

# Programação Paralela e Distribuída – Parte 3

**Gabriel P. Silva**

**Evaldo B. Costa**

# Programação Paralela e Distribuída

com MPI, OpenMP e OpenACC  
para computação de alto desempenho



 Casa do  
Código |  alura

GABRIEL P. SILVA  
CALEBE P. BIANCHINI  
EVALDO B. COSTA

Os códigos fontes utilizados neste material  
estão disponíveis em:

<https://github.com/Programacao-Paralela-e-Distribuida/OPENACC>

# OpenACC

- O OpenACC foi desenvolvido pelos principais fabricantes de hardware e software como Cray, CAPS, NVIDIA e PGI, com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e tornando possível a portabilidade do código independente de qual arquitetura em que foi desenvolvido inicialmente.
- O OpenACC é compatível com os modelos de programação OpenMP e MPI, ambas as abordagens podem ser combinadas com o OpenACC.
- Em geral, as diretivas do OpenACC são muito semelhantes às do OpenMP.
- Em relação ao CUDA, OpenACC é totalmente compatível tornando a necessidade de alteração do código a menor possível.

# OpenACC

- Independente de fabricante;
- Oculta a complexidade do hardware dos programadores;
- Requer poucas modificações ao código fonte;
- Mais fácil de programar e depurar que o CUDA;
- Possui algumas facilidades que o CUDA não oferece;
- Mesmo código por ser usado em multicore, manycore e GPUs;
- Similar ao OpenMP (familiaridade);
- Fácil transição para o OpenMP 4.5 (futuro).

# OpenACC

- Em um processamento convencional, as threads são executadas por processadores escalares.
- A GPU é construída em torno de um arranjo escalável de multiprocessadores de fluxo (Streaming Multiprocessor - SM) capazes de executar um conjunto de threads simultaneamente, que recebem o nome de ``blocos de thread''.
- Um laço de programa é transformado em uma grade de kernel, contendo vários blocos de threads, que são numerados e distribuídos para os SM com capacidade de execução disponível.

# OpenACC

- Um bloco de threads pode ser executado concorrentemente em um multiprocessador (SM).
- Assim que um bloco de thread termina a sua execução, novos blocos são lançados para os multiprocessadores de fluxo(SMs) ociosos.

