Introdução à Programação com OpenACC

Evaldo B. Costa - UFRJ
Gabriel P. Silva - UFRJ

Minicurso da XX Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)

Programa

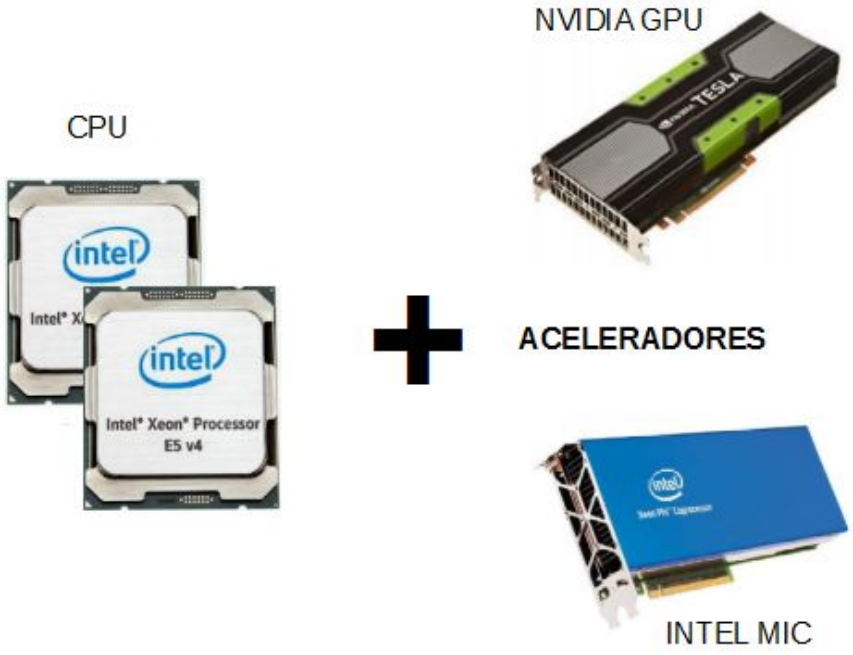
- Introdução a programação paralela
- Introdução ao OpenACC
- Diretivas e Cláusulas
- Diretiva kernels
- Diretiva Parallel
- Diretiva Parallel Loop
- Diretiva kernels vs Diretiva parallel
- Movimentação de Dados
- Compilação
- Exemplos

Introdução a Programação Paralela

Minicurso da XX Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)

Introdução à Programação Paralela

- As arquiteturas paralelas tem alcançado um alto grau de paralelismo com a utilização de um número cada vez maior de processadores, como podem ser encontrados nas arquiteturas multicore, manycore e GPUs.



Introdução à Programação Paralela

- Uma das questões que surgem ao elaborar um programa paralelo é saber se o mesmo apresenta um desempenho adequado quando executado em um ambiente paralelo.

- Deste modo, para avaliar o desempenho de um sistema de processamento paralelo, as métricas mais importantes para serem consideradas são: **speedup** (também conhecido como ganho de desempenho ou aceleração), **eficiência** e **escalabilidade**.

Introdução à Programação Paralela

- O **speedup**, ou aceleração, mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador (T(1)) e o tempo gasto na execução com P processadores (T(P)), como visto na Equação:

$$S(P) = \frac{T(1)}{T(P)}$$

Introdução à Programação Paralela

- A **eficiência** é uma medida de quão efetiva é a adição de novos processadores para ajudar na resolução de um problema. Seu valor é obtido pela razão entre o speedup (S(P)) e o número de processadores (P) utilizados para obter este speedup, conforme podemos observar na equação:

$$E(P) = \frac{S(P)}{P}$$

Introdução à Programação Paralela

- **Escalabilidade**, um sistema é dito escalável quando sua eficiência se mantém constante à medida que o número de processadores P aplicado à solução do problema cresce.

- Se o tamanho do problema é mantido constante e o número de processadores aumenta, o *overhead* de comunicação tende a crescer e a eficiência a diminuir.

