int));
5
6  cudaMemcpy(d_grafo, grafo, n * n * sizeof(int), cudaMemcpyHostToDevice);
7  cudaMemcpy(d_Q, Q, n * sizeof(int), cudaMemcpyHostToDevice);
8  cudaMemcpy(d_vniv, vniv, n * sizeof(int), cudaMemcpyHostToDevice);
9  cudaMemcpy(d_pf, pf, sizeof(int), cudaMemcpyHostToDevice);
```

Fonte: O autor (2024)

No *kernel*, as *threads* são identificadas utilizando blocos unidimensionais. Cada *thread* realiza a busca dos vértices vizinhos de um dos vértices do nível atual da fila `Q` (linha 6). O índice do vértice, cujos vizinhos cada thread deve buscar (`thid`), é composto pelo identificador do bloco (`blockIdx`), quantidade de *threads* por bloco (`blockDim`) e pelo identificador da *thread* (`threadIdx`). Devido ao índice `thid` começar sempre a partir do valor 0, é adicionada a variável `po`, responsável por armazenar a posição do vértice inicial do nível atual. Caso seja encontrado um vértice vizinho ainda não visitado (linha 9), esse é adicionado à fila `Q`. A variável `pf`, responsável por armazenar o número de vértices a serem buscados, é incrementada sendo utilizada a função `atomicAdd`, para evitar problemas de concorrência. Por fim, o nível do vértice no vetor de níveis é atualizado (linha 11).

Algoritmo 31 – BFS: Paralelização em CUDA

```
1  __global__ void gpu_bfs(int n, int *grafo, int *vniv, int *Q, int *pf,
2                          int po, int n_vert_nivel_atual){
3      int thid = blockIdx.x * blockDim.x + threadIdx.x;
4
5      if (thid < n_vert_nivel_atual){
6          int v = Q[po+thid], i;
7
8          for (i = 0; i < n; i++){
9              if (grafo[v*n+i] != 0 && vniv[i] == -1){
10                 Q[atomicAdd(pf,1)] = i;
11                 vniv[i] = vniv[v] + 1;
12             }
13         }
14     }
15 }
```

Fonte: O autor (2024)

Na chamada do kernel, é verificado se ainda existem vértices na fila `Q` (linha 1 do Algoritmo 32). O número de blocos é definido com base no número de vértices do nível atual (linha 5). Ao terminar a execução do *kernel* do nível atual, a variável `pf` é transferida para a CPU por meio da função `cudaMemcpy`, para um nova verificação se ainda existem vértices a serem buscados. Em caso negativo, o vetor com os níveis dos vértices é transferido para a memória principal através da função `cudaMemcpy` e as variáveis são desalocadas da memória da GPU através da função `cudaFree`.

Algoritmo 32 – BFS: Chamada da função *kernel* e liberação da memória na GPU

```
1  while (po != *pf){
2      int n_vert_nivel_atual = *pf - po;
3
4      int threadsPerBlock = 32;
5      int numBlocks = (n_vert_nivel_atual + threadsPerBlock - 1)
6                      / threadsPerBlock;
7
8      gpu_bfs<<<numBlocks, threadsPerBlock>>>(n, d_grafo, d_vniv
9                                              , d_Q, d_pf, po
10                                             , n_vert_nivel_atual);
11
12     cudaMemcpy(pf, d_pf, sizeof(int), cudaMemcpyDeviceToHost);
13
14     po += n_vert_nivel_atual;
15 }
16
17 cudaMemcpy(nivel, d_vniv, n * sizeof(int), cudaMemcpyDeviceToHost);
18
```

```
19 cudaFree(d_grafo);
20 cudaFree(d_Q);
21 cudaFree(d_vniv);
22 cudaFree(d_pf);
```

Fonte: O autor (2024)

4.3.3 Paralelização do BFS com OpenACC

O Algoritmo 33 apresenta a paralelização do método BFS utilizando o OpenACC. As variáveis `grafo` e `Q` são copiadas da memória da memória principal para a memória da GPU, utilizando a cláusula `acc data copyin` (linha 13). No caso da variável `vniv`, que armazena, o nível dos vértices, foi utilizada a cláusula `copy`, indicando que essa é copiada no início da região paralela e, ao final, é atualizada na memória do *host*. Caso existam vértices a serem buscados (linha 14), essa essa busca é realiza em paralelo através da diretiva `acc parallel loop` (linha 18). As variáveis de controle de laço `j` e `i` são privadas automaticamente. A diretiva `acc data` é utilizada em conjunto com a cláusula `copyin` (linha 17) para indicar que os valores das variáveis `n_vert_nivel_atual` (número de vértices do nível atual) e `po` (número de vértices percorridos) devem ser copiados para a memória da GPU no início da região paralela. No caso da variável `pf` (número total de vértices da fila) é utilizada a cláusula `copy`, indicando que a variável deve ser copiada para a memória da GPU no início da região paralela, e atualizada na memória do *host* após a conclusão da mesma.

Durante a execução da região paralela, caso uma *thread* identifique um novo vértice vizinho (linha 23), este será adicionado à fila `Q` (linhas 24 a 28), e o nível correspondente será registrado no vetor de níveis `vniv` (linha 29). A diretiva `acc atomic capture` é utilizada para evitar problemas de concorrência no incremento da variável `pf` (tamanho da fila `Q`). Após a conclusão da busca simultânea pelos vértices vizinhos (região paralela), a variável `po` é atualizada (linha 35).

Algoritmo 33 – BFS: Paralelização em OpenACC

```
1  void bfs(int r, int n, int *grafo)
2  {
3      int *Q = (int *)malloc(n * sizeof(int));
4      Q[0] = r;
5      int po = 0, pf = 1;
6      int *vniv = (int *)malloc(n * sizeof(int));
7      int i;
8
9      for(i = 0; i < n; i++) vniv[i] = -1;
10
11     vniv[r] = 0;
12
13     #pragma acc data copyin(grafo[0:n*n], Q[0:n]) copy(vniv[0:n])
```