# Modelo de Programação

## Software



Thread



Bloco  
Thread

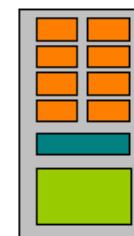


Grade

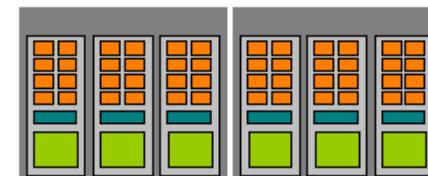
## Hardware



Processador  
Escalar



Multiprocessadores

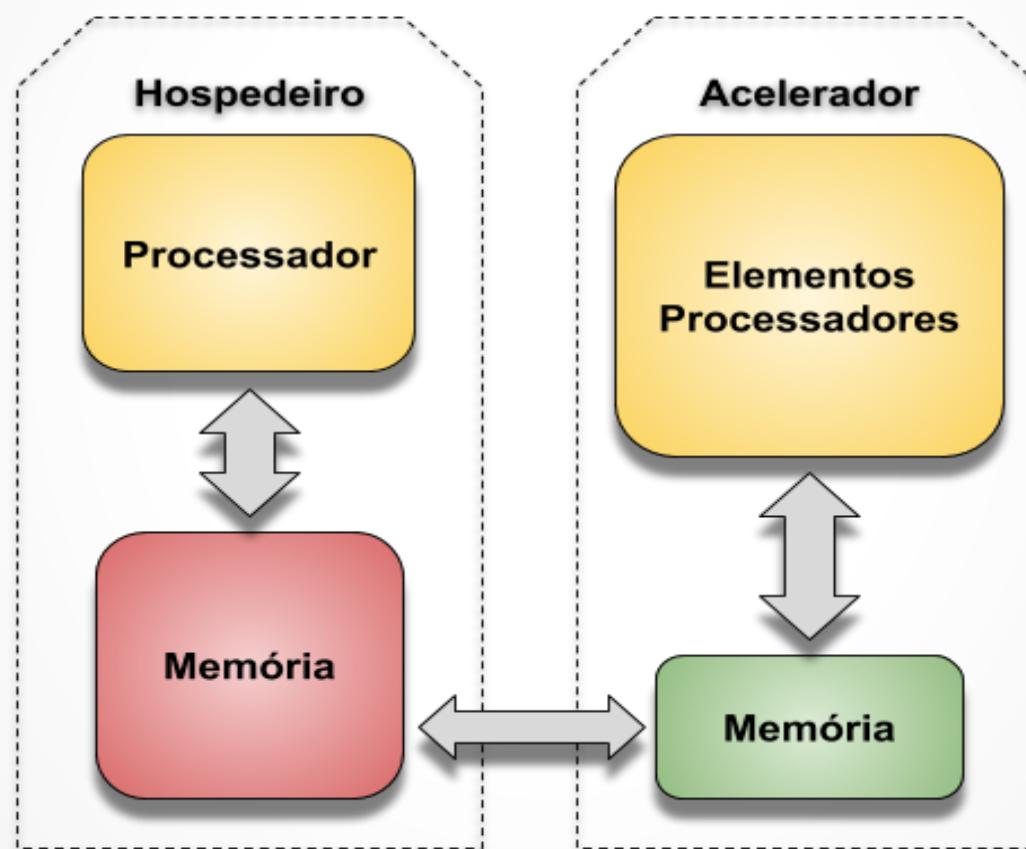


Dispositivo

# OpenACC

- Uma observação importante a ser feita, ao tentar transportar um código sequencial para uma GPU, é que normalmente os aceleradores e as CPUs não compartilham a mesma memória.
- Em outras palavras, o acelerador não tem acesso direto à memória da CPU. A memória da CPU normalmente é maior, porém mais lenta que a memória do acelerador.
- Para usar a GPU os dados devem ser transferidos da memória principal, no host, para a memória da GPU através de barramentos de E/S, que possuem largura de banda (velocidade) muito menor que a velocidade de acesso das GPUS e da CPUs suas respectivas memórias.
- Isso significa que o gerenciamento das transferências de dados entre as memória da GPU e da CPU é de capital importância para um bom desempenho da aplicação.

# Modelo de Dados



# Níveis de Paralelismo

- 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, e são chamados de gangue (gang), trabalhador (worker) e vetor (vector). Adicionalmente a execução também pode ser marcada como sequencial (seq).
- Esse tipo de paralelismo é mais útil quando estamos trabalhando com laços multi-dimensionais aninhados, ou seja, com duas ou mais varáveis de iteração.

# Níveis de Paralelismo

- Uma *gangue* é composta por um ou vários *trabalhadores*, sendo que todos os *trabalhadores* de uma *gangue* podem compartilhar os mesmos recursos, como memória cache ou multiprocessador de fluxo (SM).
- A cláusula **worker** é uma forma de o programador ter vários vetores em uma única *gangue*.
- O objetivo principal do *trabalhador* é dividir um vetor maior em vários vetores de menor tamanho.
- Isso pode ser útil quando os laços internos são muito pequenos, e não vão se beneficiar pelo uso de um vetor maior.
- Os *trabalhadores* então computam um vetor, sendo que os *vetores* trabalham de forma sincronizada (lockstep) usando paralelismo SIMD/SIMT.