Introdução ao OpenACC

Introdução ao OpenACC

- O OpenACC é um modelo de programação aberta para computação paralela desenvolvido com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e portabilidade entre diversos tipos de arquiteturas: *multicore*, *manycore* e GPUs.

Introdução ao OpenACC

- OpenACC foi desenvolvido pelos principais fabricantes de hardware e software.

Características do 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)

Introdução ao OpenACC

- O OpenACC pode ser utilizado em códigos programados em linguagens de programação C/C++ ou Fortran.

C/C++

```
#pragma acc data [clause]
{
    code region ...
}
```

Fortran

```
!$acc data
    code region ...
!$acc end data
```

Diretivas e Cláusulas

Diretivas e Cláusulas

- O modelo de programação usado no OpenACC é baseado em diretivas e cláusulas.
- As **diretivas** são comandos de instrução passadas pelo código ao compilador.
- As **cláusulas** são os valores atribuídos às instruções usadas nas diretivas.
- As diretivas do OpenACC são muito parecidas com as diretivas do OpenMP. As diretivas são escritas na forma de **pragmas**.

Diretiva Kernels

Diretiva Kernels

- Quando é utilizada a diretiva **kernels**, isto significa que é informado ao compilador que existem regiões do código que podem ser paralelizadas e que o compilador será o responsável por identificar quais são as regiões e qual a estratégia que será utilizada.

- A estratégia definida pelo compilador pode ser extrair o máximo de paralelismo do código ou executar somente o mínimo de paralelismo.

Diretiva Kernels

- Os passos do processo de compilação do código usando a diretiva **kernels** são:
  - Analisar o código para identificar regiões de paralelismo
  - Se encontrado, identificar quais dados devem ser transferidos
  - Criar um kernel
  - Movimentar os dados para dispositivo

#pragma acc kernels [clause-list]

Diretiva Parallel

Diretiva Parallel

- A paralelização usando o construtor **parallel** identifica uma região de código que será paralelizada, quando executada em conjunto com a diretiva **loop**, o compilador gerará uma versão paralela do laço para o acelerador.

- Desta forma o compilador executa o paralelismo do código de forma direta.

#pragma acc paralell [clause-list]

Cláusulas Parallel

- Algumas cláusulas são usadas para melhorar o desempenho da região a ser paralelizada, essas cláusulas permitem ter um maior controle e processamento dos laços.

  - **private**
  - **reduction**
  - **async**

Diretiva Parallel Loop

Diretiva Parallel Loop

- Para fazer o paralelismo de vários laços, é necessário que cada laço seja acompanhado de uma diretiva **parallel loop**.
- A diretiva **parallel loop** é uma afirmação do programador de que é seguro e desejável paralelizar o laço afetado.

#pragma acc paralell loop [clause-list]

Diretiva Parallel Loop

- Cada região **parallel loop** pode ter diferentes laços e cada laço pode ser paralelizado e otimizado de forma independente entre eles.

