

# Programe sua GPU com OpenMP

**Dr. Hermes Senger**

**M.Sc. Jaime Freire de Souza**

email: [hermes@ufscar.br](mailto:hermes@ufscar.br) , [jaimeFreire@estudante.ufscar.br](mailto:jaimeFreire@estudante.ufscar.br)

Departamento de Computação  
Universidade Federal de São Carlos- UFSCar

Escola Regional de Alto Desempenho de São Paulo

ERAD-SP 2023

17 a 19 de Julho de 2023

São José dos Campos - SP



# Bibliografia

O conteúdo deste minicurso se baseou principalmente no **Cap. 6** deste livro:

Using OpenMP—The Next Step

Affinity, Accelerators, Tasking, and SIMD

By Ruud van der Pas, Eric Stotzer and Christian Terboven, MIT Press, 2017



O material desenvolvido para este minicurso está disponível em:

<https://github.com/HPCSys-Lab/Curso-OpenMP-GPU>

Criamos um canal no Slack para ajudar você minicurso aqui:

<https://tinyurl.com/yvz25amn>


Poste suas dúvidas aqui!

## Ambiente para “hands-on”

- Utilizaremos o serviço Colab  
<https://colab.research.google.com/>
- Você precisará ter uma conta Google, e estar “logado”
  - Pode ser uma conta gratuita
  - Ou uma conta de estudante, caso a sua instituição utilize os serviços da empresa

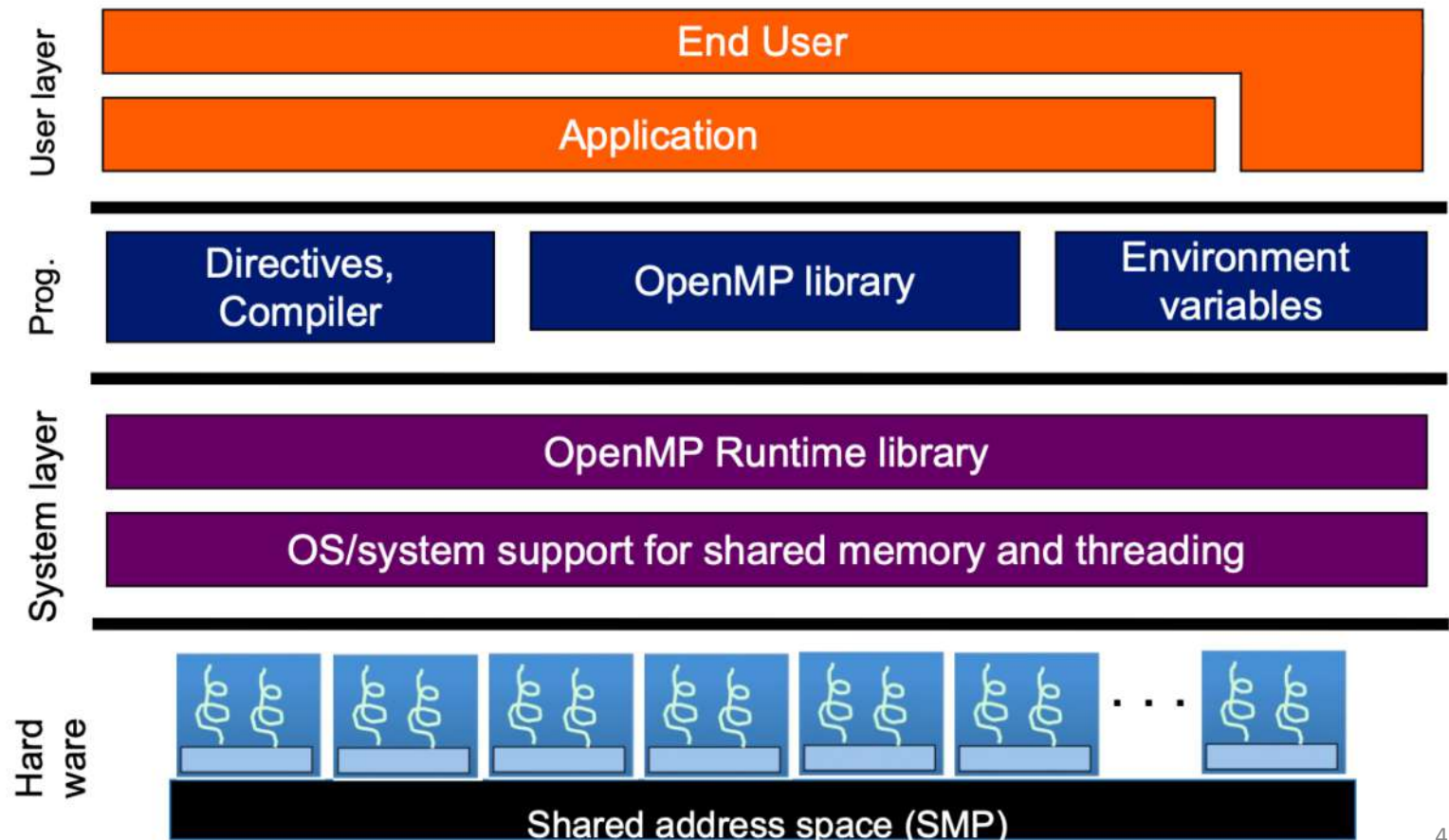
**Agora é um bom momento para criar essa conta, caso não tenha!**

# Agenda

- 
- Visão geral do OpenMP
  - Primeiros passos
  - Aceleradores
  - Movimentando dados de/para o device
  - Obtendo o paralelismo massivo
  - Otimização de código
    - Exemplos
    - Exercícios

# OpenMP – Visão Geral

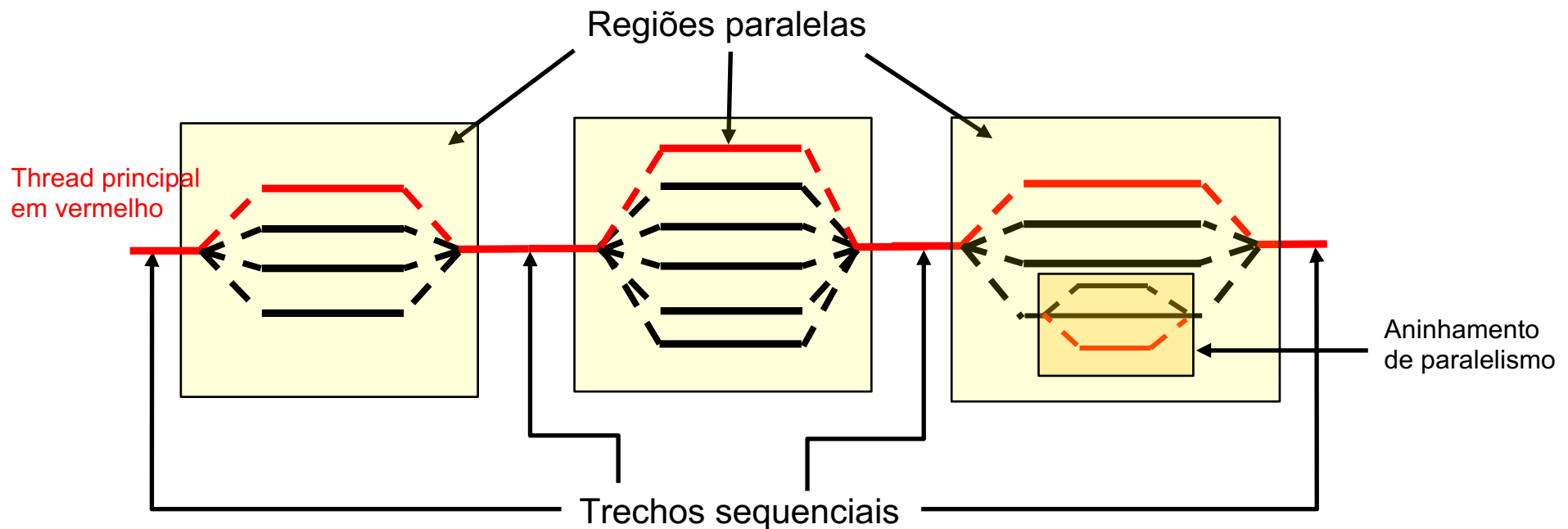
No início, OpenMP suportava apenas **multiprocessamento simétrico**, isto é, múltiplos threads acessando uma memória compartilhada, tempo de acesso uniforme ...