# Níveis de Paralelismo

- Um vetor é o nível mais baixo de paralelismo, sendo que cada *gangue* vai ter pelo menos um vetor. Cada vetor tem a habilidade de executar uma única instrução em vários elementos de dados.
- O OpenACC procurar defini-los de uma maneira genérica, sem vinculá-los a um tipo específico de *hardware*.
- Os níveis de paralelismo usados no OpenACC podem ser comparados aos níveis de execução usados na programação CUDA. Podendo assim admitir a relação entre eles: **gang = block, worker = warp e vector = threads**.
- Essas cláusulas também podem ser combinadas em um laço específico

# Níveis de Paralelismo

**Gangue**



**Trabalhadores**

# Níveis de Paralelismo

- A especificação OpenACC reforça que o laço mais externo deve ser um laço de uma *gangue*, o laço paralelo mais interno deve ser um laço *vetor* e um laço *trabalhador* pode aparecer no meio, mas usualmente é omitido.
- Um laço sequencial (cláusula **seq**) pode aparecer em qualquer nível.
- Os aceleradores podem ter limitações quanto aos valores em relação aos tamanhos que podem ser atribuídos a esses particionamentos.
- Por exemplo, para GPUS da NVIDIA, as seguintes limitações existem:
  - O comprimento de um *vetor* deve ser um múltiplo de 32 (até 1024)
  - O tamanho de uma *gangue* é dado pelo número de *trabalhadores* vezes o tamanho de um *vetor*, e não pode ser maior que 1024.

# Diretivas Principais

# Diretivas OpenACC

```
#pragma acc diretiva [cláusula]
```

```
{
```

```
    região de código ...
```

```
}
```

# Diretiva kernels

## #pragma acc kernels [cláusulas]

- Com uso da diretiva **kernel**s, o compilador analisará o código e apenas paralelizará quando tiver certeza de que é seguro fazê-lo.
- Em alguns casos, o compilador pode não ter informações suficientes para determinar se é seguro paralelizar um laço, nesse caso essa paralelização não será feita.
- O compilador OpenACC e o ambiente de execução cuidam de tudo isso de forma transparente para o programador; mas que pode não ótima.

# Diretiva kernels

```
#include <stdlib.h>
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
#pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
int main(int argc, char *argv[ ])          /* acc_saxpy.c */
int n = 1<<20; // 1 milhão de floats
float *x = (float*) malloc(n*sizeof(float));
float *y = (float*) malloc(n*sizeof(float));
#pragma acc kernels
    for (int i = 0; i < n; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(n, 3.0f, x, y);
    free(x);
    free(y);
    return(0);
}
```

# Aliasing

- No caso do código C/C++, em que as matrizes são passadas para as funções como ponteiros, o compilador nem sempre pode ser capaz de determinar se as duas matrizes não compartilham a mesma área de memória, também conhecido como *aliasing* de ponteiros.
- Se o compilador não puder saber que os dois ponteiros não possuem *aliasing*, não será capaz de paralelizar um laço que acessa essas matrizes.
- Uma prática recomendada para os programadores em C é usar a palavra-chave **restrict** (ou o decorador **restrict** em C++) sempre que possível, para informar ao compilador que os ponteiros não têm aliasing e frequentemente fornecerá ao compilador informações suficientes para paralelizar laços que seriam executados sequencialmente de outra forma.

# Aliasing

- Se o programador violar essa declaração, o comportamento do programa não poderá ser definido. Ou seja, o programador deve garantir que os valores escritos pelo ponteiro não afetarão os valores lidos por qualquer outro ponteiro existente no mesmo contexto.
- Além da palavra-chave **restrict**, pode-se declarar variáveis constantes usando a palavra-chave **const**, para permitir que o compilador armazene essa variável em memória apenas de leitura, caso esse tipo de memória exista no acelerador.
- O uso de **const** e **restrict** é uma boa prática de programação em geral, pois fornece ao compilador informações adicionais que podem ser usadas na otimização do código.

# Diretiva parallel

## #pragma acc parallel [cláusulas]

A diretiva **parallel** informa ao compilador para criar uma região paralela, mas diferentemente da região **ernels**, o código na região paralela será executado redundantemente por todas as *gangues*. Não haverá nenhum compartilhamento de tarefas!