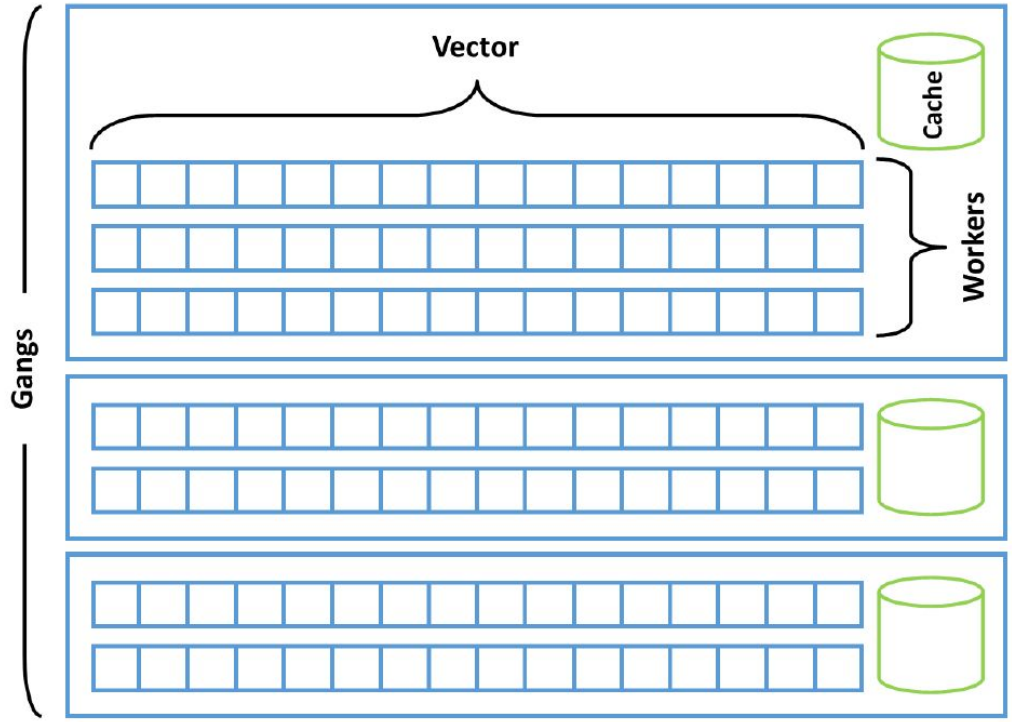
```
#pragma acc parallel loop
for (int i = 0 ; i < N; i++)
     a[i] = 0;
#pragma acc parallel loop
for (int j = 0 ; j < M; j++)
     b[j] = 0;
```

Cláusulas Parallel Loop

- Como o OpenACC serve como uma linguagem para aceleradores genéricos existem três níveis de paralelismo que podem ser usados no OpenACC.

gang	Particiona o laço entre as gangs
worker	Particiona o laço entre os workers
vector	Vetoriza o laço
seq	Não particiona o laço, executar sequencialmente

Cláusulas Parallel Loop



Gangs x Workers x Vector

Cláusulas Parallel Loop

- O uso dos níveis de paralelismo são aplicados na diretiva **parallel loop** para gerar maior ganho na execução do laço. Também podem ser usadas da diretiva **kernels**.

```
#pragma acc parallel loop gang
    for(i = 0 ; i < size; i++)
        #pragma acc loop worker
        for(j = 0 ; j < size; j++)
            #pragma acc loop vector
            for(k = 0 ; k < size; k++)
                c[i][j]+=a[i][k]*b[k][j];
```

Diretiva kernels
vs
Diretiva parallel

Diretiva kernels vs parallel

- As diretivas **kernels** e **parallel** no OpenACC, tem 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 usando a diretiva **parallel**, porém existem códigos que possuem grande dificuldade de alteração não sendo possível usar diretiva **parallel**, neste caso é usado a diretiva **kernels**, pois as alterações são as mínimas possíveis.

Diretiva kernels vs parallel

kernels	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];`
`}`
	`#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];`
	`}`

