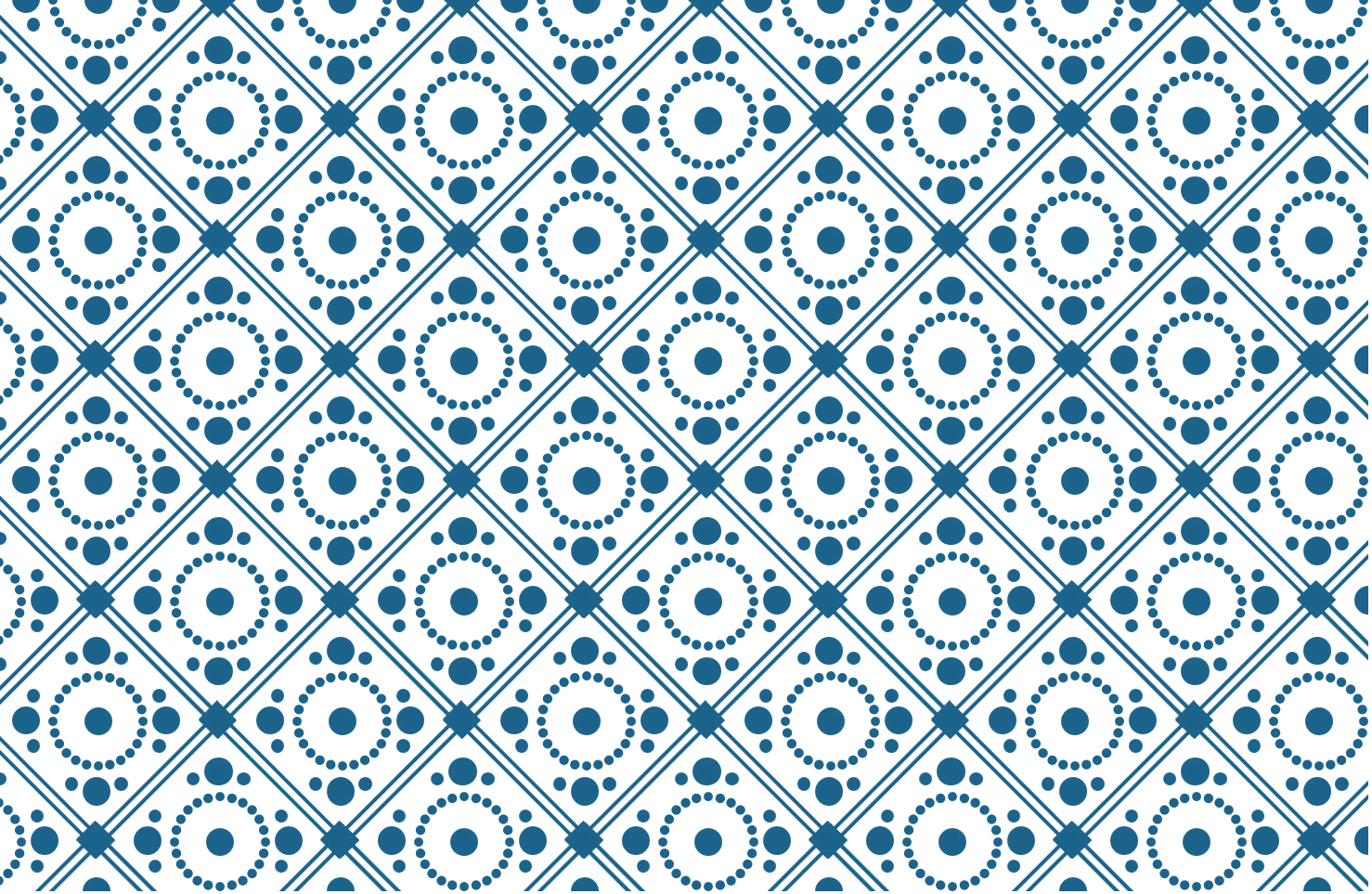
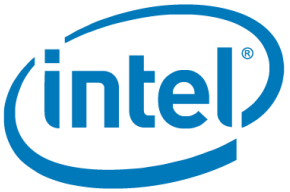
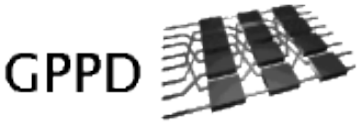