- Dentro da região paralela os laços que serão paralelizados precisam ser indicados no código pelo programador.
- Isso pode ser feito com o uso da diretiva **loop**, fazendo o compilador gerar uma versão paralela do laço para o acelerador. Caso a diretiva **loop** não seja especificada, a paralelização dos laços não será feita.
- Se não for utilizada uma cláusula **async**, vai haver uma barreira implícita no final da região paralela, ou seja, a execução da *thread* no *host* local não vai prosseguir enquanto todas as *gangues* não chegarem ao final da região paralela.

# Diretiva parallel

```
#include <stdlib.h>
void saxpy(int n, float a, float * restrict x, float *restrict y) {
    #pragma acc parallel
    #pragma acc loop
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
int main(int argc, char **argv) { /* acc_saxpy2.c */
    int n = 1<<20; // 1 milhão de floats
    if (argc > 1)
        n = atoi(argv[1]);
    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));
    #pragma acc parallel
    #pragma acc loop
    for (int i = 0; i < n; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(n, 3.0f, x, y);
    return 0;
}
```

# Diretiva kernels x parallel

- O uso da construção **kernel**s pode cobrir um grande trecho de código com apenas uma única diretiva.
- Essa construção fornece ao compilador uma margem de manobra maior para paralelizar e otimizar o código da forma que ele considere adequada para o tipo de acelerador utilizado, mas também depende muito da capacidade do compilador de paralelizar automaticamente o código.
- Como resultado, o programador pode ver diferenças de desempenho e de nível de otimização quando utilizar compiladores diferentes.

# Diretiva kernels x parallel

- O uso da diretiva **parallel** é uma afirmação do programador de que é seguro e desejável paralelizar a região afetada.
- Isso depende de o programador ter identificado corretamente o paralelismo no código e removido qualquer coisa no código que não seja segura paralelizar.
- Se o programador afirmar incorretamente que um laço pode ser paralelizado, o programa resultante poderá produzir resultados incorretos.
- Em outras palavras: a construção **kernels** pode ser pensada como uma dica para onde o compilador deve procurar paralelismo, enquanto a diretiva **parallel** é uma afirmação para o compilador onde há paralelismo.

# Diretiva kernels x parallel

- Um dos maiores pontos de confusão para os novos programadores de OpenACC é o motivo pelo qual a especificação possui as diretivas **paralell** e **kernels**, que parecem fazer a mesma coisa.
- Embora elas estejam fortemente relacionadas, existem diferenças sutis entre elas, que possuem características distintas para cada tipo de aplicação e são usadas de acordo com a necessidade de execução do código.
- Existem códigos que são fáceis de alterar e obtém melhor desempenho usando a diretiva **parallel**, porém existem códigos que possuem grande dificuldade de alteração não sendo possível usar diretiva **parallel**, nesses casos é utilizada a diretiva **kernels**, pois as alterações são as mínimas possíveis.

# Diretiva kernels x parallel

```
#pragma acc kernels
{
    for (i = 0; i < n; i++)
        a[i] = 3.0f*(float)(i+1);
    for (i = 0; i < n; i++)
        b[i] = 2.0f*a[i];
}
```

# Diretiva kernels x parallel

```
#pragma acc parallel
{
#pragma acc loop
    for (i = 0; i < n; i++)
        a[i] = 3.0f*(float)(i+1);
#pragma acc loop
    for (i = 0; i < n; i++)
        b[i] = 2.0f*a[i];
}
```

# Diretiva parallel loop

- As diretivas combinadas do OpenACC **parallel loop** e **kernels loop** são atalhos para especificar uma diretiva **loop** imediatamente após uma diretiva **parallel** ou **kernel**.
- O resultado é o mesmo que especificar explicitamente uma diretiva **parallel** ou **kernels** contendo uma diretiva **loop**.
- Qualquer cláusula permitida em uma diretiva **parallel** ou na diretiva **loop** é permitida na diretiva **parallel loop**.
- Qualquer cláusula permitida em uma diretiva **kernels** ou na diretiva **loop** é permitida na diretiva **kernels loop**.