# O modelo de programação

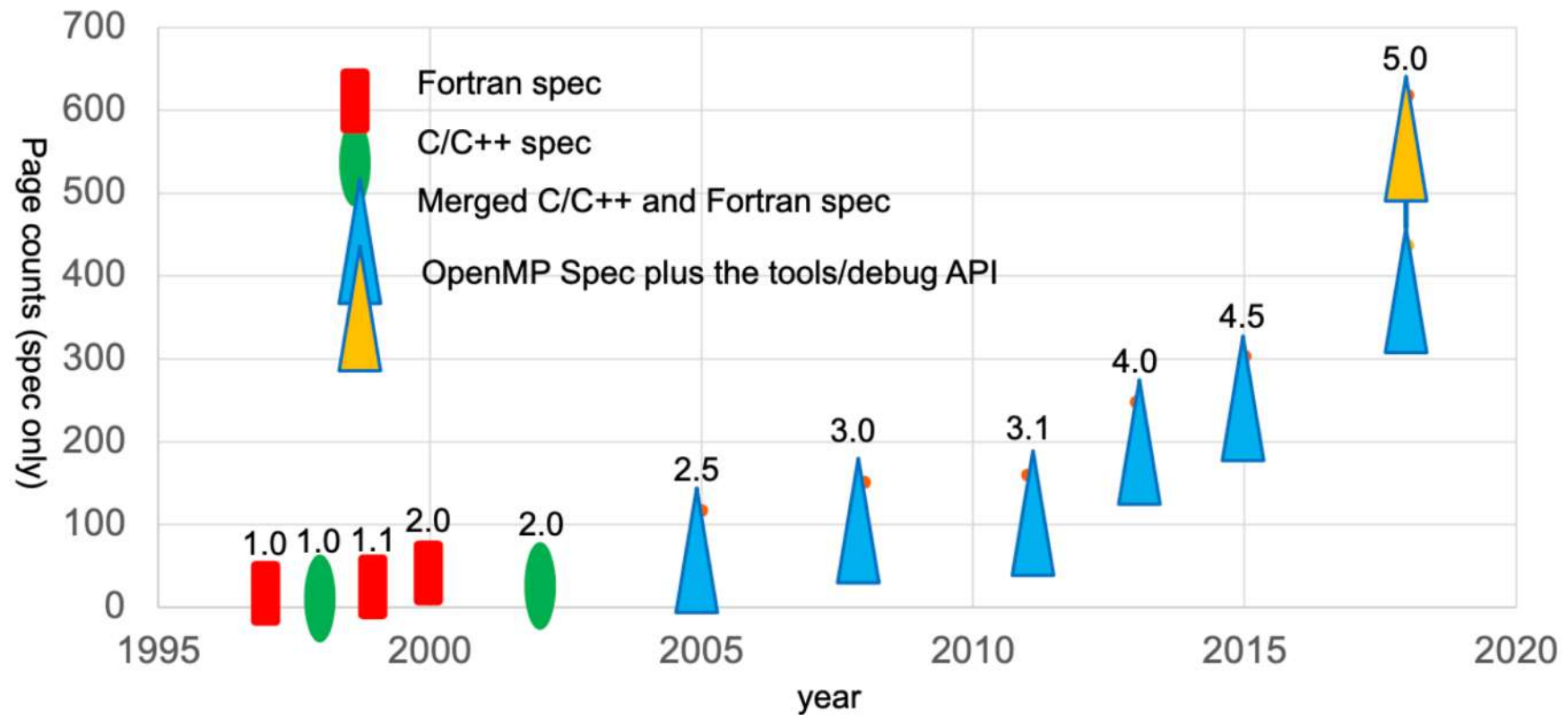
Usa paralelismo do tipo *fork-join*

- Thread principal “dispara” um time de threads se necessário
- O paralelismo vai sendo criado sob demanda, até conseguir o desempenho desejado



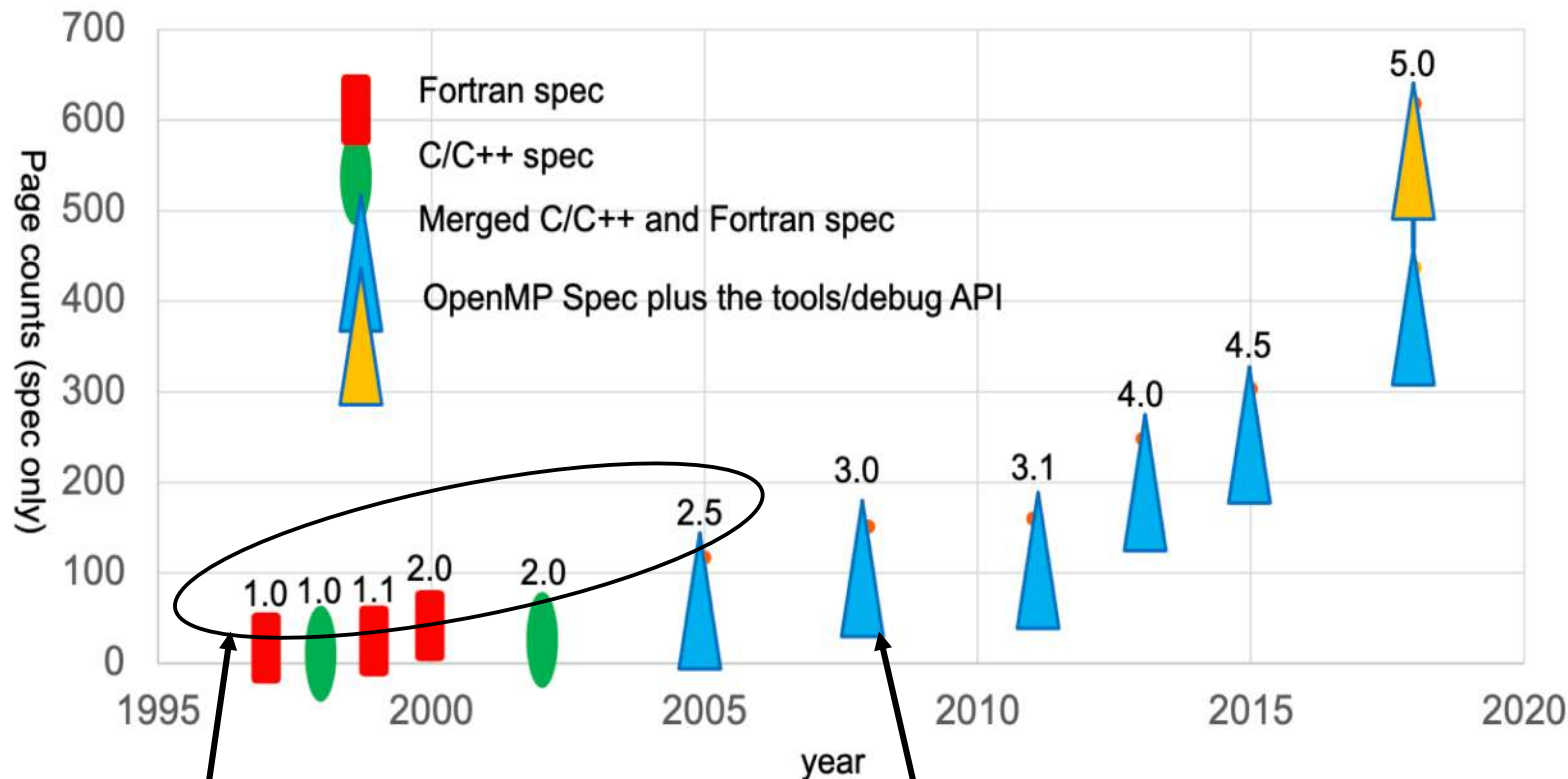
# Histórico do padrão OpenMP

Quantidade de páginas (excluindo capa, appendices) do padrão OpenMP ao longo das versões



# Histórico do padrão OpenMP

Quantidade de páginas (excluindo capa, appendices) do padrão OpenMP ao longo das versões



Suporte a multithreading, limitado a loops paralelos (times de threads)

Intruduziu **tasks** (recursão, algoritmos irregulares)

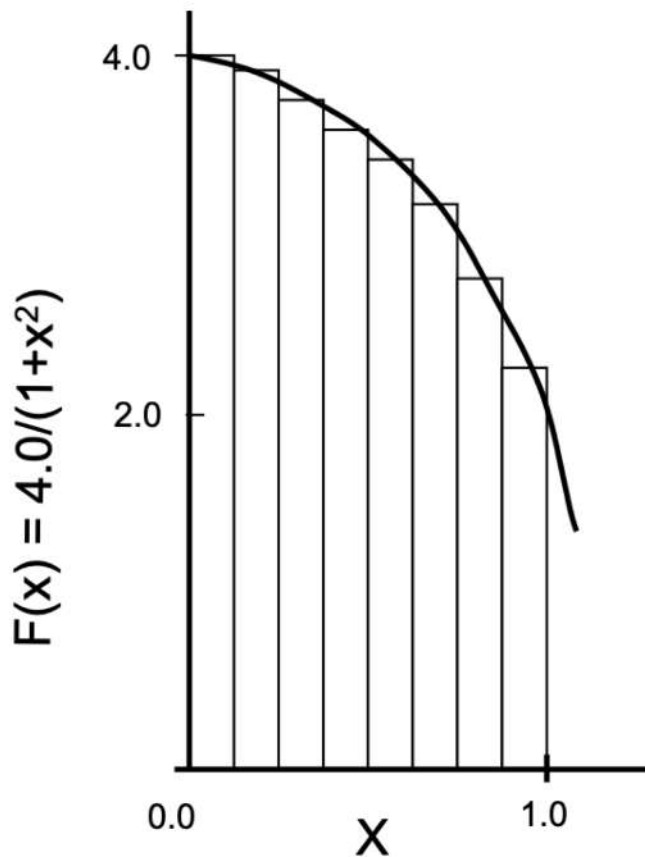


# Agenda

- Visão geral do OpenMP
- ➔ • Primeiros passos
- Aceleradores
- Movimentando dados de/para o device
- Obtendo o paralelismo massivo
- Otimização de código
  - Exemplos
  - Exercícios

# Primeiros passos

## Exemplo 1 – Cálculo e Pi



Nosso primeiro exemplo será o cálculo de Pi por integração numérica

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

Pode ser aproximado pela soma das áreas dos retângulos:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

# Pi serial

- Executa em CPU
- Execução serial
- Calcula a somatória:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000000;
double step;
int main ()
{
    int i;
    double x, pi, sum = 0.0;
    double start_time, run_time;

    step = 1.0 / (double) num_steps;

    start_time = omp_get_wtime();

    for (i = 0; i < num_steps; i++){
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }

    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("pi = %lf, %ld steps, %lf secs\n ",
        pi, num_steps, run_time);
}
```

# Atividade prática 1

## Exemplo 1: Cálculo de Pi serial na CPU

1. Acesse o notebook 1 no ambiente colab  
Você precisará ter uma conta Google!  
Precisará estar “logado” nessa conta
2. Execute o Exemplo 1: ex1-pi\_serial.c no notebook
3. Qual foi o tempo total de execução?
4. Ele executou em CPU ou GPU?
5. Qual é o tipo de CPU alocada?  
Tente o commando “lscpu”

## Primeiros passos

- Antes de programarmos uma GPU, é preciso entender como funciona o paralelismo em OpenMP
  - Regiões paralelas
  - Loops paralelos
  - Worksharing

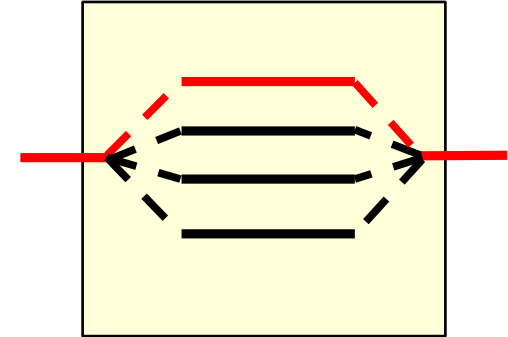
# A diretiva parallel

## Criação de uma região paralela

```
#pragma omp parallel [clause[,] clause]... ]
```

Exemplo:

```
...  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    //... Código executado por cada uma das threads  
    printf("Hello from thread %d",  
          omp_get_thread_num());  
}
```



- A diretiva **parallel** cria (**fork**) um time de threads (*team of threads*) ao entrar na região paralela.
- Ao final da região, **todas as threads devem aguardar** a sincronização (**join**).
- A primeira *thread* que chegar poderá seguir adiante. As demais são “destruídas”.

**Obs.:** compiladores podem implementar otimizações que não destroem de verdade, permitindo que sejam reutilizados em uma próxima região paralela.

# Worksharing

## Distribuição de trabalho

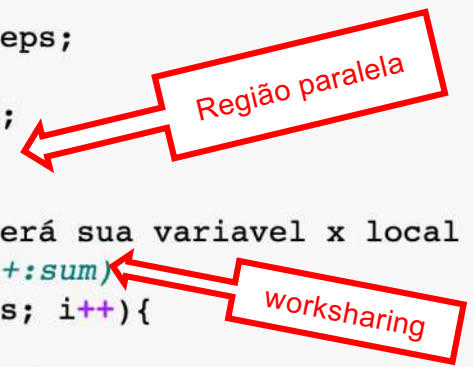
```
#pragma omp parallel
{
    // região paralela ...
    #pragma omp for
    for (i = 0; i < num_steps; i++){
        // blocos de iterações são
        // distribuídos pelas threads
        ...
    }
}
```

- A diretiva **parallel** apenas cria um **time de threads** (*thread team*).
- É preciso atribuir trabalho às threads (isto se chama **Worksharing**).

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000000;
double step;
int main ()
{
    int i;
    double x, pi, sum = 0.0;
    double start_time, run_time;

    step = 1.0 / (double) num_steps;

    start_time = omp_get_wtime();
    #pragma omp parallel
    {
        double x; /* cada thread terá sua variavel x local */
        #pragma omp for reduction(+:sum)
        for (i = 0; i < num_steps; i++){
            x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }
    }
    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("pi = %20.15lf, %ld steps, %lf secs\n ", pi,
        num_steps, run_time);
}
```





# Worksharing

## Medindo o tempo de execução

```
double start_time = omp_get_wtime();
#pragma omp parallel
{
    // região paralela ...
    #pragma omp for
    for (i = 0; i < num_steps; i++){
        // blocos de iterações são
        // distribuídos pelas threads
        ...
    }
}
run_time = omp_get_wtime() - start_time;
```

- Há várias formas de medir, mas o tempo de relógio (*wall clock*) em geral é suficiente!

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000000;
double step;
int main ()
{
    int i;
    double x, pi, sum = 0.0;
    double start_time, run_time;

    step = 1.0 / (double) num_steps;

    start_time = omp_get_wtime();
    #pragma omp parallel
    {
        double x; /* cada thread terá sua variavel x local */
        #pragma omp for reduction(+:sum)
        for (i = 0; i < num_steps; i++){
            x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }
    }
    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("pi = %20.15lf, %ld steps, %lf secs\n ", pi,
           num_steps, run_time);
}
```

Medir antes da região paralela

Medir no final e ver a diferença

# Atividade prática 2

## Exemplo 2: Pi paralelo na CPU - Pi-V1.0.c

1. Acesse o notebook 1 no ambiente colab
2. Execute o Exemplo2: Pi loop paralelo + reduction
3. Compare os tempos de execução de ambos
4. Descubrir tipo de CPU e de GPU alocados
5. Flags de compilação

# Agenda

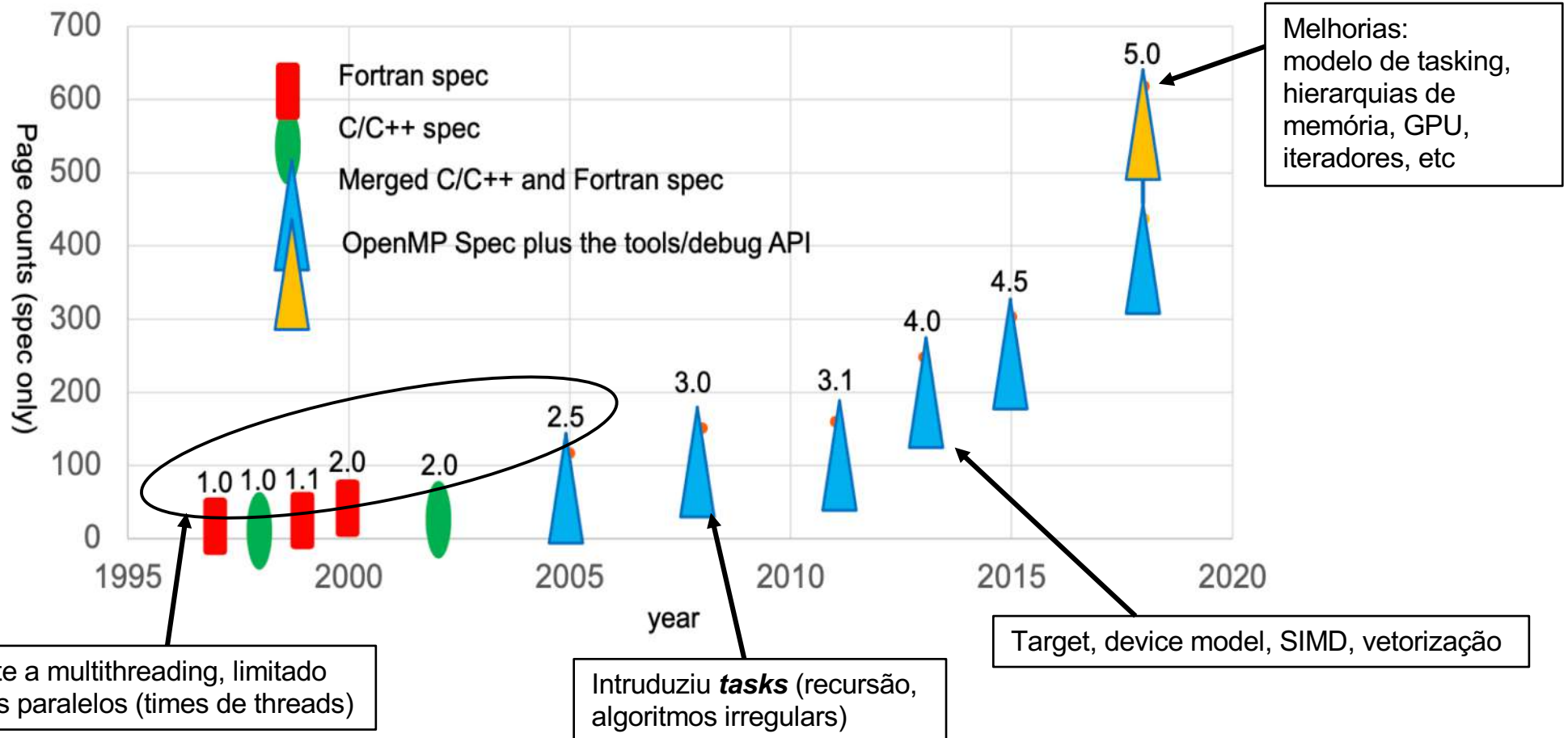


- Visão geral do OpenMP
- Primeiros passos
- Aceleradores
- Movimentando dados de/para o device
- Obtendo o paralelismo massivo
- Otimização de código
  - Exemplos
  - Exercícios

# OpenMP e Aceleradores

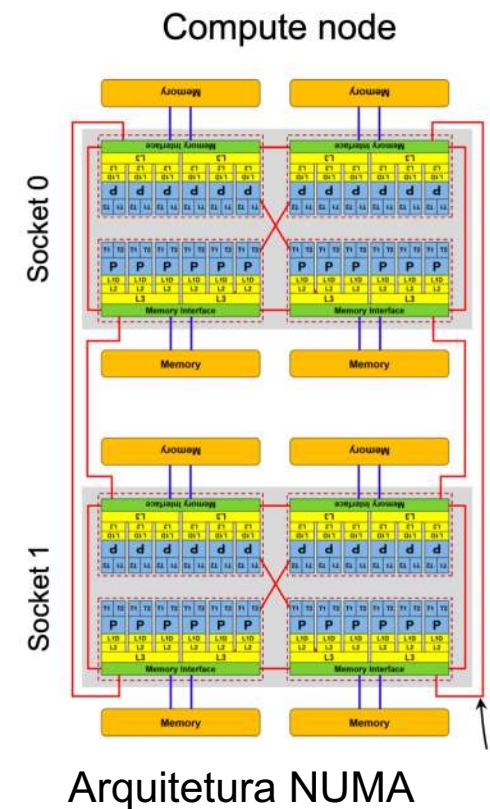
# Histórico do padrão OpenMP

Quantidade de páginas (excluindo capa, appendices) do padrão OpenMP ao longo das versões

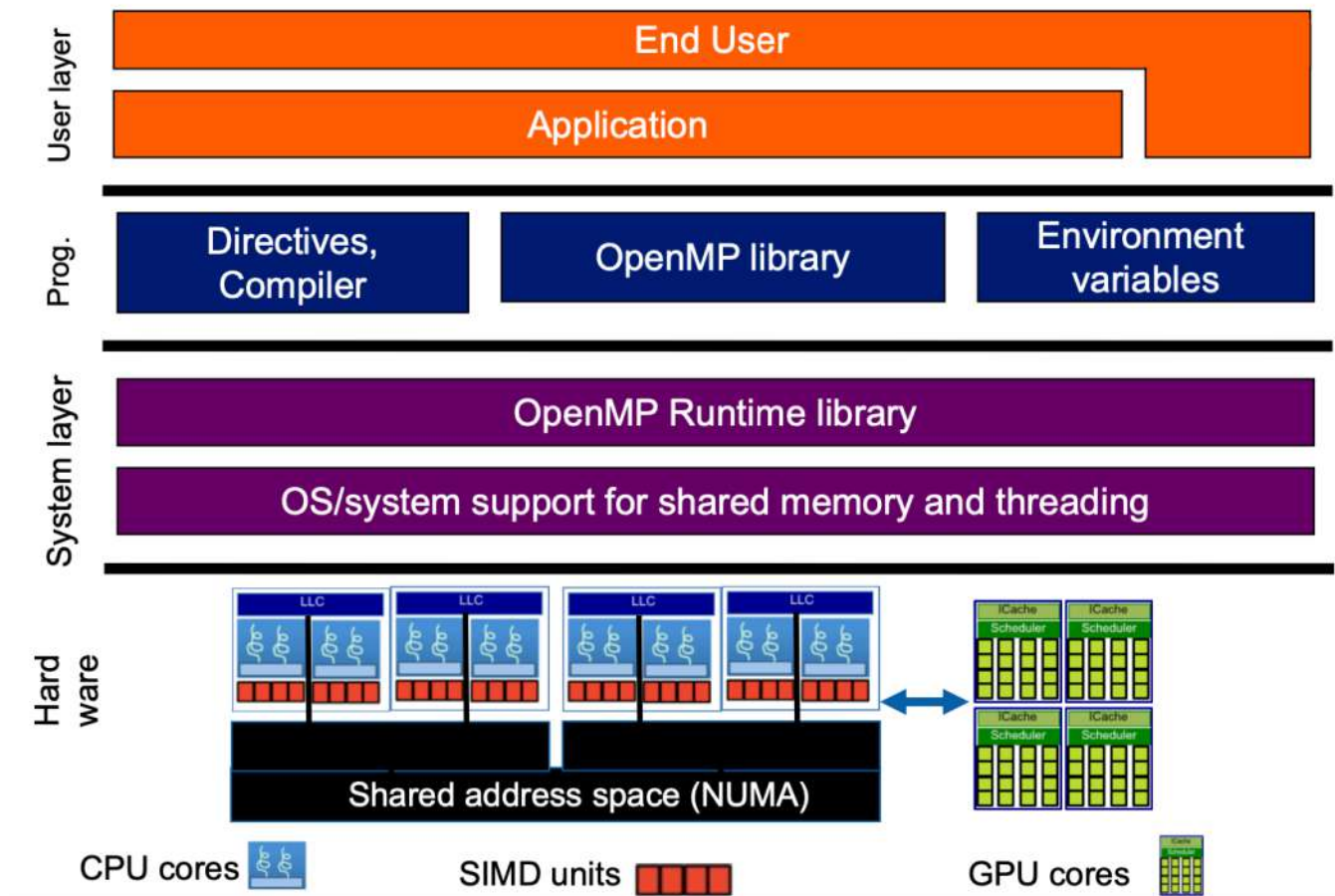


# Modelo de Programação de OpenMP

- Até o OpenMP 3.0:
  - Foco em CPUs multi-core
  - Todos os **cores** podem acessar a memória principal toda
  - OpenMP define um espaço de endereçamento único, que pode ser acessado por todos os threads paralelos:  
**Programação com memória COMPARTILHADA!**
- A versão OpenMP 4.x mudou isso
  - Incluiu controles NUMA (***non-uniform memory access***):
    - Reconhece que a memória **não possui desempenho uniforme**
    - Mas ainda é compartilhada entre todos os cores da CPU!
- O modelo de dispositivo alvo (**target device**) foi criado:
  - Esse modelo separa o espaço de memória (memória do **host** e memória do **device**)
  - Isto cria a **programação heterogênea**

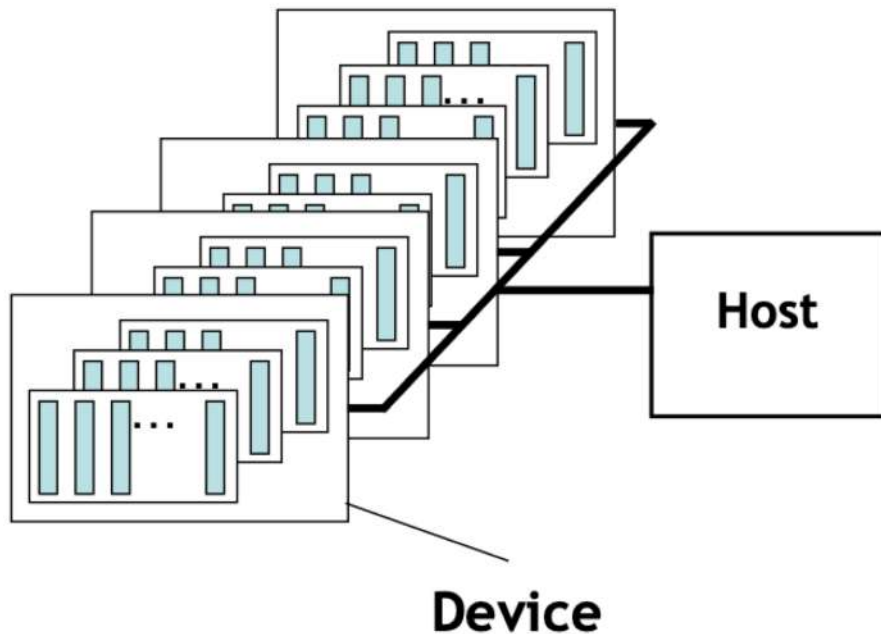


# O modelo *host/device*



# O modelo *host/device* do OpenMP

- Modelo host/device do OpenMP **assume** que:
  - O **host** é onde o thread inicial do programa inicia sua execução
  - Zero ou mais **devices** são conectados ao **host**
  - A memória do host e a memória do device possuem **espaços de endereçamento distintos**