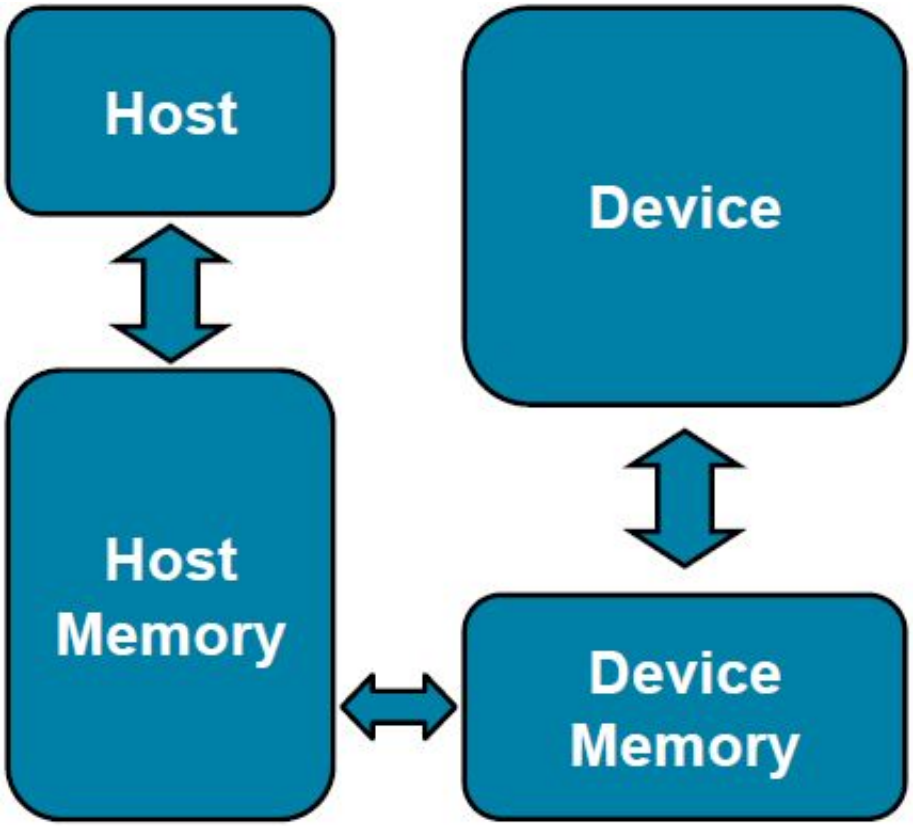
Movimentação de dados

Movimentação de Dados

- Um grande fator de impacto de desempenho no processamento paralelo é a movimentação de dados, principalmente quando se faz o processamento dos dados em lugares diferentes.

  - Define a região do código na qual os dados permanecem no acelerador;
  - Os dados são compartilhados entre todos os *kernels* da região;
  - Transferências de dados explícitas.

Movimentação de Dados



Movimentação de Dados

- 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 podem ser usados nas diretivas **kernels** ou **parallel**.

#pragma acc data [clause]

Movimentação de Dados

- Principais **cláusulas** utilizadas para a movimentação de dados:

  - copy
  - copyin
  - copyout
  - create
  - present
  - deviceptr

Compilação

Compilação

- Antes de compilar qualquer código é importante saber quais dispositivos aceleradores estão configurados para uso no sistema.

- Existem alguns comandos que fornecem informações de modelos e características desses dispositivos.

```
# nvidia-smi -q | grep "Product Name"

# pgaccelinfo | grep "Device Name"
```

Compilação

- Para compilar os códigos feitos em OpenACC, é necessário o uso de compiladores que suportem OpenACC.

  - PGI (Portland Group)
  - Cray
  - GNU GCC (a partir da versão 7)

Compilação

- Nos exemplos apresentados neste minicurso usaremos o compilador da PGI disponibilizado para a comunidade, o compilador pode ser baixado do sítio da PGI através do link:

  https://www.pgroup.com/products/community.htm

- Para compilar códigos em C usaremos o comando **pgcc**, e para compilação de códigos em C++ usar o comando **pgc++**.

Compilação

Parâmetro	Descrição
-fast	faz a otimização do código
-acc	habilita o uso de diretivas OpenACC
-Minfo=accel	informações sobre quais partes do código foram aceleradas
-ta=nvidia	compila o código usando NVIDIA

Compilação

- Alguns exemplos de uso do compilador PGI e seus parâmetros básicos.

```
$ pgcc -acc -ta=nvidia -Minfo=accel main.c
$ pgc++ -acc -ta=nvidia -Minfo=accel main.cpp
```

- Para usar o GNU GCC especificar o parâmetro -fopenacc.

```
$ gcc -fopenacc main.c
```

Exemplos

Cálculo de PI

- Implementação sequencial simples para o cálculo de PI (serial).

```
#include <stdio.h>
#define N 1000000000

int main(void) {
    double pi = 0.0f; long i;
    for (i=0; i<N; i++)
    {
      double t=(double) ((i+0.5)/N);
      pi += 4.0/(1.0+t*t);
    }
    printf("pi=%f\n",pi/N);
    return 0;
}
```