# Diretiva kernels loop

```
#pragma acc kernels loop
for (int i = 0 ; i < N; i++)
    a[i] = 0;
#pragma acc kernels loop
for (int j = 0 ; j < M; j++)
    b[j] = 0;
```

# Diretiva parallel loop

```
#pragma acc parallel loop
  for (int i = 0 ; i < N; i++)
    a[i] = 0;
#pragma acc parallel loop
  for (int j = 0 ; j < M; j++)
    b[j] = 0;
```

# Movimentação de Dados

# Movimentação de Dados

- A movimentação de dados entre o hospedeiro e o acelerador é feita através de um barramento de E/S, que é lento em comparação com a largura de banda de memória. Por sua vez o acelerador não pode executar o processamento dos dados até que eles estejam na sua memória local.
- Para realizar a movimentação de dados entre o *host* e o acelerador durante a execução do programa é necessário o uso das cláusulas de dados.
- As cláusulas de dados dão ao programador controle adicional sobre como e quando os dados são criados e copiados de/para um dispositivo.
- As cláusulas de movimentação de dados podem ser usadas nas construções **data**, **kernels** ou **parallel**.

# Cláusula default none

- Caso a cláusula **default(none)** seja utilizada na região paralela, o programador deve definir o escopo de todas as variáveis explicitamente, em caso contrário o compilador vai emitir uma mensagem de erro.
- Se não houver uma cláusula **default(none)** o compilador vai determinar implicitamente os atributos para as variáveis que não estiverem listadas na diretiva **data**.
- Por exemplo, as variáveis escalares receberão um atributo **firstprivate**; os índices dos laços serão **private** por padrão; vetores e matrizes serão compartilhados por padrão, sendo que o compilador escolhe o tipo: **copyin, copyout, etc.**

# Diretiva data

- **#pragma acc data [cláusula]**
- A diretiva **data** define quais escalares, arranjos e sub-arranjos devem ser alocados na memória do dispositivo durante o tempo de duração da região, definindo também se os dados devem ser copiados da memória do hospedeiro para o dispositivo na entrada da região e/ou copiados da memória do dispositivo para o hospedeiro, na saída da região.

# Cláusulas data

- Usualmente as cláusulas das diretiva **data** são utilizadas para definir o seguinte escopo das variáveis compartilhadas:
- **copyin**: uma variável compartilhada que é utilizada apenas como **leitura** pelo dispositivo.
- **copyout**: uma variável compartilhada que é usada apenas para **escrita**.
- **copy**: uma variável compartilhada que é usada para **leitura e escrita**.
- **create**: uma variável compartilhada que é apenas um espaço de rascunho para dados temporários (embora haja uma cópia não utilizada no *host* nesse caso).

# Cláusulas data

- **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 *host* 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 *host*.
- **copyout** - Cria espaço para as variáveis listadas no dispositivo, mas não as inicia. No final da região, copia os resultados de volta para o *host* e libera o espaço no dispositivo.

# Cláusulas data

- **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 já movimentou os dados para o dispositivo.
- **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.

# **Cláusulas da Diretiva Parallel**

# Cláusulas private e firstprivate

## #pragma acc parallel private (lista-de-variáveis)

- A cláusula **private** da construção **parallel** vai privatizar as variáveis listadas para cada gangue na região paralela, ou seja cada gangue vai ter uma cópia própria das variáveis da lista.

## #pragma acc parallel firstprivate (lista-de-variáveis)

- A cláusula **firstprivate** da construção **parallel** vai privatizar as variáveis listadas para cada gangue na região paralela, e a cópia será iniciada com o valor desse item na thread do hospedeiro quando a construção **parallel** for encontrada.
- A diretiva **loop** também faz uso da cláusula **private**, como veremos mais adiante.