```
14      while (po != pf){
15          int j, n_vert_nivel_atual = pf - po;
16
17          #pragma acc data copyin(n_vert_nivel_atual, po) copy(pf)
18          #pragma acc parallel loop
19          for (j = po; j < po + n_vert_nivel_atual; j++){
20
21              int v = Q[j];
22              for (i = 0; i < n; i++){
23                  if (grafo[v*n+i] != 0 && vniv[i] == -1){
24                      int old_pf;
25                      #pragma acc atomic capture
26                      old_pf = pf++;
27
28                      Q[old_pf] = i;
29                      vniv[i] = vniv[v] + 1;
30                  }
31              }
32
33          }
34
35          po += n_vert_nivel_atual;
36      }
37
38      free(Q);
39      free(vniv);
40  }
```

Fonte: O autor (2024)

4.4 PARALELIZAÇÃO DO RADIX SORT

Nesta seção, é apresentada a paralelização do algoritmo *radix sort* utilizando as bibliotecas OpenMP, CUDA e OpenACC.

4.4.1 Paralelização do Radix Sort com OpenMP

A implementação do *Radix Sort* possui como parâmetros de entrada o vetor a ser ordenado `A`, o número de elementos do vetor (`n`) e o número de *threads* a serem utilizadas (Algoritmo 34). Primeiramente, é obtido o maior valor de A, sendo armazenado na variável `m` (linha 4). Após, tem-se uma iteração partindo do dígito menos significativo até o número de dígitos de `m` (linha 7). A cada iteração, é extraído o dígito atual de cada elemento do vetor `A`. Esse procedimento é realizado em paralelo, ou seja, cada *thread* é responsável por uma parte dos elementos de `A` (diretiva `omp parallel` da linha 15). Neste caso, cada *thread* gera uma histograma local, com a ocorrência de cada dígito (vetor `balde_local`) (linhas 15 a 20). Após,

é realizada a montagem de um histograma global (vetor `balde`) (linhas 22 a 26). Nesse caso, foi utilizada a diretiva `omp critical` para garantir que apenas uma *thread* por vez atualize o vetor, evitando problemas de concorrência.

Após uma única *thread* é responsável por calcular a posição correta dos elementos de A (linha 29 a 33). O cálculo das posições é realizado utilizando o vetor de histogramas global `balde`. Para tanto, utilizou-se a diretiva `omp barrier` para garantir que vetor de histogramas global `balde` esteja completamente montado (linha 28). Em seguida, de forma paralela, é construído o vetor `saida` contendo a ordenação até a iteração atual. Por fim, o vetor `saida` copiado para o vetor `A`, para o início de uma nova iteração.

Algoritmo 34 – *Radix sort*: Paralelização com OpenMP

```
1  void radix(int * A, int n, int nthreads) {
2
3      int *saida = (int *)malloc(n * sizeof(int));
4      int m = getMax(A, n);
5
6      int exp, j;
7      for (exp = 1; m / exp > 0; exp *= 10) {
8          int balde[DIGITOS] = {0};
9          int *balde_local= (int*) calloc(DIGITOS * nthreads, sizeof(int));
10
11         #pragma omp parallel
12         {
13             int tid = omp_get_thread_num();
14
15             #pragma omp for
16             for (j = 0; j < n; j++)
17             {
18                 int d = (A[j] / exp) % 10;
19                 balde_local[tid*DIGITOS+d]++;
20             }
21
22             #pragma omp critical
23             for (j = 0; j < DIGITOS; j++)
24             {
25                 balde[j] += balde_local[tid*DIGITOS+j];
26             }
27
28             #pragma omp barrier
29             #pragma omp single
30             for (j = 1; j < DIGITOS; j++)
31             {
32                 balde[j] += balde[j-1];
33             }
34
```

```
35              int offset[DIGITOS];
36              for (int j = 0; j < DIGITOS; j++) {
37                  offset[j] = balde[j];
38                  for (int t = nthreads - 1; t >= tid; t--) {
39                      offset[j] -= balde_local[t * DIGITOS + j];
40                  }
41              }
42
43              #pragma omp for
44              for(j = 0; j < n; j++)
45              {
46                  int d = (A[j] / exp) % 10;
47                  saida[offset[d]++] = A[j];
48              }
49
50              #pragma omp for
51              for (j = 0; j < n; j++)
52                  A[j] = saida[j];
53
54          }
55
56          free(balde_local);
57      }
58
59      free(saida);
60
61  }
```

Fonte: O autor (2024)

4.4.2 Paralelização do Radix Sort com CUDA

Inicialmente, como pode ser observado no Algoritmo 35, as estruturas necessárias foram alocadas na GPU por meio da função `cudaMalloc` e o vetor `A` foi transferido da memória do *host* para a memória da GPU através da função `cudaMemcpy`.

Algoritmo 35 – *Radix sort*: Alocação e transferência dos dados para a memória da GPU

```
1  cudaMalloc(&d_A, n * sizeof(int));
2  cudaMalloc(&d_balde, DIGITOS * sizeof(int));
3  cudaMalloc(&d_balde_local, DIGITOS * numBlocks * sizeof(int));
4  cudaMalloc(&d_offset, DIGITOS * numBlocks * sizeof(int));
5  cudaMalloc(&d_saida, n * sizeof(int));
6
7  cudaMemcpy(d_A, A, n * sizeof(int), cudaMemcpyHostToDevice);
```

Fonte: O autor (2024)

Cada etapa do *radix sort* foi implementada em um *kernel* diferente. O primeiro *kernel*, que é responsável por calcular os histogramas locais pode ser visto no Algoritmo 36. Neste caso, cada bloco CUDA monta seu próprio histograma local, e cada *thread* é responsável por extrair o dígito de um único elemento, sendo que a posição do elemento (`thid`) é composta pelo identificador do bloco (`blockIdx`), quantidade de *threads* por bloco (`blockDim`) e pelo identificador da thread (`threadIdx`). A função `atomicAdd` (linha 6), é utilizada para evitar problemas de concorrência no acesso ao vetor `balde_local`.

Algoritmo 36 – *Radix sort*: Cálculo dos histogramas locais

```
1 __global__ void calc_local_hist(int *A, int n, int exp, int *balde_local){
2     int thid = blockIdx.x * blockDim.x + threadIdx.x;
3
4     if (thid < n){
5         int d = (A[thid] / exp) % 10;
6         atomicAdd(&balde_local[blockIdx.x*DIGITOS+d], 1);
7     }
8 }
```

Fonte: O autor (2024)

O segundo *kernel* (Algoritmo 37), responsável por montar o histograma global a partir dos histogramas locais. Nesse caso, cada thread identifica as posições que deve atualizar no vetor responsável por armazenar o histograma global (vetor `balde`). A função `atomicAdd` (linha 8) foi utilizada para evitar problemas de concorrência no acesso ao vetor com o histograma global (`balde`).

Algoritmo 37 – *Radix sort*: Cálculo do histograma global