# **INTEL MODERN CODE:** INTRODUÇÃO À PROGRAMAÇÃO VETORIAL E PARALELA PARA O PROCESSADOR INTEL XEON PHI KNIGHTS LANDING


Matheus S. Serpa, Vinícius G. Pinto, Philippe O. A. Navaux
**Contato:** msserpa@inf.ufrgs.br

INTEL MODERN CODE PARTNER

# EQUIPE

GPPD
Grupo de Processamento Paralelo e Distribuído

intel

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

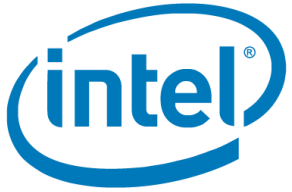
**Philippe O. A. Navaux**

Vinícius Garcia Pinto

Matheus S. Serpa

**Demais membros do GPPD.**

# REVISÃO DE OPENMP



# INTRODUÇÃO

OpenMP é um dos modelos de programação paralelas mais usados hoje em dia.

Esse modelo é relativamente fácil de usar, o que o torna um bom modelo para iniciar o aprendizado sobre escrita de programas paralelos.

Observações:

- Assumo que todos sabem programar em linguagem C. OpenMP também suporta Fortran e C++, mas vamos nos restringir a C.

# SINTAXE BÁSICA - OPENMP

Tipos e protótipos de funções no arquivo:

**#include <omp.h>**

A maioria das construções OpenMP são diretivas de compilação.

**#pragma omp construct [clause [clause]…]**

- Exemplo:

**#pragma omp parallel private(var1, var2) shared(var3, var4)**

A maioria das construções se aplicam a um **bloco estruturado**.

**Bloco estruturado**: Um bloco com um ou mais declarações com um ponto de entrada no topo e um ponto de saída no final.

Podemos ter um **exit()** dentro de um bloco desses.

# NOTAS DE COMPILAÇÃO

Linux e OS X com **gcc** or **intel icc**:

```
gcc -fopenmp foo.c #GCC
icc -qopenmp foo.c #Intel ICC
export OMP_NUM_THREADS=40
./a.out
```

Para shell bash

Por padrão é o nº de proc. virtuais.

Também funciona no Windows!

Até mesmo no Visual Studio!

**Mas vamos usar Linux ☺**

# FUNÇÕES

Funções da biblioteca OpenMP.

```
// Arquivo interface da biblioteca OpenMP para C/C++
#include <omp.h>

// retorna o identificador da thread.
int omp_get_thread_num();

// indica o número de threads a executar na região paralela.
void omp_set_num_threads(int num_threads);

// retorna o número de threads que estão executando no momento.
int omp_get_num_threads();

// Comando para compilação habilitando o OpenMP.
icc -o hello hello.c -qopenmp
```

# DIRETIVAS

Diretivas do OpenMP.

```
// Cria a região paralela. Define variáveis privadas e
compartilhadas entre as threads.
#pragma omp parallel private(...) shared(...)
{ // Obrigatoriamente na linha de baixo.

// Apenas a thread mais rápida executa.
#pragma omp single


}
```

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS

A construção de divisão de trabalho em laços divide as iterações do laço entre as *threads* do time.

```
#pragma omp parallel private(i) shared(N)
{
 #pragma omp for
 for(i = 0; i < N; i++)
   NEAT_STUFF(i);
}
```

A variável **i** será feita privada para cada *thread* por padrão. Você poderia fazer isso explicitamente com a cláusula **private(i)**

