XXV Escola Regional de Alto Desempenho da Região Sul

ERAD/RS 2025

MINICURSOS

XXV ERAD/RS 25 anos

Foz do Iguaçu, PR

23, 24 e 25 de Abril de 2025

ORGANIZAÇÃO

FOMENTO

REALIZAÇÃO

PATROCÍNIO

DIAMANTE

OURO

PRATA

BRONZE

LUCAS MELLO SCHNORR
GUILHERME PIEGAS KOSLOVSKI

MINICURSOS DA XXV ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD/RS)

Porto Alegre
Sociedade Brasileira de Computação – SBC
2025

Dados Internacionais de Catalogação na Publicação (CIP)

E74 Escola Regional de Alto Desempenho da Região Sul (25. : 23 – 25 abril 2025 : Foz do Iguaçu)
Minicursos da XXV ERAD-RS 2024 [recurso eletrônico] / coordenadores: Lucas Mello Schnorr; Guilherme Piêgas Koslovski. Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2025.
62 p. : il. : PDF

Modo de acesso: World Wide Web.
Inclui bibliografia
ISBN 978-85-7669-630-8 (e-book)

1. Computação – Brasil – Evento. 2. Processamento de Alto Desempenho. 3. Programação paralela. I. Mello Schnorr, Lucas. II. Koslovski, Guilherme Piegâs. III. Sociedade Brasileira de Computação. VI. Título.

CDU 004(063)

Ficha catalográfica elaborada por Annie Casali – CRB-10/2339

Biblioteca Digital da SBC – SBC OpenLib

 DOI: 10.5753/sbc.16630.8

ERAD/RS 2025

XXV Escola Regional de Alto Desempenho da Região Sul
23, 24 e 25 de abril de 2025
Foz do Iguaçu - PR
https://cradrs.github.io/eradrs2025/

A XXV Escola Regional de Alto Desempenho da Região Sul (ERAD/RS 2025) foi realizada entre os dias 23 e 25 de abril de 2025, no Itaipu Parquetec, em Foz do Iguaçu/PR. A escola é realizada anualmente pela Sociedade Brasileira de Computação (SBC), pela Comissão Especial de Arquitetura de Computadores e Processamento de Alto Desempenho (CE-ACPAD) e pela Comissão Regional de Alto Desempenho da Região Sul (CRAD/RS). Esta edição foi especialmente significativa, pois celebrou os 25 anos da escola, reforçando sua importância na formação de profissionais e pesquisadores da área.

A ERAD/RS tem como público-alvo estudantes, profissionais e pesquisadores que atuam em Computação de Alto Desempenho e áreas correlatas, abrangendo toda a região Sul do Brasil. Seus principais objetivos incluem a qualificação de profissionais no setor, a promoção de um espaço para a apresentação e discussão dos avanços recentes e o aprimoramento do ensino de HPC nas universidades.

A programação da ERAD/RS 2025 contou com diversas atividades, incluindo sessões técnicas com a apresentação de 55 trabalhos nos fóruns de Iniciação Científica e Pós-Graduação, além de uma palestra acadêmica, cinco palestras industriais, uma palestra no Workshop de Inteligência Artificial (WIA), sete minicursos e uma maratona de programação paralela. Para incentivar a participação feminina na área, o evento também promoveu um workshop sobre mulheres na Computação de Alto Desempenho, em parceria com o Women in HPC (WHPC). Como parte das comemorações pelos 25 anos da escola, foi realizado o painel ”25 Anos de ERAD/RS: Legado, Evolução e Perspectivas para o Futuro”, resgatando a trajetória do evento e discutindo os rumos da área.

A organização da ERAD/RS 2025 foi conduzida pelos professores Guilherme Galante (UNIOESTE), Fauzi de Moraes Shubeita e Arthur Francisco Lorenzon (UFRGS) na Coordenação Geral. O Fórum de Iniciação Científica foi coordenado pelos professores Daniel F. Pigatto (UTFPR) e Fabio Rossi (IF-Farroupilha), enquanto o Fórum de Pós-Graduação ficou sob a responsabilidade de Marco Z. Alves (UFPR) e Dalvan Griebler (PUC-RS). Os minicursos foram organizados por Lucas Mello Schnorr (UFRGS) e Guilherme P. Koslovski (UDESC). A maratona de programação paralela foi coordenada por Calebe Bianchini (Mackenzie) e Matheus Serpa (UFRGS), enquanto o workshop Women in HPC foi conduzido pela professora Marcia Pasin (UFSM). A equipe organizadora local foi coordenada pelo professor Antonio Marcos Hashisuca (UNIOESTE) e pela técnica Maria Lucia Honorio dos Santos (UNIOESTE).

Índice

Mensagem da Coordenação Geral iii
Mensagem da Coordenação dos Minicursos iv
Comitês Organizadores v
Minicursos vi

 DOI: 10.5753/sbc.16630.8

Mensagem da Coordenação Geral

É com imenso prazer e entusiasmo que saudamos e damos as boas vindas à vigésima quinta edição da Escola Regional de Alto Desempenho da Região Sul (ERAD/RS 2025), que neste ano aconteceu de forma presencial entre os dias 23 e 25 de abril de 2025, em Foz do Iguaçu, Paraná. A ERAD/RS é um evento anual, promovido pela Sociedade Brasileira de Computação (SBC), por meio da Comissão Regional de Alto Desempenho da Região Sul (CRAD/RS), desde 2001.

Coordenar e organizar um evento da magnitude da ERAD/RS é um grande desafio, que só foi realizado com a colaboração de todos os membros da organização e apoio de diversas instituições. Especificamente, neste ano a ERAD/RS contou com fomento da Fundação Araucária de apoio ao desenvolvimento científico e tecnológico do estado do Paraná, além de apoio governamental através da Pró-Reitoria de Extensão da Unioeste e da Secretaria da Ciência, Tecnologia e Ensino Superior do Paraná. Ainda, a ERAD/RS contou com patrocínio da empresa VIRCOS na categoria diamante, enquanto na categoria ouro foi apoiada pelas empresas LexisNexis, SDC/Intel, SCHERM/Quantum e HPE/AMD. Tivemos também patrocínio na categoria prata da Fineasy Tech e NVidia/Dell e na categoria bronze da Kaiak.

Quanto as instituições organizadores, a ERAD/RS foi organizada pela UNIOESTE e UFRGS, contando com apoio da UFPR, UFSM, Mackenzie, UDESC, UTFPR, PUCRS, IF-Farroupilha e Instituto Friol. Também é importante citar o Parquetec Itaipu que gentilmente cedeu seus espaços e infraestrutura para a realização do evento.

Agradecemos, em especial, a todos os(as) autores(as) que submeteram seus trabalhos às sessões técnicas, às empresas patrocinadoras e aos diversos convidados que aceitaram prontamente nosso convite e, sem dúvida alguma, engrandeceram a ERAD/RS com as suas participações.

Por fim, agradecemos todos os coordenadores dos eventos da ERAD/RS, que tomaram para si diversos encargos e os conduziram com muito sucesso.

Obrigado pela presença de todos.

Esperamos que a ERAD/RS 2025 seja muito proveitosa.

Um abraço,

Guilherme Galante (UNIOESTE),
Fauzi de Moraes Shubeita (SETREM) e
Arthur Francisco Lorenzon (UFRGS)
Coordenadores gerais da ERAD/RS 2025

 DOI: 10.5753/sbc.16630.8

Mensagem da Coordenação dos Minicursos

A Escola Regional de Alto Desempenho da Região Sul (ERAD/RS) é um evento anual promovido pela Sociedade Brasileira de Computação (SBC) e pela Comissão Regional de Alto Desempenho da Região Sul (CRAD/RS). A escola, que neste ano completa os seus vinte e cinco anos, foi realizada entre os dias 23 e 25 de abril de 2025, no Itaipu Parquetec, na cidade de Foz do Iguaçu – PR. Essa foi a primeira vez que a escola é realizada no sudoeste do estado do Paraná. Um dos objetivos da ERAD/RS é qualificar profissionais da região sul nas diversas áreas do Processamento de Alto Desempenho (PAD). Com este intuito, todo o ano, são selecionados minicursos introdutórios, intermediários e avançados em tópicos de interessa à comunidade.

O Livro de Minicursos da XXV Escola Regional de Alto Desempenho da Região Sul (ERAD/RS) apresenta a contribuição de pesquisadores em computação paralela do Brasil. Constituído de três capítulos, todos focados em computação de alto desempenho, o livro possui assuntos convergentes abordando sobretudo APIs de programação para aceleradores. No primeiro capítulo, intitulado "Programação de Alto Desempenho em GPUs com C++", o autor fornece um apanhado geral das APIs de programação paralela para GPUs em C++, indo desde aquelas de mais baixo nível, como CUDA e OpenCL, até soluções mais portáveis como OpenMP e SYCL. No capítulo dois, "Diretivas Paralelas de Programação", os autores apresentam as APIs de programação paralela OpenMP e OpenACC. Estas são consideradas métodos mais simples de produzir operações paralelas, permitindo um acesso inicial facilitado a mecanismos de paralelização. No terceiro capítulo, os autores de "Programação Avançada de Múltiplas GPUs com OpenACC", os autores trazem um enfoque no emprego da API OpenACC para programação com aceleradoras, inclusive com aspectos de heterogeneidade. Além destes minicursos para os quais apresentamos os textos dos autores, a ERAD/RS contou também com minicursos que foram unicamente apresentados no evento.

Os coordenadores dos minicursos agradecem aos autores desses minicursos, por compartilharem seus conhecimentos através da submissão de propostas de alto nível para esta edição da escola, aos coordenadores e organizadores da ERAD/RS, pelo apoio dado na realização do evento, além de desejar uma boa ERAD/RS a todas as pessoas que puderem participar presencialmente!

Lucas Mello Schnorr (UFRGS) e Guilherme P. Koslovski (UDESC)
Coordenadores dos Minicursos da ERAD/RS 2025

 DOI: 10.5753/sbc.16630.8

Comitês Organizadores

Coordenação Geral

- Guilherme Galante (Unioeste)
- Fauzi de Moraes Shubeita (SETREM)
- Arthur Francisco Lorenzon (UFRGS)

Coordenação Local

- Antonio Marcos Hashisuca (UNIOESTE)
- Maria Lucia Honorio dos Santos (UNIOESTE)

Fórum de Pós-Graduação

- Marco Z. Alves (UFPR)
- Dalvan Griebler (PUC-RS)

Fórum de Iniciação Científica

- Daniel F. Pigatto (UTFPR)
- Fabio Rossi (IF-Farroupilha)

Minicursos

- Lucas Mello Schnorr (UFRGS)
- Guilherme P. Koslovski (UDESC)

Minicursos

Minicurso 1

Programação de Alto Desempenho em GPUs com C++ 1
Pablo Alessandro Santos Hugen

Minicurso 2

Diretivas Paralelas de Programação 15
Claudio Schepke, Vinícius Garcia Pinto

Minicurso 3

Programação Avançada de Múltiplas GPUs com OpenACC 30
Calebe P. Bianchini, Evaldo B. Costa, Gabriel P. Silva

 DOI: 10.5753/sbc.16630.8

Capítulo 1

Programação de Alto Desempenho em GPUs com C++[1]

Pablo Alessandro Santos Hugen[2]

Resumo

A programação paralela heterogênea tem se consolidado como uma abordagem essencial na área de HPC*, especialmente com o uso crescente de* GPGPUs*. Nesse cenário, a linguagem de programação* C++ *desempenha um papel central. Contudo, a diversidade e a complexidade dos modelos e ferramentas disponíveis representam uma barreira significativa para iniciantes na área. Assim, o objetivo deste minicurso é fornecer uma visão geral das principais ferramentas e modelos de programação paralela para GPUs em* C++*, destacando suas características, peculiaridades e aplicabilidades.*

1.1. Introdução

Nos últimos anos, a Computação de Alto Desempenho (*HPC – High-Performance Computing*) emergiu como uma ferramenta fundamental para a resolução de problemas em diversos campos, desde a simulação de fenômenos físicos (HONG; JANG; JEONG, 2022) até sua aplicação em vários subcampos da Inteligência Artificial, como a análise de dados e a aprendizagem de máquina (ELSEBAKHI et al., 2015). Paralelamente, Barlas (2014) reitera que as Unidades de Processamento Gráfico de Propósito Geral (*GPGPUs, General Purpose Graphics Processing Units*) têm desempenhado um papel cada vez mais crítico nessa área, oferecendo capacidades de processamento massivamente paralelo. Consequentemente, a complexidade dos problemas modernos tem expandido continuamente os limites da computação de alto desempenho. Como exemplo, destaca-se o uso recente da aceleração de múltiplas *GPGPUs* no treinamento de Modelos de Linguagem Massivos (*LLMs, Large Language Models*), que utiliza técnicas de paralelismo de dados para escalar o treinamento desses modelos em *clusters* de *GPUs*.

[1] DOI: 10.5753/sbc.16630.8.1

[2] Formado em Ciência da Computação pela Universidade Estadual do Oeste do Paraná (2024), atua profissionalmente como Programador de Sistemas. Entusiasta de programação de baixo nível, computação paralela/distribuída, FOSS e Data Oriented Design.

Nesse contexto, a linguagem de programação C++ desempenha um papel central no desenvolvimento de aplicações para HPC, oferecendo um equilíbrio entre abstração de alto nível e controle eficiente sobre os recursos de hardware. Especialmente no caso das *GPUs*, a linguagem serviu como base para a criação de todo o ecossistema de programação paralela para esses aceleradores. Assim, o objetivo deste minicurso é apresentar os principais modelos de programação paralela em *GPUs* disponíveis na linguagem. A Seção 1.2 explica brevemente o modelo de computação das *GPUs* e a Seção 1.3 enumera as alternativas em C++ para programação desses aceleradores.

1.2. Modelo de computação das GPUs

Um pré-requisito essencial para escrever algoritmos e estruturas de dados eficientes em *GPUs* é compreender sua arquitetura e como ela difere do modelo tradicional das *CPUs*. Estas, projetadas para execução sequencial, priorizam a redução da latência por meio de otimizações como pipeline de instruções, execução fora de ordem, especulativa e caches multinível (HENNESSY; PATTERSON, 1990; HENNESSY; PATTERSON, 2018). Por outro lado, as *GPUs* foram projetadas para alto *throughput* e paralelismo massivo, mesmo com maior latência na execução de instruções (OWENS, 2007). Esse design, influenciado por aplicações em gráficos, computação numérica e aprendizado de máquina, prioriza cálculos de álgebra linear em alta velocidade. Como visto na Figura 1.1, enquanto *CPUs* dedicam mais área do chip a caches e *UCs* para reduzir a latência, *GPUs* maximizam o poder de cálculo com um grande número de *ULAs*, sacrificando latência para otimizar *throughput* (OWENS, 2007).

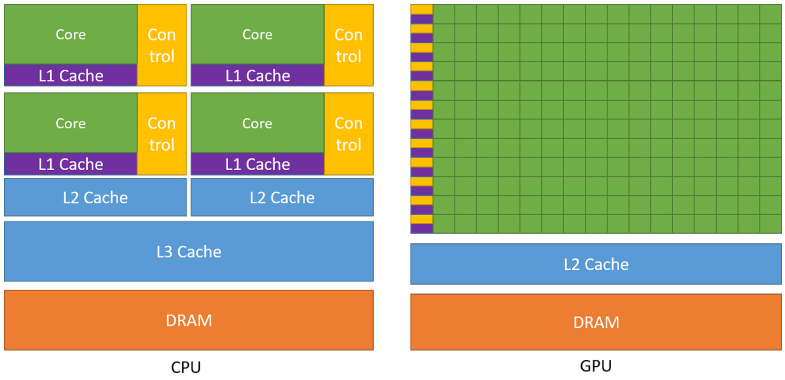


Figura 1.1: Comparativo arquitetural entre *CPUs* e *GPUs*. Fonte: (NVIDIA, 2023a)

Assim, as *GPUs* toleram altas latências mantendo desempenho superior ao escalonar milhares de *threads* em paralelo. Embora instruções individuais possam ser lentas, a alternância eficiente entre *threads* garante o uso contínuo dos recursos computacionais. Como destacado por Owens et al. (2008), enquanto algumas *threads* aguardam resultados, outras são executadas, maximizando a ocupação das unidades de processamento e o *throughput*. A Figura 1.2 ilustra essa organização e a execução paralela dentro da *GPU*. Internamente, a Unidade de Processamento Gráfico é composta por um conjunto de Multiprocessadores de Fluxo (*SMs, Streaming Multiprocessors*) (OWENS, 2007), com cada *SM* possuindo uma quantidade de memória compartilhada. Esses *SMs* são fundamentais para o desempenho paralelo da *GPU*, permitindo o processamento simultâneo massivo.

Por sua vez, cada *SM* é composto por vários núcleos de processamento. Estes núcleos, frequentemente referidos como *threads*, operam de maneira paralela. Importante destacar que dentro de cada *SM*, todas essas *threads* compartilham memória e outros recursos de processamento, facilitando a execução de tarefas paralelas de forma eficiente.

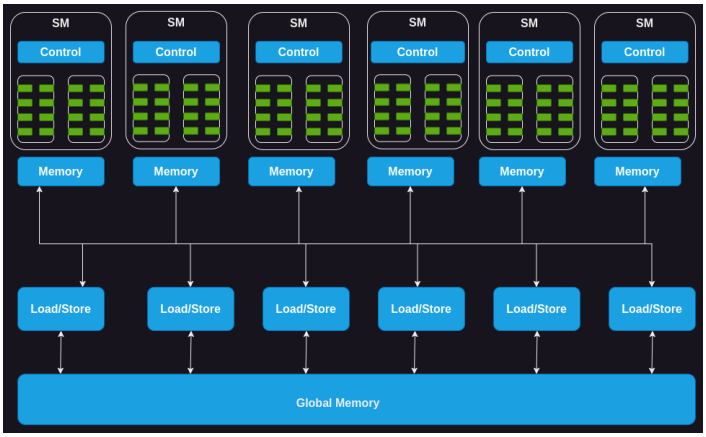


Figura 1.2: Arquitetura do Modelo de computação das *GPUs*. Fonte: (NVIDIA, 2023a).