```
1  __global__ void calc_hist(int *balde, int *balde_local){
2      int thid = blockIdx.x * blockDim.x + threadIdx.x;
3
4      if(thid < gridDim.x){
5          int j;
6          for (j = 0; j < DIGITOS; j++)
7          {
8              atomicAdd(&balde[j],balde_local[thid*DIGITOS+j]);
9          }
10     }
11 }
```

Fonte: O autor (2024)

O terceiro *kernel*, responsável por calcular a posição final dos elementos no vetor de saída (Algoritmo 38). Esse *kernel* é executado em apenas uma *thread* devido ao fato desse procedimento apresentar uma natureza sequencial.

Algoritmo 38 – Radix sort: Cálculo da posição final

```
1 __global__ void updt_hist(int *balde, int *balde_local){
2     if(threadIdx.x == 0){
3         int j;
4         for (j = 1; j < DIGITOS; j++)
5         {
6             balde[j] += balde[j-1];
7         }
8     }
9 }
```

Fonte: O autor (2024)

O quarto *kernel*, responsável por montar o vetor com as posições dos elementos no vetor final (Algoritmo 39). Para cada bloco CUDA, é montado um vetor local (vetor `offset`), que armazena as posições dos elementos para a montagem do vetor `saida`.

Algoritmo 39 – Radix sort: Calcula vetor de posições finais

```
1  __global__ void calc_offset(int *balde, int *balde_local, int *offset){
2      int thid = blockIdx.x * blockDim.x + threadIdx.x;
3      if (thid < gridDim.x){
4          for (int j = 0; j < DIGITOS; j++) {
5              offset[thid*DIGITOS+j] = balde[j];
6              int t;
7              for (t = gridDim.x - 1; t >= (int) thid; t--) {
8                  offset[thid*DIGITOS+j] -= balde_local[t * DIGITOS + j];
9              }
10         }
11     }
12 }
```

Fonte: O autor (2024)

O quinto *kernel* é responsável por montar o vetor temporário `saida` com os elementos ordenados até a iteração atual (Algoritmo 40). A função `atomicAdd` é utilizada para evitar problemas de concorrência no incremento da variável `offset`.

Algoritmo 40 – *Radix sort*: Montagem do vetor temporário

```
1  __global__ void create_output(int *A, int *saida, int n
2                                , int exp, int *offset){
3      int thid = blockIdx.x * blockDim.x + threadIdx.x;
4
5      if (thid < n){
6          int d = (A[thid] / exp) % 10;
7          saida[atomicAdd(&offset[blockIdx.x*DIGITOS+d], 1)] = A[thid];
8      }
9
10 }
```

Fonte: O autor (2024)

Por fim, o sexto *kernel* é responsável por atualizar o vetor original `A`, copiando os elementos do vetor `saida` para o mesmo (Algoritmo 41).

Algoritmo 41 – Radix sort: Atualiza o vetor original com valores ordenados

```
1  __global__ void swap_arr(int *A, int *saida, int n){
2      int thid = blockIdx.x * blockDim.x + threadIdx.x;
3
4      if (thid < n){
5          A[thid] = saida[thid];
6      }
7  }
```

Fonte: O autor (2024)

Como pode ser observado no Algoritmo 42, a cada iteração do *Radix sort*, todos *kernels* são chamados em sequência. Neste caso, apenas o *kernel* que calcula a posição final de cada elemento no vetor de saída é executado de forma sequencial (*kernel* `updt_hist` na linha 16). Ao atingir o número máximo de dígitos, o vetor ordenado `A` é copiado da memória da GPU para a memória do *host*, por meio da função `cudaMemcpy`. Ao final, as estruturas utilizadas são desalocadas utilizando a função `cudaFree`.

Algoritmo 42 – Radix sort: Chamada dos *kernels* e liberação da memória da GPU

```
1  int max = getMax(A, n);
2  int threadsPerBlock = 256;
3  int numBlocks = (n + threadsPerBlock - 1) / threadsPerBlock;
4  int numB = (numBlocks+ threadsPerBlock - 1) / threadsPerBlock;
5  int exp;
6  for (exp = 1; max / exp > 0; exp *= 10) {
7
8      cudaMemset(d_balde, 0, DIGITOS * sizeof(int));
9      cudaMemset(d_balde_local, 0, DIGITOS * numBlocks * sizeof(int));
10
11     calc_local_hist<<<numBlocks, threadsPerBlock>>>(d_A,n,exp
12                                                     ,d_balde_local);
13
14     calc_hist<<<numB, threadsPerBlock>>>(d_balde, d_balde_local);
15
16     updt_hist<<<1, 1>>>(d_balde, d_balde_local);
17
18     calc_offset<<<numB, threadsPerBlock>>>(d_balde,d_balde_local,d_offset);
19
20     create_output<<<numBlocks, threadsPerBlock>>>(d_A,d_saida,n,exp
21                                                   ,d_offset);
22
23     swap_arr<<<numBlocks, threadsPerBlock>>>(d_A,d_saida,n);
24 }
25
26 cudaMemcpy(A, d_A, length * sizeof(int), cudaMemcpyDeviceToHost);
27
28 cudaFree(d_A);
29 cudaFree(d_balde);
30 cudaFree(d_balde_local);
31 cudaFree(d_offset);
32 cudaFree(d_saida);
```

Fonte: O autor (2024)

4.4.3 Paralelização do Radix Sort com OpenACC

No Algoritmo 43 tem-se a paralelização do método *Radix sort* em OpenACC. Primeiramente, é obtido o maior valor de A, sendo armazenado na variável `m` (linha 4). Após, tem-se uma iteração partindo do dígito menos significativo até o número de dígitos de `m` (linha 7). Devido ao OpenACC não possuir um método para se obter o identificador de *threads* ou blocos, como o OpenMP e CUDA, o número de elementos que cada *thread* deve processar foi calculado manualmente e armazenado na variável `SLICES`.

O procedimento utilizado foi similar a paralelização com OpenMP. A cada iteração (dígito) (linhas 18 a 27) é gerado um histograma local, ou seja, cada *thread* calcula um histograma correspondente aos elementos da sua parte (`acc parallel loop` (linha 17)). A diretiva usada é a `copyin` para identificar que, no início da região paralela, o vetor `A` deve ser copiado da memória do *host* para a memória GPU. A diretiva `copy` foi utilizada para identificar que o vetor `balde_local` deverá ser copiado da memória da *host* para a memória da GPU, e ao final esse deve ser copiado de volta para memória do *host*. Quando todas as *threads* finalizarem a montagem dos histogramas locais é realizada a montagem do histograma global (linhas 29 a 38). A diretiva `acc atomic update` foi utilizada para evitar problemas de concorrência no acesso ao vetor que armazena o histograma global (`balde`). Em seguida, a CPU calcula a posição de cada elemento no vetor ordenado. O vetor temporário (vetor `saida`) é construído em paralelo, contendo os elementos organizados de acordo com a ordenação alcançada até a iteração atual. Por fim, o vetor `A` é atualizado com os valores do vetor `saida`, avançando para a próxima iteração do *Radix Sort*.

Algoritmo 43 – Radix sort: Código OpenACC