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS UM EXEMPLO MOTIVADOR

Código sequencial

```
for(i = 0; i < N; i++)
  a[i] = a[i] + b[i];
```

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS UM EXEMPLO MOTIVADOR

Código sequencial

```
for(i = 0; i < N; i++)
  a[i] = a[i] + b[i];
```

Região OpenMP parallel

```
#pragma omp parallel
{
  int id, i, Nthrds, istart, iend;
  id = omp_get_thread_num();
  Nthrds = omp_get_num_threads();
  istart = id * N / Nthrds;
  iend = (id+1) * N / Nthrds;
  if(id == Nthrds-1) iend = N;
  for(i = istart; i < iend; i++)
    a[i] = a[i] + b[i];
}
```

# CONSTRUÇÕES DE DIVISÃO DE LAÇOS UM EXEMPLO MOTIVADOR

Código sequencial

```
for(i = 0; i < N; i++)
  a[i] = a[i] + b[i];
```

Região OpenMP parallel

```
#pragma omp parallel
{
  int id, i, Nthrds, istart, iend;
  id = omp_get_thread_num();
  Nthrds = omp_get_num_threads();
  istart = id * N / Nthrds;
  iend = (id+1) * N / Nthrds;
  if(id == Nthrds-1) iend = N;
  for(i = istart; i < iend; i++)
    a[i] = a[i] + b[i];
}
```

**Região paralela OpenMP com uma construção de divisão de laço**

```
#pragma omp parallel
#pragma omp for
for(i = 0; i < N; i++) a[i] = a[i] + b[i];
```

# CONSTRUÇÕES PARALELA E DIVISÃO DE LAÇOS COMBINADAS

Algumas cláusulas podem ser combinadas.

```
double res[MAX]; int i;
#pragma omp parallel
{
  #pragma omp for
  for(i=0; i < MAX; i++)
    res[i] = huge();
}
```

=

```
double res[MAX]; int i;
#pragma omp parallel for
  for(i=0; i < MAX; i++)
    res[i] = huge();
```

# REDUÇÃO

Combinação de variáveis locais de uma *thread* em uma variável única.

- Essa situação é bem comum, e chama-se **redução**.
- O suporte a tal operação é fornecido pela maioria dos ambientes de programação paralela.

# DIRETIVA REDUCTION

reduction(op : list_vars)

Dentro de uma região paralela ou de divisão de trabalho:

- Será feita uma cópia local de cada variável na lista
- Será inicializada dependendo da **op** (ex. 0 para +, 1 para *).
- Atualizações acontecem na cópia local.
- Cópias locais são “reduzidas” para uma única variável original (global).

**#pragma omp for reduction(* : var_mult)**

# EXERCÍCIO 1 - VECTOR SUM

cd vectorSum/ make ./vectorSum.exec <elementos>

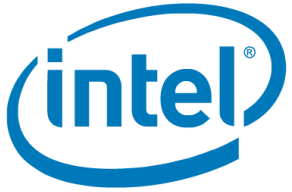
```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  for(i = 0; i < N; i++)
    sum += v[i];

  return sum

}
```

# SOLUÇÃO 1 - VECTOR SUM

cd vectorSum/ make ./vectorSum.exec <elementos>

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for private(i) reduction(+ : sum)
  for(i = 0; i < N; i++)
    sum += v[i];

  return sum

}
```

# INTRODUÇÃO A PROGRAMAÇÃO VETORIAL



# *SINGLE INTRUCTION MULTIPLE DATA* (SIMD)

Técnica aplicada por unidade de execução

- Opera em mais de um elemento por iteração.
- Reduz número de instruções significativamente.

Elementos são armazenados em registradores SIMD

**Scalar**

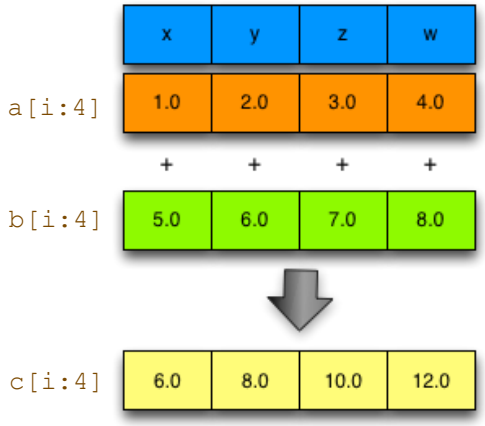
Uma instrução. Uma operação.