# Cláusula default(**none**)

- A cláusula **default(*none*)** é opcional.
- Ela indica que o compilador não deve determinar implicitamente o atributo de dados das variáveis, mas sim obrigar que todas as variáveis ou arranjos utilizados na região, que não tenham os atributos de dados pré-determinados, apareçam explicitamente em uma cláusula **data** para a região ou para uma região em que essa região esteja contida.
- Caso isso não seja feito o compilador vai emitir uma mensagem de erro.

**#pragma acc parallel default (*none*)**

# Cláusula reduction

- A cláusula **reduction** funciona de forma similar à cláusula **private** de forma que é gerada uma cópia privada das variáveis, mas existe uma redução ao final da região paralela de todas as cópias privadas em um único resultado final, que é retornado ao sair da região paralela.
- A redução pode apenas ser especificada em uma variável escalar e as operações de redução possíveis são: soma; multiplicação; máximo; mínimo; **e** bit a bit; **ou** bit a bit; **not**; **e** lógico; **ou** lógico.

**#pragma acc parallel reduction (operador:variavel)**

# Operadores da Cláusula reduction

operador	valor inicial
+	0
*	1
max	menor rep.
min	maior rep.
&	$\sim 0$
	0
^	0
& &	1
	0

# Cláusula if

- A cláusula **if** é opcional nas construções **parallel** e **kernels**.
- Quando não houver nenhuma cláusula **if**, o compilador vai gerar código para executar a região no dispositivo acelerador.
- Quando houver uma cláusula **if**, o compilador vai gerar duas cópias da região paralela, uma cópia para executar a região no acelerador e uma cópia para executar na *thread local* no hospedeiro.
- Quando a condição avaliada for diferente de ``0", a cópia do acelerador será executada. Em caso contrário, a cópia da região para a *thread local* será executada.

**#pragma acc parallel if (condição)**

**#pragma acc kernels if (condição)**

# Cláusulas num\_gangs, num\_workers, vector\_length

**#pragma acc parallel num\_gangs(expr-int)**

**#pragma acc parallel num\_workers(expr-int)**

**#pragma acc parallel vector\_length(expr-int)**

- O comportamento padrão é deixar ao compilador determinar quantos trabalhadores e o comprimento dos seus vetores. O número de gangues é definido pelo ambiente de execução quando o programa for executados, sendo que o uso da memória é normalmente o critério limitante.
- Contudo, o programador pode definir esses parâmetros dentro de uma região parallel ou kernels com o uso das cláusulas num\_gangs(N), num\_workers(M), vector\_length(Q).
- Essas cláusulas são úteis principalmente se o código utiliza uma estrutura de dados que é difícil para o compilador analisar. O número ótimo de gangues é altamente dependente do tipo de arquitetura, portanto use-as com cautela.

# **Cláusulas da Diretiva loop**

# Cláusula gang

- A cláusula **gang** em uma região **parallel**, especifica que as iterações dos laços associados devem ser executadas em paralelo através da distribuição das iterações entre as gangues criadas pela diretiva **parallel**.
- O número de gangues é controlado pela diretiva **parallel**, sendo que apenas o argumento **static**, visto a seguir, é permitido.

**#pragma acc loop gang [(expr-int)] [(static:[\*] [expr-int])]**

- O escalonamento de iterações do laço para as *gangues* não é especificado a menos que o argumento **static** seja utilizado, nesse caso o valor inteiro informado após o argumento é o tamanho do *chunksize*, ou seja, a quantidade de iterações que é atribuída a cada *gangue*.

# Cláusula worker

- Em uma região **parallel**, a cláusula **worker** especifica que as iterações do laço ou laços associados devem ser executadas em paralelo, distribuindo as iterações entre os vários trabalhadores dentro de uma única gangue.
- Em uma região **kernels** do acelerador, a cláusula **worker** especifica que as iterações do laço ou laço associados devem ser executadas em paralelo entre os trabalhadores de uma gangue criada por qualquer kernel contido no laço ou laços.
- Se um argumento for especificado, ele indica quantos trabalhadores por gangue devem ser usados para executar as iterações deste laço.