Cálculo de PI

- Podemos paralelizar o código em OpenMP adicionando A linha **#pragma omp parallel for reduction(+: pi)** antes do laço.

```
#pragma omp parallel for reduction(+: pi)
for (i=0; i<N; i++)
{
   double t=(double) ((i+0.5)/N);
   pi += 4.0/(1.0+t*t);
}
```

Cálculo de PI

- Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC. Da mesma forma que foi usado no OpenMP, adicionar a linha **#pragma acc parallel loop reduction(+: pi)** antes do laço.

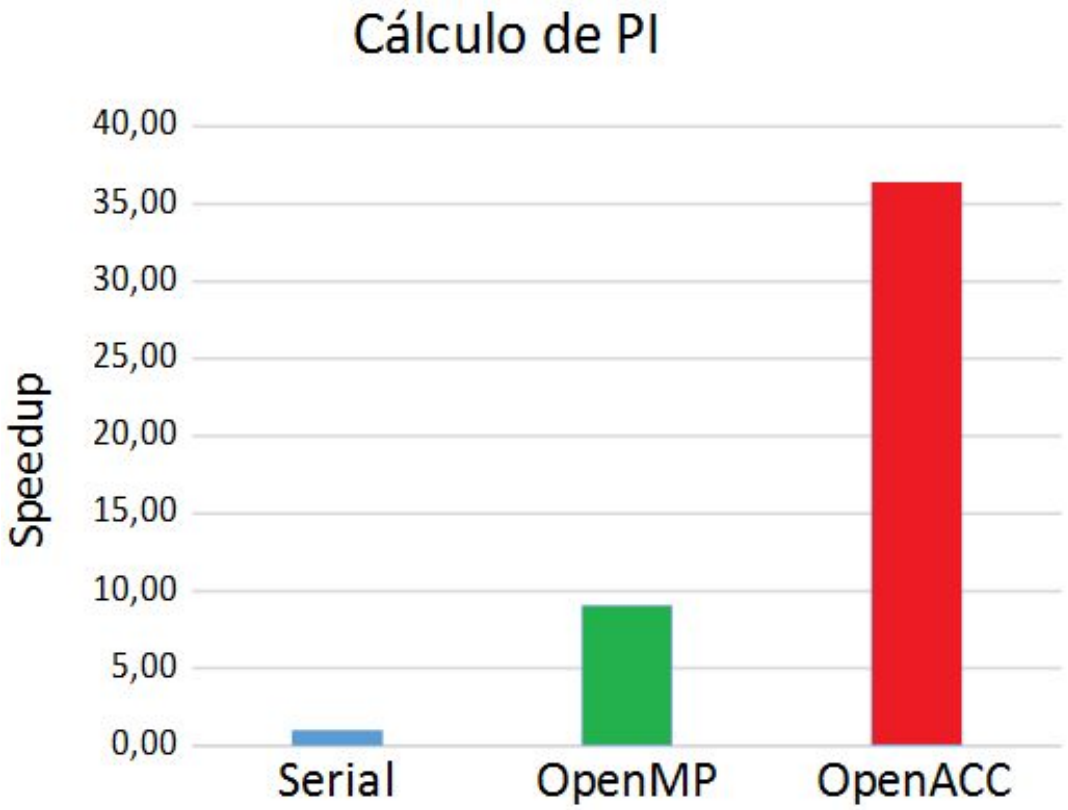
```
#pragma acc parallel loop reduction(+: pi)
for (i=0; i<N; i++)
{
    double t=(double) ((i+0.5)/N);
    pi += 4.0/(1.0+t*t);
}
```

Cálculo de PI

- O speedup encontrado usando programação OpenMP foi de 9,10 em comparação a execução em sequencial.

- Utilizando OpenACC o speedup foi de 36,40 em relação ao executado sequencialmente.