```
for(i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

**Vector**

Uma instrução. Quatro operações, por exemplo.

```
for(i = 0; i < N; i += 4)
  c[i:4] = a[i:4] + b[i:4];
```



# *SINGLE INTRUCTION MULTIPLE DATA* (SIMD)

Técnica aplicada por unidade de execução

- Opera em mais de um elemento por iteração.
- Reduz número de instruções significativamente.

Elementos são armazenados em registradores SIMD

**Scalar**

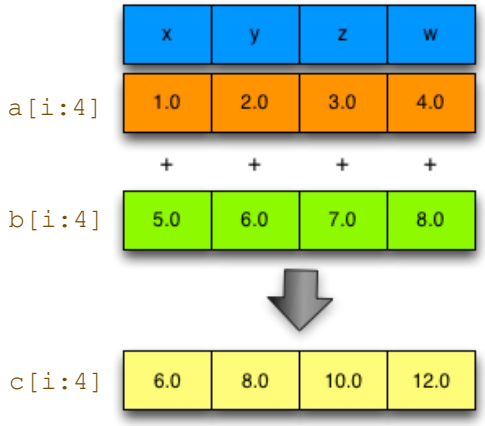
Uma instrução. Uma operação.

```
for(i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

**Vector**

Uma instrução. Quatro operações, por exemplo.

```
for(i = 0; i < N; i += 4)
  c[i:4] = a[i:4] + b[i:4];
```




Dados contíguos para desempenho ótimo
c[0] c[1] c[2] c[3] …

# PROGRAMAÇÃO VETORIAL

**Vetorização**

```
#pragma simd
for(i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

**Vetorização com redução**

```
#pragma simd reduction(+ : v)
for(i = 0; i < N; i++)
  v += a[i] + b[i];
```

# EXERCÍCIO 2, PARTE A: DOT PRODUCT SIMD

cd dotProduct/ make ./dotProduct.exec <elementos>

```
double dotproduct(double *a, int *b, long long int N){
  long long int i;
  double dot = 0.0;



  for(i = 0; i < N; i++)
    dot += a[i] * b[i];

  return dot;

}
```

# SOLUÇÃO 2.1, PARTE A: DOT PRODUCT SIMD

cd dotProduct/ make ./dotProduct.exec <elementos>

```
double dotproduct(double *a, int *b, long long int N){
  long long int i;
  double dot = 0.0;


  #pragma simd reduction(+ : dot)
  for(i = 0; i < N; i++)
    dot += a[i] * b[i];

  return dot;

}
```

# EXERCÍCIO 2, PARTE B: DOT PRODUCT PARALLEL

cd dotProduct/ make ./dotProduct.exec <elementos>

```
double dotproduct(double *a, int *b, long long int N){
  long long int i;
  double dot = 0.0;



  for(i = 0; i < N; i++)
    dot += a[i] * b[i];

  return dot;

}
```

# SOLUÇÃO 2.2, PARTE B: DOT PRODUCT PARALLEL

cd dotProduct/ make ./dotProduct.exec <elementos>

```
double dotproduct(double *a, int *b, long long int N){
  long long int i;
  double dot = 0.0;


  #pragma omp parallel for private(i) reduction(+ : dot)
  for(i = 0; i < N; i++)
    dot += a[i] * b[i];

  return dot;

}
```

# EXERCÍCIO 2, PARTE C: DOT PRODUCT PARALLEL SIMD

cd dotProduct/ make ./dotProduct.exec <elementos>

```
double dotproduct(double *a, int *b, long long int N){
  long long int i;
  double dot = 0.0;



  for(i = 0; i < N; i++)
    dot += a[i] * b[i];

  return dot;

}
```

# SOLUÇÃO 2.3, PARTE C: DOT PRODUCT PARALLEL SIMD

cd dotProduct/ make ./dotProduct.exec <elementos>

```
double dotproduct(double *a, int *b, long long int N){
  long long int i;
  double dot = 0.0;


  #pragma omp parallel for simd reduction(+ : dot)
  for(i = 0; i < N; i++)
    dot += a[i] * b[i];

  return dot;

}
```