**#pragma acc loop worker([num:]int-expr)]{}**

- A região de um laço com a cláusula **worker** não pode conter um laço com uma cláusula **gang** ou **worker**, a menos que esteja dentro de uma região **kernels** ou **parallel** aninhada.
- Todos os trabalhadores concluirão a execução de todas as iterações que lhe forem atribuídas, antes que qualquer trabalhador prossiga além do final do laço.

# Cláusula vector

- Em uma região **parallel** do acelerador, a cláusula **vector** especifica que as iterações do laço ou laços associados devem ser executadas no modo vetorial ou SIMD. Uma construção de laço com uma cláusula **vector** faz com que um trabalhador faça a transição do modo de vetor único para o modo de particionamento de vetor.

**#pragma acc loop vector [([length:]int-expr)]**

- Em uma região **kernels** do acelerador, a cláusula **vector** especifica que as iterações do laço ou laços associados devem ser executadas com processamento vetorial ou SIMD.
- Se um argumento for especificado, as iterações serão processadas em vetores desse comprimento; se nenhum argumento for especificado, a implementação escolherá um comprimento de vetor apropriado.
- A região de um laço com a cláusula **vector** não pode conter um laço com uma cláusula **gang**, **worker** ou **vector** a menos que dentro de uma região **parallel** ou **kernels** aninhada. Todas os vetores irão completar a execução de suas iterações atribuídas antes de qualquer faixa de vetor possa prosseguir além do final do laço.

# Cláusula seq e auto

- A cláusula **seq** especifica que o laço ou laços associados devem ser executados sequencialmente pelo acelerador.
- Essa cláusula vai se sobrepor a qualquer paralelização ou vetorização automáticas que poderiam ser feitas pelo compilador.
- A cláusula **auto** especifica que a implementação deve selecionar se vai ser aplicado paralelismo gangue, trabalhador ou vetor ao laço, que deve ainda analisar o laço para identificar se as iterações do laço são independentes de dados.
- Em uma construção **kernels**, um laço sem nenhuma cláusula **gang**, **worker**, **vector** ou **seq** é tratado como se tivesse uma cláusula **auto**.

# Cláusula private

- A cláusula private especifica que cada iteração do laço terá sua própria cópia das variáveis listadas.
- Variáveis referenciadas em um laço e que não estejam em uma cláusula private e que não sejam privadas por definição, não serão privatizadas para nenhuma thread que executar as iterações do laço.

## **#pragma acc loop private (lista-de-variáveis)**

- Os iteradores do laço são privatizados por padrão, então eles não precisam ser listados na diretiva private.
- A menos que seja especificado em contrário, qualquer variável escalar acessada dentro de um laço com a diretiva combinada parallel loop será definida como firstprivate por padrão, o que significa que uma cópia será feita para cada iteração do laço e que terá um valor inicial igual ao que havia na variável escalar ao se entrar na região.
- Quaisquer variáveis (escalares ou não) que são declaradas dentro de um laço em C ou C++ serão feitas privadas para as iterações do laço por padrão.

# Cláusula reduction

- Em uma região paralela, se a cláusula de redução for utilizada em um laço com cláusulas vector ou worker (e sem a cláusula gang), e a variável escalar também aparecer em uma cláusula private na construção parallel, o valor da cópia privada da variável escalar será atualizado na saída do laço.
- Se a variável escalar não aparecer na lista da cláusula private na construção parallel, ou se a cláusula gang tiver sido utilizada no laço, o valor da variável escalar só será atualizado no final da região paralela.

**#pragma acc loop reduction(operador:variavel)**

# Cláusula collapse

- Em OpenACC, se nenhuma cláusula **collapse** estiver presente, apenas o laço imediatamente a seguir é associado com a diretiva **loop**, ou seja, 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 a criar um novo laço.
- É dependente de implementação se uma cláusula **gang**, **worker** ou **vector** na diretiva será aplicada a cada laço ou ao espaço de iteração linearizado.

