 DOI: 10.5753/sbc.16630.8

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

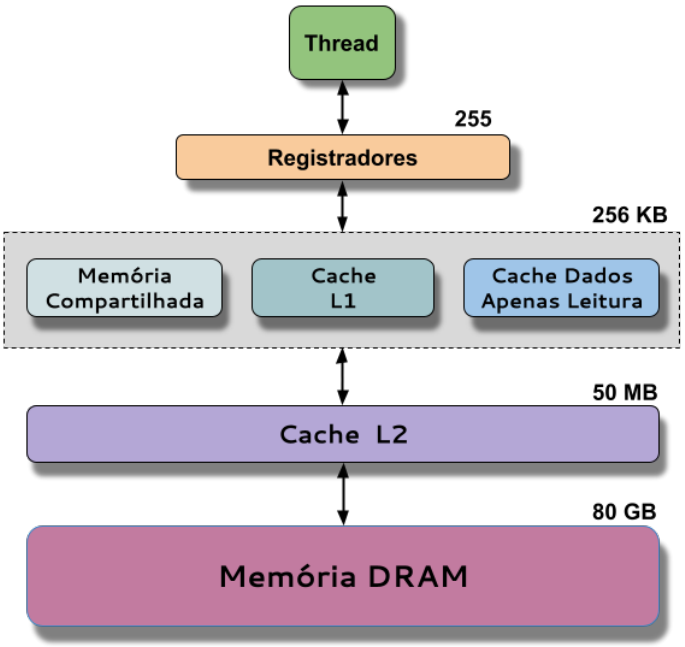
Figura 3.2: Multiprocessador de Fluxo (SM)

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>

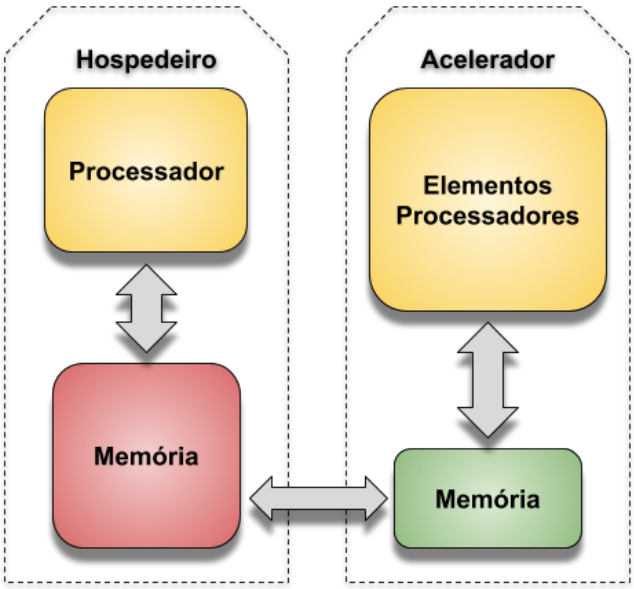
Maiores detalhes podem ser vistos na referência (NVIDIA, 2022).

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.

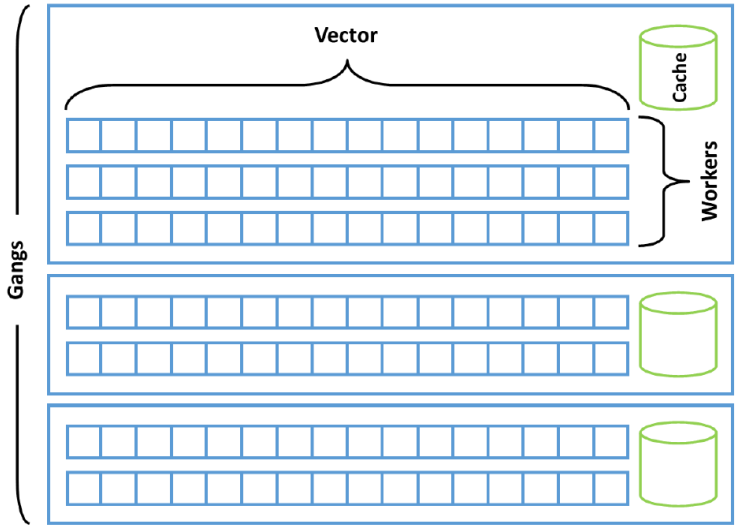


Figura 3.5: Gangs x Workers x Vector

- 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:

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

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.

```c
#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:

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

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.

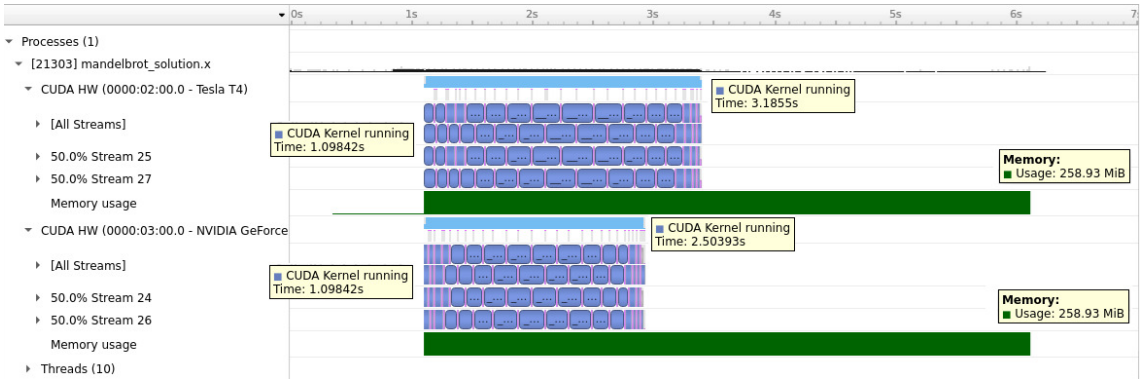
Em seguida, compilamos e executamos o código conforme mostrado a seguir:

```
$ 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.