# EXERCÍCIO 3, PARTE A: MM - PARALLEL

cd matrixMultiplication/ make ./matrixMultiplication.exec <elementos>

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++){
  for(j = 0; j < N; j++){
   for(k = 0; k < N; k++){
    C[i * N + j] += A[i * N + k] * B[k * N + j];
   }
  }
 }

}
```

# SOLUÇÃO 3.1, PARTE A: MM - PARALLEL

cd matrixMultiplication/ make ./matrixMultiplication.exec <elementos>

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;

 #pragma omp parallel for private(i, j, k)
 for(i = 0; i < N; i++){
  for(j = 0; j < N; j++){
   for(k = 0; k < N; k++){
    C[i * N + j] += A[i * N + k] * B[k * N + j];
   }
  }
 }

}
```

# EXERCÍCIO 3, PARTE B: MM - SIMD

cd matrixMultiplication/ make ./matrixMultiplication.exec <elementos>

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++){
  for(j = 0; j < N; j++){
   for(k = 0; k < N; k++){
    C[i * N + j] += A[i * N + k] * B[k * N + j];
   }
  }
 }

}
```

# ~~SOLUÇÃO 3.2~~, PARTE B: MM – SIMD WRONG

cd matrixMultiplication/ make ./matrixMultiplication.exec <elementos>

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++){
  for(j = 0; j < N; j++){
   #pragma simd
   for(k = 0; k < N; k++){
    C[i * N + j] += A[i * N + k] * B[k * N + j];
   }
  }
 }

}
```

# EXERCÍCIO 3, PARTE C: MM - SIMD

cd matrixMultiplication/ make ./matrixMultiplication.exec <elementos>

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++){
  for(j = 0; j < N; j++){
   #pragma simd
   for(k = 0; k < N; k++){
    C[i * N + j] += A[i * N + k] * B[k * N + j];
   }
  }
 }

}
```

# SOLUÇÃO 3.3, PARTE C: MM - SIMD

cd matrixMultiplication/ make ./matrixMultiplication.exec <elementos>

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++){
  for(k = 0; k < N; k++){
   #pragma simd
   for(j = 0; j < N; j++){
    C[i * N + j] += A[i * N + k] * B[k * N + j];
   }
  }
 }

}
```