```
#include <omp.h>
#include <stdio.h>

int main() {
    int num_devices = omp_get_num_devices();
    printf("Temos %d devices alocados\n", num_devices);
}
```



# A Diretiva Target

1. A execução começa, criando um **thread inicial** no **host**

2. Uma região paralela implícita cerca o programa inteiro

3. A tarefa inicial começa a executar

4. A tarefa inicial encontra a diretiva **target**

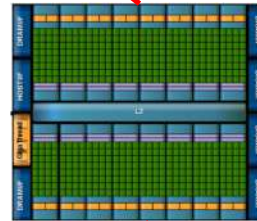
5. A **tarefa inicial** cria uma **tarefa target**, que é uma tarefa *mesclável e incluída*

6. A **tarefa inicial** cria uma **região target** no **device**

10. A **tarefa inicial** no **host** retoma a execução assim que a **região target** termina

A diretiva **target** descarrega a execução no **device**  
**#pragma omp target**  
{....} // a structured block of code

**#pragma omp target**



7. Uma nova **tarefa inicial** executa no **device**

8. Uma **região paralela** implícita circunda o programa no **device**

9. A **tarefa inicial** executa o Código na **região target** no **device**

```
#pragma omp target nowait
{
    // trabalho do device
}
```

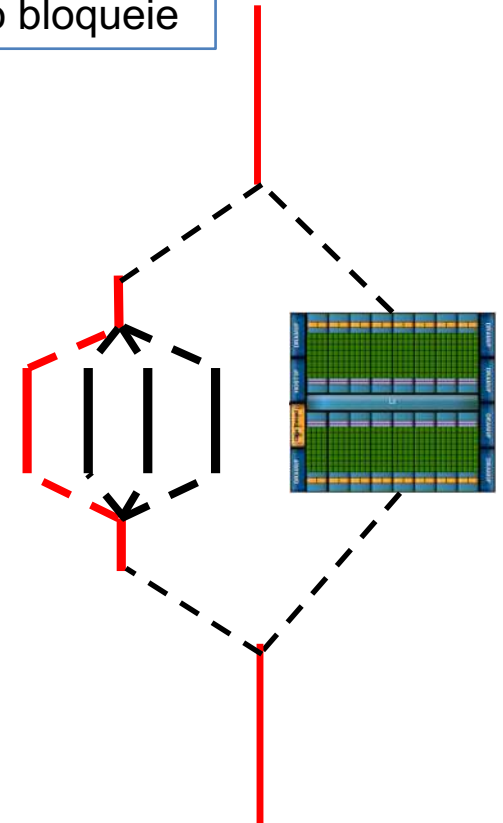
Faz com que a tarefa no host não bloqueie

```
#pragma omp parallel for
for(int i=0;i<N;i++) {
    trabalho_do_host (i);
}
```


A tarefa (implícita) do host pode executar outro trabalho (que pode ser paralelo)

```
#pragma omp taskwait
```

Faz com que a tarefa (implícita) do host bloqueie e aguarde a região target completar a execução



# Agenda

- Visão geral do OpenMP
- Primeiros passos
- Aceleradores
-  • Movimentando dados de/para o device
- Obtendo o paralelismo massivo
- Otimização de código
  - Exemplos
  - Exercícios

# Movimentação de Dados - Implícita

## Movimentação dos dados - Implícita

- **Lembre-se:** espaços de memória do *host* e do *device* separados
- OpenMP usa uma combinação de movimentos de dados *implícitos* e *explícitos*
- Dados são movidos entre o *host* e o *device* em lugares bem definidos:
  - Primeiramente, veremos como isso é feito no início e no fim de uma **região target**:

### #pragma omp target

```
{      // pode mover dados do host para o device
    ...
}      // e mover dados do device para o host
```

## Movimentação dos dados - Implícita

- Variáveis escalares:
  - OpenMP adota a semântica **firstprivate**
  - A variável não é copiada de volta para o *host*
  - Exemplo
    - `int N; double x;`

## Movimentação dos dados - Implícita

- Variáveis **não-escalares**:
  - Devem ter **tipagem completa**
  - Exemplo: Vetor de tamanho fixo (alocado na pilha)  
**double X[1000];**
  - São copiadas para o device no início da região **target** e copiadas de volta para o **host** no final
  - OpenMP chama essa semântica de **tofrom**

# Movimentação dos dados - Implícita

- Ponteiros são copiados implicitamente, mas **não os dados que eles apontam**
  - Exemplo: Vetores alocados no *heap*  
`double *X = malloc (sizeof(double) * 1000);`
  - O **valor de X** será copiado (i.e., o endereço armazenado em X)
  - Mais adiante veremos como copiar os dados de forma explícita



# Exemplo de cópia implícita

```
int main(void) {  
    int N = 1024; // variáveis criadas na memória do host  
    double A[N], B[N];  
  
    #pragma omp target // E, A e B são copiadas para o device  
    { // a execução no device começa  
  
        for (int ii = 0; ii < N; ++ii) // ii é declarada dentro da região  
            A[ii] = A[ii] + B[ii]; // target, portanto é privada  
  