Esses aceleradores possuem uma hierarquia de memórias complexa e otimizada para desempenho (MEI; CHU, 2016). A Figura 1.3 ilustra a organização da memória em um *SM*, incluindo registradores privados por *thread* e *caches* de constantes, que exigem declaração explícita para otimização. Cada *SM* possui uma memória compartilhada (*scratchpad*) de baixa latência, útil para reduzir acessos à memória global e sincronizar *threads* (OWENS, 2007). O cache L1 armazena dados acessados com frequência do cache L2, que é compartilhado entre os *SMs* (PICCHI; ZHANG, 2015). A memória global externa, como a *DRAM* de alta largura de banda na arquitetura *Hopper* da Nvidia (CHOQUETTE, 2023), tem alta latência, mas seu impacto é reduzido pelo escalonamento eficiente das *threads* e pelo uso de caches.

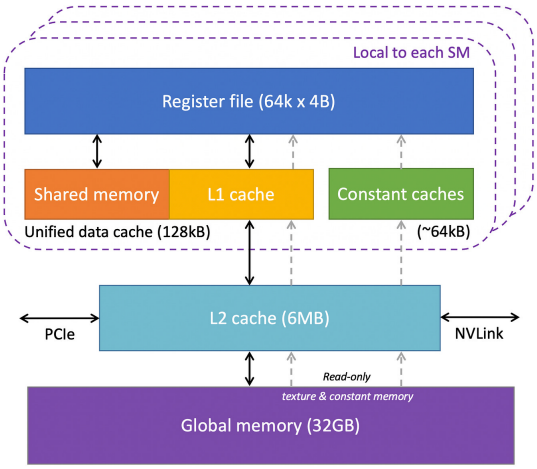


Figura 1.3: Arquitetura de memória das *GPUs*. Fonte: (NVIDIA, 2023a)

A Memória Unificada permite que *CPUs* e *GPUs* acessem um mesmo endereço de memória (NVIDIA, 2023b). O driver *CUDA* e o *hardware* gerenciam automaticamente

a migração das páginas de memória quando necessário. Esse mecanismo é vantajoso para aplicações com padrões de acesso dispersos, evitando a necessidade de pré-carregar grandes arrays ou recorrer a acessos de alta latência fora do dispositivo (*Zero Copy*), com suporte a falhas de página, onde apenas as páginas necessárias são migradas.

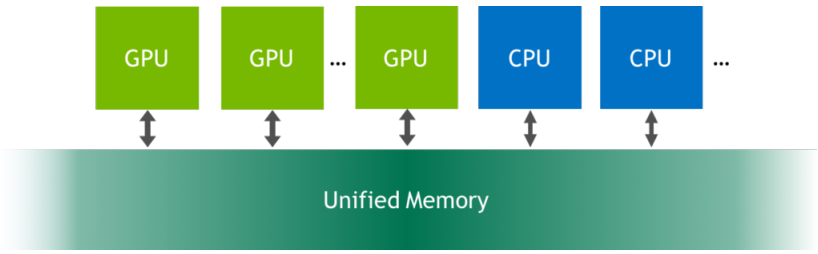


Figura 1.4: Arquitetura de Memória Unificada CUDA. Fonte: (NVIDIA, 2023a)

1.3. Modelos de programação paralela em GPUs disponíveis no C++

As *GPUs*, inicialmente projetadas para processamento gráfico, evoluíram para se tornar componentes essenciais em diversas aplicações computacionais. Esse avanço acelerado gerou um ecossistema fragmentado, repleto de modelos de programação paralela, compiladores e ferramentas, cada um oferecendo diferentes níveis de abstração e otimização.

Nesse contexto, a linguagem *C++* se firmou como a principal escolha para o desenvolvimento em *GPUs*, fornecendo suporte a uma ampla gama de modelos de programação paralela. Sua combinação de flexibilidade e alto desempenho possibilitou a criação de APIs e frameworks que vão desde abordagens de baixo nível, como *Compute Shaders* e CUDA, até soluções mais portáveis e genéricas, como OpenCL, OpenMP e SYCL. Essa diversidade reflete a necessidade de equilibrar controle granular sobre o hardware com portabilidade e facilidade de uso, permitindo atender a diferentes domínios computacionais. As próximas seções exploram esses modelos em detalhes.

1.3.1. Compute Shaders

Os *shaders* são pequenos programas executados diretamente na *GPU*, projetados para processar e manipular dados em estágios específicos do *pipeline gráfico*. Os *vertex shaders* transformam as coordenadas de vértices em um espaço tridimensional, enquanto os *fragment shaders* determinam a cor, textura e iluminação de cada pixel antes da renderização final. No entanto, o *pipeline* gráfico tradicional nem sempre é suficiente para atender a demandas computacionais que vão além da renderização, como simulações físicas e processamento de grandes volumes de dados. Para suprir essa necessidade, *APIs* gráficas modernas introduziram os *Compute Shaders*, que operam fora do *pipeline* e permitem a execução de cálculos arbitrários diretamente na *GPU* usando a mesma *API* gráfica. A introdução desse tipo de *shader* possibilitou o uso da *GPU* para tarefas gerais, expandindo suas aplicações para além da renderização gráfica. A Figura 1.5 ilustra a estrutura do *pipeline* gráfico do *OpenGL* e a integração dos *compute shaders* em diferentes estágios.

No *OpenGL*, os *compute shaders* diferem dos demais por não terem entradas ou saídas fixas, exigindo que busquem e armazenem dados manualmente (em texturas, images ou *uniforms*). Eles operam divididos em *work groups*, definidos pelo usuário na

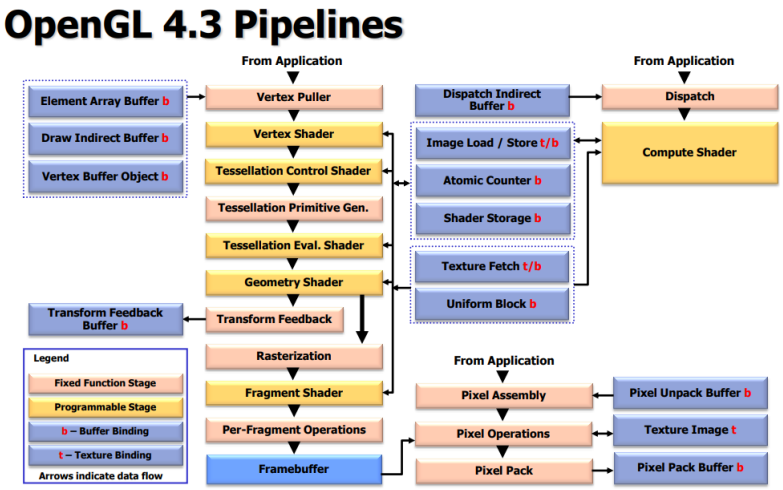


Figura 1.5: Estrutura do *pipeline* gráfico e *compute shaders*. Fonte: (LICHTENBELT, 2012)

execução. Cada grupo contém várias invocações do *shader* (definidas pelo *local size*) que podem se comunicar entre si, mas não com outros grupos. Como a ordem de processamento é indefinida, não há garantia de sequência fixa. Essa estrutura é ideal para tarefas como processamento de imagens, onde cada grupo lida paralelamente com regiões da imagem (como demonstrado no Código 1). Para executar um *compute shader* em C++, é necessário ativá-lo com `glUseProgram` ou `glBindProgramPipeline` e então despachar a computação com `glDispatchCompute`. Esse comando define quantos *work groups* serão executados nas três dimensões. Também é possível despachar a computação de forma indireta, lendo os valores do número de grupos a partir de um *Buffer Object*, usando `glDispatchComputeIndirect`.

1.3.2. CUDA e HIP

Lançada pela *NVIDIA* em novembro de 2006, *CUDA (Compute Unified Device Architecture)* é uma plataforma de computação paralela que permite o uso de suas *GPUs* para processamento geral. Ela fornece uma API baseada em C/C++ para que desenvolvedores possam escrever programas que executem de forma massivamente paralela, aproveitando milhares de núcleos disponíveis nas GPUs modernas. Já o *HIP (Heterogeneous-Compute Interface for Portability)*, é a solução equivalente da *AMD*, que além de permitir a programação de *GPUs* da própria *AMD*, também fornece uma camada de compatibilidade para código *CUDA*, facilitando a portabilidade entre os dois fornecedores de *hardware*.

Essas *APIs* são baseadas em *kernels* (código executado na *GPU*), que são funções chamadas pelo código *host* (código executado na *CPU*) e executadas *N* vezes em paralelo por *N* diferentes *threads CUDA*. Um *kernel* é definido usando o especificador de declaração *__global__*, e o número de *threads* que o executam para uma determinada chamada é especificado por meio de uma sintaxe específica (<<< ... >>>). Cada *thread* recebe um *ID* único, acessível dentro do próprio *kernel*.

Como discutido anteriormente na Seção 1.2, o conceito de *hierarquia de threads e memória* na arquitetura das *GPUs* é essencial para obter um bom desempenho na exe-

```
#version 460

layout(local_size_x = 16, local_size_y = 16) in;
layout(binding = 0, rgba8) uniform readonly image2D inputImage;
layout(binding = 1, rgba8) uniform writeonly image2D outputImage;

void main() {
    ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy);
    vec4 pixelColor = imageLoad(inputImage, pixelCoords);
    vec4 invertedColor = vec4(vec3(1.0) - pixelColor.rgb, pixelColor.a);
    imageStore(outputImage, pixelCoords, invertedColor);
}
```

```
GLuint createComputeShader() {/*...*/ }

auto main() -> int {
    /*...*/
    GLuint computeShaderProgram = createComputeShader();
    /*...*/
    glUseProgram(computeShaderProgram);
    glDispatchCompute(16, 16, 1);
    glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
    /*...*/
}
```

Código 1: Exemplo de *compute shader (glsl)* para inverter cores de imagens.

cução de *kernels*. O Código 2 exemplifica um *kernel* simples de adição de matrizes em *CUDA*. Nele, podemos observar a organização das *threads* em blocos e sua indexação dentro da *grid* de execução. Cada bloco de *threads* é identificado por um índice *(blockIdx.x e blockIdx.y)*, enquanto cada *thread* dentro do bloco possui um índice local *(threadIdx.x e threadIdx.y)*, permitindo calcular sua posição global na grade ao combiná-los com o tamanho do bloco *(blockDim.x e blockDim.y)*.

O *nvcc* é o compilador *CUDA* proprietário da *NVIDIA* responsável por traduzir código-fonte contendo *kernels* em código executável para a *GPU*. Ele separa o código *host* do código *device*, compilando o código *device* para *PTX* ou *cubin* e modificando o código *host* para gerenciar as chamadas aos *kernels*. A compilação pode ser feita de forma *offline*, gerando código binário antecipadamente, ou *just-in-time*, onde o código *PTX* é compilado pelo driver durante a execução da aplicação, garantindo compatibilidade com novas arquiteturas. O uso do *nvcc* simplifica esse processo, fornecendo uma interface unificada para a compilação e a geração de código otimizado para *GPUs*.

```
__global__ void matadd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    /*...*/
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    matadd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    /*...*/
}
```

Código 2: Kernel de adição de matrizes em *CUDA*.

Todo o runtime *CUDA* exposto pelo compilador é baseado em uma *API* de baixo nível escrita em C, que também pode ser acessada pela aplicação. Essa interface oferece um controle adicional ao expor conceitos como *contextos*, semelhantes a processos do sistema operacional, e *módulos*, que funcionam como bibliotecas carregadas dinamicamente no dispositivo. A maioria das aplicações não utiliza essa *API* de baixo nível, pois não necessita dessa quantidade extra de controle, o que deixa o código mais simples.

1.3.3. OpenMP e OpenACC

O OpenMP (*Open Multi-Processing*) é uma *API* para programação paralela em sistemas de memória compartilhada, como *multi-core CPUs*. Por meio de diretivas de compilador (*pragmas*), ele facilita a paralelização de loops e seções de código com mínima modificação no código sequencial, permitindo distribuir o trabalho entre múltiplas *threads*. Além disso, o OpenMP simplifica o acesso à memória compartilhada, um desafio comum em outros ambientes, onde pode levar a *data races* e problemas semelhantes. Versões recentes também introduziram suporte para offloading para GPUs e outros aceleradores, expandindo suas capacidades além das *CPUs*.

Inicialmente desenvolvido pela empresas *Portland Group (PG)*, *Cray* e *NVIDIA*, o OpenACC (*Open Accelerators*), assim como o OpenMP, utiliza diretivas de compilador para paralelizar o código, mas é focado em aceleração com *GPUs* e outros dispositivos especializados. Com o OpenACC, os desenvolvedores podem facilmente transferir trechos de código e para diferentes tipos de aceleradores, aproveitando as especialidades e poder de computação paralela de cada um deles. Embora também ofereça uma abordagem de alto nível, o OpenACC é especialmente voltado para quem deseja acelerar aplicações sem precisar se aprofundar em conceitos de muito baixo nível.

Apesar de ambas as especificações definirem uma *API* declarativa baseada em diretivas de compilação, o OpenACC foi projetado desde o início para plataformas heterogêneas, como *GPUs* e outros aceleradores, enquanto o OpenMP passou a suportar esses dispositivos apenas a partir da versão 4. Também, o *OpenACC* foca em fornecer computação de alto desempenho de maneira simples e abstraída, especialmente para a comunidade científica. A seguir, o Código 3 mostra a soma de arrays paralela utilizando *OpenACC* e *OpenMP*.

```
template <typename T, size_t N>
auto vector_add(const std::array<T, N>& A, const std::array<T, N>& B,
                std::array<T, N>& C) -> void {
#pragma acc parallel loop copyin(A, B) copyout(C)
  for (auto i = 0ull; i < N; ++i) C[i] = A[i] + B[i];
}
```

```
template <typename T, size_t N>
auto vector_add(const std::array<T, N>& A, const std::array<T, N>& B,
                std::array<T, N>& C) -> void {
#pragma omp target teams distribute parallel for map(to : A, B) map(from : C)
  for (auto i = 0ull; i < N; ++i) C[i] = A[i] + B[i];
}
```

Código 3: Exemplo de adição de arrays paralelo usando *OpenACC* e *OpenMP*.

Ambos os códigos são compilados com compiladores comuns como o *GCC* e o *Clang*, utilizando as flags específicas: `-fopenacc` e `-fopenmp`, respectivamente.

1.3.4. OpenCL

A Open Computing Language (*OpenCL*) é uma especificação aberta para programação paralela heterogênea desenvolvida pela *Khronos Group*, permitindo a execução de código em *CPUs*, *GPUs*, e vários outros aceleradores. Desenvolvida para alto desempenho, a *API* oferece portabilidade e controle de baixo nível entre diferentes arquiteturas de hardware e é amplamente utilizada em aplicações que exigem processamento massivo de dados, como simulações científicas e processamento de imagens. O Código 4 ilustra o controle de baixo nível ao implementar um kernel para a redução de arrays em *OpenCL*.

O objetivo dessa *API* é fornecer uma camada de abstração próxima ao hardware, suficientemente flexível para ser utilizada tanto no desenvolvimento direto quanto como destino de compilação por compiladores e *frameworks* de mais alto nível. A Figura 1.6 ilustra a importância dessa *API* como destino de *frameworks* como o *SYCL* ou até mesmo como alvo de compiladores como o *clang*, por meio da representação intermediária *SPIR*.

```
__kernel void sum(
    __global const double *input,
    __global double *partialSums,
    __local double *localSums) {
  uint local_id = get_local_id(0);
  uint group_size = get_local_size(0);
  localSums[local_id] = input[get_global_id(0)];

  for (uint stride = group_size / 2; stride > 0; stride /= 2) {
    barrier(CLK_LOCAL_MEM_FENCE);

    if (local_id < stride)
        localSums[local_id] += localSums[local_id + stride];
  }

  if (local_id == 0) partialSums[get_group_id(0)] = localSums[0];
}
```

Código 4: Kernel de redução (soma) de arrays para *OpenCL*.

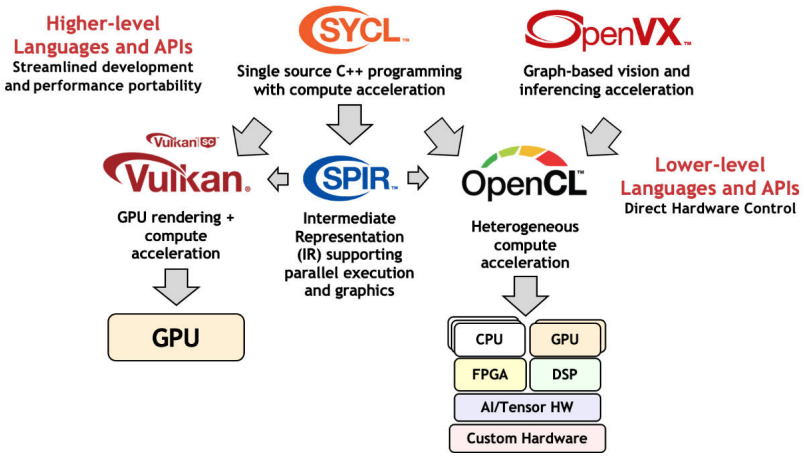


Figura 1.6: Ecossistema do *OpenCL*. Fonte: (The Khronos Group, 2025).