# EXERCÍCIO 3, PARTE D: MM – PARALLEL SIMD

cd matrixMultiplication/ make ./matrixMultiplication.exec <elementos>

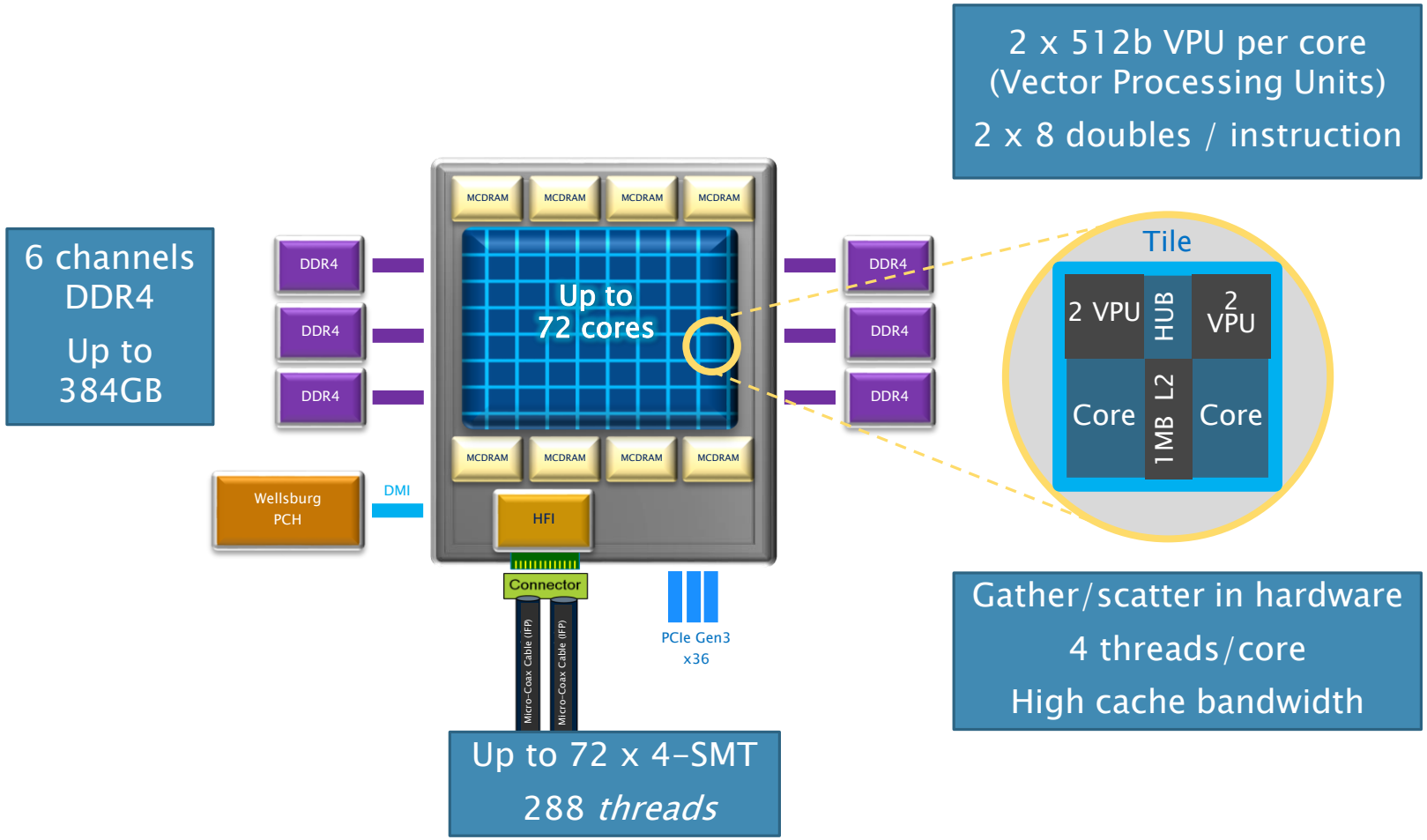
```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++){
  for(k = 0; k < N; k++){
   #pragma simd
   for(j = 0; j < N; j++){
    C[i * N + j] += A[i * N + k] * B[k * N + j];
   }
  }
 }

}
```