    } // fim da região target – A e B são copiadas para o host  
    // A execução prossegue no host  
}
```

Pergunta: ao final, N será copiada de volta para o *host*?

# Diretivas comumente usadas com o target

**#pragma omp target [clausula[,]c clausula]...] { \\* bloco estruturado\*\}**

**if(scalar-expression)**

– Se **scalar-expression** for falsa, a região **target** será executada pelo host no ambiente de dados do host

**device(integer-expression)**

– O valor de **integer-expression** determina qual device deverá executar a região

**private(list) firstprivate(list)**

– Cria no device variáveis cujos nomes constam na lista. No caso de **firstprivate**, o valor da variável no host é copiado para a variável privada no device


**map(map-type: list)**

– map-type pode ser **to**, **from**, **tofrom**, ou **alloc**. A cláusula define como as variáveis da lista devem ser movidas entre o host e o device

**nowait**

– A tarefa target será atrasada, o que significa que o host pode executar seu código em paralelo com a execução da região target no device

# Agenda

- Visão geral do OpenMP
- Primeiros passos
- Aceleradores
-  • Movimentando dados de/para o device
- Obtendo o paralelismo massivo
- Otimização de código
  - Exemplos
  - Exercícios

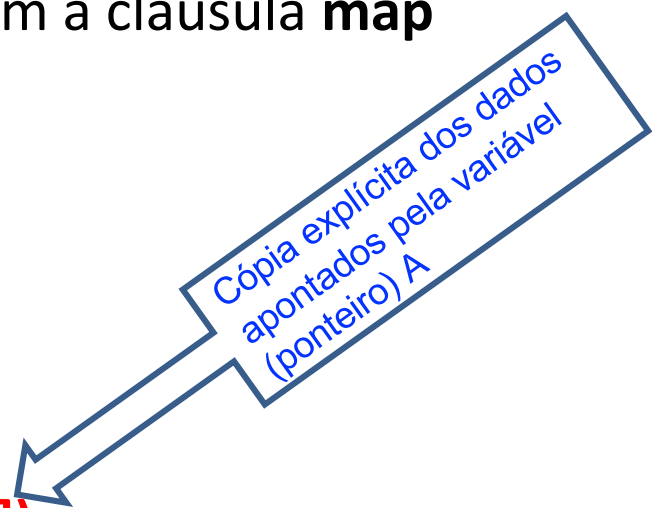
Movimentação **explícita** de dados

# Movimentação Explícita de Dados

A movimentação explícita é feita com a cláusula **map**

**Exemplo:**

```
int main(void) {  
    int i=0, N = 1024;  
    int* A = malloc(sizeof(int)*N);  
  
    #pragma omp target map (A[0:N])  
    {  
        // N, i e A existem AQUI  
        // Os dados apontados por A (*A , A[ii]) também EXISTEM aqui!  
    }  
}
```



Cópia explícita dos dados apontados pela variável (ponteiro) A

# Notação de vetores

- É preciso tomar cuidado com a notação para mapear vetores:
  - Notação em C é **ponteiro[inicio: comprimento]**
  - **map(to: a[0:N])**
  - Começa no elemento **a[0]** e copia N elementos para região target
  - **Cuidado!**
    - É comum confundir com **inicio: fim**, mas é **comprimento**
  - Sem o mapeamento, OpenMP entende que o ponteiro **a** deve ser mapeado como um vetor de comprimento ZERO  
Como se fosse **A[:0]**

# Movimentação de dados de/para o device

```
int i, a[N], b[N], c[N];
```

```
#pragma omp target map(to:a,b) map(tofrom:c)
```

- As várias formas da cláusula map:
  - **map(to:list)**: ao entrar na região, inicializa as variáveis da lista com os valores que elas têm no host (cópia host -> device)
  - **map(from:list)**: ao sair da região, os valores das variáveis da lista são copiadas do device para o host. Ao entrar na região, elas não serão inicializadas no device
  - **map(tofrom:list)**: soma os efeitos de map-to e map-from (host → device ao entrar na região, device → host ao sair da região)
  - **map(alloc:list)**: ao entrar na região, os dados são alocados no device, mas não serão inicializados
  - **map(list)**: equivalente a map(tofrom:list)

# Atividade prática 3



10 minutos para fazer!

Não cole a resposta!

## Exercício 1: Soma de vetores

1. Acesse o notebook 1 no ambiente colab
2. Paralelize a soma de vetores com a diretiva **#pragma omp target** para executar na GPU
3. Paralelize o loop da inicialização na CPU com **#pragma omp parallel for**
4. Paralelizar o loop de teste na CPU.

Obs.: Você pode utilizar redução para totalizar a contagem de erros:

**#pragma omp parallel for reduction(+:err)**

5. O programa está disponível aqui, caso precise restaurá-lo:

<https://github.com/UoB-HPC/openmp-tutorial/blob/master/vadd.c>



## Atividade prática 3

10 minutos para fazer!

A solução está no próximo slide. Não cole!



# Solução: Soma de Vetores

```
#include <stdio.h>
#include <omp.h>
#define N 100000
#define TOL 0.0000001
// Written by Tim Mattson, November 2017
int main()
{
    float a[N], b[N], c[N], res[N];
    int err=0;

    // fill the arrays
    #pragma omp parallel for
    for (int i=0; i<N; i++){
        a[i] = (float)i;
        b[i] = 2.0*(float)i;
        c[i] = 0.0;
        res[i] = i + 2*i;
    }

    // add two vectors
    #pragma omp target
    for (int i=0; i<N; i++){
        c[i] = a[i] + b[i];
    }
```

```
// test results
#pragma omp parallel for reduction(+:err)
for(int i=0;i<N;i++){
    float val = c[i] - res[i];
    val = val*val;
    if(val>TOL) err++;
}