```
1  void radix(int * A, int length) {
2
3      int *saida = (int *)malloc(length * sizeof(int));
4      int m = getMax(A, length);
5
6      int exp, i, j;
7      for (exp = 1; m / exp > 0; exp *= 10) {
8
9              int balde[DIGITOS] = {0};
10             int balde_local[DIGITOS * SLICES] = {0};
11             int slice_size = length/SLICES;
12             int resto = length % SLICES;
13             int offset[DIGITOS * SLICES];
14
15             #pragma acc data copyin(A[0:length])
16                              copy(balde_local[0:DIGITOS * SLICES])
17             #pragma acc parallel loop private(j)
18             for (i = 0; i < SLICES; i++){
19                 int ini = slice_size * i, fim = ini + slice_size +
20                           (i == SLICES-1 ? resto : 0);
21                 for (j = ini; j < fim; j++)
22                 {
23                     int d = (A[j] / exp) % 10;
24                     #pragma acc atomic update
25                     balde_local[i*DIGITOS+d]++;
26                 }
27             }
28
29             #pragma acc data copy(balde[0:DIGITOS])
```

```
30                                  copyin(balde_local[0:DIGITOS * SLICES])
31              #pragma acc parallel loop
32              for (i = 0; i < SLICES; i++){
33                  for (j = 0; j < DIGITOS; j++)
34                  {
35                      #pragma acc atomic update
36                      balde[j] += balde_local[i*DIGITOS+j];
37                  }
38              }
39
40              for (j = 1; j < DIGITOS; j++)
41              {
42                  balde[j] += balde[j-1];
43              }
44
45              #pragma acc data copyout(offset[0:DIGITOS*SLICES])
46                      copyin(balde[0:DIGITOS],balde_local[0:DIGITOS * SLICES])
47              #pragma acc parallel loop
48              for (i = 0; i < SLICES; i++){
49                  for (int j = 0; j < DIGITOS; j++) {
50                      offset[i*DIGITOS+j] = balde[j];
51                      for (int t = SLICES - 1; t >= i; t--) {
52                          offset[i*DIGITOS+j] -= balde_local[t * DIGITOS + j];
53                      }
54                  }
55              }
56
57              #pragma acc data copyin(A[0:length], offset[0:DIGITOS*SLICES])
58                                  copyout(saida[0:length])
59              #pragma acc parallel loop
60              for (i = 0; i < SLICES; i++){
61                  int ini = slice_size * i, fim = ini + slice_size +
62                            (i == SLICES-1 ? resto : 0);
63                  for (j = ini; j < fim; j++)
64                  {
65                      int d = (A[j] / exp) % 10;
66                      saida[offset[i*DIGITOS+d]++] = A[j];
67                  }
68              }
69
70              for (j = 0; j < length; j++)
71                  A[j] = saida[j];
72      }
73
74  }
```

Fonte: O autor (2024)

5 TESTES E RESULTADOS

Todos os testes foram realizados utilizando um computador com um processador Intel® Core i5-7400 com 4 núcleos de processamento, com frequência de 3.0 GHz. O processador possui uma *cache* L1 de 64 KB, *cache* L2 compartilhada de 256 KB e *cache* L3 compartilhada de 6 MB. Esse computador possui 24 GB de memória RAM DDR4, operando a 3200 MHz. O disco rígido é um HDD de 1 TB. O sistema operacional utilizado é o Ubuntu 22.04 LTS de 64 bits, com *kernel* versão 6.5. A GPU utilizada nos testes é uma GeForce GTX 1060. Essa placa possui uma arquitetura Pascal, com 1280 CUDA *cores*. Ela possui 6 GB de memória com uma *interface* de 192 *bits*.

As implementações foram compiladas utilizando o *nvc* versão 24.9, com otimização de nível 2 (`-O2`) (NVIDIA Corporation, 2024b). A avaliação das implementações foi realizada a partir de uma comparação entre os tempos de execução (média aritmética de 10 execuções). Os tempos obtidos com as implementação em OpenACC foram comparados aos tempos de execução: das aplicações sequenciais, ou seja, executando em um único núcleo de processamento; com as implementações paralelizadas com OpenMP; e com as implementações desenvolvidas com CUDA.

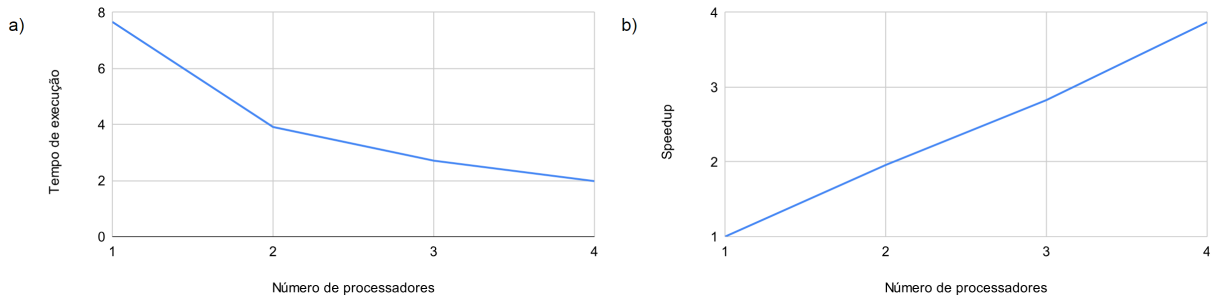
5.1 RESULTADOS - MULTIPLICAÇÃO DE MATRIZES

Para a realização de testes foram utilizadas matrizes de dimensões de 1500×1500. As matrizes foram inicializadas com valores randômicos. Nos testes, foi utilizada a mesma semente para a geração de números aleatórios. Na Figura 7 a), é apresentado o tempo de execução em função do número de processadores para a implementação com OpenMP. Na Figura 7 b), é apresentado o *speedup*[1] em função do número de processadores. Como pode ser observado, a paralelização com OpenMP apresentou um *speedup* quase que linear. De fato, com 4 núcleos de processamento foi obtido um *speedup* de 3,87. Esse resultado está de acordo com o *speedup* de aproximadamente 3,85 que foi obtido no trabalho de Hellberg e Bhamidipati (2022).