# SOLUÇÃO 3.4, PARTE D: MM – PARALLEL SIMD

cd matrixMultiplication/ make ./matrixMultiplication.exec <elementos>

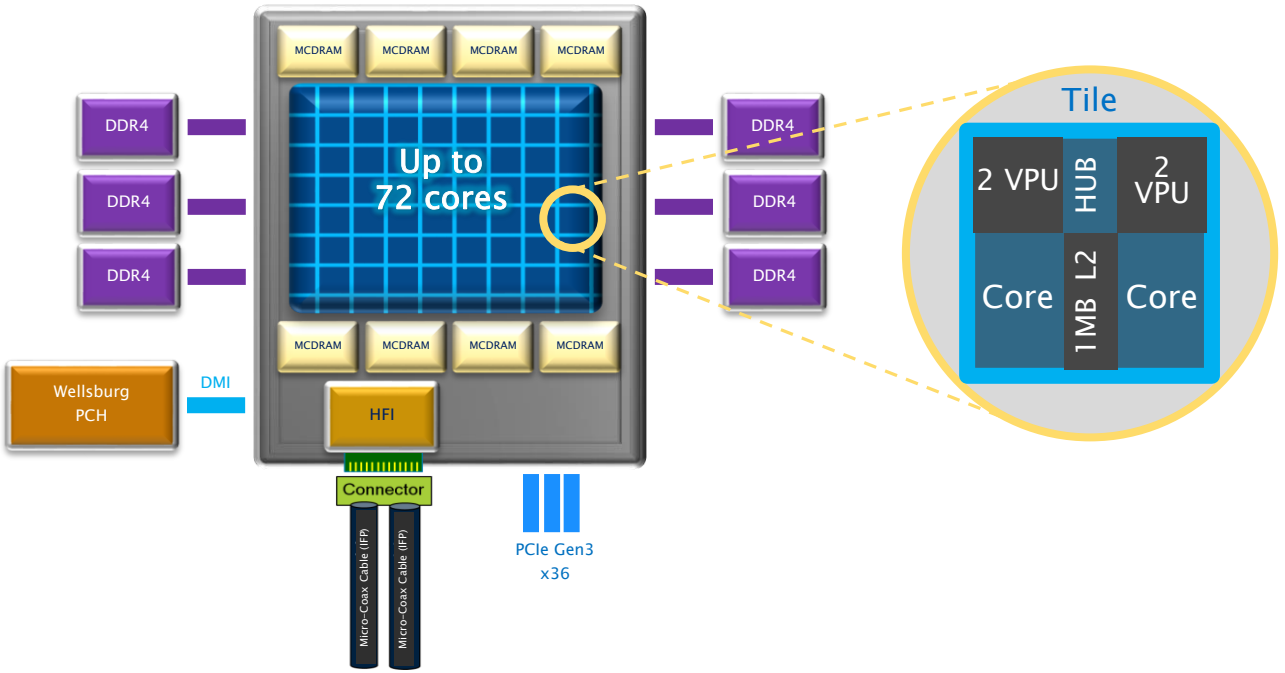
```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;

 #pragma omp parallel for private(i, j, k)
 for(i = 0; i < N; i++){
  for(k = 0; k < N; k++){
   #pragma simd
   for(j = 0; j < N; j++){
    C[i * N + j] += A[i * N + k] * B[k * N + j];
   }
  }
 }

}
```