printf(" vectors added with %d errors\n",err);
return 0;
}
```

# Atividade prática 4




10 minutos para fazer!

Não cole a resposta!

## Exercício 2: Movimentação explícita de dados

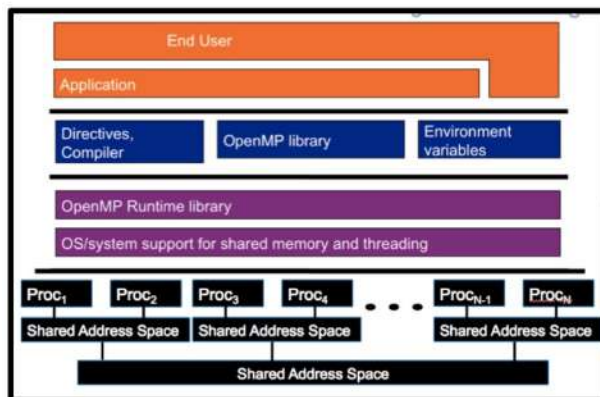
1. Acesse o notebook 1 no ambiente colab
2. Paralelize o programa de soma de vetores sequencial disponível em:  
[https://github.com/UoB-HPC/openmp-tutorial/blob/master/vadd\\_heap.c](https://github.com/UoB-HPC/openmp-tutorial/blob/master/vadd_heap.c)
3. Aloque os vetores no heap em vez do stack:
  - Troque **double a[N]**
  - por **\*a = malloc(sizeof(double) \* N)**
4. Modifique o Código para executar na GPU
  - Use a diretiva **target** para descarregar a execução na GPU
  - #pragma omp target**
5. Copie os dados dos arrays no heap para/da GPU com as cláusulas map:  
**map(tofrom:...), map(to:...), map(from:...)**

# Agenda

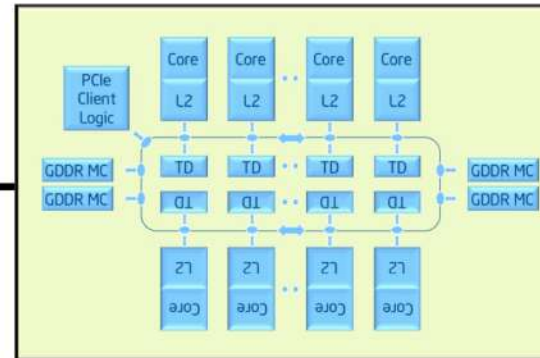
- Visão geral do OpenMP
- Primeiros passos
- Aceleradores
- Movimentando dados de/para o device
-  • Obtendo o paralelismo massivo
- Otimização de código
  - Exemplos
  - Exercícios

# Obtendo o Paralelismo Massivo

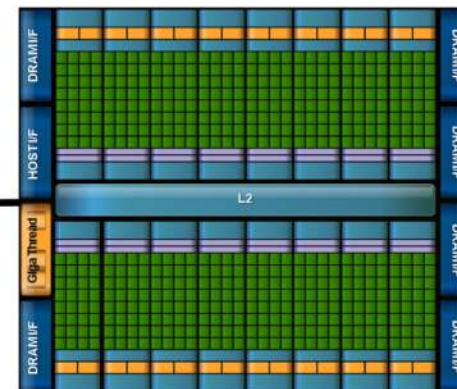
Some key devices that were considered when designing the device model in OpenMP



Host



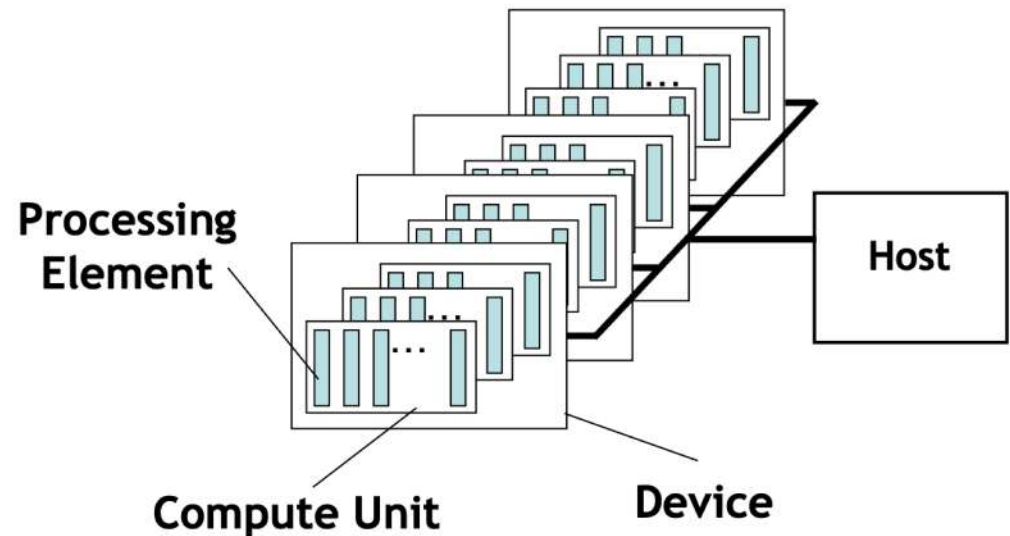
Target Device: Intel® Xeon Phi™ processor



Target Device: GPU

# O modelo *host/device* do OpenMP

Existem vários tipos de aceleradores, como GPUs (discretas ou integradas, de diferentes fabricantes), co-processadores (Xeon Phi), DSPs, etc . Para lidar com isso, OpenMP definiu um modelo.



- Existe apenas um **Host** e um ou mais **Devices**
  - Cada **device** é composto por um ou mais **Compute Units**
  - Cada **Compute Unit** possui um ou mais **Processing Elements**
  - A memória é dividida entre **memória do host** e **memória do device**

# O modelo *host/device* do OpenMP

- GPUs são feitas de muitas ***compute units***
  - NVIDIA P100 tem 56 ***streaming multiprocessors*** (SMs) – nada mais são do que *compute units*! Cada ***compute unit*** tem 64 ***processing elements*** (**cuda cores** que operam em FP32)  
No total, uma P100 possui 56 SMs x 64PEs = 3.584 ***processing elements*** (ou cuda cores)
  - V100 (arquitetura Volta) tem 80 SMs x 64 Pes = 5120 ***processing elements*** (ou cuda cores)
  - A100 (arquitetura Ampere) tem 128 SMs x 64 FP32 PEs = 8192 FP32 CUDA Cores/GPU
  - H100 (arquitetura Hopper) tem 132 SMs x 128 FP32 processing elements = 14592 Pes/GPU
  - As GPUs da AMD e Intel têm estrutura similar, com **compute units** e **processing elements**
  - ...
- A questão chave é:  
“Você precisa oferecer múltiplas unidades de trabalho para cada processing element para conseguir bom desempenho!”
- Estratégia: Explorar quantidades massivas de processamento (hierárquico)



# As diretivas para paralelismo massivo

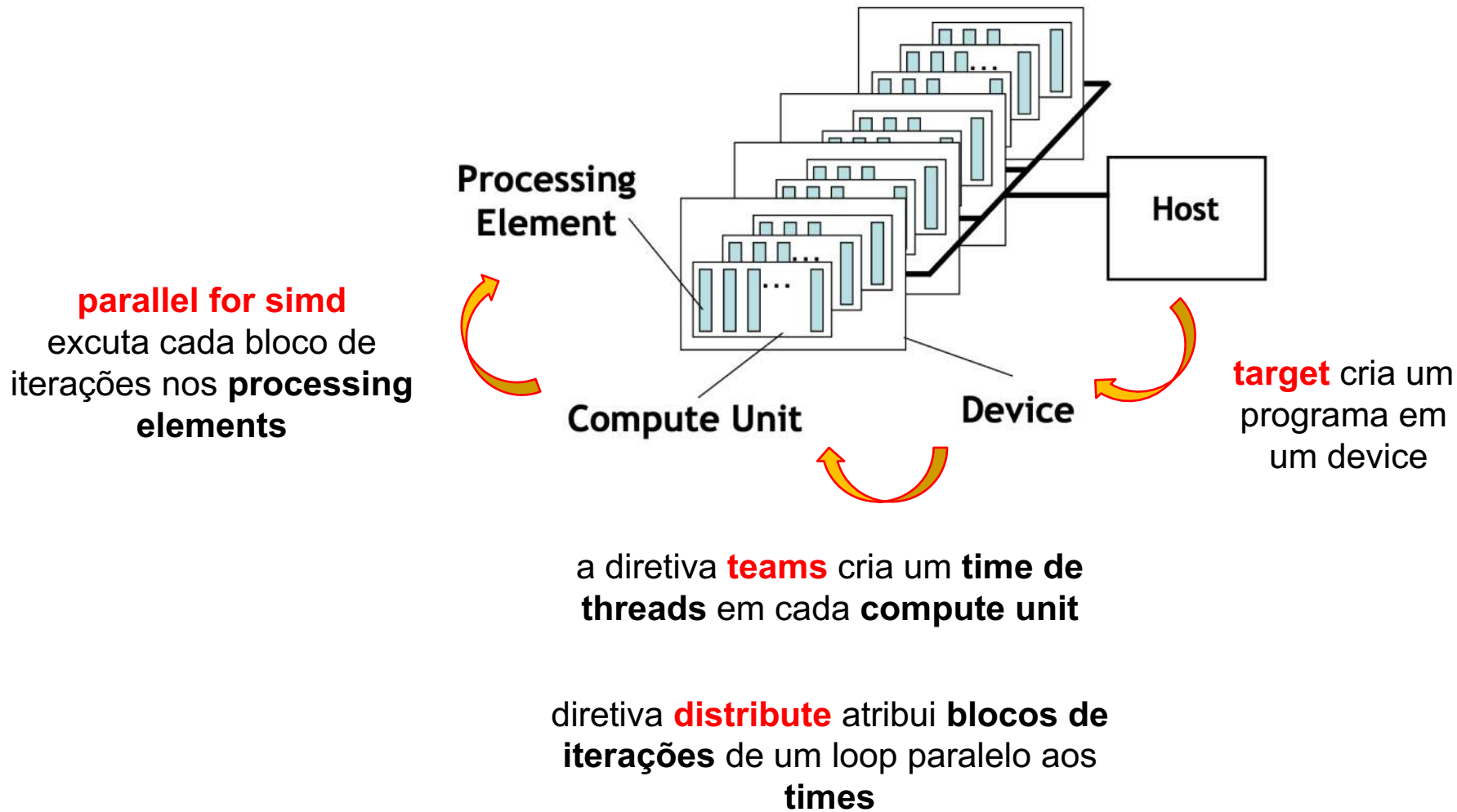


O modelo GH100 possui 144 SMs



Um SM da GPU GH100

# As diretivas para paralelismo massivo



## Detalhes de implementação

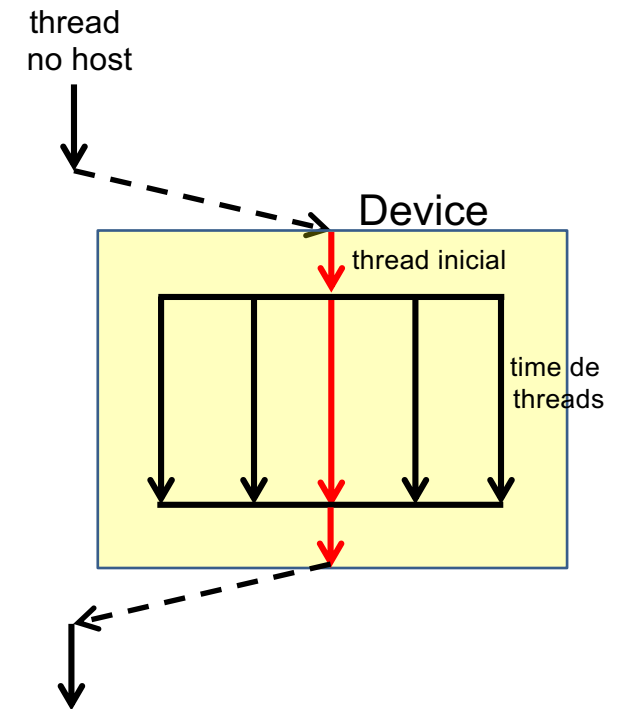
- OpenMP define algumas abstrações de paralelismo, junto com uma terminologia específica
- As implementações de OpenMP (compilador, runtime, etc) tem alguma liberdade para decidir como isto se aplica ao hardware
- Permite que as implementações tomem decisões críticas em busca de melhor desempenho
- A seguir, veremos como funcionam essas abstrações

# Threads Paralelos

- Lembre-se do modelo fork-join e regiões paralelas na CPU
  - **#pragma omp parallel**
- Threads são criados na entrada da região paralela
- Todos os threads criados pertencem a um **time**
- Os **threads** de um **time** podem sincronizar  
**#pragma omp barrier**

```
#pragma omp target
#pragma omp parallel for
for (i=0; i<N; i++)
...
```

Impede que qualquer thread de um time passe além da barreira, até que todos no time encontrem a barreira.



# Atividade prática 5



3 minutos para fazer!

## Exemplo 3: Pi V2.0 – threads na GPU

1. Acesse o notebook 1 – Programa **Pi-par-V2.c**
2. Teste as seguintes diretivas:

```
#pragma omp target map (sum)
#pragma omp parallel for reduction(+: sum) private(x)
for (i=0; i<N; i++)
    ...
```

3. Execute o programa e anote o tempo de execução.
4. Foi atingido o paralelismo massivo desejado? Justifique sua resposta.

# As diretivas teams e distribute

- A diretiva **teams**
  - É semelhante à diretiva **parallel** em CPUs
  - Cria uma **liga** (*league*) de **times**
  - Cada **time** na **liga** é criado com **um thread inicial** – i.e. um time de um thread – threads em diferentes times **não podem** sincronizar uns com os outros
  - A diretiva deve ser “perfeitamente” aninhada em uma diretiva **target**
- A diretiva **distribute**
  - Semelhante à diretiva **for** de CPUs
- As iterações do loop são distribuídas (*workshared*) entre os threads iniciais de de uma liga de times – Não há uma barreira implícita no final da região
  - **dist\_schedule(kind[, chunk\_size])**
    - Se for especificado, o escalonamento deve ser **static**
    - Blocos de tamanho **chunk\_size** são distribuídos de forma circular (*round-robin*)
    - Se nenhum tamanho for especificado, os blocos (aproximadamente) de mesmo tamanho serão distribuídos; cada **time** recebe pelo menos um bloco

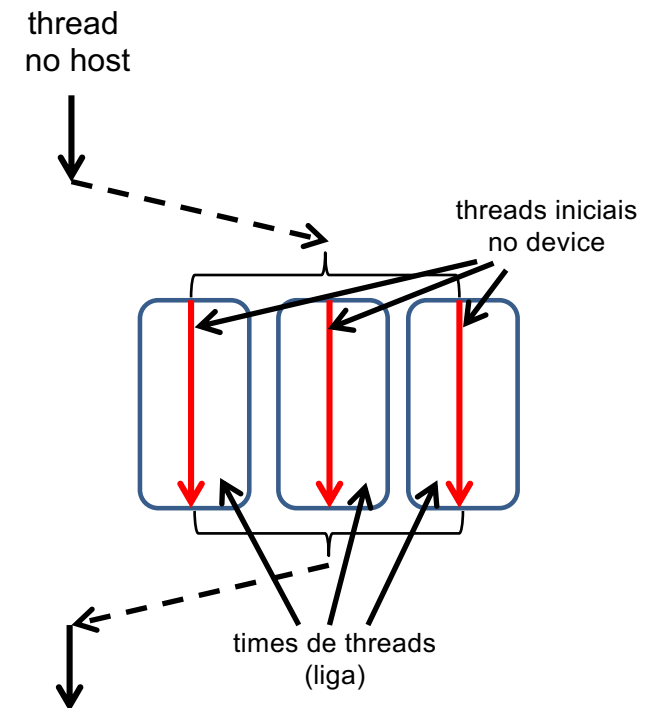
# Multiplos times

- A diretiva teams
- A diretiva distribute