1.3.5. SYCL

Também desenvolvida pelo *Khronos Group*, o *SYCL* é uma especificação multi-plataforma focada na abstração de algoritmos em diferentes tipos de aceleradores, com alta expressividade com mínima modificação no código. Delegar parte do código para aceleradores específicos é comum na computação de alto desempenho, mas exige conhecimento de bibliotecas e modelos de programação específicos para cada hardware. O *SYCL* resolve isso ao fornecer uma abstração comum para a programação heterogênea, maximizando a conformidade com o padrão *C++*. Sua popularidade resultou no surgimento de implementações como **Open SYCL**, **neoSYCL**, **triSYCL** e **Intel® oneAPI tools**.

Essas implementações são na maioria das vezes compatíveis, mas variam em recursos devido a diferenças no desenvolvimento e foco arquitetural, porém todas suportam *CPUs* populares. O Código 5 mostra um exemplo de adição de arrays paralelo em *GPU* usando o *SYCL* (implementação *oneAPI*). O uso de *filas* de trabalho e *seletores de dispositivo* deixa o códio genérico ao ponto de, se mudarmos o seletor para `cpu_selector_v` ou `fpga_selector_v` a função executará no respectivo dispositivo (não necessáriamente com o mesmo desempenho, levando em conta as características de cada um).

```
1  #include <sycl/sycl.hpp>
2
3  using namespace sycl;
4
5  auto vecadd(queue &q, const int *a, const int *b, int *sum, size_t size)
6      -> void {
7    range<1> num_items{size};
8    auto e = q.parallel_for(
9          num_items,
10         [=](auto i) { sum[i] = a[i] + b[i]; }
11     );
12   e.wait();
13 }
14
15 auto main(int argc, char *argv[]) -> int {
16   constexpr auto array_size = 10000;
17   auto selector = gpu_selector_v;
18
19   sycl::queue q(selector);
20
21   int *a = malloc_shared<int>(array_size, q);
22   int *b = malloc_shared<int>(array_size, q);
23   int *sum = malloc_shared<int>(array_size, q);
24
25   vecadd(q, a, b, sum, array_size);
26
27   free(a, q);
28   free(b, q);
29   free(sum, q);
30 }
```

Código 5: Exemplo de adição de arrays paralelo usando *SYCL*.

O *SYCL* vem evoluindo com a intenção de influenciar a direção do *ISO C++* em torno da computação heterogênea, criando pontos de prova e features que podem ser considerados no contexto da evolução e adoção na linguagem.

1.3.6. std::exec: Execução de tarefas assíncronas/heterogêneas diretamente no C++

A maturidade do *C++* no ecossistema de *HPC* levou ao desenvolvimento de modelos de programação paralela de alto nível diretamente baseados na linguagem, reduzindo o tempo e o esforço necessários para manter e implantar aplicações, ao mesmo tempo em que garante portabilidade e desempenho com uma única base de código (DEAKIN; MCINTOSH-SMITH, 2020). Esses modelos têm como objetivo possibilitar a programação paralela heterogênea com desempenho próximo ao nativo (BREYER; CRAEN; PFLÜGER, 2022).

O padrão *C++17* introduziu algoritmos paralelos síncronos (bloqueantes) na biblioteca padrão de *C++*, com suporte para *CPUs* multi-core e *GPUs* com desempenho competitivo (LIN; DEAKIN; MCINTOSH-SMITH, 2022). Esse modelo é inerentemente síncrono, bloqueando a execução do programa até que as tarefas paralelas sejam concluídas, limitando o paralelismo concorrente (LIN; DEAKIN; MCINTOSH-SMITH, 2022). Para resolver isso, os esforços estão focados no desenvolvimento do modelo de programação paralela assíncrona (não bloqueante) **std::exec** para o próximo padrão *C++26*, facilitando a execução assíncrona flexível e eficiente de tarefas usando abstrações como *senders*, *receivers* e *schedulers* (DOMINIAK et al., 2024). Um exemplo de execução assíncrona nesse modelo pode ser visto no Código 6.

```
1 const auto fn1 = [...](auto i) noexcept {/*...*/};
2 const auto fn2 = [...](auto i) noexcept {/*...*/};
3
4 const auto work = stdexec::when_all(
5     stdexec::just() | exec::on(gpu.get_scheduler(),
6                         stdexec::bulk(N, insert_s_human)),
7     stdexec::just() | exec::on(cpu.get_scheduler(),
8                         stdexec::bulk(N, insert_s_mosquito)))
9 stdexec::sync_wait(work);
```

Código 6: Exemplo de execução assíncrona com std::exec.

Nas abordagens tradicionais de programação paralela e concorrente, utiliza-se threads para paralelismo e primitivas de sincronização (como variáveis atômicas e *mutexes*) para evitar interações incorretas, como *race conditions* e *deadlocks*. Podemos pensar nesse modelo como "desestruturado", onde é difícil ter um raciocínio local sobre uma tarefa, pois é preciso considerar o programa como um todo para determinar a sincronização necessária. Com isso, esse *framework* adota o conceito de concorrência estruturada, onde um programa pode ser decomposto em tarefas independentes, executadas concorrentemente e coordenadas de forma assíncrona, abstraindo a programação paralela heterogênea dentro de um mesmo conjunto de abstrações. A Figura 1.7 ilustra os principais conceitos disponíveis no *framework*.

Um *sender* é uma entidade que descreve uma tarefa concorrente. Essa tarefa possui um único ponto de entrada e um ponto de saída, com três possíveis resultados de finalização: sucesso (com retorno de um valor), erro (exceção) ou cancelamento (sem

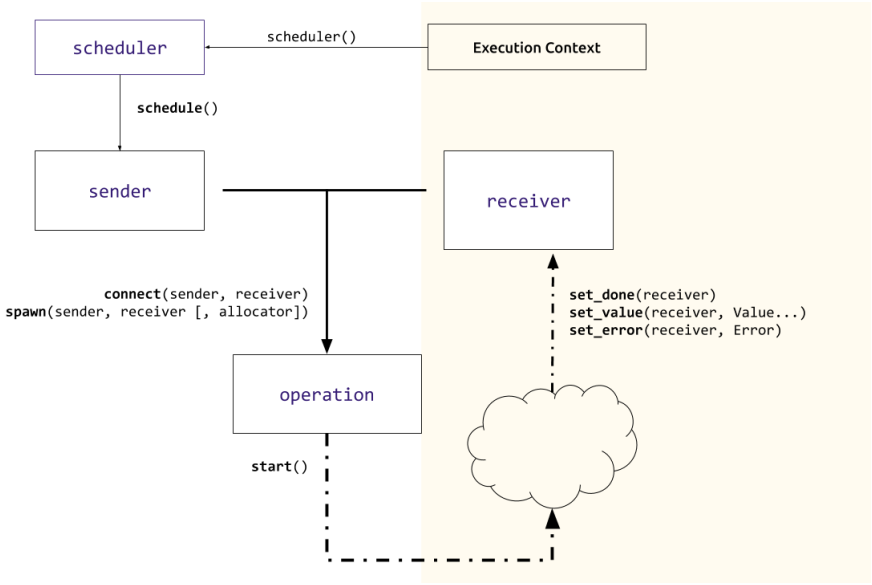


Figura 1.7: Framework *std::exec*. Fonte: (JABOT, 2019)

retorno). Os schedulers abstraem o local de execução dessas tarefas, possibilitando que sejam realizadas em *CPUs*, *GPUs* ou outros dispositivos, otimizando o uso dos recursos disponíveis. Por fim, os receivers são responsáveis por capturar o resultado de um sender e processá-lo conforme o tipo de conclusão. Com esses blocos de construção, é possível desenvolver programas paralelos e concorrentes de forma estruturada, clara e eficiente. A implementação atual do *framewrk* é fornecida pela *NVIDIA* no compilador *nvc++*, parte do *NVIDIA HPC SDK*. Ela inclui *schedulers* otimizados para as suas *GPUs* proprietárias, inclusive com suporte a *multi-GPUs*.

1.4. Considerações finais

A programação paralela em *GPUs* com *C++* é essencial para o desenvolvimento de aplicações de alto desempenho que aproveitem o máximo do hardware moderno, mas requer o domínio de diversos modelos e ferramentas. Este minicurso apresentou as principais abordagens, desde *APIs* de baixo nível, como *CUDA* e *OpenCL*, até soluções mais portáveis, como *SYCL* e *OpenMP*. Cada modelo tem suas vantagens e desafios, permitindo a escolha mais adequada para cada caso de uso específico. Com esse conhecimento, espera-se que os participantes adquiram uma base sólida sobre as alternativas para explorar e desenvolver aplicações eficientes em *GPUs*. Para aprofundar o entendimento sobre o tema, recomenda-se a leitura do clássico trabalho de Barlas (2014) e das outras referências citadas, além das páginas de comunidade e documentação das ferramentas apresentadas.

Referências

BARLAS, G. *Multicore and GPU Programming: An integrated approach*. [S.l.]: Elsevier, 2014.

BREYER, M.; CRAEN, A. V.; PFLÜGER, D. A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor

hardware. p. 1–12, 2022.

CHOQUETTE, J. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro*, IEEE, 2023.

DEAKIN, T.; MCINTOSH-SMITH, S. Evaluating the performance of hpc-style sycl applications. p. 1–11, 2020.

DOMINIAK, M. et al. *P2300R10: std::execution*. 2024. ISO/IEC JTC1/SC22/WG21. Available from Internet: <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html>.

ELSEBAKHI, E. et al. Large-scale machine learning based on functional networks for biomedical big data with high performance computing platforms. *Journal of Computational Science*, Elsevier, v. 11, p. 69–81, 2015.

HENNESSY, J.; PATTERSON, D. A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced. 2018.

HENNESSY, J. L.; PATTERSON, D. A. Computer architecture. Los Altos, CA (USA); Morgan Kaufman Publishers, Inc., 1990.

HONG, S.; JANG, G.; JEONG, W.-K. Mg-fim: A multi-gpu fast iterative method using adaptive domain decomposition. *SIAM Journal on Scientific Computing*, SIAM, v. 44, n. 1, p. C54–C76, 2022.

JABOT, C. *A Universal Async Abstraction for C++*. 2019. Accessed: 2025-03-26. Available from Internet: <https://cor3ntin.github.io/posts/executors/>.

LICHTENBELT, B. *Latest news and features in OpenGL*. 2012. Presented at SIGGRAPH 2012, Los Angeles, CA, USA. Available from Internet: <https://www.khronos.org/assets/uploads/developers/library/2012-siggraph-opengl-bof>.

LIN, W.-C.; DEAKIN, T.; MCINTOSH-SMITH, S. Evaluating iso c++ parallel algorithms on heterogeneous hpc systems. p. 36–47, 2022.

MEI, X.; CHU, X. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 28, n. 1, p. 72–86, 2016.

NVIDIA. *CUDA C++ Programming Guide*. 2023. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. [Acessado em 19/01/2024].

NVIDIA. *CUDA C++ Unified Memory Guide*. 2023. <https://developer.nvidia.com/blog/unified-memory-cuda-beginners/>. [Acessado em 19/01/2024].

OWENS, J. Gpu architecture overview. In: *ACM SIGGRAPH 2007 courses*. [S.l.: s.n.], 2007. p. 2–es.

OWENS, J. D. et al. Gpu computing. *Proceedings of the IEEE*, IEEE, v. 96, n. 5, p. 879–899, 2008.

PICCHI, J.; ZHANG, W. Impact of l2 cache locking on gpu performance. p. 1–4, 2015.

The Khronos Group. *OpenCL: The open standard for parallel programming of heterogeneous systems*. 2025. Accessed: 2025-03-26. Available from Internet: <https://www.khronos.org/opencl/>.

Capítulo 2

Diretivas Paralelas de Programação[1]

Claudio Schepke - claudioschepke@unipampa.edu.br[2]
Vinícius Garcia Pinto - vinicius.pinto@furg.br[3]

Resumo

Este capítulo tem como objetivo abordar a paralelização de aplicações usando diretivas das interfaces de programação paralela OpenMP e OpenACC. O uso de diretivas tem sido a forma mais fácil de acelerar uma aplicação tanto em CPU como em GPU, sem a necessidade de conhecimentos profundos a respeito de programação paralela. Através de exemplos práticos de aplicações consideradas simples, mas não triviais como as vistas geralmente em cursos introdutórios, almeja-se apresentar os principais recursos para a execução concorrente de tarefas e de laços paralelos. As interfaces paralelas selecionadas também permitem o direcionamento da arquitetura-alvo, o que possibilita o uso de uma mesma interface tanto para CPU como para GPU.

[1]DOI: 10.5753/sbc.16630.8.2

[2]Claudio Schepke possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2005) e mestrado (2007) e doutorado (2012) em Computação pela Universidade Federal do Rio Grande do Sul, sendo este feito na modalidade sanduíche na Technische Universität Berlin, Alemanha (2010-2011). Realizou um pós-doutorado no Instituto Politécnico de Bragança, Portugal (2024-2025), trabalhando com programação paralela. É professor associado da Universidade Federal do Pampa (UNIPAMPA), campus Alegrete/RS desde 2012, ministrando disciplinas de arquitetura de computadores, sistemas operacionais, compiladores e processamento paralelo. Tem experiência na área de Ciência da Computação, com ênfase em Computação de Alto Desempenho, atuando principalmente nos seguintes temas: interfaces de programação paralela, aplicações científicas e computação em nuvem.

[3]Vinícius Garcia Pinto é doutor em Computação pelo Programa de Pós-Graduação em Computação da UFRGS em co-tutela com a Université Grenoble Alpes / França (2018); Mestre em Computação pelo Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande do Sul (2013). Bacharel em Ciência da Computação pela Universidade Federal de Santa Maria (2010). Foi professor da Faculdade São Francisco de Assis (2013-2014), da Universidade de Caxias do Sul (2018-2019) e professor substituto da Universidade Federal do Rio Grande do Sul (2019-2021) onde também realizou estágio de pós-doutorado. Desde 2021 é professor do Centro de Ciências Computacionais da Universidade Federal do Rio Grande, atuando em disciplinas de programação e sistemas operacionais.

2.1. Introdução

Programar paralelamente, isto é, gerar múltiplos fluxos de execução ou processos que executam colaborativamente, já foi considerado muito difícil. Para simplificar o desenvolvimento de aplicações de alto desempenho, pode-se fazer uso de diferentes interfaces de programação paralela, o que inclui bibliotecas de programação, linguagens de programação paralelas, extensões de linguagem, linguagens de domínio específico ou *frameworks* de desenvolvimento. No que tange à aceleração via execução concorrente de códigos sequenciais já existentes, o uso de diretivas paralelas de programação simplifica a etapa de escrita de código-fonte.

Diretivas são comandos ou palavras-chave utilizadas para a geração de código em tempo de pré-compilação. Isto é, antes que o código-fonte seja propriamente compilado, os termos inseridos nos códigos são substituídos e novas linhas de código são injetadas. No caso das diretivas (*pragmas*) paralelas, estas são convertidas para um código que gere, por exemplo, fluxos concorrentes de código (*threads*) que serão executados em arquiteturas multi-core ou GPU, por exemplo.

Neste contexto, este capítulo mostra como o desenvolvimento de programas pode beneficiar-se do uso de diretivas de programação paralelas. A Seção 2.2 irá apresentar as métricas de desempenho que vêm sendo utilizadas para avaliar os ganhos de desempenho no caso de execuções paralelas. Na sequência, são mostrados conceitos e exemplos de técnicas de programação usando OpenMP, na Seção 2.3, e OpenACC, na Seção 2.4. Por fim, na Seção 2.5, a conclusão discute as considerações finais sobre o assunto. Desta forma, a contribuição deste capítulo é oferecer uma introdução de forma intuitiva e fácil sobre como iniciar um programa paralelo, sem a necessidade de conceitos mais profundos de arquitetura de computadores e sistemas operacionais.

2.2. Métricas de desempenho

O uso de diretivas de compilação não necessariamente agrega alto desempenho a um código. Neste sentido, é importante fazer uso de métricas para avaliar o desempenho de programas, com o intuito de investigar o real ganho paralelo. Pela **Lei de Amdahl**, nem todo um programa pode ser paralelamente executado. Considerando o clássico conceito de Entrada, Processamento e Saída baseado no **Modelo de Von Neumann** de computação, operações de pré- e pós-processamento são normalmente seriais, pois envolvem o acesso ou escrita linear de informações, através da memória, de arquivos, por exemplo. Para alguns casos, esse armazenamento também poderia ser fragmentado e realizado concorrentemente, ainda que com limitações tecnológicas.

De todo modo, presume-se que a maior parte do tempo será demandada no processamento, muitas vezes composta de operações iterativas ou etapas independentes. No caso de **operações com laços**, se cada iteração depender linearmente da anterior, não será possível a quebra em paralelo das operações. Essa situação ocorre em dependências temporais. Todavia, mesmo em dependências temporais, internamente, em cada etapa iterativa, haverá a chance de que existam operações que poderão ser executadas concorrentemente sobre os dados. No caso de **etapas de processamento**, novamente podem haver dependências entre cada fase ou computações naturalmente sobre tarefas paralelas. Ainda que possam existir dependências entre as etapas, é possível dividir a computação

dos dados, gerando uma espécie de duto de processamento.

O **tempo de processamento** é talvez o primeiro fator a ser considerado para avaliar o desempenho das implementações. Em um mundo computacional multi-tarefa, vale o tempo real transcorrido e não apenas o tempo em que a CPU ficou alocada para um determinado processo. Ou seja, é preciso considerar formas de coletar o tempo de processamento. Gerenciadores de tarefas, monitores de processamento em GPU ou a execução de chamadas externas à execução de um programa são algumas das alternativas. No caso da execução do comando `time`, é possível obter os tempos de execução específicos da aplicação executada em modo usuário, modo sistema operacional e valores totais. Alguns programadores preferem utilizar a sua própria função baseada em chamadas de sistema Linux, como `gettimeofday`, `clock_gettime` ou `cpu_time` de bibliotecas de funções de relógio ou tempo disponíveis em linguagens como C e Fortran. Outros preferem fazer uso de funções ou rotinas encapsuladas nas próprias interfaces de programação paralela como `omp_get_wtime` de OpenMP ou a flag `pgi_acc_time` de OpenACC. É importante lembrar que as métricas podem ter uma determinada granularidade de coleta (microssegundos, por exemplo) ou cumulativamente retornar a soma dos tempos de execução de cada fluxo paralelo, o que pode não ser precisamente satisfatório para execuções curtas ou não ser o exato objetivo da medição.