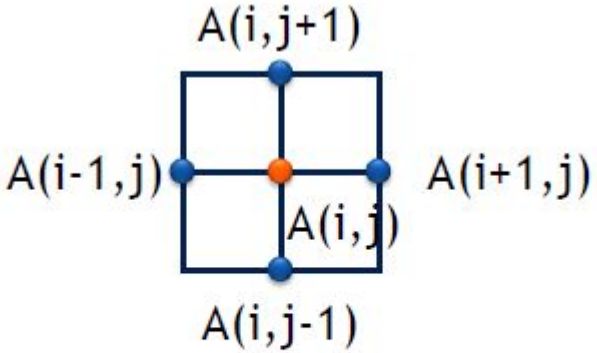
Speedup


Cálculo de PI
Speedup
40,00
35,00
30,00
25,00
20,00
15,00
10,00
5,00
0,00
Serial
OpenMP
OpenACC

Método Jacobi

- O Método de Jacobi é um procedimento iterativo para a resolução de sistemas lineares.

- Neste exemplo faremos o cálculo da temperatura na placa usando a equação de Laplace.



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Método Jacobi

- Implementação sequencial simples para o cálculo da temperatura da placa usando o método Jacobi (serial).

```
while ( dt > MAX_TEMP_ERROR && iteration <=
              max_iterations ) {
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
          Anew[i][j] = 0.25 * (A[i+1][j] + A[i-1][j] +
   A[i][j+1] + A[i][j-1]);
     }
}
dt = 0.0;
for(i = 1; i <= ROWS; i++){
   for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
          A[i][j] = Anew[i][j];
   }
}
```

Método Jacobi

- Podemos paralelizar o código em OpenMP adicionando as linhas **#pragma omp parallel for** antes do primeiro laço e a linha **#pragma omp parallel for reduction(max:dt)** antes do segundo laço, desta forma pode-se utilizar mais de uma thread para o cálculo.

Método Jacobi

```
#pragma omp parallel for
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
          Anew[i][j] = 0.25 * (A[i+1][j] + A[i-1][j] +
   A[i][j+1] + A[i][j-1]);
     }
}
dt = 0.0;
#pragma omp parallel for reduction(max:dt)
for(i = 1; i <= ROWS; i++){
   for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
          A[i][j] = Anew[i][j];
   }
}
```

Método Jacobi

- Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC.

- Antes do primeiro laço adicionar a linha **#pragma acc parallel loop**, e por fim a linha **#pragma acc parallel loop reduction(max:dt)** no segundo laço.