```
#pragma omp target  
#pragma omp teams  
#pragma omp distribute  
for (i=0; i<N; i++)  
...
```

- Cria uma **liga** de (**VÁRIOS** times com) **threads iniciais** no **device**
- Distribui o trabalho (iterações do loop) entre os **threads iniciais**

Obs.: A escolha sobre o número de **times** fica a cargo do compilador, para melhor “portabilidade de desempenho”. Os compiladores podem escolher como fazer o mapeamento desses **times** e **threads**.



# Atividade prática 6



3 minutos para fazer!

## Exemplo 4: Pi V3.0 – múltiplos times

1. Acesse o notebook 1 – Programa **Pi-par-V3.c**
2. Teste as seguintes diretivas:

```
#pragma omp target  
#pragma omp teams  
#pragma omp distribute  
for (i=0; i<N; i++)  
...
```

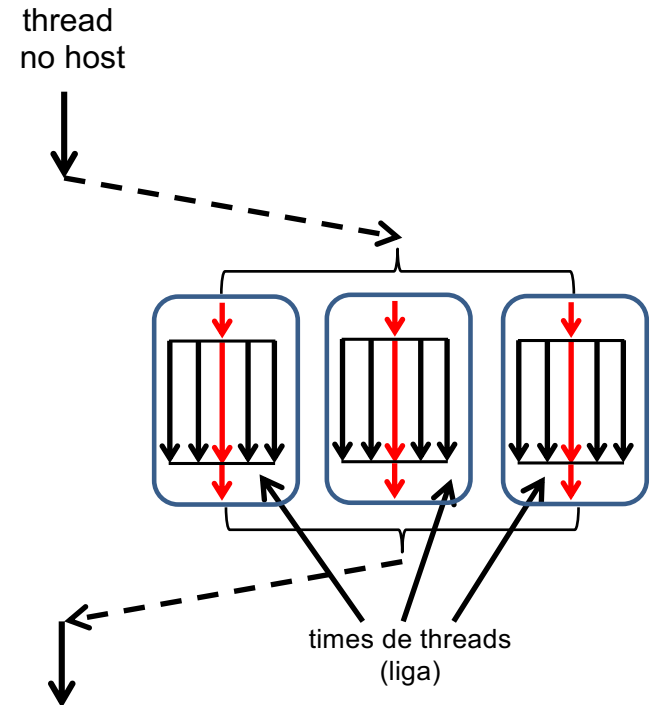
3. Execute o programa e anote o tempo de execução.
4. Foi atingido o paralelismo massivo desejado? Justifique sua resposta.



# Paralelismo times + threads + SIMD

- É preciso combinar tudo isso para ter bom desempenho (lembre-se, paralelismo massivo)!

```
#pragma omp target  
#pragma omp teams distribute  
for (i=0; i<N; i++)  
#pragma omp parallel for simd  
for (i=0; i<M; i++)  
...
```



- Cria uma **liga** de **VÁRIOS** times, com **threads iniciais** no **device** (um por time)
  - Distribui o trabalho (blocos de iterações do loop) entre os **threads iniciais** (distribuição para times)
- Cada **thread** inicial se torna o principal (*master*) do seu time
  - Distribui o trabalho (iteraões do loop) entre os **threads de um time** (**parallel for simd**)
  - O compilador decide como – pode vetorizar

# Atividade prática 7



3 minutos para fazer!

## Exemplo 5: Pi V4.0 – times+threads+SIMD

1. Acesse o notebook 1 – Programa **Pi-par-V4.c**
2. Teste as seguintes diretivas:

```
#pragma omp target  
#pragma omp teams distribute  
for (i=0; i<N; i++)  
#pragma omp parallel for simd  
for (i=0; i<M; i++)  
...
```

3. Execute o programa e anote o tempo de execução.
4. Foi atingido o paralelismo massivo desejado? Justifique sua resposta.

# Composição de diretivas

- Os padrões de distribuição podem ser um tanto confusos
- OpenMP define composições de diretivas para padrões típicos
  - **distribute simd**
  - **distribute parallel for**
  - **distribute parallel for simd**
  - ... mais algumas combinações para **teams** e **target**
- Deixe o compilador decidir como dividir o loop em blocos (“ladrilhamento”)

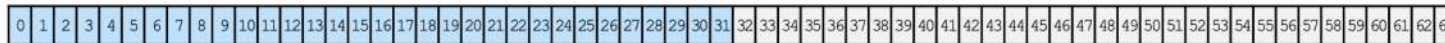
```
#pragma omp target teams  
{  
    #pragma omp distribute parallel for simd  
    for (int i = 0; i < n; i++) {  
        F(i) = G(i);  
    }  
}
```

# Exemplo

```
#pragma omp target teams distribute parallel for simd \
num_teams(2) num_threads(4) simdlen(2)
for (i=0; i<64; i++)
...
```

64 iterações atribuídas a 2 times  
Cada time tem 4 threads  
Cada thread tem 2 faixas SIMD

A diretiva **distribute** divide as iterações entre 2 times



Dentro de cada **time**: o trabalho (iterações do loop) é dividido entre os 4 threads



Agora sim, temos  
Paralelismo nos 3  
níveis!


Por fim, cada thread  
implementa paralelismo SIMD



# Cláusulas comumente utilizadas com ... teams distribute parallel for simd

- Nossa diretiva “padrão ouro” é:  
**#pragma omp teams distribute parallel for simd [clause[,]clause...] for-loops**
- As cláusulas mais comumente utilizadas são:
  - **private(list) firstprivate(list) lastprivate(list) shared(list)**
    - Se comporta como as cláusulas para diretivas de manipulação de dados do OpenMP, mas note que os valores só são criados ou copiados para dentro da região, mas não para fora da região no final
  - **reduction(reduction-identifier : list)**
    - Se comporta como no resto do OpenMP, mas a variável deve aparecer em uma cláusula map(tofrom) na região target associada para que possa ter copiado de volta o valor no final (falaremos mais sobre isto)
  - **collapse(n)**
    - Combina loops antes que a diretiva distribute divida as iterações entre os times
  - **dist\_schedule(kind[, chunk\_size])**
    - Admite somente kind = static. Caso contrário funciona como se fosse aplicado a uma diretiva for.  
Obs.: Isto se aplica à operação da diretiva distribute e controla a distribuição das iterações do loop aos times (mas NÃO a distribuição das iterações dentro de um time).

# Agenda

- Visão geral do OpenMP
- Primeiros passos
- Aceleradores
- Movimentando dados de/para o device
- Obtendo o paralelismo massivo
-  • Otimização de código
  - Exemplos
  - Exercícios

# Equação da Onda Acústica

## Aplicação: Simulação da Onda Acústica

- Aplicação geofísica - Simula a propagação de uma onda acústica por (PDEs)

$$\frac{1}{\rho v_p^2} \frac{\partial p(\mathbf{x}, t)}{\partial t} - \nabla \cdot \mathbf{v}(\mathbf{x}, t) = f, \rho \frac{\partial \mathbf{v}(\mathbf{x}, t)}{\partial t} - \nabla p(\mathbf{x}, t) = 0,$$

— Precisa conhecer as propriedades do meio

- Mas como fazer o inverso? Como descobrir as propriedades dos materiais?

- **Full waveform inversion (FWI)**

- É um problema inverso

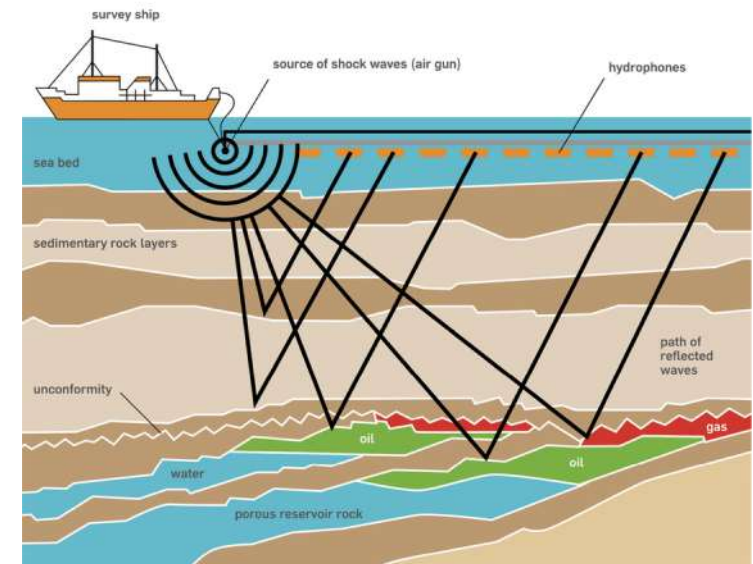
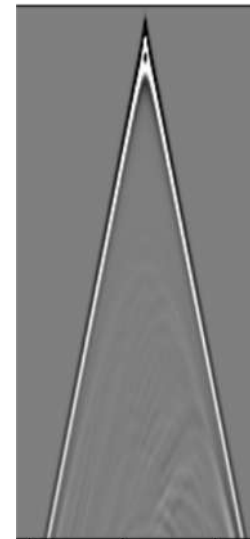
*“Minimize the misfit between a set of observed arrival times on the receivers and those synthetically generated using an estimate of the velocity model”*  
(Virieux & Operto, 2009)

$$\min_{\mathbf{m}} f(\mathbf{m}) = \sum_{i=1}^{n_s} \frac{1}{2} \|\mathbf{d}_i^{pred}(\mathbf{m}, \mathbf{q}_i) - \mathbf{d}_i^{obs}\|_2^2,$$

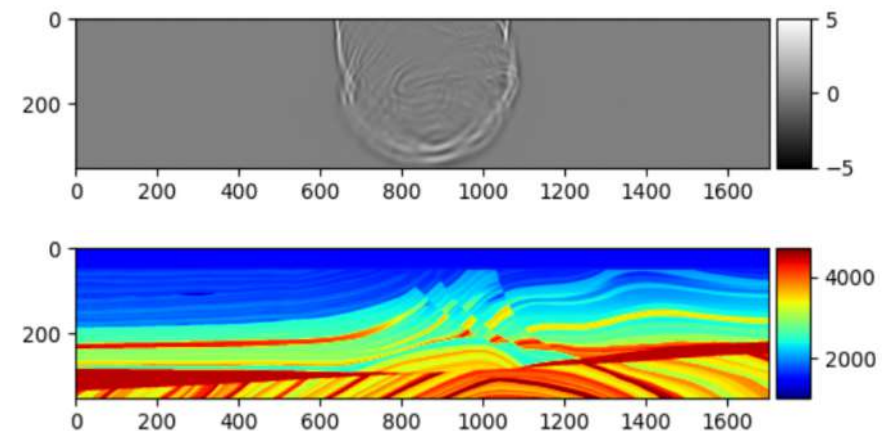
$$\mathbf{d}_i^{pred}(\mathbf{m}, \mathbf{q}_i) = \mathbf{P}_r \mathbf{u}(\mathbf{m})$$

$$\mathbf{u}(\mathbf{m}) = \mathbf{A}(\mathbf{m})^{-1} \mathbf{P}_s^T \mathbf{q}_i.$$

- O processo todo é **muito custoso**



63





# Simulação da Onda Acústica

- Para resolver essa equação, precisamos da sua forma discretizada:

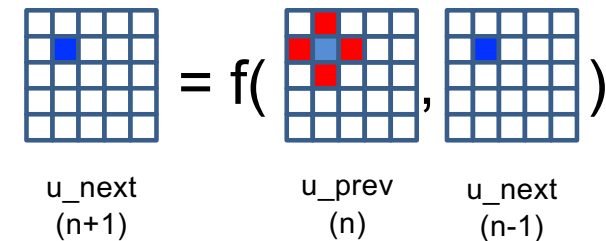
$$p_{i,j}^{n+1} = 2p_{i,j}^n - p_{i,j}^{n-1} + 2\Delta t^2 \cdot v^2 \left( \frac{p_{i+1,j}^n - 2p_{i,j}^n + p_{i-1,j}^n}{\Delta x^2} + \frac{p_{i,j+1}^n - 2p_{i,j}^n + p_{i,j-1}^n}{\Delta y^2} \right)$$

- Algoritmo