[1] Métrica utilizada em computação paralela para avaliar o ganho de desempenho de um programa paralelo em comparação com sua versão sequencial. Ele é definido como a razão entre o tempo de execução sequencial (T_s) e o tempo de execução paralelo (T_p):

$$S = \frac{T_s}{T_p}.$$

Figura 7 – Multiplicação de Matrizes: *Speedup* da implementação em OpenMP



Fonte: O autor (2024).

É possível visualizar na Tabela 1 os tempos de execução (em segundos) das diferentes implementações do método de multiplicação de matrizes. Neste caso, observa-se um significativo ganho de desempenho das implementações utilizando GPU, em relação a implementação sequencial e a implementação utilizando OpenMP. A implementação utilizando CUDA foi 76,60 vezes mais rápida que a implementação sequencial e a implementação com OpenACC para GPU foi 5,85 vezes mais rápida. Isso se deve ao fato das GPUs serem projetadas para se beneficiarem de cálculos paralelos com operações de ponto flutuante.

Tabela 1 – Multiplicação de Matrizes: Tempos de execução

Implementação	**Tempo de execução (s)**	**Desvio padrão**
Sequencial	7,66	0,049
OpenMP (4 *threads*)	1,98	0,022
CUDA	0,10	0,007
OpenACC (CPU com 4 *threads*)	2,42	0,028
OpenACC (GPU)	1,31	0,005

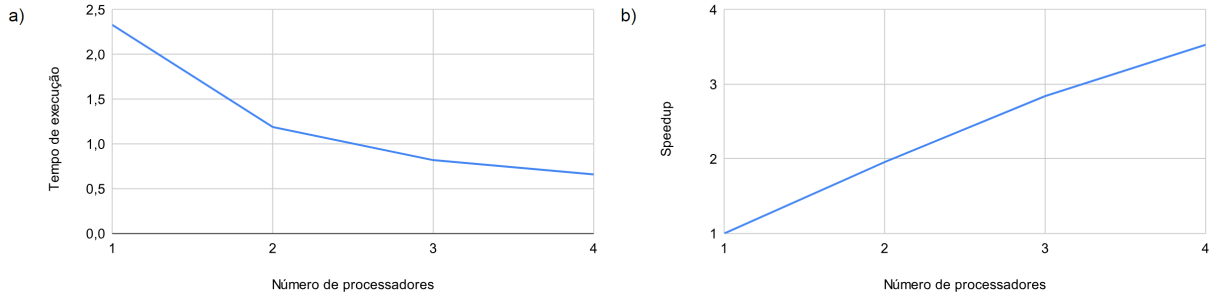
Fonte: O Autor (2024).

Observa-se que as implementações utilizando OpenACC obtiveram resultados inferiores, quando comparados às outras ferramentas de paralelização. Em relação à implementação em OpenMP, a implementação com OpenACC para CPU foi cerca de 22% mais lenta. Observa-se ainda, que a implementação utilizando OpenACC para GPU, apresentou um desempenho inferior à implementação utilizando CUDA. Isso deve-se ao OpenACC ser uma ferramenta de mais alto nível, não disponibilizando o mesmo nível de controle que as ferramentas de mais baixo nível como o OpenMP e o CUDA.

5.2 RESULTADOS - MÉTODO DE JACOBI

Para a realização dos testes foi utilizado um sistema com 700 equações, sendo que a matriz de coeficientes A e o vetor de incógnitas b foram inicializados com valores aleatórios. Todos os testes utilizaram o mesma semente para geração da matriz de coeficientes A e vetor de termos independentes b. O vetor de incógnitas foi inicializado com todos os elementos zerados. O número máximo de iterações foi definido como 15000 e o valor de tolerância TOL foi definido como 10^{-10}. Nestas condições, o método de Jacobi convergiu em 10622 iterações. Na Figura 8 a), é apresentado o tempo de execução do método de Jacobi, em função do número de processadores para a implementação com OpenMP. Na Figura 8 b), é apresentado o *speedup* em função do número de processadores. Como pode ser observado, a paralelização com OpenMP apresentou desempenho satisfatório, com um *speedup* crescente em função do número de processadores. Neste caso, para 4 processadores foi obtido um *speedup* de 3,53.

Figura 8 – Método de Jacobi: *Speedup* da implementação em OpenMP



Fonte: O autor (2024).

Na Tabela 2 tem-se o tempo os tempos de execução (em segundos) das diferentes implementações do método de Jacobi. Observa-se que as implementações para GPU, apresentaram um desempenho superior as implementações sequencial e com OpenMP. A implementação com CUDA foi 3,69 vezes mais rápida que a implementação sequencial e a implementação com OpenACC foi 3,58 vezes mais rápida. No entanto, o ganho de desempenho com a paralelização em relação à multiplicação de matrizes foi menor. Isso ocorre devido à necessidade de sincronização das threads após a atualização do vetor de incógnitas `X`, para o cálculo da convergência. Além disso, o cálculo de convergência é realizado de forma sequencial.

Tabela 2 – Método de Jacobi: Tempos de execução

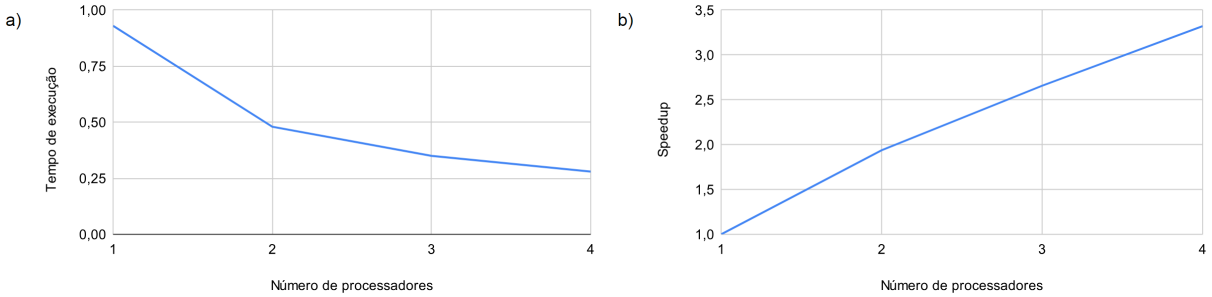
Implementação	**Tempo de execução (s)**	**Desvio padrão**
Sequencial	2,33	0,015
OpenMP (4 *threads*)	0,66	0,007
CUDA	0,63	0,006
OpenACC (CPU com 4 *threads*)	0,71	0,009
OpenACC (GPU)	0,65	0,013

Fonte: O Autor (2024).

5.3 RESULTADOS - BFS

Para o desenvolvimento e realização de testes, a matriz de adjacência foi inicializada de forma randômica. Além disso, o número de vértices foi definido como 15000. A mesma semente foi utilizada para a geração da matriz de adjacência em todos os testes. Na Figura 9 a), é apresentado o tempo de execução em função do número de processadores para a implementação com OpenMP. Na Figura 9 b), é apresentado o *speedup* em função do número de processadores. Como pode ser observado, a paralelização com OpenMP apresentou um *speedup* inferior as implementações da multiplicação de matrizes e o método de Jacobi. Neste caso, para 4 procesadores foi obtido um *speedup* de 3,32. A presença de seções críticas e a necessidade de utilizar mecanismos de exclusão mútua podem ter impactado negativamente no desempenho da implementação.

Figura 9 – BFS: *Speedup* da implementação em OpenMP



Fonte: O autor (2024).