Além do tempo, são considerados também como métricas o ganho de desempenho (*speedup*) e a eficiência computacional.

No primeiro caso, a ideia do **ganho de desempenho** é avaliar quantas vezes um programa paralelo é mais rápido em relação ao tempo de execução sequencial. Hipoteticamente, usando N recursos replicados, o ganho deveria ser de N, para o trecho em paralelo considerado. Mesmo considerando apenas os trechos que podem ser paralelizados, não é possível acelerar linearmente uma aplicação, especialmente quando um número maior de fluxos paralelos é utilizado. Isso decorre de custos inerentes com a instanciação de tarefas paralelas, da divisão irregular da carga de processamento e da própria natureza da computação das instruções, como níveis de cache, acesso concorrente à memória, maior número de operações de saltos de instruções. Este último, inclusive, pode prover um ganho levemente superlinear para execuções de entradas bastante específicas, considerando um pequeno número de fluxos de execução. Todavia, o mais natural é um comportamento de aceleração em escala logarítmica, tendendo a um limite e eventualmente caindo depois que a granularidade de cada tarefa paralela tornar-se muito pequena.

No segundo caso, a **eficiência** considera uma valoração percentual. Ou seja, a razão entre o tempo paralelo dividido pelo tempo sequencial e o número de recursos paralelos considerados resulta em um valor entre 0 e 1. Quanto mais próximo de 1, maior é a eficiência. Pelos mesmos motivos apresentados para o ganho paralelo, é impossível a eficiência permanecer constante à medida que o número de unidades de computação paralela aumenta. Assim, o comportamento da curva de eficiência tenderá a 0 para um número grande de fluxos de execução. Diante disso, o que se deseja em uma implementação paralela é adiar o máximo possível o declínio da eficiência à medida em que há mais processamento ocorrendo em paralelo.

Uma análise crítica que também precisa ser feita em relação à aceleração paralela é o tempo sequencial de referência a ser considerado. Alguns advogam medir apenas

a proporção entre os trechos que foram paralelizados. No entanto, isso é injusto, pois o que importa é reduzir o tempo total de processamento de uma aplicação. Por outro lado, o tempo de computação também não deveria ser comparado quando arquiteturas computacionais distintas são avaliadas.

Para solucionar esse problema, pode-se utilizar a noção de número de **operações de ponto flutuante por intervalo de tempo**. A escolha tradicional por operações com números reais deve-se à predominância delas na computação de propósito geral e, mais expressivamente, na computação científica. Através desta métrica de coleta, é possível comparar arquiteturas distintas como multicore e GPU.

Outra métrica muito em voga é a capacidade de processamento em relação ao **consumo energético**. A preocupação com a computação verde incorporou medições ou estimativas aproximadas do uso de energia em relação ao número de pontos flutuantes computados. Assim, a métrica de desempenho (flop) por Watt pode ser utilizada.

Outra forma de tratar a capacidade de desempenho de um programa é afixar o tempo de processamento de acordo com o valor sequencial e ajustar o **volume de processamento** para que o mesmo se adéque ao tempo sequencial. Por exemplo, se a complexidade computacional é quadrática, então é possível, em teoria, computar o dobro de dados tendo quatro vezes mais capacidade de processamento à disposição em relação ao volume de dados originais e tempo sequencial. Desta forma, o que se deseja é aumentar o tamanho do problema ou domínio a ser computado, de maneira a garantir a execução em um tempo aceitável.

Por fim, cabe citar as informações estatísticas realizadas por **contadores de hardware** como um meio mais fino de avaliar o desempenho de uma aplicação. Ou seja, é interessante coletar a quantidade de cada tipo de operação elementar a ser executada pelo hardware, de maneira a comparar as execuções sequenciais e paralelas. Em um cenário ideal, o número e o tipo de instruções mantêm-se bastante próximos dos valores da execução sequencial. Todavia, é natural um acréscimo no somatório do número total de instruções paralelas, uma vez que determinadas operações terão que ser computadas em cada fluxo paralelo distinto. Somado a isso, tem-se os níveis de cache e as replicações de alguns desses níveis nos diferentes cores. No caso de execuções paralelas, espera-se também distribuir a carga de computação da maneira mais **balanceada** possível, especialmente se o hardware for homogêneo.

Ainda em relação à estatística, é importante usar as melhores práticas recomendadas. Assim, aspectos como o **número mínimo de execuções** dos experimentos para garantir um percentual de **confiança** aceitável, **média**, **mediana** e **desvio padrão** são passos que precisam ser seguidos e cujos valores precisam ser coletados. Outros pontos que envolvem a coleta de dados nos experimentos têm a ver com a execução dos testes. Por exemplo, que abordagem deve ser seguida: executar várias vezes consecutivamente sem intervalos, excluindo da coleta a primeira execução que foi gerada “a frio” ou intercalar cada execução de maneira que cada teste seja feito “isoladamente”, como de fato aconteceria na realidade? No primeiro caso, os valores estatísticos de controle estariam menos discrepantes. No segundo caso, os experimentos são mais condizentes com o que aconteceria em um cenário real. Uma maior discussão a respeito de boas práticas de experimentação em computação paralela pode ser vista em (PINTO; NESI; SCHNORR,

2020).

2.3. Diretivas de OpenMP

OpenMP (*Open Multi-Processing*) é uma API aberta focada na escrita de aplicações paralelas. As diferentes versões da API são padronizadas pelo *OpenMP Architecture Review Board* e são suportadas por diversos compiladores. Desde a versão 1.0 lançada ainda nos anos 1990, a API tem sido incrementada, adicionando suporte a novos recursos de *Sardware* como arquiteturas NUMA, vetorização e GPUs, bem como novos recursos de *Software*, como paralelismo de tarefas e mecanismos de sincronização. A versão mais recente é a 6.0 lançada em novembro de 2024 (OpenMP, 2024). Um aspecto importante a ser destacado é que as funcionalidades mais recentes adicionadas na especificação nem sempre são plenamente suportadas pelos compiladores mais populares como GCC e LLVM/Clang. No momento da escrita deste capítulo, o Clang[4] e o GCC[5] suportam plenamente a versão 4.5 do OpenMP e apenas parcialmente as versões mais recentes como 5.0, 5.1 e 6.0. Neste material, vamos dar maior ênfase naquelas funcionalidades da API que já estão consolidadas e com suporte integral nos mais diversos compiladores.

Podemos interagir com a API do OpenMP de três maneiras distintas:

- adicionando diretivas de compilador;
- fazendo chamadas à biblioteca de rotinas; e
- definindo variáveis de ambiente.

Do ponto de vista da pilha de Software, pode-se dizer que OpenMP fica localizado numa camada intermediária. No topo temos a aplicação do usuário final, enquanto na base temos o suporte do Sistema Operacional à funcionalidades como *threads* e primitivas de sincronização de memória. A execução de um programa OpenMP faz uso de um ambiente de execução conhecido como *OpenMP Runtime Library* que é responsável por questões como a criação e o gerenciamento das *threads*, sincronização e escalonamento. Cada compilador usualmente implementa o seu próprio ambiente de execução como o `libgomp` do GCC ou o `libomp` do LLVM.

OpenMP surgiu como uma ferramenta para paralelização incremental de códigos já existentes. Dessa forma, vamos começar o estudo das diretivas com o mais simples e primitivo dos programas: um "Olá Mundo", ilustrado no Código 2.1. Neste exemplo temos um programa em linguagem `C`. Cabe ressaltar que OpenMP também suporta as linguagens `C++` e `Fortran`. Alguém familiarizado com a programação em `C` notará rapidamente que a versão paralela com OpenMP é muito similar a um "Olá Mundo" tradicional (e sequencial). As diferenças se resumem às linhas de código 2 e 5. Ambas são diretivas de pré-processamento. A primeira é o já conhecido `#include` que serve para incluir um arquivo de cabeçalho. Já a segunda (`#pragma omp parallel`) é uma diretiva específica do OpenMP e é utilizada para delimitar uma região paralela. Isto é, o trecho de código associado a esta diretiva será executado em paralelo por múltiplas *threads*. Antes

[4]<https://clang.llvm.org/docs/OpenMPSupport.html#openmp-6-0-implementation-details>
[5]<https://gcc.gnu.org/projects/gomp/#implementation-status>

de executarmos tal código, precisamos compilá-lo adicionando a *flag* `fopenmp` à linha de compilação como, por exemplo: `gcc ola-mundo.c -fopenmp`.

```
1 #include<stdio.h>
2 #include<omp.h>
3 int main (){
4
5  #pragma omp parallel
6  printf("Ola Mundo!\n");
7
8 }
```

Código 2.1. Ola Mundo Paralelo Com OpenMP

Em seguida, podemos executar tal programa e observar sua possível saída:

```
Ola Mundo!
Ola Mundo!
Ola Mundo!
Ola Mundo!
```

Nosso programa exibiu quatro vezes a mensagem `"Ola Mundo!"`. Isto aconteceu pois o `printf` estava associado a uma região paralela. Logo, ele foi executado, em paralelo, por várias *threads*. No exemplo acima, podemos deduzir que o número de *threads* usado foi quatro.

O número de *threads* a ser utilizado na execução é um dos parâmetros chave a ser configurado em um programa paralelo. O número ideal usualmente depende de várias variáveis como a carga de trabalho, a estrutura do algoritmo utilizado e a interação com outras bibliotecas paralelas. Por padrão, na ausência de qualquer outra informação, OpenMP utiliza como número de *threads* o número de CPUs virtuais/lógicas disponíveis no sistema. Caso necessário, este valor pode ser alterado de pelo menos três maneiras. A primeira delas não requer alterar o código e pode ser feita para a execução do programa, ao setar a variável de ambiente `OMP_NUM_THREADS`. A segunda é fazendo uso da chamada de rotina `omp_set_num_threads()` diretamente no código do programa. Isto valerá para as regiões paralelas subsequentes. Já a terceira consiste em especificar o número de *threads* diretamente na diretiva `#omp parallel` adicionando a cláusula `num_threads`.

Um dos principais desafios ao se paralelizar um programa diz respeito ao gerenciamento do acesso (possivelmente concorrente) às variáveis. Em programas sequenciais mais simples costumamos levar em conta que as variáveis podem ser globais ao programa inteiro ou locais a uma determinada função ou região. Isto vai determinar os trechos de código onde a variável é acessível (escopo) e seu tempo de vida. Quando adicionamos paralelismo, também acrescentamos mais níveis de escopo e tempo de vida já que podemos ter variáveis que são totalmente privadas a cada *thread*, variáveis que são compartilhadas (i.e., globais) entre *threads* mas locais a uma determinada parte do programa, ou ainda

variáveis que em dado momento são privadas mas que ao final são reduzidas a um valor único compartilhado.

Vejamos o comportamento padrão do OpenMP para algumas situações. São compartilhadas, isto é, há uma única instância, as variáveis declaradas fora de uma região paralela, as variáveis estáticas e as variáveis alocadas dinamicamente (e.g., uso de `malloc`). São privadas, isto é, há uma instância para cada *thread*, as variáveis declaradas dentro de uma região paralela e as variáveis de funções chamadas dentro de uma região paralela.

Há situações, porém, em que o comportamento padrão é inadequado. Nestes casos, podemos alterar tal comportamento acrescentando informações a uma diretiva utilizando cláusulas. A cláusula `shared` define que a variável em questão será compartilhada entre todas as *threads* que executam a região paralela em questão. Já a cláusula `private` indica ao OpenMP que devem ser criadas novas instâncias da variável para que cada *thread* acesse apenas a sua cópia privada. Por padrão, estas novas instâncias não são inicializadas. Caso necessário, podemos inicializar as cópias com o valor original utilizando a variação `firstprivate`.

Até aqui, vimos que o código interno a uma região paralela é executado igualmente por todas as *threads*. Porém, há várias situações em que alguns trechos de código não necessitam ser executados por todas as *threads* (e.g., inicialização, leitura de entradas) ou que não podem ser executados mais de uma vez (e.g., envio de mensagens, alocação/-liberação de memória) ou ainda que podem executar mais de uma vez mas não simultaneamente. Nestes casos, podemos empregar as diretivas `single`, `master`, `masked` e `critical`. Nos dois primeiros casos, os comandos associados serão executados uma única vez porém há duas diferenças importantes. Na diretiva `single`, o comando associado poderá ser executado por qualquer uma das *threads*, i.e, a primeira a alcançar a diretiva, enquanto a diretiva `master` restringe a execução à *thread* cujo identificador é 0. A segunda diferença é que na diretiva `single` há uma barreira implícita, que obriga as demais *threads* a aguardarem até que a execução dos comandos associados seja concluída, o que não ocorre no caso da diretiva `master`. Importante ressaltar também que a diretiva `master` está marcada como obsoleta desde a versão 5.1. Sugere-se que seja substituída pela diretiva `masked` que tem a mesma semântica desde que seja usada sem uma cláusula `filter`. A vantagem é que, usando `masked filter(<id>)`, podemos restringir a execução a outras *threads* que não a *thread* de identificador 0. A expressão associada à cláusula `filter` aceita construções mais complexas desde que resultem em um inteiro. Por exemplo, a estrutura `masked filter(omp_get_thread_num()%3)` restringe a execução dos comandos às *threads* 0, 1 e 2. Por fim, vamos analisar o comportamento da diretiva `critical`. Esta diretiva restringe a execução dos comandos temporariamente a apenas uma *thread* de cada vez. Ou seja, permite definir os comandos associados como uma região crítica.

O Código 2.2 apresenta uma nova versão do programa "Olá Mundo" visto anteriormente. Este código exemplifica o uso de algumas das estruturas que discutimos. Na linha 8 vemos que o número de *threads* foi definido para três, utilizando a chamada de rotina `omp_set_num_threads`. Na linha 10 definimos uma região paralela que engloba o bloco de código que começa na linha 11 e termina na linha 20. Todo este trecho executará em paralelo nas três *threads*, exceto o comando da linha 17 que é precedido

pela diretiva `single`. Vamos agora analisar o escopo das três variáveis declaradas no código. Inicialmente, na linha 6, foram declaradas as variáveis `id` e `total`. A linha 6 está fora da região paralela, logo, por padrão, ambas variáveis seriam compartilhadas entre as *threads*. Porém, na linha 10, usamos as cláusulas `shared` e `private`. A cláusula `shared(total)` reforça que a variável `total` continuará sendo única e compartilhada entre todas as *threads* enquanto a cláusula `private(id)` altera o comportamento padrão para a variável `id`. Dessa forma, OpenMP criará uma nova variável `id` para cada *thread*, apesar de possuírem o mesmo nome. O fato é que dentro da região paralela teremos três novas variáveis, cada uma delas sendo privada a uma dada *thread*. Na linha 11, temos a declaração da variável `x`. Como tal declaração acontece já dentro da região paralela (linhas 11 a 20), cada *thread* declarará a sua variável `x` de uso privativo.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<omp.h>
4
5  int main (){
6   int id, total;
7
8   omp_set_num_threads(3);
9
10  #pragma omp parallel shared(total) private(id)
11  {
12   id = omp_get_thread_num();
13
14   int x = rand() % 100;
15
16   #pragma omp single
17   total = omp_get_num_threads();
18
19   printf("Ola Mundo! (thread %d de %d sorteou %d) \n", /
        id, total, x);
20  }
21  printf("Tchau!\n");
22 }
```

Código 2.2. Nova versão do Olá Mundo Paralelo Com OpenMP

Com isso, temos uma possível saída para o programa anterior:

```
Ola Mundo! (thread 2 de 3 sorteou 86)
Ola Mundo! (thread 0 de 3 sorteou 77)
Ola Mundo! (thread 1 de 3 sorteou 83)
Tchau!
```

Esta não é a única resposta possível uma vez que a maneira como o Sistema Operacional escalona as *threads* pode mudar de uma execução pra outra. Como o número de

threads foi configurado em três, teremos a mensagem "Ola Mundo" sendo impressa três vezes com valores distintos para `id` e `x`. Cada *thread* descobre seu identificador e sorteia um número inteiro. Como cada uma delas possui instâncias locais das variáveis `id` e `x`, não há conflitos de escrita concorrente. A variável `total` é compartilhada entre as três *threads*. Mas graças à diretiva `single`, seu valor sempre será consistente, uma vez que é escrito por apenas uma *thread* e porque existe uma barreira implícita obrigando as demais *threads* a não prosseguirem até a conclusão da execução do comando de escrita. Cabe ressaltar que, leituras concorrentes à variável `total`, como temos na linha `19`, não geram conflitos, pois não fazem alteração na mesma. Por fim, temos a impressão de uma única mensagem de "Tchau". Isto ocorre pois o comando da linha 21 está fora da região paralela. Por consequência, neste momento o código está sendo executado apenas pela *thread* principal do programa.

O paralelismo frequentemente é empregado em programas cuja execução é dominada pelo tempo necessário para efetuar operações custosas contidas em laços de repetição. Para tratar destes casos, OpenMP provê a diretiva `omp for`. Dentre as diretivas de divisão de trabalho de OpenMP, esta é a mais conhecida e amplamente utilizada (QUEVEDO et al., 2024). Ela consiste em dividir as iterações do laço entre as diversas *threads*. Com isso, várias iterações podem ser processadas em paralelo por diferentes *threads*. É importante ressaltar que cabe a quem programa analisar se não há dependências que impeçam a execução das iterações em paralelo. O Código 2.3 ilustra um exemplo desta diretiva. A diretiva `for` da linha 6 provoca o particionamento das iterações do laço das linhas 7 e 8, entre as quatro *threads* da região paralela. Por fins didáticos, na linha 8 fizemos cada *thread* escrever o respectivo `id` na posição do vetor em que esta trabalhou. Ao observar a saída do programa, notamos que as *threads* com `id` 0 e 1 receberam cada uma 8 iterações e, por consequência, escreveram em 8 posições do vetor. Já as *threads* com `id` 2 e 3 receberam 7 iterações cada.