```

u_prev = nova_matriz(nz*nx)
u_next = nova_matriz(nz*nx)
...
para (n = 0; n < iterations; n++)
    para (i = 1; i < nz - 1; i++) {
        para (j = 1; j < nx - 1; j++) {
            current = i * nx + j;
            value = prev_u[current + 1] - 2.0 * prev_u[current] + prev_u[current
- 1]) / dxSquared;
            value += (prev_u[current + nx] - 2.0 * prev_u[current] +
prev_u[current - nx]) / dzSquared;
            value *= dtSquared * vel_model[current] * vel_model[current];
            next_u[current] = 2.0 * prev_u[current] - next_u[current] + value;
        }
    }

```

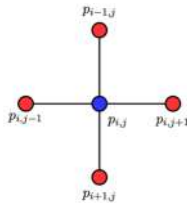


# Simulação da Onda Acústica

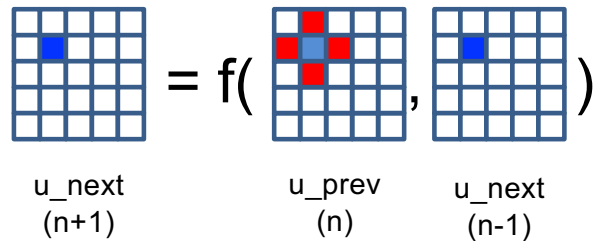
- Para resolver essa equação, precisamos da sua forma discretizada:

$$p_{i,j}^{n+1} = 2p_{i,j}^n - p_{i,j}^{n-1} + 2\Delta t^2 \cdot v^2 \left( \frac{p_{i+1,j}^n - 2p_{i,j}^n + p_{i-1,j}^n}{\Delta x^2} + \frac{p_{i,j+1}^n - 2p_{i,j}^n + p_{i,j-1}^n}{\Delta y^2} \right)$$

- Calcular todos os pontos da matriz para cada passo temporal
- Cálculo dos pontos é um padrão estêncil



- Duas matrizes são suficientes



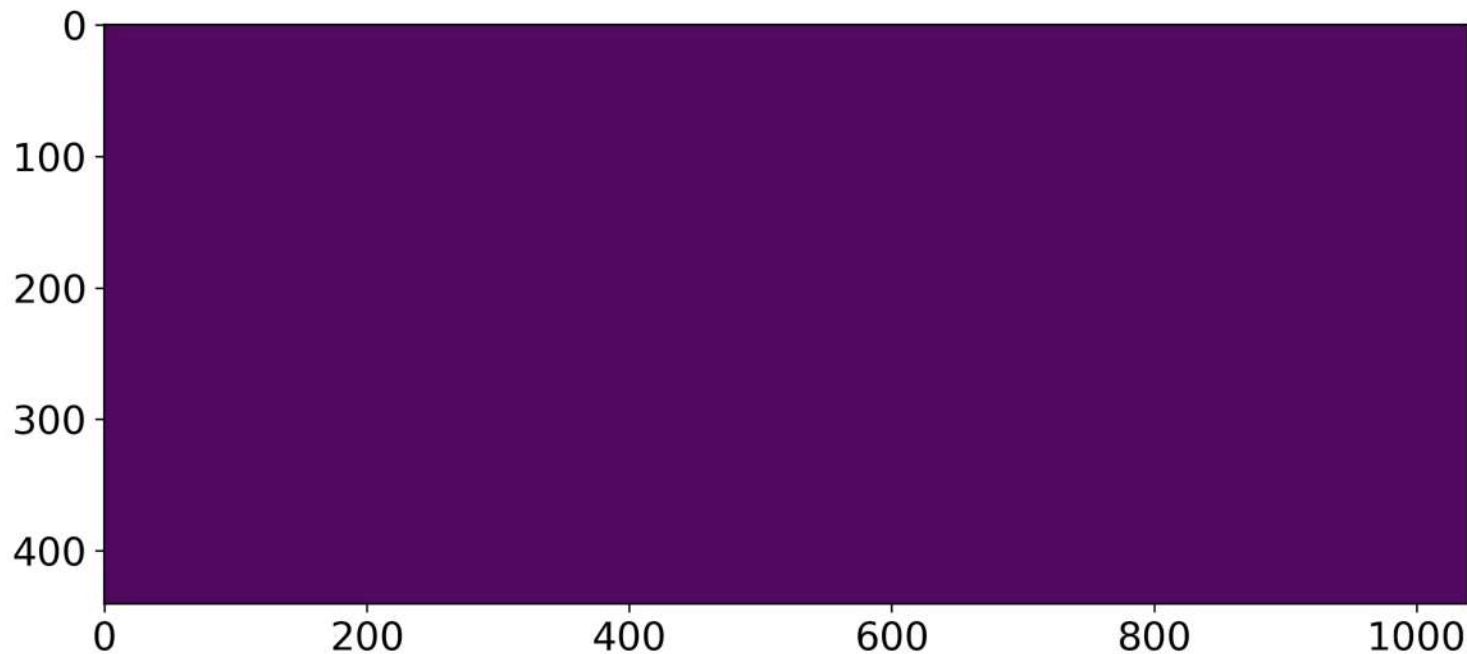
## Algoritmo

```

u_prev = nova_matriz(nz*nx)
u_next = nova_matriz(nz*nx)
...
// loop do passo temporal - não pode ser paralelizado
para (n = 0; n < iterations; n++) {
    para (i = 1; i < nz - 1; i++) { // loop eixo Z
        para (j = 1; j < nx - 1; j++) { // loop eixo X
            current = i * nx + j;
            value = prev_u[current + 1] - 2.0 *
                    prev_u[current] + prev_u[current - 1])
                / dxSquared;
            value += (prev_u[current + nx] - 2.0 *
                    prev_u[current] + prev_u[current - nx])
                / dzSquared;
            value *= dtSquared * vel_model[current] *
                    vel_model[current];
            next_u[current] = 2.0 * prev_u[current] -
                    next_u[current] + value;
        }
    }
    // swap das matrizes
    aux = next_u;
    next_u = prev_u;
    prev_u = aux;
}
    
```

## Propagação de uma onda acústica

- Eis o resultado para vários passos no tempo



# Atividade prática 8



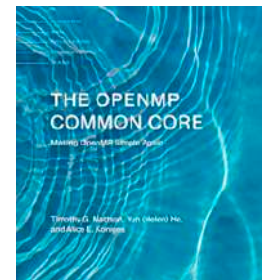
15 minutos para fazer!

## Equação da onda: paralelismo massivo

1. Acesse o notebook 2
2. O programa **wave\_1.c** simula a equação da onda, porém com alguns problemas que causam ineficiência:
  - Ele copia os dados do host para o device, e devolta para o host a cada iteração temporal;
  - Talvez o paralelismo hierárquico nos loops possa ser melhorado

# Bibliografia

- Using OpenMP—The Next Step  
Affinity, Accelerators, Tasking, and SIMD  
By Ruud van der Pas, Eric Stotzer and Christian Terboven, MIT Press, (2017)
- OpenMP Common Core: Making OpenMP Simple Again  
by Tim Mattson, Helen He, Alice Koniges, MIT Press, (2019)  
Código: \$ git clone <https://github.com/tgmattso/OmpCommonCore.git>
- High Performance Parallel Runtimes  
by Michael Klemm and Jim Cownie, De Gruyter Oldenbourg (2021)  
Código: \$ git clone <https://github.com/parallel-runtimes/lomp.git>
- Programming Your GPU with OpenMP: A Hands-On Introduction  
by Simon McIntosh-Smith, Tom Deakin, Tim Mattson  
Githubb: <https://github.com/UoB-HPC/openmp-tutorial>  
(Minicurso ministrado todos os anos na Supercomputing)



# Agradecimentos

Os autores agradecem o apoio da Shell Brasil através do projeto “ANP 20714-2 Desenvolvimento de técnicas numéricas e software para problemas de inversão com aplicações em processamento sísmico” da Universidade de São Paulo e reconhecem a importância estratégica do apoio da ANP por meio do regulamento de arrecadação de P&D.

Hermes Senger agradece o apoio do Projeto Temático “Trends on High Performance Computing, from Resource Management to New Computer Architectures”, coordenado pelo IME/USP e que tem o apoio da FAPESP (Processo 2019/26702-8).

Hermes Senger  
email: [hermes@ufscar.br](mailto:hermes@ufscar.br)

Computing Department  
Federal University of São Carlos- UFSCAR