Na Tabela 3 tem-se os tempos de execução (em segundos) das diferentes implementações do método BFS. Observa-se que as implementações para GPU em *CUDA* e OpenACC apresentaram resultados similares, sendo aproxidamente 8 vezes mais rápidas que a sequencial. A presença de níveis com poucos vértices pode ter impactado negativamente no desempenho, pois, nesses casos, não foi possível aproveitar plenamente o poder computacional da GPU. Observa-se ainda que as implementações em OpenACC apresentaram resultados levemente inferiores à implementações em OpenMP e CUDA.

Tabela 3 – BFS - Tempos de execução

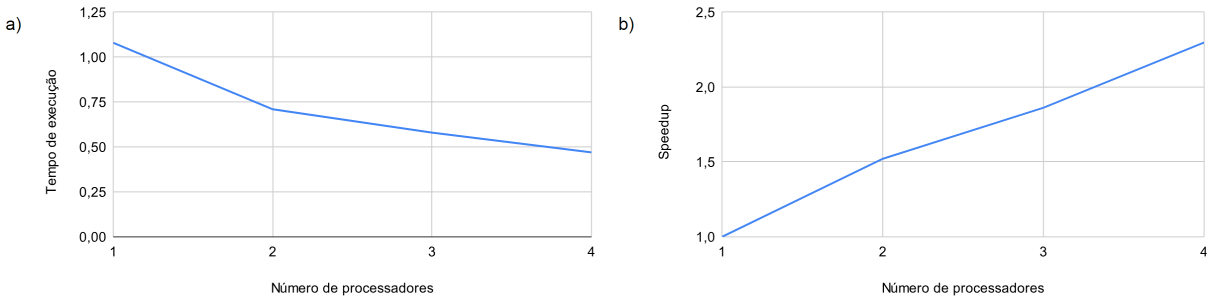
Implementação	**Tempo de execução (s)**	**Desvio padrão**
Sequencial	0,93	0,032
OpenMP (4 *threads*)	0,28	0,011
CUDA	0,19	0,004
OpenACC (CPU com 4 *threads*)	0,35	0,013
OpenACC (GPU)	0,21	0,004

Fonte: O Autor (2024).

5.4 RESULTADOS - RADIX SORT

Os testes foram realizados utilizando um vetor de 5 milhões de números, inicializado com números randômicos de 10 dígitos. Na Figura 9 a), é apresentado o tempo de execução em função do número de processadores para a implementação com OpenMP. Na Figura 9 b), é apresentado o *speedup* em função do número de processadores. Como pode ser observado, apesar de haver melhora no desempenho com o aumento do número de *threads*, esse aumento é inferior ao obtido nas demais implementações. De fato, com 4 processadores foi obtido um *speedup* de 2,30. O desempenho pode ter sido impactado negativamente pela existência de partes sequenciais na implementação (cálculo da posição final no vetor ordenado).

Figura 10 – *Radix sort*: *Speedup* da implementação em OpenMP



Fonte: O autor (2024).

Na Tabela 4 são apresentados os tempos de execução (em segundos) das diferentes implementações do método *Radix Sort*. Neste caso, a implementação em *CUDA* foi cerca de 9 vezes mais rápida que a implementação sequencial. Já a implementação em OpenACC para GPU, apesar de ter sido 2,76 vezes mais rápida que a sequencial, foi cerca de três vezes mais lenta que a implementação em CUDA. Observa-se ainda que a implementação em OpenACC para CPU foi 1,93 vezes mais rápida que a sequencial, sendo mais lenta que a implementação em OpenMP, que foi 2,30 vezes mais rápida que a implementação sequencial.

Tabela 4 – *Radix sort*: Tempos de execução

Implementação	**Tempo de execução (s)**	**Desvio padrão**
Sequencial	1,08	0,005
OpenMP (4 *threads*)	0,47	0,012
CUDA	0,12	0,008
OpenACC (CPU com 4 *threads*)	0,56	0,006
OpenACC (GPU)	0,39	0,003

Fonte: O Autor (2024).

6 CONSIDERAÇÕES FINAIS

Neste trabalho foi realizado um estudo sobre ferramentas para o desenvolvimento de aplicações paralelas para arquiteturas híbridas com CPUs e GPUs. Entre as ferramentas disponíveis foram analisados o StarPU, OpenACC e OmpSs. Entre essas optou-se pela utilização do OpenACC, pois esse é uma ferramenta mais madura e consolidada, possuindo apoio de empresas como a Nvidia, AMD e HP. Além disso, ele está disponível nativamente no NVCC, o compilador da NVIDIA, fabricante da GPU utilizada neste trabalho.

Optou-se pela não utilização do OmpSs, devido à falta de atualizações e suporte. Com a última atualização feita em 2019, não é possível garantir a confiabilidade da ferramenta. Além disso, a documentação é escassa, o que pode dificultar o desenvolvimento. O uso do StarPU foi descartado devido à necessidade de fornecer implementações específicas para cada arquitetura, exigindo que o programador tenha um conhecimento detalhado sobre essas arquiteturas e das ferramentas utilizadas para desenvolvimento. Essas características não atendem um dos principais objetivos do trabalho, que consiste em avaliar ferramentas capazes de facilitar o desenvolvimento em arquiteturas híbridas com CPUs *multicores* e GPUs.

Para a avaliação do OpenACC foram definidas 4 aplicações: multiplicação de matrizes, método de Jacobi, busca em largura em grafos e o método de ordenação *radix sort*. A escolha dessas aplicações foi baseada em *benchmarks* que tradicionalmente são utilizados para avaliação de desempenho de arquiteturas híbridas. Além disso, essas aplicações foram escolhidas pois o desenvolvimento e a paralelização das mesmas não é altamente complexa, permitindo assim a comparação de diferentes versões paralelas. No entanto, destaca-se que essas implementações permitem realizar comparações de desempenho utilizando operações com números inteiros e de ponto flutuante, que são comumente utilizadas em aplicações que demandam alto desempenho.

Os tempos de execução das implementações paralelizadas com o OpenACC foram comparadas com: a implementação sequencial; com a implementação paralelizada para processadores *multicore*, utilizando OpenMP; com a implementação paralelizada para GPUs, utilizando CUDA. Optou-se pela utilização do OpenMP pois esse é um padrão amplamente utilizado. Além disso, ele está disponível nativamente no compilador *nvc* e possui um modelo de programação muito semelhante ao OpenACC, permitindo o desenvolvimento sem necessidade de mudanças significativas no código. A escolha pelo ambiente CUDA se deu pelo fato de que esse foi desenvolvido para ser utilizado exclusivamente em GPUs da Nvidia, fabricante da placa utilizada neste trabalho. Além disso, as implementações em CUDA frequentemente possuem um maior desempenho quando comparados com outras ferramentas, como por exemplo, o OpenCL.