```
1  #pragma omp parallel private(id) num_threads(4)
2  {
3   id = omp_get_thread_num();
4   printf("Thread %d executando!\n", id);
5
6   #pragma omp for
7   for(int i = 0; i < MAX; i++)
8    v[i] = id;
9  }
10 for(int i = 0; i < MAX; i++)
11  printf("%d ", v[i]);
```

Código 2.3. Diretiva de divisão de Trabalho `for`

```
Thread 2 executando!
Thread 1 executando!
Thread 3 executando!
Thread 0 executando!
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3 3 3
```

A diretiva `for` opcionalmente pode ser estendida com a cláusula `schedule`, que permite controlar se a distribuição das iterações entre as *threads* será estática (`static`) ou dinâmica (`dynamic`), bem como definir o tamanho (máximo) do subconjunto de iterações distribuído. Veja, no Código 2.4, um trecho que exemplifica o uso da diretiva `schedule` no particionamento das *threads* e o reflexo na saída do programa.

```
#pragma omp for schedule(static, 3)
for(int i = 0; i < MAX; i++)
 v[i] = id;
```

Código 2.4. Diretiva schedule

```
0 0 0 1 1 1 2 2 2 3 3 3 0 0 0 1 1 1 2 2 2 3 3 3 0 0 0 1 1 1
```

Outra situação recorrente em programas paralelos é aquela em que cada *thread* calcula localmente uma parte da solução e estas partes precisam ser combinadas para que se obtenha a solução integral. Tal situação é tratada com uma operação de redução, que no OpenMP é provida pela diretiva `reduction`. Uma operação de redução aplica um operador sobre uma variável. Os operadores mais frequentes são soma (+), multiplicação (/), máximo (`max`) e mínimo (`min`). O Código 2.5 a seguir ilustra o uso da diretiva `reduction` para computar um somatório de valores. Cada *thread* terá uma instância privada da variável `total`, que será inicializada com o elemento neutro da operação em questão, que neste caso é 0. Todas as atualizações parciais ocorrem na instância privada. Ao final, cada valor computado localmente é incorporado por OpenMP à variável original de maneira segura sem que seja necessária a inclusão explícita de uma diretiva de sincronização como `critical`.

```
#pragma omp for reduction(+ : total)
for(int i = 0; i < MAX; i++){
 v[i] = id;
 total += v[i];
}
```

Código 2.5. Exemplo de uma operação de redução

Além da paralelização de laços, OpenMP também suporta o paralelismo de tarefas com a diretiva `task`. Neste modelo não há um mapeamento exato e fixo do trabalho às *threads*. A ideia é desacoplar o algoritmo da plataforma, definindo tarefas, ou seja, trechos de código que podem executar em paralelo. Além do código, cada tarefa está associada a um ambiente de dados. Na ausência de pontos de sincronização ditos "globais", as tarefas podem ser executadas em qualquer ordem e por qualquer *thread*. Uma prática usual é criar uma grande quantidade de tarefas que serão posteriormente consumidas pelas *threads*.

Embora o suporte inicial à tarefas tenha sido adicionado em 2008 na versão 3.0 da especificação, seu uso ainda permanece menos frequente que as diretivas de laços paralelos. Algumas categorias de algoritmos como divisão e conquista, operações em matrizes por blocos e busca em grafos podem ser mais facilmente paralelizados utilizando

o paralelismo de tarefas. O trecho apresentado no Código 2.6 ilustra o funcionamento das diretivas `task` e `taskwait`. A diretiva `task` (linhas 1 e 6) é usada para definir duas novas tarefas. A primeira consiste do código entre as linhas 2 e 5 e a segunda do código entre as linhas 7 e 10. As duas novas tarefas, assim como o comando da linha 11, podem ser executados em qualquer ordem por qualquer *thread* (veja a seguir uma saída possível). O comando da linha 13 faz uso das variáveis `x` e `y`, que são escritas pelas tarefas criadas anteriormente. Dessa forma, antes de executá-lo precisamos garantir que tais tarefas já tenham terminado. Esta sincronização é obtida por meio da diretiva `taskwait`, da linha 12, que bloqueia a execução até que todas as tarefas criadas anteriormente no mesmo escopo já tenham terminado. Caso necessário, é possível estender a diretiva `task` com cláusulas como `if`, para limitar a criação de novas tarefas ao atendimento de alguma condição, e `priority` que permite dar ao ambiente de execução alguma sugestão de quais tarefas podem ter sua execução priorizada. A prioridade é dada por um número inteiro. Quanto maior este valor, mais prioritária é a tarefa sugerindo ao ambiente de execução que ela deve ser executada antes daquelas cujo valor de prioridade é menor.

```
1  #pragma omp task
2  {
3   printf("tarefa A!! \n");
4   x = 123;
5  }
6  #pragma omp task
7  {
8   printf("tarefa B!! \n");
9   y = -10;
10 }
11 printf("outra mensagem qualquer\n");
12 #pragma omp taskwait
13 printf("as duas tarefas acabaram e x*y=%d\n", x*y);
```

Código 2.6. Exemplo de uso da diretiva `task`

```
outra mensagem qualquer
tarefa B!!
tarefa A!!
as duas tarefas acabaram e x*y=-1230
```

Outra maneira de se impor alguma ordem na execução das tarefas é por meio da cláusula `depend`. Esta cláusula permite especificar para cada tarefa quais dados são lidos e quais dados são escritos. Com base nestas informações combinadas com a ordem de criação das tarefas, OpenMP pode inferir quais tarefas são independentes e, portanto, podem executar em paralelo e quais não podem. Veja no Código 2.7 uma versão alternativa que utiliza `depend` para descrever como os dados compartilhados são acessados por cada tarefa. As diretivas das linhas 1 e 6 agora especificam quais tarefas produzem os dados `x` e `y`. Já a diretiva da linha 12 especifica que a tarefa a seguir consome os dados `x` e `y`. Logo, tal tarefa não pode ser executada antes que tais dados já tenham sido produzidos. Além

dos parâmetros `in` e `out`, a cláusula `depend` também pode ser usada com o parâmetro `inout` para os casos em que um dado é tanto lido quanto escrito. Uma discussão mais detalhada sobre o paralelismo de tarefas em OpenMP pode ser encontrada em (NESI et al., 2021). Complementarmente ao paralelismo de tarefas, formato das diretivas em Fortran, bem como exemplos e avaliações comparativas das diretivas de OpenMP podem ser encontradas em (SCHEPKE; LUCCA; LIMA, 2023).

```
1  #pragma omp task depend(out:x)
2  {
3   printf("tarefa A!! \n");
4   x = 123;
5  }
6  #pragma omp task depend(out:y)
7  {
8   printf("tarefa B!! \n");
9   y = -10;
10 }
11 printf("outra mensagem qualquer\n");
12 #pragma omp task depend(in:x) depend(in:y)
13 printf("as duas tarefas acabaram e x*y=%d\n", x*y);
```

Código 2.7. Exemplo de uso da diretiva `depend`

2.4. Diretivas de OpenACC

Embora tanto OpenMP como OpenACC possam gerar código tanto para CPU como para GPU, tradicionalmente associa-se a segunda interface à programação paralela para GPUs. Quando se fala em programação para GPUs, logo tem-se a ideia de que é preciso fazer uso de CUDA. Todavia, esta interface torna muito explícitas as características do hardware e exige a compreensão de noções da arquitetura e do modelo de programação. Algo similar acontece com a interface OpenCL. Com isso, OpenACC torna-se uma boa alternativa de abstração das camadas inferiores, possibilitando ver apenas os aspectos de aplicação.

OpenACC é uma API desenvolvida em 2011 para programação em arquiteturas heterogêneas com aceleradores. O modelo de programação usa diretivas de alto nível, assim como o OpenMP, para expressar as áreas ou blocos paralelos (CHANDRASEKARAN; JUCKELAND, 2017). A versão mais recente da API do OpenACC é a 3.3, lançada em novembro de 2022.

A API está disponível para as linguagens C/C++ e FORTRAN (KIRK; WENMEI, 2016). Tal qual OpenMP, OpenACC possui diretivas padronizadas por `#pragma acc` para C/C++, seguida de `[atributos]` opcionais. Essas diretivas especificam *loops* e blocos de código que podem ser enviados da CPU para um acelerador. A seguir, são descritas algumas diretivas compatíveis com C/C++ e Fortran.

Antes de mais nada, é preciso lembrar que a programação em GPU envolve o uso de *Non-Uniform Memory Access* (NUMA). Isto é, há pelo menos duas memórias distintas: a que a CPU acessa e a que a GPU acessa, que inclusive podem ser fabricadas com

diferentes tecnologias. Por mais que haja abordagens de unificar este acesso à memória, de modo geral é preciso fazer a movimentação (via placa-mãe) entre a memória principal e a da GPU. Para tanto, pode-se utilizar `data copy`, como diretiva para a cópia de dados entre CPU e GPU, garantindo a sincronização dos dados, e `present`, como diretiva específica para o compilador "estar ciente" de que as variáveis estão presentes na memória da GPU e que não é necessário fazer cópia. As diretivas `copyin` e `copyout` são opções similares, mas cuja transferência de dados ocorre apenas para GPU e da GPU para a CPU. Também é possível usar a diretiva `create`, para apenas alocar memória na GPU, sem a cópia de dados. Por fim, pode-se utilizar `enter data` ou `exit data`, como uma diretiva que garanta a presença e término de existência dos dados, ou formalmente define o início e o fim de uma vida útil de dados não estruturados.

Em relação ao paralelismo, há algumas diretivas bastante úteis. A diretiva `kernels` permite que o compilador tome as decisões de paralelismo. A diretiva `parallel` impõe ao compilador que o trecho de código indicado na sequência deve ser paralelizado. Dependendo do uso, a diretiva pode produzir resultados equivocados. Já a diretiva `loop independent` especifica que o laço indicado pode ser paralelizado e ainda executado independentemente. Algumas dessas diretivas possuem outras opções parecidas e com funcionalidades bastante próximas (LUCCA, 2020).

Um exemplo de utilização de exploração do paralelismo de laços está apresentado no Código 2.8.