# Cláusula collapse

- Quais as vantagens em usar a cláusula **collapse**?
  - Colapsar os laços externos para permitir a criação de mais gangues.
  - Colapsar os laços internos para permitir comprimentos de vetor mais longos.
  - Colapsar todos os laços, quando for possível, para fazer as duas coisas: ter mais gangues criadas e vetores maiores.
- Esta cláusula é especialmente útil quando alguns laços não tem um número total de iterações suficientemente grande para fazer uso efetivo do acelerador.

# Cláusula collapse

- Esta cláusula é especialmente útil quando alguns laços não tem 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)  
#pragma acc parallel loop collapse(2)  
  for (int i = 0; i < N; i++)  
    for (int j = 0; j < M; j++)  
#pragma acc parallel loop  
  for (int ij = 0; ij < N*M; ij++)...
```

# Cláusula tile

- Com a cláusula **tile** é possível otimizar o laço através da operação de blocos menores para explorar melhor o acesso aos dados. Considere o seguinte exemplo de transposição de matriz.

```
#pragma acc parallel loop private(i,j) tile(8,8)
    for (i = 0; i < rows; i++)
        for (j = 0; j < cols; j++)
            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 bloco 8x8 “ladrilhos” da matriz antes de passar para o próximo bloco. Com isso o compilador faz a otimização dentro do bloco, com o objetivo de obter melhor desempenho.

# Diretiva atomic

- 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;
- Finalmente, uma operação **capture** executa uma atualização, mas salva o valor calculado nessa região para ser utilizada no código seguinte à região.

# Diretiva atomic

```
#pragma acc data copyin(a[0:N])
copyout(h[0:HN])

for (int it = 0; it < ITERS; it++) {

#pragma acc parallel loop
for (int i = 0; i < HN; i++)
    h[i] = 0;

#pragma acc parallel loop
for (int i = 0; i < N; i++) {
    #pragma acc atomic update
        h[a[i]] += 1; }

}
```

# Dicas

- (PGI) Usar a opção “time” para saber quanto tempo gasta em cada trecho do código
  - `ta=nvidia, time`
- Eliminar aritmética com ponteiro
- Fazer o “Inline” das chamadas de função nas regiões paralelas
  - (PGI): `-inline ou -inline, levels(<N>)`
- Usar memória contínua para os arranjos multi-dimensionais (linearizar)
- Usar regiões de dados para evitar transferências excessivas de dados
- Pode ser utilizada a macro `_OPENACC` para compilação condicional

# Variáveis de Ambiente e Compilação

`ACC_DEVICE device` Especifica qual o tipo de dispositivo a se conectar para a computação paralela

`ACC_DEVICE_NUM num` Especifica qual o número do dispositivo a se conectar para a computação paralela.

`_OPENACC` Diretiva do pre-processador para a compilação condicional. Diz também qual a versão do OpenACC

# Multiplicação de Matrizes

```
#pragma acc parallel loop collapse(2)
for (i = 0; i < SIZE; ++i) {
    for (j = 0; j < SIZE; ++j) {
        C[i][j] = 0;
        for (k = 0; k < SIZE; ++k) {
            aux += A[i][k] * B[k][j];
        }
        C[i][j] = aux;
        aux = 0;
    }
}
```

# Referências

- 1) Gabriel P. Silva, Calebe Bianchini e Evaldo B. Costa  
“Programação Paralela – Um Curso Introdutório” Editora Casa do Código, 2022
- 2) Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, 2nd ed., Englewood Cliffs, NJ, Prentice--Hall, 1988.
- 3) <https://www.openacc.org>
- 4) <https://developer.nvidia.com/openacc>

# Obrigado!

Gabriel P. Silva

[gabriel@ic.ufrj.br](mailto:gabriel@ic.ufrj.br)

<http://github.com/gpsilva2003>