Os resultados obtidos com a utilização do OpenACC foram satisfatórios, apresentando reduções consideráveis no tempo de execução das aplicações desenvolvidas. O OpenACC é uma opção atrativa, pois permite explorar os recursos das GPUs sem exigir modificações extensivas no código fonte. Além disso, o OpenACC simplifica a programação para GPUs, tornando a implementação menos complexa em comparação com outras ferramentas, como por exemplo, CUDA e OpenCL.

Ressalta-se que o OpenACC deve ser utilizado com cuidado, especialmente em aplicações que demandam de um maior desempenho, visto que essa apresentou desempenho inferior quando comparada às ferramentas OpenMP e CUDA. Isso deve-se ao fato dessa ser uma ferramenta de mais alto nível, o que restringe a possibilidade de gerenciamento dos recursos, além de dificultar a identificação de gargalos nas aplicações paralelizadas.

6.1 TRABALHOS FUTUROS

Como possíveis trabalhos futuros sugere-se:

- Comparar os resultados obtidos com as ferramentas OmpSs e StarPU;
- Realizar um perfilamento mais detalhado das implementações em OpenACC, com o objetivo de otimizar e aumentar o desempenho das soluções desenvolvidas;
- Utilizar o OpenACC no desenvolvimento de uma simulação mais realista e complexa, com o objetivo de aprofundar a avaliação dessa ferramenta.

REFERÊNCIAS

AUGONNET, C. **Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective**. Tese (Doutorado) — Bordeaux 1, 2011.

AUGONNET, C.; THIBAULT, S.; NAMYST, R. **StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines**. [S.l.], 2010. 33 p. Disponível em: <https://inria.hal.science/inria-00467677>.

AUGONNET, C. *et al.* StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. **Concurrency and Computation: Practice and Experience**, Wiley, v. 23, n. 2, p. 187–198, 2011. Disponível em: <https://inria.hal.science/inria-00550877>.

Barcelona Supercomputing Center. **OmpSs Specification**. [S.l.], 2019. Acesso em: 11 jun. 2024. Disponível em: <https://pm.bsc.es/ftp/ompss/doc/spec/OmpSsSpecification.pdf>.

BURDEN, R. L.; FAIRES, J. D. **Numerical analysis**. [S.l.]: Brooks Cole, 1997.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)**. [S.l.]: The MIT Press, 2007. ISBN 0262533022.

CHE, S. *et al.* Rodinia: A benchmark suite for heterogeneous computing. In: **2009 IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.: s.n.], 2009. p. 44–54.

CHEN, L.; HUO, X.; AGRAWAL, G. Scheduling methods for accelerating applications on architectures with heterogeneous cores. In: **2014 IEEE International Parallel & Distributed Processing Symposium Workshops**. [S.l.: s.n.], 2014. p. 48–57.

COHN, H. *et al.* Group-theoretic algorithms for matrix multiplication. In: IEEE. **46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)**. [S.l.], 2005. p. 379–388.

CORMEN, T. H. *et al.* **Introduction to algorithms**. [S.l.]: MIT press, 2022.

DABAH, A. *et al.* Hybrid multi-core cpu and gpu-based b&b approaches for the blocking job shop scheduling problem. **Journal of Parallel and Distributed Computing**, v. 117, p. 73–86, 2018. ISSN 0743-7315. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0743731518300820>.

DANALIS, A. *et al.* The scalable heterogeneous computing (shoc) benchmark suite. In: **Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units**. New York, NY, USA: Association for Computing Machinery, 2010. (GPGPU-3), p. 63–74. ISBN 9781605589350. Disponível em: <https://doi.org/10.1145/1735688.1735702>.

DONGARRA, J. J.; LUSZCZEK, P.; PETITET, A. The linpack benchmark: past, present and future. **Concurrency and Computation: practice and experience**, Wiley Online Library, v. 15, n. 9, p. 803–820, 2003.

DURAN, A.; CORBALAN, J.; MARTORELL, X. Ompss: A proposal for programming heterogeneous multi-core architectures. In: IEEE COMPUTER SOCIETY. **Proceedings of the 2011 International Conference on Parallel Processing**. [S.l.], 2011. p. 407–416.

FARBER, R. **CUDA application design and development**. [S.l.]: Elsevier, 2011.

FLYNN, M. J. **Some Computer Organizations and Their Effectiveness**. [S.l.]: IEEE Computer Society Press, 1972.

FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. [S.l.]: Addison-Wesley, 1995.

GOUGH, B. J.; STALLMAN, R. **An Introduction to GCC.** [S.l.]: Network Theory Limited, 2004.

HELLBERG, L.; BHAMIDIPATI, B. **Performance evaluation of Web Workers API and OpenMP**. 2022.

HUGUE, M. M. **Types of Benchmarks**. University of Maryland, Department of Computer Science, 1998. Acesso em: 15 jun. 2024. Disponível em: <https://www.cs.umd.edu/~meesh/cmsc411/website/projects/morebenchmarks/types.html>.