```
1 !$acc parallel loop collapse(2)
2   DO i=2,imax
3     DO j=2,jmax
4       a(i,j) = b(i,j) * c(i,j)
5     ENDDO
6   ENDDO
7 !$acc end parallel
```

Código 2.8. Exemplo do uso da diretiva `!$acc parallel loop`

Apesar das interfaces OpenMP, OpenACC e OpenMP Target ((LIMA; SCHEPKE; LUCCA, 2021)) utilizarem diretivas para especificar o paralelismo, cada interface tem suas características e diretivas, conforme apresentado em (LUCCA, 2020). A Tabela 2.1 apresenta de forma resumida uma comparação da equivalência entre a sintaxe do OpenMP clássico com diretivas que são executadas em CPU, do OpenACC e do OpenMP Target para execução de tarefas em GPU. As APIs diferem em sua abordagem para especificar a paralelização. Como todas as versões utilizam diretivas, a proposta é especificar as equivalências entre as APIs. Embora as diretivas sejam semelhantes em cada API, o paralelismo é especificado de uma forma diferente (SILVA et al., 2022).

2.5. Conclusão

Injetar diretivas é o meio mais simples de produzir operações paralelas, sem a necessidade de conhecer diversos aspectos específicos relacionados à arquitetura de computadores e ao gerenciamento de fluxos de execução. Isso não significa que magicamente o programa

Tabela 2.1. Relação entre as diretivas de OpenMP, OpenACC e OpenMP Target para GPU

OpenMP CPU	OpenACC	OpenMP Target (GPU)
omp parallel	acc parallel	omp target
omp parallel for	acc loop gang	omp teams distribute
-	acc loop vector	omp parallel for
-	acc data copyin(<var>)	omp data map(to:<var>)
-	acc data copyout(<var>)	omp data map(to:<var>)
-	acc data create(<var>)	omp data map(alloc:<var>)
omp parallel simd	-	omp parallel simd
omp task	-	omp task
-	routine	-
-	acc update	target update

resultante produzirá alto desempenho. Se assim o fosse, ambientes de auto-paralelização poderiam existir e serem adotados.

Assim, após conhecer como as diretivas funcionam e os resultados que elas produzem, é importante saber identificar em que trecho de código vale a pena aplicar a paralelização e qual abordagem produziria um efeito de ganho de desempenho mais expressivo. O assunto aqui apresentado está longe de ser esgotado, e possui lacunas, especialmente na questão de como identificar em códigos fonte, onde vale a pena buscar uma paralelização. Todavia, as ferramentas foram expostas, basta agora praticar.

Referências

CHANDRASEKARAN, S.; JUCKELAND, G. *OpenACC for Programmers: Concepts and Strategies*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2017. ISBN 0134694287.

KIRK, D. B.; WEN-MEI, W. H. *Programming massively parallel processors: a hands-on approach*. [S.l.]: Morgan Kaufmann, 2016.

LIMA, J. V. F.; SCHEPKE, C.; LUCCA, N. Além de Simplesmente: #pragma omp parallel for. In: CHARãO, A.; SERPA, M. da S. (Ed.). *Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre/RS: Sociedade Brasileira de Computação, 2021. chp. 4, p. 86–103.

LUCCA, C. S. N. Programando Aplicações com Diretivas Paralelas. In: BOIS, M. C. A. D. (Ed.). *Minicursos da XX Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre: Sociedade Brasileira de Computação, 2020. p. 89–104. ISBN https://doi.org/10.5753/sbc.4400.9.5.

NESI, L. L. et al. Desenvolvimento de Aplicações Baseadas em Tarefas com OpenMP Tasks. In: CHARãO, A.; SERPA, M. (Ed.). *Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul*. Sociedade Brasileira de Computação - SBC, 2021. p. 131–152. ISBN 9786587003504. Available from Internet: <http://dx.doi.org/10.5753/sbc.6150.4.6>.

OpenMP. *OpenMP Application Program Interface Version 6.0*. OpenMP Architecture Review Board, 2024. Available from Internet: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf>.

PINTO, V. G.; NESI, L. L.; SCHNORR, L. M. Boas Práticas para Experimentos Computacionais de Alto Desempenho. In: BOIS, A. D.; CASTRO, M. (Ed.). *Minicursos da XX Escola Regional de Alto Desempenho da Região Sul*. Sociedade Brasileira de Computação - SBC, 2020. p. 1–19. ISBN 9786587003504. Available from Internet: <https://doi.org/10.5753/sbc.4400.9.1>.

QUEVEDO, C. et al. An Empirical Study of OpenMP Directive Usage in Open-Source Projects on GitHub. In: *Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho*. Porto Alegre, RS, Brasil: SBC, 2024. p. 144–155. ISSN 0000-0000. Available from Internet: <https://sol.sbc.org.br/index.php/sscad/article/view/30993>.

SCHEPKE, C.; LUCCA, N.; LIMA, J. V. F. Diretivas Paralelas de OpenMP: Um Estudo de Caso. In: PADOIN, E. L.; GALANTE, G.; RIGHI, R. (Ed.). *Minicursos da XXIII Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre: Sociedade Brasileira de Computação, 2023. ISBN https://doi.org/10.5753/sbc.11938.7.1.

SILVA, H. U. da et al. Parallel OpenMP and OpenACC Porous Media Simulation. *The Journal of Supercomputing*, 2022. Available from Internet: <https://doi.org/10.1007/s11227-022-05004-2>.

Capítulo 3

Programação Avançada de Múltiplas GPUs com OpenACC[1]

Calebe P. Bianchini[2], Evaldo B. Costa[3], Gabriel P. Silva[4]

Resumo

Este minicurso tem como objetivo apresentar técnicas de programação paralela para processadores multinúcleo e aceleradores utilizando OpenACC usando cláusulas como, por exemplo, async e wait, com ênfase nos modelos de paralelismo em multiaceleradores. Serão abordadas as modificações necessárias no código-fonte para implementar esses modelos, bem como diretivas para movimentação de dados, como data, copy e create, além de diretivas e cláusulas avançadas como routine e atomic. Ao final, um estudo de caso é apresentado, observando o uso desses diretivas e também das APIs disponíveis no OpenACC.

[1]DOI: 10.5753/sbc.16630.8.3

[2]Calebe de Paula Bianchini é formado em Ciência da Computação na UFSCar, onde também obteve o título de mestre. Finalizou seu doutorado em Engenharia da Computação pela POLI/USP em 2009. Somados, já são mais de 19 anos de experiência com Computação de Alto Desempenho, com participação em diversos projetos de pesquisa e desenvolvimento com Petrobras, Shell, Intel, CERN, dentre outros. Atualmente é professor e pesquisador na Universidade Presbiteriana Mackenzie e de Centro Universitário FEI. É um membro ativo na comunidade brasileira de Computação de Alto Desempenho, especialmente na organização de competições e desafios nesta área. É também Embaixador Universitário do Deep Learning Institute da NVIDIA, ministrando cursos relacionados à Computação de Alto Desempenho em GPUs NVidia.

[3]Evaldo Bezerra da Costa é graduado em Engenharia Elétrica, ênfase em Telecomunicações pela Universidade Estácio de Sá (2010), com mestrado (2014) e doutorado (2022) em Informática pela Universidade Federal do Rio de Janeiro. Tem experiência na área de Ciência da Computação, atuando principalmente nos seguintes temas: programação paralela, computação de alto desempenho, inteligência artificial e bioinformática.

[4]Gabriel P. Silva tem doutorado em Engenharia de Sistemas e Computação pela Universidade Federal do Rio de Janeiro. Atualmente é Professor Titular do Instituto de Computação da UFRJ. Tem larga experiência na área de Ciência da Computação, com ênfase em Arquitetura de Sistemas de Computação, atuando principalmente nos seguintes temas: arquitetura de computadores, programação paralela, computação de alto desempenho e internet das coisas.

3.1. Introdução

A crescente demanda por poder computacional em diversas áreas impulsiona a necessidade de explorar o paralelismo oferecido por arquiteturas heterogêneas, como processadores multinúcleo e aceleradores, incluindo GPGPUs (KLEMM; COWNIE, 2021; BIANCHINI; COSTA; SILVA, 2024). Dentre as diversas abordagens para programação paralela nesses sistemas, o OpenACC se destaca como uma API baseada em diretivas que simplifica a tarefa de paralelização de código, permitindo aos desenvolvedores direcionar o compilador para gerar código paralelo para um dispositivo alvo.

Os aceleradores gráficos de propósito geral (*GPGPU - General-Purpose Graphics Processing Unit*) oferecem um modelo de paralelismo que pode ser explorado em multiprocessadores de fluxo (*streaming multiprocessors - SM*), que executam simultaneamente trechos computacionalmente intensivos chamados *kernels* (DEAKIN; MATTSON, 2023). Esse modelo de execução maciçamente paralelo é ideal para resolver problemas que exigem alta capacidade de processamento.

Para sistemas que contêm mais de um acelerador, o OpenACC fornece uma API para que as operações ocorram em um dispositivo específico. Caso um sistema contenha aceleradores de tipos diferentes, a especificação também permite consultar e selecionar dispositivos de uma arquitetura específica.

Entre os temas importantes do OpenACC está a necessidade de otimizar a localidade dos dados para reduzir o custo das transferências de dados em sistemas onde o hospedeiro e o acelerador têm memórias fisicamente distintas. Sempre haverá alguma quantidade de transferências de dados que simplesmente não podem ser otimizadas e ainda produzir resultados corretos.

Depois de minimizar as transferências de dados, pode ser possível reduzir ainda mais a penalidade de desempenho associada a essas transferências, sobrepondo as cópias com outras operações no hospedeiro, no dispositivo ou em ambos.

Por fim, é importante contextualizar o OpenACC em relação a outras tecnologias de programação paralela:

- CUDA é uma plataforma proprietária da NVIDIA que oferece controle de baixo nível sobre suas GPUs, possibilitando alto desempenho, mas exigindo um conhecimento mais aprofundado da arquitetura da GPU.
- OpenMP é uma API para programação paralela de memória compartilhada, amplamente utilizada em CPUs multi-core e com funcionalidades para *offloading* a aceleradores. O Exemplo 3.8 demonstra o uso de OpenMP em conjunto com OpenACC para gerenciar a execução em múltiplas GPUs.
- OpenCL é um padrão aberto para programação paralela em diversas plataformas, oferecendo maior portabilidade entre diferentes fabricantes de *hardware*.

Neste minicurso apresentaremos algumas técnicas de programação paralela avançada com múltiplas GPUs com OpenACC. Serão exploradas técnicas que vão além do uso

básico da API, como a gestão de operações assíncronas com as cláusulas **async** e a diretiva **wait**, a otimização da movimentação de dados entre o hospedeiro e os aceleradores utilizando diversas cláusulas da diretiva **data**, e a utilização de múltiplas GPUs através de funções específicas da API OpenACC.

Dado o nível avançado dos tópicos abordados, é fundamental que o participante possua um conhecimento básico sobre computação paralela como processadores multinúcleo e aceleradores e OpenACC para acompanhar o conteúdo deste minicurso de forma eficaz. Dentre os tópicos básicos, espera-se que o leitor esteja familiarizado com os seguintes conceitos fundamentais:

- A estrutura básica de uma diretiva OpenACC (**#pragma acc ...**)
- As diretivas essenciais para a criação de regiões paralelas, como **#pragma acc parallel loop** e **#pragma acc kernels**.
- O conceito de regiões de dados (**#pragma acc data**) e o uso primário das cláusulas de dados como **copy**, **copyin** e **copyout** para gerenciar a transferência de dados entre a memória do hospedeiro e a memória do dispositivo.
- A compreensão da distinção entre a execução no hospedeiro (CPU) e no dispositivo (acelerador), e a necessidade de explicitar a movimentação de dados para o processamento paralelo.

3.2. Arquitetura de Aceleradores

As arquiteturas com aceleradores são um tipo particular de exploração de paralelismo no nível de *thread*, onde trechos computacionalmente intensivos do programa, chamados de *kernels* são enviados para execução nos aceleradores, que têm memória distinta da memória do hospedeiro.

Isso requer que tanto o código como os dados sejam transferidos da memória do hospedeiro para a memória do acelerador, através de barramentos dedicados de alta velocidade. Os tipos de aceleradores mais utilizados nesse tipo de paralelismo são as GPUs (*Graphics Processing Unit*), cujos detalhes iremos apresentar mais adiante com a biblioteca de programação OpenACC.

As arquiteturas dos aceleradores gráficos (GPUs) são bem diferenciadas das arquiteturas dos processadores convencionais. O paralelismo nos aceleradores gráficos é explorado através de um conjunto maciço de multiprocessadores de fluxo (*streaming multiprocessors – SM*), executando em paralelo e de forma sincronizada trechos computacionalmente intensivos, chamados de *kernels*, das diversas aplicações.

Para o melhor entendimento dos aceleradores gráficos (GPUs), vamos estudar, sem perda de generalidade, a arquitetura de um tipo de acelerador gráfico desenvolvido pela NVIDIA, a arquitetura Hopper (NVIDIA, 2022).

Na Figura 3.1 verificamos que o acelerador gráfico possui uma arquitetura distinta, com diversos níveis de hierarquia de memória, algumas delas compartilhadas, outras exclusivas de cada multiprocessador de fluxo (SM).

Figura 3.1: Arquitetura NVIDIA Hopper

Cada unidade de multiprocessamento (SM) possui 128 núcleos CUDA de precisão simples e 64 de precisão dupla, com cada núcleo dedicado à execução de operações matemáticas envolvendo números inteiros e cálculos de ponto flutuante. Esses núcleos operam em um *pipeline* paralelo, permitindo a execução simultânea de múltiplas operações, o que aumenta consideravelmente a eficiência e a velocidade de processamento. A arquitetura foi projetada com foco na relação desempenho/consumo de energia, um aspecto essencial para a computação de alto desempenho moderna, onde o uso eficiente de energia é fundamental para maximizar a capacidade de processamento dentro de limites ambientalmente sustentáveis. Uma unidade SM da arquitetura Hopper denominado H-100 pode ser visto na Figura 3.2.

O escalonador do multiprocessador de fluxo (SM) dispara as *threads* em grupos de 32 *threads* chamadas de *warps*. Cada SM possui quatro escalonadores de *warp*, permitindo um máximo de quatro *warps* disparadas e executadas concorrentemente. O número de registradores pode chegar até 255 registradores utilizados simultaneamente por cada *thread*.

Para melhorar ainda mais o desempenho, a arquitetura Hopper possui 32 instruções *shuffle*. Essas instruções são usadas para otimizar operações de troca de dados entre *threads* dentro de um *warp*, melhorando a eficiência e o desempenho das operações paralelas, impactando em muito o desempenho de aplicações como, por exemplo, a transformada de Fourier (FFT).

Outro tipo de instrução disponível são as operações atômicas em memória, permitindo que as *threads* realizem adequadamente operações de *read-modify-write*, como soma, máximo, mínimo e compare-e-troque em estruturas de dados compartilhadas. Operações atômicas são amplamente utilizadas para ordenação em paralelo e para o acesso em paralelo a estruturas de dados compartilhadas sem a necessidade de travas que serializam a execução do código.

A arquitetura de memória da arquitetura Hopper é bastante sofisticada e está organizada em diversos níveis. No nível mais alto estão os registradores, em um total de

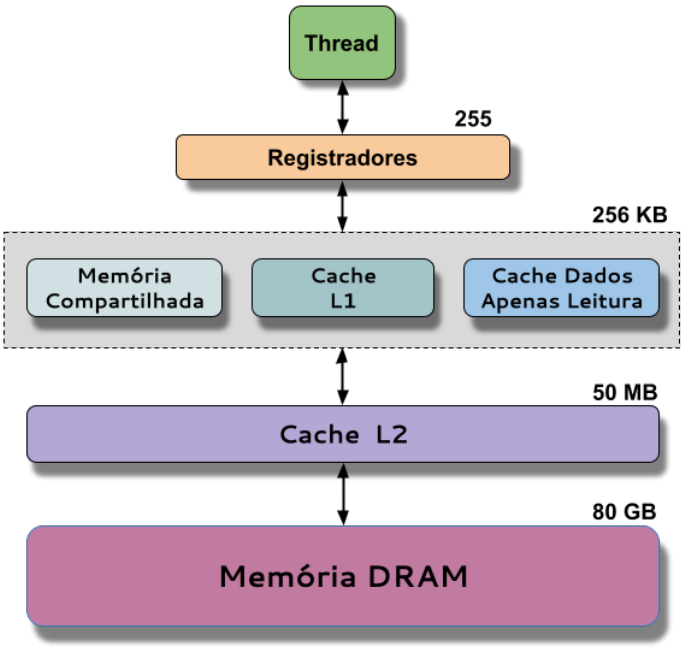
Figura 3.2: Multiprocessador de Fluxo (SM)

65.536 registradores de 32 bits por multiplexador de fluxo (SM). No nível imediatamente inferior, temos 256 KB de memória disponíveis para cache L1, cache de textura e memória compartilhada em cada SM. Um esquema da hierarquia de memória da H-100 pode ser visto na Figura 3.3.

A memória local de 256 KB da arquitetura Hopper é configurável para otimizar o desempenho em diferentes tipos de aplicações, podendo ser ajustada entre cache L1 e memória compartilhada, com esta última suportando até 228 KB. Esse espaço de armazenamento flexível permite aos desenvolvedores personalizá-lo conforme as necessidades específicas de cada aplicação.

A memória compartilhada pode ser dividida em até 8 bancos de memória, permitindo acessos paralelos por múltiplas *threads*. Essa divisão é projetada para melhorar a eficiência das operações, reduzindo a latência e aumentando a largura de banda, o que contribui para um desempenho geral mais rápido e eficiente.

Figura 3.3: Hierarquia de Memória



Além da cache L1, a arquitetura Hopper possui uma cache apenas de leitura de até 64 MB. O gerenciamento dessa cache pode ser feito automaticamente pelo compilador ou explicitamente pelo programador. O acesso a uma variável ou estrutura de dados que o programador identifica como apenas de leitura pode ser declarado com a palavra-chave *const __restrict*, permitindo ao compilador carregá-la na cache apenas de leitura. Há indicações de que a cache de constantes seria também armazenada junto com a memória de 256 KB disponível para cache L1 e memória compartilhada[5].

A arquitetura Hopper possui também uma cache L2 de 50 MB que permite um melhor fluxo de dados para as cargas de trabalho de processamento intensivo característicos de aplicações de IA e HPC. A cache L2 é o ponto primário de unificação de dados entre os diversos SMs, servindo operações de *load, store* e de textura, provendo um compartilhamento de dados eficiente e de alta velocidade.

Algoritmos onde o endereço dos dados é conhecido previamente, tais como solucionadores de física, *ray tracing* e multiplicação esparsa de matrizes, se beneficiam especialmente da hierarquia de cache. Os *kernels* de filtro e convolução onde é necessário que diversos SMs leiam os mesmos dados, também se beneficiam dessa hierarquia.

Finalmente, no último nível temos a memória global compartilhada, que pode ter até 80 GB, sendo acessada com uma largura de banda entre 2.000 e 3.000 GB/s, dependendo se for utilizada uma memória HBM2e ou HBM3.

A arquitetura possui uma série de outras facilidades, como código de correção de erro, paralelismo dinâmico, gerenciamento de filas de trabalho e unidade de gerenciamento de *grids*, que servem para melhorar o desempenho e a confiabilidade do acelerador.

[5]<https://chipsandcheese.com/p/nvidias-h100-funny-l2-and-tons-of-bandwidth>

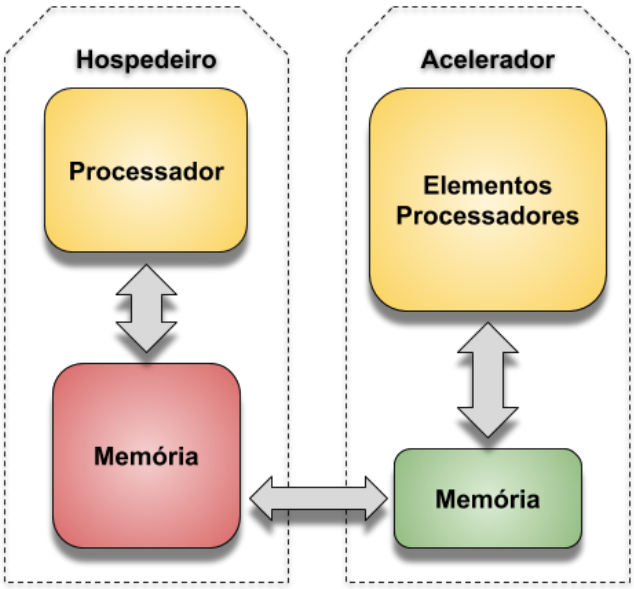
Maiores detalhes podem ser vistos na referência (NVIDIA, 2022).

3.3. Movimentação de dados

Um grande fator de impacto no desempenho do processamento paralelo é a movimentação de dados, havendo necessidade de se otimizar a localidade dos dados em sistemas onde o hospedeiro e o acelerador possuem memórias fisicamente distintas.

A movimentação de dados entre o hospedeiro e o acelerador é feita através de um barramento, que usualmente é mais lento em comparação com a largura de banda das respectivas memórias. Por sua vez, o acelerador não pode executar o processamento dos dados até que eles estejam na sua memória local. A Figura 3.4 apresenta o modelo básico dessa movimentação de dados.

Figura 3.4: Modelo básico de movimentação de dados entre hospedeiro e o acelerador (SILVA; BIANCHINI; COSTA, 2022)



No OpenACC, as cláusulas de dados são utilizadas para controlar e minimizar a movimentação de dados entre o hospedeiro e o acelerador durante a execução do programa.

Deste modo, os dados iniciais podem ser movimentados do hospedeiro para o acelerador e, somente após o término de todo o processamento, os resultados necessários são transferidos de volta para o hospedeiro. As cláusulas de movimentação de dados podem ser utilizadas em conjunto com as diretivas **data**, **kernels** ou **parallel**.

3.3.1. Diretiva data

A diretiva **data** facilita o compartilhamento de dados entre múltiplas regiões paralelas. Uma região de dados pode ser adicionada ao redor de uma ou mais regiões paralelas

dentro da mesma função ou posicionada em um nível superior na árvore de chamadas do programa, permitindo o compartilhamento de dados entre regiões em múltiplas funções.

A diretiva **data** é uma construção estruturada, o que significa que deve começar e terminar no mesmo escopo (como na mesma função ou sub-rotina), e possui o seguinte formato:

```
#pragma acc data [cláusula]
```

A Tabela 3.1 apresenta as cláusulas de dados, com uma breve descrição de seus significados.

Cláusula	Descrição
copy	Cria espaço para as variáveis listadas no dispositivo, inicia as variáveis copiando dados para o dispositivo no início da região, copia os resultados de volta para o hospedeiro no final da região e finalmente libera o espaço no dispositivo quando terminar.
copyin	Cria espaço para as variáveis listadas no dispositivo, inicia a variável copiando os dados para o dispositivo no início da região e libera o espaço no dispositivo quando terminar, sem copiar os dados de volta para o hospedeiro.
copyout	Cria espaço para as variáveis listadas no dispositivo, mas não as inicia. No final da região, copia os resultados de volta para o hospedeiro e libera o espaço no dispositivo.
create	cria espaço para as variáveis listadas e as libera no final da região, mas não copia nenhum dos dados de/para o dispositivo.
present	As variáveis listadas já estão presentes no dispositivo, portanto, nenhuma outra ação precisa ser executada. Isso é usado com mais frequência quando existe uma região de dados em uma rotina de maior nível que inicia os dados.
deviceptr	As variáveis listadas usam a memória do dispositivo que foi gerenciada fora do OpenACC, portanto as variáveis devem ser usadas no dispositivo sem qualquer conversão de endereço. Esta cláusula é geralmente usada quando o OpenACC é misturado com outro modelo de programação.

Tabela 3.1: Cláusulas da Diretiva Data

A região de dados de laços paralelos no exemplo a seguir permite o compartilhamento de dados entre os dois blocos de laços.

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