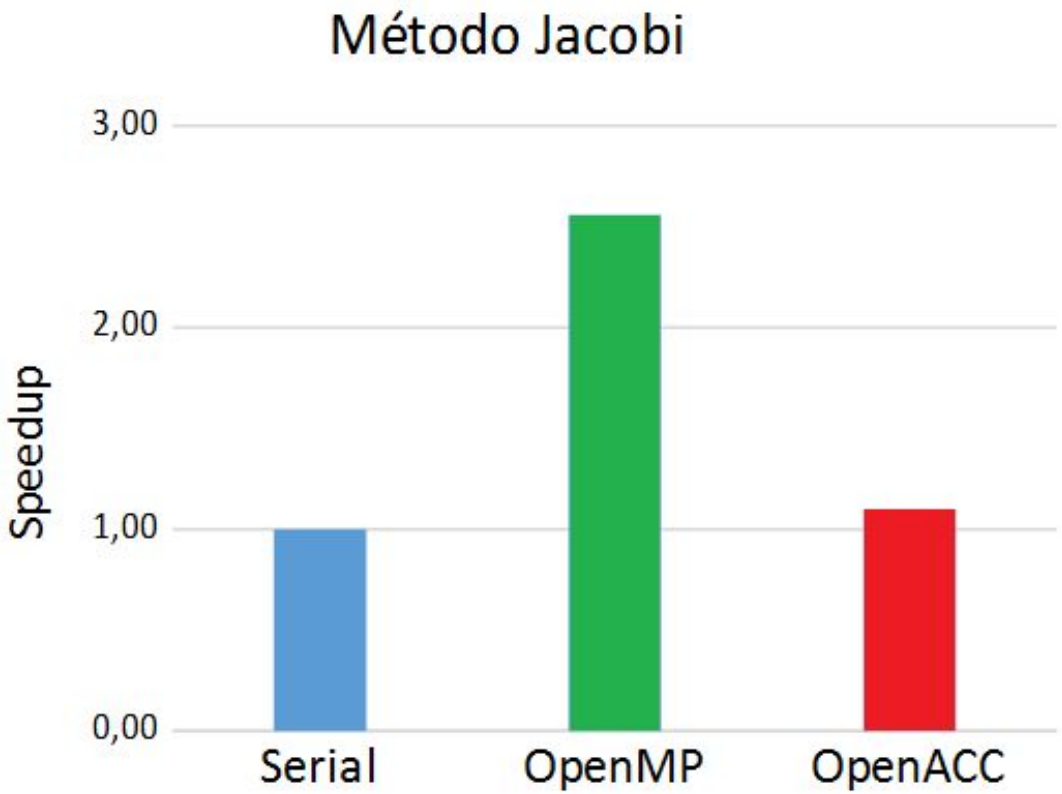
Método Jacobi

```
#pragma acc parallel loop
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
          Anew[i][j] = 0.25 * (A[i+1][j] + A[i-1][j]
  + A[i][j+1] + A[i][j-1]);
     }
}
dt = 0.0;
#pragma acc parallel loop reduction(max:dt)
for(i = 1; i <= ROWS; i++){
   for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
          A[i][j] = Anew[i][j];
   }
}
```

Método Jacobi

- O speedup encontrado usando programação OpenMP foi de 2,56 em comparação a execução em sequencial
- Utilizando OpenACC o speedup foi de 1,10 em relação ao executado em sequencial.

Speedup


Método Jacobi
3,00
2,00
1,00
0,00
Speedup
Serial
OpenMP
OpenACC

Método Jacobi

Qual é o problema?

Método Jacobi

- A matriz de cálculo não é executada no acelerador. Toda vez que o acelerador faz a operação ele usa a matriz que está no host.

- Para resolver este problema, usar a diretiva **data** do OpenACC. Adicionar a linha **#pragma acc data copy(A) create (Anew)** antes dos dois laços para fazer a cópia da matriz para o acelerador.