KARIMI, K.; DICKSON, N. G.; HAMZE, F. A performance comparison of cuda and opencl. **arXiv preprint arXiv:1005.2581**, 2010.

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

KIRK, D. B.; HWU, W. mei W. **Programming Massively Parallel Processors**: A hands-on approach. 2. ed. [S.l.]: Elsevier, 2013.

KOPYTOV, A. Sysbench manual. **MySQL AB**, p. 2–3, 2012.

LI, X.; SHIH, P.-C. Performance comparison of cuda and openacc based on optimizations. In: **Proceedings of the 2018 2nd High Performance Computing and Cluster Technologies Conference**. [S.l.: s.n.], 2018. p. 53–57.

LONGBOTTOM, R. **Whetstone benchmark history and results**. 2005.

MEMETI, S. *et al.* Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. In: **Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing**. [S.l.: s.n.], 2017. p. 1–6.

MOHAMED, K. S. Parallel computing: Openmp, mpi, and cuda. In: ____. **Neuromorphic Computing and Beyond: Parallel, Approximation, Near Memory, and Quantum**. Cham: Springer International Publishing, 2020. p. 63–93. ISBN 978-3-030-37224-8. Disponível em: <https://doi.org/10.1007/978-3-030-37224-8_3>.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics**, v. 38, n. 8, p. 114–117, April 1965.

MUNSHI, A. The opencl specification. In: IEEE. **2009 IEEE Hot Chips 21 Symposium (HCS)**. [S.l.], 2009. p. 1–314.

NICHOLS, B.; BUTTLAR, D.; FARRELL, J. **Pthreads programming: A POSIX standard for better multiprocessing**. [S.l.]: "O'Reilly Media, Inc.", 1996.

NVIDIA Corporation. **NVIDIA CUDA C Programming Guide**. 2010. Version 3.2.

____. **NVIDIA CUDA Compiler Driver NVCC**. [S.l.], 2024. <https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/>.

____. **NVIDIA HPC SDK Documentation**. [S.l.], 2024. <https://docs.nvidia.com/hpc-sdk//pdf/hpc249ug.pdf>.

OpenACC.org. **OpenACC Application Programming Interface Version 3.3**. [S.l.], 2022. Disponível em: <https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf>.

____. **OpenACC Programming and Best Practices Guide**. [S.l.], 2022. Disponível em: <https://openacc-best-practices-guide.readthedocs.io/en/latest/index.html>.

PACHECO, P. **An introduction to parallel programming**. [S.l.]: Elsevier, 2011.

SAAD, Y. **Iterative methods for sparse linear systems**. [S.l.]: SIAM, 2003.

SCARPAZZA, D. P.; VILLA, O.; PETRINI, F. Efficient breadth-first search on the cell/be processor. **IEEE Transactions on Parallel and Distributed Systems**, v. 19, n. 10, p. 1381–1395, 2008.

SCOGLAND, T. R. *et al.* Coretsar: Core task-size adapting runtime. **IEEE Transactions on Parallel and Distributed Systems**, v. 26, n. 11, p. 2970–2983, 2015.

SHAFFER, C. A. **A practical introduction to data structures and algorithm analysis**. [S.l.]: Prentice Hall Upper Saddle River, NJ, 1997.

SONG, P. *et al.* Implementation of the cpu/gpu hybrid parallel method of characteristics neutron transport calculation using the heterogeneous cluster with dynamic workload assignment. **Annals of Nuclear Energy**, v. 135, p. 106957, 2020. ISSN 0306-4549. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0306454919304529>.

STALLINGS, W. **Arquitetura e organização de computadores**. 8. ed. [S.l.]: Pearson, 2010.

STERLING, T.; BRODOWICZ, M.; ANDERSON, M. **High performance computing: modern systems and practices**. [S.l.]: Morgan Kaufmann, 2017.

STRATTON, J. A. *et al.* Parboil: A revised benchmark suite for scientific and commercial throughput computing. In: . [s.n.], 2012. Disponível em: <https://api.semanticscholar.org/CorpusID:497928>.

SWAHN, H. **Pthreads and OpenMP: A performance and productivity study**. 2016.

TEAM, S. D. **STARPU User Manual**. [S.l.], 2022. Disponível em: <https://files.inria.fr/starpu/doc/starpu.pdf>.

VOKOLOS, F. I.; WEYUKER, E. J. Performance testing of software systems. In: **Proceedings of the 1st International Workshop on Software and Performance**. [S.l.: s.n.], 1998. p. 80–87.

WEICKER, R. An overview of common benchmarks. **Computer**, v. 23, n. 12, p. 65–75, 1990.

WITKOWSKI, W. **Moore's Law's dead, Nvidia CEO Jensen says in justifying gaming card price hike**. 2022. Acessado em: April 3, 2024. Disponível em: <https://www.marketwatch.com/story/moores-laws-dead-nvidia-ceo-jensen-says-in-justifying-gaming-card-price-hike-11663798618>.

XU, G. *et al.* A distributed parallel computing environment for bioinformatics problems. In: **Sixth International Conference on Grid and Cooperative Computing (GCC 2007)**. [S.l.: s.n.], 2007. p. 593–599.

YANG, C.-T.; HUANG, C.-L.; LIN, C.-F. Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. **Computer Physics Communications**, Elsevier, v. 182, n. 1, p. 266–269, 2011.

YASUI, Y.; FUJISAWA, K. Fast and scalable numa-based thread parallel breadth-first search. In: **2015 International Conference on High Performance Computing & Simulation (HPCS)**. [S.l.: s.n.], 2015. p. 377–385.