```
8       #pragma acc parallel loop
9         for (i=0; i<N; i++)
10             y[i] = 2.0f * x[i] + y[i];
11    }
```

3.4. Níveis de paralelismo do OpenACC

Como o OpenACC serve como uma linguagem para aceleradores genéricos existem três níveis de paralelismo que podem ser usados no OpenACC. Eles especificam o nível de paralelismo contidos em uma região paralela, são chamados de **gang, worker e vector**. Adicionalmente a execução também pode ser marcada como sequencial (**seq**). Uma *gang* é composta por um ou vários *workers*. Todos os *workers* de uma *gang* podem compartilhar os mesmos recursos, como memória cache ou processador.

Os níveis de paralelismo usado no OpenACC podem ser comparados ao níveis de execução usados na programação CUDA. Podendo assim admitir a relação entre eles: **gang = block**, **worker = warp** e **vector = threads**.

Cláusula	**Descrição**
gang	Particiona o laço entre as *gangs*
worker	Particiona o laço entre os *workers*
vector	Vetoriza o laço
seq	Não particiona o laço, que é executado sequencialmente

Tabela 3.2: Níveis de Paralelismo do OpenACC

Essas diretivas também podem ser combinadas em um laço específico. Por exemplo, um laço **gang vector** pode ser particionado entre *gangs*, cada uma delas com um *worker* implicitamente, e depois vetorizado, como exemplificado na Figura 3.5.

A especificação OpenACC reforça que o laço mais externo deve ser um laço de uma *gang*, o laço paralelo mais interno deve ser um laço *vector* e um laço *worker* pode aparecer no meio. Um laço sequencial (**seq**) pode aparecer em qualquer nível.

3.5. Diretivas e Cláusulas Avançadas

3.5.1. Cláusula collapse

A execução de um laço em OpenACC está associada ao laço imediatamente a seguir. Uma diretiva é necessária para cada laço. Isso tende a ser complicado, especialmente se vários laços devem ser tratados da mesma maneira. A cláusula **collapse** é útil nesse caso. O argumento para a cláusula **collapse** é um número inteiro positivo constante, que especifica quantos laços fortemente aninhados serão associados para criar um novo laço. Alguns benefícios que a cláusula **collapse** pode trazer são:

- colapsar os laços externos para permitir a criação de mais *gangs*.
- colapsar os laços internos para permitir comprimentos de vetor mais longos.

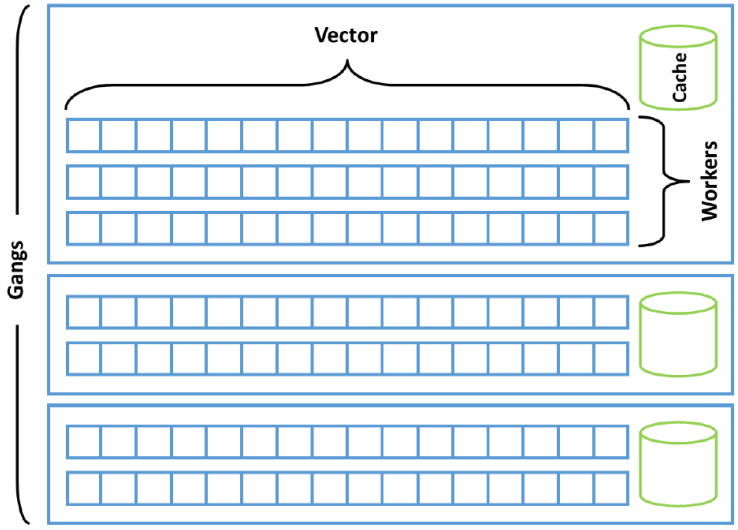


Figura 3.5: Gangs x Workers x Vector

- colapsar todos os laços, quando for possível, para fazer as duas coisas: ter mais *gangs* criadas e vetores maiores.

Esta cláusula é especialmente útil quando alguns laços não têm um número total de iterações suficientemente grande para fazer uso efetivo do acelerador. A sua sintaxe é vista a seguir:

```
#pragma acc loop collapse(n)
```

O exemplo a seguir apresenta um trecho de código com um laço com o uso desta cláusula, seguido de um laço que exemplifica o efeito do seu uso.

```
1 #pragma acc parallel loop collapse(2)
2     for (int i = 0; i < N; i++)
3          for (int j = 0; j < M; j++)
4
5 #pragma acc parallel loop
6     for (int ij = 0; ij < N*M; ij++)...
```

3.5.2. Diretiva routine

As chamadas de função ou sub-rotina em laços paralelos podem ser problemáticas para os compiladores, pois nem sempre é possível para o compilador ver todos os laços de uma só vez. Os compiladores OpenACC 1.0 eram forçados a fazer *inline* de todas as rotinas chamadas em regiões paralelas ou a não paralelizar laços contendo chamadas de rotina.

O OpenACC 2.0 introduziu a diretiva **routine**, que instrui o compilador a criar uma versão de dispositivo da função ou sub-rotina para que possa ser chamada de uma região de dispositivo. Para leitores já familiarizados com a programação CUDA, essa

funcionalidade é semelhante ao especificador da função *__device__*.

Para orientar a otimização, você pode usar cláusulas para informar ao compilador se a rotina deve ser criada para paralelismo de nível de **gang**, **work**, **vector** ou **seq** (sequencial). Você pode especificar várias cláusulas para rotinas que podem ser chamadas com vários níveis de paralelismo[6]. Fazer isso corretamente exige que você coloque uma cláusula **routine** apropriada antes da definição da rotina para chamar a rotina com o nível certo de paralelismo. O Exemplo 3.1 apresenta o uso da diretiva **routine**.

Exemplo 3.1: Diretiva **routine**

```
1  #pragma acc routine vector
2  void foo(float* v, int i, int n) {
3    #pragma acc loop vector
4    for (int j = 0; j < n; ++j)
5        v[i*n+j] = 1.0f/(i*j);
6  }
7
8  #pragma acc parallel loop
9  for ( int i=0; i<n; ++i) {
10   foo(v,i);
11   //chamada no dispositivo
12 }
```

Quando a rotina *foo* é chamada a partir do código do hospedeiro, ela será executada no hospedeiro, incrementando os valores do hospedeiro. Quando ela for chamada de dentro de uma construção paralela do OpenACC, ela incrementará os valores do dispositivo.

Teoricamente, esta diretiva permitirá o uso de funções recursivas. Contudo, há alguns fatores que limitam a profundidade da recursão. Por exemplo, os dispositivos NVIDIA estão limitados a 16 níveis de recursão, assim como os dispositivos AMD possuem seus próprios limites.

3.5.3. Operações Atômicas

Quando uma ou mais iterações de um laço precisam acessar um elemento na memória ao mesmo tempo, condições de corrida podem ocorrer. Por exemplo, se uma iteração do laço está modificando o valor contido em uma variável e outra está tentando ler a mesma variável em paralelo, diferentes resultados podem ocorrer dependendo de qual iteração ocorra primeiro.

Em programas seriais, os laços sequenciais garantem que a variável será modificada e lida em uma ordem previsível, mas os programas paralelos não garantem que uma iteração específica de um laço irá ocorrer antes da outra. Em casos simples, como encontrar uma soma, um valor máximo ou um valor mínimo, uma operação de redução

[6]O comportamento dessas cláusulas pode ser modificado em função de especificidades do compilador ou fabricante. Por exemplo, a partir da versão 14.9 do compilador PGI, uma diretiva **routine** sem nenhuma cláusula de nível de paralelismo (**gang**, **worker** ou **vector**) será tratada como se uma cláusula **seq** estivesse presente.

garantirá que o programa será executado corretamente.

Para operações mais complexas, a diretiva **atomic** garantirá apenas uma única thread executando a operação nela contida de forma exclusiva. O uso da diretiva **atomic** às vezes é uma parte necessária do processo de paralelização para garantir a correta execução do código.

A diretiva **atomic** aceita uma das quatro cláusulas seguintes para declarar o tipo de operação contida na região:

- A operação **read** assegura que duas iterações de um laço não farão leituras da região ao mesmo tempo.
- A operação **write** garantirá que não haja duas iterações realizando escrita na região ao mesmo tempo.
- Uma operação **update** é uma operação de leitura e de escrita combinadas.
- Uma operação **capture** executa uma atualização, mas salva o valor calculado nessa região para ser utilizada no código seguinte à região.

Se nenhuma cláusula for definida, uma operação **update** é assumida.

Uma forma de aplicar a diretiva **atomic** pode ser visto no Exemplo 3.2. Esse código percorre uma série de números inteiros de um intervalo conhecido e conta o total de ocorrências de cada número nesse intervalo. Ou seja, esse código implementa um histograma, que é uma técnica comum para contar quantas vezes os valores ocorrem em um conjunto de entrada de acordo com o seu valor. Como cada número no intervalo pode ocorrer várias vezes, precisamos garantir que cada elemento no vetor de histograma seja atualizado atomicamente. Ao final, esse exemplo demonstra o uso da diretiva **atomic** para gerar um histograma.

Exemplo 3.2: Diretiva **atomic**

```
1 #pragma acc parallel loop
2     for(int i=0; i < N; i++)
3         h[i]=0;
4 #pragma acc parallel loop
5     for(int i=0; i < N; i++) {
6 #pragma acc atomic update
7         h[a[i]]+=1; }
```

Observe que as atualizações no vetor do histograma *h* são executadas atomicamente. Como estamos incrementando o valor do elemento de um vetor, uma operação **update** é usada para ler o valor, modificá-lo e gravá-lo novamente.

3.5.4. Cláusula tile

A cláusula **tile** é usada conjuntamente com a diretiva **acc loop**. Com ela, é possível otimizar o laço através da operação de blocos menores para explorar o acesso aos dados,

normalmente já armazenados em uma memória do tipo *cache*. Considere a transposição de matriz no Exemplo 3.3.

Exemplo 3.3: Cláusula **tile**

```
1 #pragma acc parallel loop private(i,j) tile(8,8)
2 for(i=0; i<rows; i++)
3      for(j=0; j<cols; j++)
4         out[i*rows + j] = in[j*cols + i];
```

Ao adicionar a cláusula tile $(8,8)$ ao laço paralelo, serão criados automaticamente pelo compilador dois laços adicionais que funcionam em um *chunk* 8×8 (*tile*) da matriz antes de passar para o próximo *chunk*. Com isso, o compilador faz a otimização dentro do bloco com o objetivo de obter melhor desempenho. Embora uma transposição de matriz não tenha muita reutilização de dados, outros algoritmos podem ter uma melhora significativa no desempenho, explorando a localidade e a reutilização de dados nos laços disponíveis.

3.6. Operações Assíncronas

Após minimizar as transferências de dados, pode ser possível reduzir ainda mais as perdas de desempenho associadas a essas transferências sobrepondo as cópias de dados com outras operações no hospedeiro, no dispositivo ou em ambos. Isso pode ser alcançado no OpenACC utilizando a cláusula **async** e a diretiva **wait**.

A cláusula **async** pode ser adicionada às diretivas **parallel**, **kernels** e **update** para indicar que, após o envio da operação ao acelerador ou ao ambiente de execução, o processador pode continuar executando outras tarefas sem aguardar sua conclusão. Isso permite, por exemplo, enfileirar novas operações no acelerador ou realizar cálculos independentes do processamento em andamento no acelerador.

3.6.1. Cláusula async

Uma forma de usar a cláusula **async** pode ser vista no Exemplo 3.4.

Exemplo 3.4: Cláusula **async**

```
1 #pragma acc parallel loop async
2     for (int i=0; i<N; i++)
3          c[i] = a[i] + b[i]
4 #pragma acc update self(c[0:N]) async
```

Neste Exemplo 3.4, a *thread* do hospedeiro enfileirará a região paralela na fila assíncrona padrão, então a execução retornará à *thread* do hospedeiro para que ele também possa enfileirar a atualização e, finalmente, a *thread* do hospedeiro continuará a execução.

Eventualmente, no entanto, a *thread* do hospedeiro precisará dos resultados com-

putados no acelerador e copiados de volta para o hospedeiro usando a atualização. Então, ela deve sincronizar com o acelerador para garantir que essas operações tenham terminado antes de tentar usar os dados.

3.6.2. Diretiva wait

A diretiva **wait** instrui o ambiente de execução a aguardar a conclusão das operações assíncronas anteriores antes de prosseguir. Assim, o exemplo anterior (Exemplo 3.4) pode ser estendido para incluir uma sincronização antes que os dados copiados pela diretiva **update** sejam utilizados.

O Exemplo 3.5 ilustra essa extensão, demonstrando o uso da diretiva **wait** para garantir a sincronização adequada.

Exemplo 3.5: Diretiva **wait**

```
1 #pragma acc parallel loop async
2     for (int i=0; i<N; i++)
3          c[i] = a[i] + b[i]
4 #pragma acc update self(c[0:N]) async
5 #pragma acc wait
```

3.6.3. Explicitando dependências

Embora isso seja útil, seria ainda mais vantajoso expor as dependências dessas operações assíncronas e os respectivos pontos de espera, de modo que operações independentes pudessem ser executadas simultaneamente. Tanto **async** quanto **wait** têm um argumento opcional, um número inteiro não negativo, que especifica o número de fila para essa operação. Todas as operações colocadas na mesma fila operarão em ordem, mas operações colocadas em filas diferentes podem operar em qualquer ordem em relação umas às outras. Operações em filas diferentes podem ocorrer em paralelo, mas isso não é garantido.

Essas filas de trabalho são exclusivas por dispositivo, portanto, dois dispositivos terão filas distintas com o mesmo número. Se uma diretiva **wait** for encontrada sem um argumento, ela aguardará todo o trabalho enfileirado anteriormente naquele dispositivo. O código apresentado no Exemplo 3.6 demonstra como usar diferentes filas de trabalho para obter sobreposição de computação e transferências de dados.

Exemplo 3.6: Uso da cláusula **async** e diretiva **wait**

```
1  #pragma acc parallel loop async(1)
2      for (int i=0; i<N; i++)
3           a[i] = i;
4  #pragma acc parallel loop async(2)
5      for (int i=0; i<N; i++)
6           b[i] = 2*i;
7  #pragma acc wait(1) async(2)
8  #pragma acc parallel loop async(2)
9      for (int i=0; i<N; i++)
10         c[i] = a[i] + b[i]
11 #pragma acc update self(c[0:N]) async(2)
```

```
12 #pragma acc wait
```

Além de permitir a colocação de operações em filas separadas, também seria útil poder unir essas filas em um ponto onde os resultados de ambas sejam necessários antes de prosseguir. Isso pode ser feito adicionando-se uma cláusula **async** a uma diretiva **wait**. Como pode parecer pouco intuitivo, mostramos também no código do Exemplo 3.6 como isso é feito.

Neste mesmo Exemplo 3.6, os vetores *a* e *b* têm seus valores iniciais atribuídos por filas de trabalho separadas, permitindo que essa operação ocorra de forma independente. O **wait**(1) **async**(2) garante que a fila de trabalho 2 não prossiga até que a fila 1 seja concluída. A adição dos vetores pode então ser enfileirada no dispositivo porque os laços anteriores terão sido concluídos antes deste ponto. Por fim, o código aguarda que todas as operações anteriores sejam concluídas. Usando esta técnica, expressamos as dependências de nossos laços para maximizar a simultaneidade entre as regiões, mas ainda fornecer resultados corretos.

3.7. Programação multi-GPUs com OpenACC

Para sistemas que contêm mais de um acelerador, o OpenACC fornece uma API para fazer as operações serem realizadas em um dispositivo específico ou para transferir dados entre dispositivos de forma eficiente. Isso permite que os desenvolvedores otimizem seus aplicativos aproveitando os recursos exclusivos de cada acelerador, garantindo que as cargas de trabalho sejam equilibradas e o desempenho seja maximizado em vários componentes de hardware (OpenACC-Standard.org, 2023).

Caso um sistema contenha aceleradores de diferentes tipos, a especificação também permite consultar e selecionar dispositivos de uma arquitetura ou recursos específicos, permitindo que os desenvolvedores adaptem seu código para execução ideal. Essa flexibilidade é crucial para aplicativos que exigem poder de processamento diversificado, pois permite o uso mais eficiente de recursos, ao mesmo tempo que reduz a complexidade do gerenciamento de vários dispositivos.

3.7.1. Rotina acc_get_num_devices()

A rotina *acc_get_num_devices()* pode ser usada para consultar quantos dispositivos de uma determinada arquitetura estão disponíveis no sistema. Ao aproveitar essa rotina, é possível ajustar dinamicamente a alocação de recursos e otimizar o desempenho com base no número de dispositivos detectados. Essa rotina não apenas simplifica o processo de desenvolvimento, mas também aprimora a experiência geral do usuário, garantindo que as aplicações sejam executadas sem problemas em vários tipos de configurações de ambiente.

```
#include <openacc.h>
int acc_get_num_devices(acc_device_t devicetype);
```

A rotina aceita um parâmetro do tipo *acc_device_t* e retorna um número inteiro

de dispositivos disponíveis de uma arquitetura específica. Isso permite que seja implementada facilmente a lógica condicional no código, permitindo caminhos de execução personalizados que aproveitam ao máximo os recursos disponíveis.

O parâmetro *devicetype* pode ter um dos seguintes valores:

- acc_device_default: dispositivo padrão configurado pelo ambiente.
- acc_device_host: hospedeiro (execução sequencial).
- acc_device_nvidia: GPU NVIDIA.
- acc_device_radeon: GPU AMD.
- acc_device_gpu: qualquer GPU disponível.

A seguir um exemplo de uso:

```
1 #include <stdio.h>
2 #include <openacc.h>
3 int main() {
4     int num_gpus = acc_get_num_devices(acc_device_gpu);
5     printf("Número de GPUs disponíveis: %d\n", num_gpus);
6     return 0;
7 }
```

3.7.2. Rotina acc_get_device_num()

A rotina *acc_get_device_num()* é utilizada para obter o número do dispositivo atualmente em uso pelo programa.