Método Jacobi

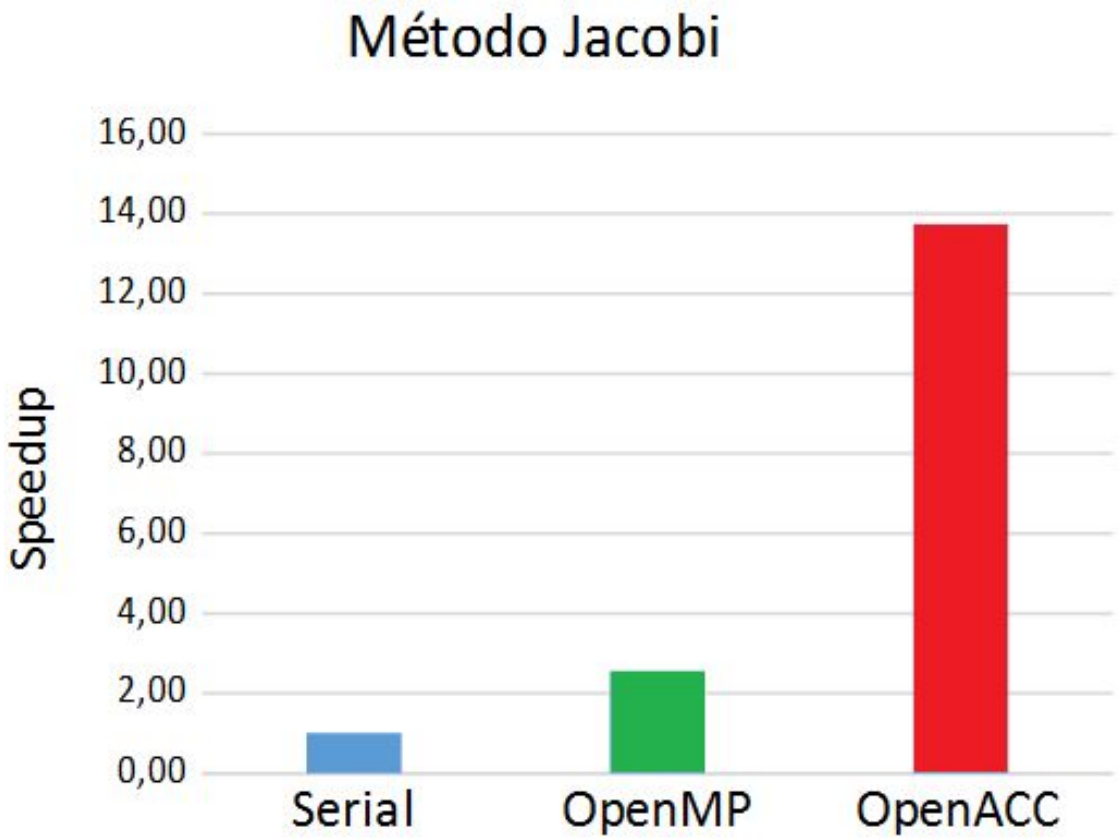
```
#pragma acc data copy(A) create(Anew)
while ( dt > MAX_TEMP_ERROR && iteration <=
              max_iterations ) {
#pragma acc parallel loop
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
          Anew[i][j] = 0.25 * (A[i+1][j] + A[i-1][j]
  + A[i][j+1] + A[i][j-1]);
     }
}
dt = 0.0;
#pragma acc parallel loop reduction(max:dt)
for(i = 1; i <= ROWS; i++){
   for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
          A[i][j] = Anew[i][j];
   }
}
```

Método Jacobi

- O speedup encontrado usando programação OpenMP foi de 2,56 em comparação a execução em sequencial

- Por sua vez usando OpenACC o speedup foi de 13,75 em relação ao executado em sequencial.

Speedup


Método Jacobi
Speedup
16,00
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
Serial
OpenMP
OpenACC

Conclusão

- O OpenACC é fácil de usar
- Trabalho com o uso de diretivas
- Em alguns casos são feitas pequenas alterações no código
- O código pode ser implementado em qualquer acelerador

Referências OpenACC

- Chen, S. (2017). Introduction to OpenACC. Research Computing Services Information Services and Technology Boston University.
- Costa, E. B., Silva, G. P., and Teixeira, M. (2018). Avaliação de desempenho do montador daligner em arquiteturas manycore. In WSCAD 2018 - WCH (), São Paulo - SP, Brazil.
- Larkin, J. (2018). Introduction to OpenACC. NVIDIA.
- Abbott, S. (2017). Advanced OpenACC. NVIDIA Corporation.
- Correa, J. C. and Silva, G. P. (2012). Analysis and performance evaluation of parallel BLAST. I. J. Comput. Appl., 20(2):112–122.

Referências OpenACC

- [OpenACC-Standard.org 2015] OpenACC-Standard.org (2015). OpenACC Programming
- and Best Practices Guide. OpenACC-Standard.org.
- [Rahman 2013] Rahman, R. (2013). Intel Xeon Phi Coprocessor Architecture and Tools
- The Guide for Application Developers. Apress Open.
- [Silva 2018] Silva, G. P. (2018). Programação Paralela com MPI Um Curso Introdutório. Amazon.
- Larkin, J. (2018). Introduction to OpenACC. NVIDIA

Obrigado

Evaldo B. Costa
ebcosta@dcc.ufrj.br

Gabriel P. Silva
gabriel@dcc.ufrj.br