```
#include <openacc.h>
int acc_get_device_num(acc_device_t devicetype);
```

O parâmetro *devicetype* pode ter os mesmos valores apresentados anteriormente. A seguir, apresentamos um exemplo da sua utilização:

```
1 #include <stdio.h>
2 #include <openacc.h>
3 int main() {
4     int device_num;
5     // Obtém o número do dispositivo atual para GPUs
6     device_num = acc_get_device_num(acc_device_gpu);
7     printf("Dispositivo GPU atual: %d\n", device_num);
8     return 0;
9 }
```

Se o programa estiver executando em um ambiente multi-GPU, essa rotina é útil para identificar e gerenciar a distribuição das tarefas entre os dispositivos disponíveis.

3.7.3. Rotina acc_set_device_num()

A rotina *acc_set_device_num()* recebe dois parâmetros: o número do dispositivo desejado e seu tipo, permitindo definir em qual dispositivo as operações subsequentes serão executadas. O uso dessa função possibilita a otimização do desempenho da aplicação, garantindo uma distribuição eficiente da carga de trabalho entre os recursos disponíveis. A sua sintaxe é apresentada a seguir:

```
#include <openacc.h>
void acc_set_device_num(int device_num, acc_device_t devicetype);
```

A chamada desta rotina se sobrepõe ao valor definido na variável de ambiente ACC_DEVICE_NUM.

Após a definição do número do dispositivo, todas as operações serão direcionadas para este dispositivo até que um outro seja especificado por uma nova chamada à função *acc_set_device_num()* ou até o término da aplicação. Essa flexibilidade possibilita ajustes dinâmicos na alocação de recursos, permitindo o uso de diferentes dispositivos computacionais ao longo da execução da aplicação, como pode ser visto a seguir:

```
1 #include <openacc.h>
2 int main() {
3     // Seleciona o dispositivo 1 de GPUs
4     acc_set_device_num(1, acc_device_gpu);
5     // Código que será executado no dispositivo selecionado
6     return 0;
7 }
```

3.7.4. Rotina acc_get_device_type()

A rotina *acc_get_device_type()* não aceita parâmetros, retorna o tipo de dispositivo do dispositivo padrão atual. O valor retornado é normalmente uma *string* que identifica o dispositivo, como “GPU” ou “CPU”. Veja o exemplo de uso a seguir.

```
1  #include <stdio.h>
2  #include <openacc.h>
3  int main() {
4      acc_device_t device_type;
5      // Obtém o tipo do dispositivo atual
6      device_type = acc_get_device_type();
7      // Exibe o tipo de dispositivo
8      if (device_type == acc_device_gpu) {
9          printf("Dispositivo atual é uma GPU.\n");
10     } else if (device_type == acc_device_host) {
11         printf("Dispositivo atual é o hospedeiro (CPU).\n");
12     } else {
13         printf("Outro dispositivo.\n");
14     }
15     return 0;
16 }
```

3.7.5. Rotina acc_set_device_type()

A função *acc_set_device_type()* especifica para o ambiente de execução o tipo de dispositivo que deve ser utilizado para as operações de aceleração, mas permite que o ambiente de execução escolha qual dispositivo desse tipo utilizar.

A rotina *acc_set_device_type()* requer um parâmetro que especifica qual o tipo do dispositivo que será utilizado para a execução das regiões paralelas seguintes. Apesar dessa indicação, ele delega para a implementação do ambiente de execução qual dispositivo escolher – idealmente, ele levará em consideração a disponibilidade e o desempenho para esta escolha.

```
#include <openacc.h>
void acc_set_device_type(acc_device_t devicetype);
```

A chamada desta rotina se sobrepõe ao valor definido na variável de ambiente ACC_DEVICE_TYPE.

3.8. Programando Múltiplas GPUs

O Exemplo 3.7 apresenta como seria a utilização dessas funções para a distribuição de trabalho entre duas GPUs.

Exemplo 3.7: Uso de Múltiplas GPUs

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <openacc.h>
4  #define N 1000000  // Tamanho do vetor
5  #define NGPU 2     // Número de GPUs
6
7  int main() {
8      int i, gpu_id;
9      float *A, *B;
10     // Alocação de memória
11     A = (float*) malloc(N * sizeof(float));
12     B = (float*) malloc(N * sizeof(float));
13     // Inicialização do vetor A
14     for (i = 0; i < N; i++)
15         A[i] = i * 1.0f;
16     // Definição da quantidade de trabalho por GPU
17     int chunk_size = N / NGPU;
18     // Processamento em múltiplas GPUs
19     #pragma acc enter data copyin(A[0:N]) create(B[0:N])
20     for (gpu_id = 0; gpu_id < NGPU; gpu_id++) {
21         int start = gpu_id * chunk_size;
22         int end = start + chunk_size;
23         // Selecionar a GPU
24         #pragma acc set device_type(acc_device_nvidia) device_num(
       ↪ gpu_id)
25
26         // Alternativamente você pode definir a GPU
```

```
27          // a ser utilizada com
28          // acc_set_device_num(gpu_id, acc_device_nvidia);
29          // e recuperá-lo com
30          // gpu_id = acc_get_device_num(acc_device_nvidia);
31
32          // Processamento assíncrono na GPU correspondente
33          #pragma acc parallel loop async(gpu_id) present(A[start:end
    ↪ ], B[start:end])
34          for (i = start; i < end; i++)
35              B[i] = A[i] * 2.0f;  // Simples multiplicação
36      }
37      // Sincronizar todas as GPUs antes de prosseguir
38      for (gpu_id = 0; gpu_id < NGPU; gpu_id++)
39          #pragma acc wait(gpu_id)
40      // Copiar resultados de volta
41      #pragma acc exit data copyout(B[0:N]) delete(A[0:N])
42      // Verificação dos primeiros valores
43      printf("B[0] = %f, B[N-1] = %f\n", B[0], B[N-1]);
44      // Liberação de memória
45      free(A);
46      free(B);
47      return 0;
48  }
```

O código acima demonstra uma abordagem correta para utilizar múltiplas GPUs, distribuindo o trabalho em blocos e utilizando a diretiva **async** para permitir a execução paralela. A diretiva **#pragma acc set** permite direcionar o trabalho para GPUs NVIDIA específicas, o que é essencial em um ambiente multi-GPU. A sintaxe, que também foi utilizada no Exemplo 3.7, é:

```
#pragma acc set device_type() device_num()
```

As diretivas **#pragma acc enter data** e **#pragma acc exit data** fornecem um controle claro sobre a movimentação dos dados entre o hospedeiro e as GPUs. O uso de **#pragma acc wait** garante que todas as operações nas GPUs sejam concluídas antes que os resultados sejam acessados no hospedeiro.

Essa flexibilidade de funções do OpenACC garante a utilização ideal de recursos disponíveis, permitindo que as aplicações sejam executadas de forma eficiente e mais adequada aos recursos disponíveis a qualquer momento no sistema.

3.9. Estudo de caso

A seguir, apresentamos um estudo de caso utilizando múltiplas GPUs para a execução de um código para geração de um fractal de Mandelbrot.

Antes de iniciar o processo de execução de uma aplicação para uso de múltiplas GPUs, é preciso verificar se o ambiente possui todas as características para o perfeito funcionamento, garantindo que existam, no mínimo, duas GPUs no ambiente e se elas estão prontas para uso.

Alguns comandos podem ser utilizados para esta verificação, dentre eles o *nvidia-smi* e o *pgaccelinfo*. Um exemplo do uso desses comandos, bem como seus parâmetros e a filtragem das respostas, está a seguir:

```
$ nvidia-smi -q

==============NVSMI LOG==============
Timestamp                                 : Wed Mar 19 12:31:33 2025
Driver Version                            : 470.74
CUDA Version                              : 11.4
Attached GPUs                             : 2
```

```
$ pgaccelinfo | grep "Device Number"
Device Number:                 0
Device Number:                 1
```

A partir deste ponto, deve-se garantir que a versão do compilador utilizado tem suporte às funções para o uso de múltiplas GPUs. O compilador escolhido em nossos testes faz parte do NVIDIA HPC SDK versão 21.7.

```
$ nvc --version

nvc 21.7-0 64-bit target on x86-64 Linux -tp nehalem
NVIDIA Compilers and Tools
Copyright (c) 2021, NVIDIA CORPORATION & AFFILIATES.  All rights
reserved.
```

Para o processamento em múltiplas GPUs, usaremos como estudo de caso o conjunto de Mandelbrot[7]. Como explicado anteriormente, será necessário o uso das rotinas *acc_get_num_devices()* e *acc_set_device_num()* para distribuição das tarefas entre as GPUs.

Exemplo 3.8: Código de Mandelbrot para uso em múltiplas GPUs

```
1  #include <cstdio>
2  #include <cstdlib>
3  #include <fstream>
4  #include <omp.h>
5  #include <openacc.h>
6  #include "mandelbrot.h"
7  #include "constants.h"
8
9  using namespace std;
10
11 int main() {
12
```

[7] <https://github.com/OpenACC/openacc-best-practices-guide/blob/main/examples/mandelbrot/cpp/task5.multithread/>

```
13    size_t bytes = WIDTH*HEIGHT*sizeof(unsigned int);
14    unsigned char *image = (unsigned char*)malloc(bytes);
15    int num_blocks=64, block_size = (HEIGHT/num_blocks)*WIDTH;
16    FILE *fp = fopen("image.pgm","wb");
17    fprintf(fp, "P5\n%s\n%d %d\n%d\n","#comment", WIDTH, HEIGHT,
        ↪ MAX_COLOR);
18
19    int num_gpus = acc_get_num_devices(acc_device_nvidia);
20  // This parallel section eats the cost of initializing the devices
21  // to prevent the initialization time from skewing the results.
22  #pragma omp parallel num_threads(num_gpus)
23  {
24    acc_init(acc_device_nvidia);
25    acc_set_device_num(omp_get_thread_num(),acc_device_nvidia);
26  }
27    printf("Found %d NVIDIA GPUs.\n", num_gpus);
28
29    double st = omp_get_wtime();
30  #pragma omp parallel num_threads(num_gpus)
31  {
32    int queue = 1;
33    int my_gpu = omp_get_thread_num();
34    acc_set_device_num(my_gpu,acc_device_nvidia);
35    printf("Thread %d is using GPU %d\n", my_gpu, acc_get_device_num(
        ↪ acc_device_nvidia));
36  #pragma acc data create(image[WIDTH*HEIGHT])
37  {
38    #pragma omp for schedule(static,1)
39    for(int block = 0; block < num_blocks; block++ ) {
40      int start = block * (HEIGHT/num_blocks),
41          end   = start + (HEIGHT/num_blocks);
42  #pragma acc parallel loop async(queue)
43      for(int y=start;y<end;y++) {
44        for(int x=0;x<WIDTH;x++) {
45          image[y * WIDTH + x] = mandelbrot(x, y);
46        }
47      }
48  #pragma acc update self(image[block * block_size : block_size])
      ↪ async(queue)
49      queue = (queue + 1) % 2;
50    }
51  }
52  #pragma acc wait
53  } // OMP Parallel
54
55    double et = omp_get_wtime();
56    printf("Time: %lf seconds.\n", (et - st));
57    fwrite(image, sizeof(unsigned char), WIDTH * HEIGHT, fp);
58    fclose(fp);
59    free(image);
60    return 0;
61  }
```

No início do código, utiliza-se a rotina *acc_get_num_devices(acc_device_nvidia)*

para identificar a quantidade de GPUs existentes no sistema. A quantidade de GPUs identificadas será utilizada como valor para a variável *num_gpus*.

Uma vez atribuído um valor à variável *num_gpus*, utilizamos a diretiva **#pragma omp parallel num_threads(num_gpus)** para especificar a quantidade de *threads* que serão criadas para a região paralela, que, neste caso, é o valor da variável *num_gpus* que indica o número de *thread* criadas.

Com o uso da diretiva **#pragma acc parallel loop async(queue)** o laço é executado em paralelo nas múltiplas *threads*. Cada *thread* identificada em *omp_get_thread_num()* será responsável pelo controle de execução de cada GPU inicializada com *acc_set_device_num()* e já identificada no sistema. Após a execução dos laços nas diferentes GPUs, usamos a diretiva *#pragma acc wait* para garantir que as tarefas executadas nas GPUs e no hospedeiro sejam sincronizadas.

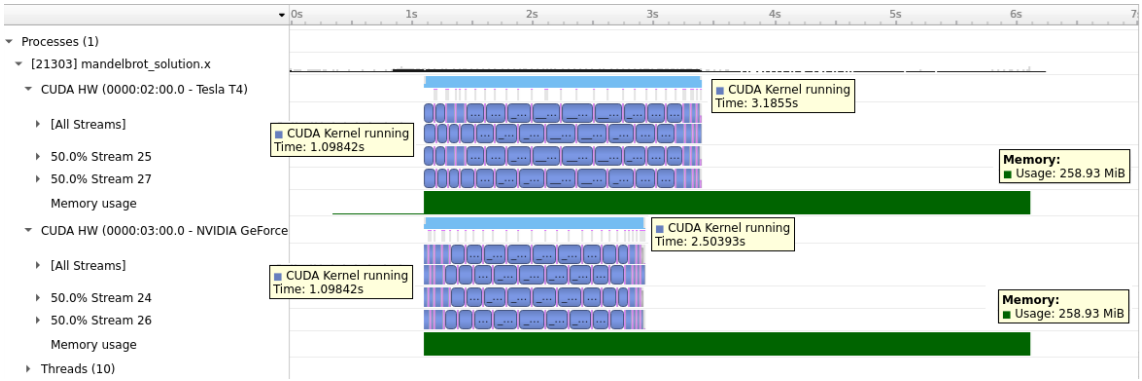
Em seguida, compilamos e executamos o código conforme mostrado a seguir:

```
$ make mandelbrot_solution.x
nvc++ -fast -acc=gpu -gpu=pinned -Minfo=all -mp -o mandelbrot_solution
.x mandelbrot.o main_solution.o

$ ./mandelbrot_solution.x
Found 2 NVIDIA GPUs.
Thread 0 is using GPU 0
Thread 1 is using GPU 1
Time: 2.280399 seconds.
```

Para a análise da utilização dos recursos durante o processo de execução do código, utilizamos o NVIDIA Nsight Graphics, cujos resultados são apresentados na Figura 3.6. Conforme podemos observar, o processo de execução foi dividido entre as duas GPUs identificadas pelo sistema. Ambas iniciaram ao mesmo tempo em 1,09 segundos; a primeira GPU finalizou o seu tempo de execução em 3,18 segundos – um total de 2,09 segundos; a segunda GPU finalizou o seu tempo de execução em 2,50 segundos – total de 1,41 segundos. A divisão entre as GPUs não foi exatamente igual e isso ocorre, principalmente, porque são GPUs que não possuem as mesmas características. Em relação ao uso de memória, ambas as GPUs utilizam praticamente a mesma quantidade: 258 MB.

Figura 3.6: Código de MandelBrot para uso em múltiplas GPUs



3.10. Conclusão

Apresentamos os principais conceitos de programação paralela para processadores multinúcleo e aceleradores com OpenACC, além de estratégias para otimizar a execução de códigos em sistemas multi-GPU. Esses princípios e estratégias são explorados em um estudo de caso para demonstrar sua eficácia. Percebemos, assim, que o OpenACC e suas primitivas permitem lidar com a crescente complexidade dos problemas computacionais e aproveitar o avanço dos hardwares modernos, que exigem estratégias eficientes de paralelismo.

Também discutimos os principais tipos de arquiteturas de aceleradores e os modelos básicos de programação com OpenACC, abordando a integração dessas APIs para criar aplicações paralelas eficientes em sistemas com mais de um acelerador. A combinação de múltiplas GPUs oferece vantagens como a redução do tempo total da aplicação, a exploração de níveis adicionais de paralelismo – incluindo o uso de CPU com OpenMP (como exemplificado no Exemplo 3.8) –, além da melhoria do balanceamento de carga. Além disso, exploramos os diferentes níveis de suporte a operações assíncronas, destacando as diretivas **async** e **wait** e seu impacto no desempenho.

A otimização da movimentação de dados entre o hospedeiro e o acelerador foi destacada como um fator crítico para o desempenho, sendo apresentadas as diretivas de dados como **data**, **copy**, **copyin**, **copyout**, **create**, **present** e **deviceptr**. Técnicas avançadas como a sobreposição de transferências com computação, utilizando as cláusulas **async** e a diretiva **wait**, foram detalhadas para mitigar as penalidades de desempenho associadas à movimentação de dados.

Também abordamos as diretivas e as cláusulas avançadas do OpenACC, incluindo **collapse** para otimizar laços aninhados, **routine** para criar versões de dispositivo de funções, **atomic** para garantir acesso seguro à memória compartilhada, e **tile** para melhorar a localidade de dados nos laços.

Para facilitar a gestão de múltiplos aceleradores, demonstramos como utilizar as rotinas da API, como **acc_get_num_devices()**, **acc_get_device_num()**, **acc_set_device_num()**, **acc_get_device_type()** e **acc_set_device_type()**, distribuindo a carga de trabalho de forma eficiente.

O estudo de caso do fractal de Mandelbrot demonstrou, na prática, como a utilização eficiente de múltiplas GPUs e técnicas de paralelismo pode reduzir significativamente o tempo de execução.

Todos os exemplos deste minicurso, juntamente com as orientações para sua compilação, estão disponíveis no seguinte repositório <https://github.com/Programacao-Paralela-e-Distribuida/OpenACC>.

Referências

BIANCHINI, C. P.; COSTA, E. B.; SILVA, G. P. Programação paralela híbrida: Mpi + openmp offloading. In: LORENZON, A. F.; FAZENDA, A. L. (Ed.). *Minicursos do SSCAD 2024*. [S.l.]: Sociedade Brasileira de Computação, 2024. p. 45–67.

DEAKIN, T.; MATTSON, T. G. *Programming your GPU with openmp: Performance portability for gpus*. 1. ed. [S.l.]: The MIT Press, 2023.

KLEMM, M.; COWNIE, J. *High performance parallel runtimes: Design and implementation*. 1. ed. [S.l.]: De Gruyter Oldenbourg, 2021.

NVIDIA. *NVIDIA H100 Tensor Core GPU Architecture*. [S.l.], 2022. Available from Internet: <https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper?ncid=no-ncid>.

OpenACC-Standard.org. *OpenACC Programming and Best Practices Guide*. OpenACC-Standard.org, 2023. Available from Internet: <https://openacc-best-practices-guide.readthedocs.io/en/latest/index.html>.

SILVA, G. P.; BIANCHINI, C. P.; COSTA, E. B. *Programação Paralela e Distribuída com MPI, OpenMP e OpenACC para computação de alto desempenho*. [S.l.]: CasaDoCodigo, 2022.