# INTEL XEON PHI KNIGHTS LANDING

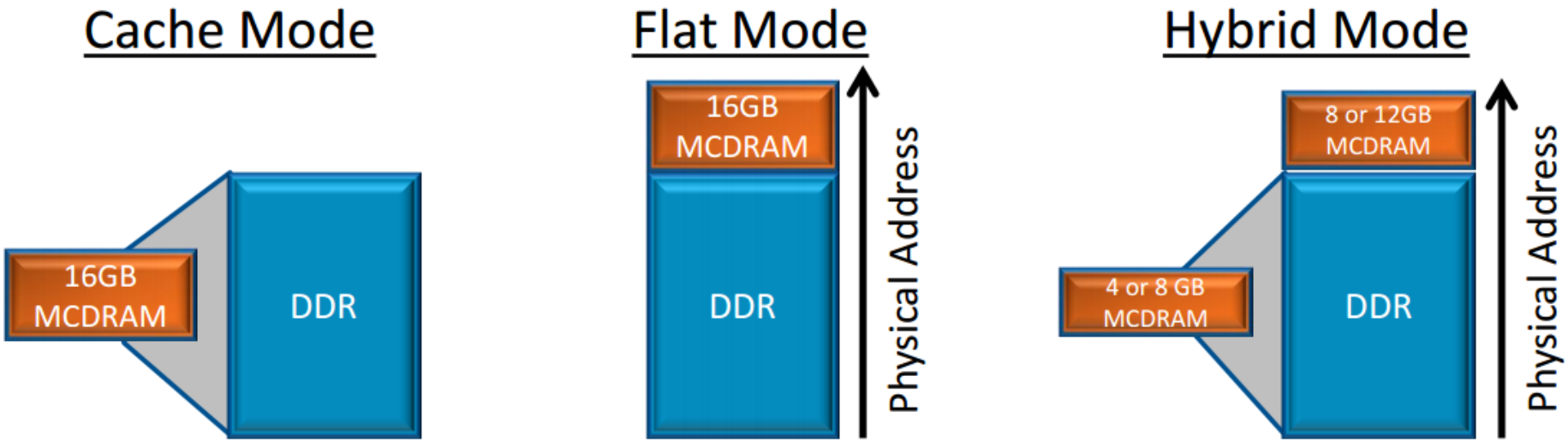


# INTEL XEON PHI KNIGHTS LANDING

FREE TEST DRIVE
colfaxresearch.com



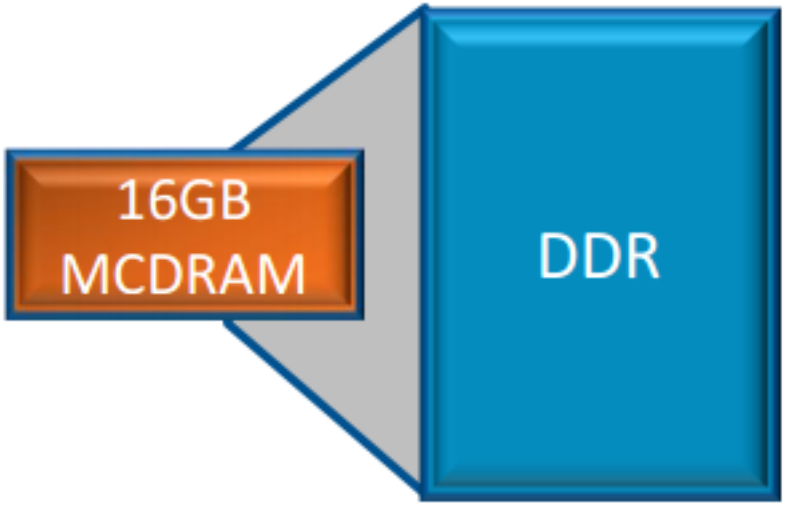
# INTEL XEON PHI
## MODOS DE MEMÓRIA



# INTEL XEON PHI
## MODE CACHE

Transparente para Software

Engloba todos os endereços da memória DDR



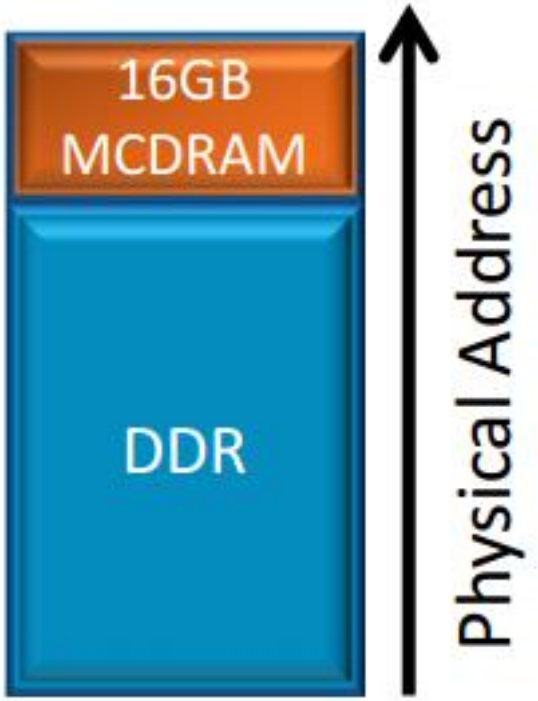
# INTEL XEON PHI
## MODO FLAT

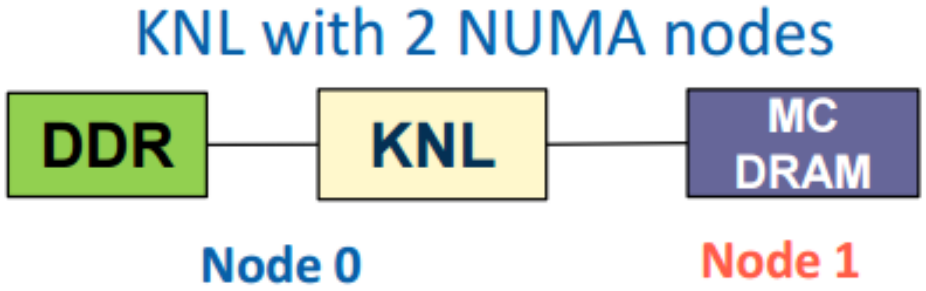
MCDRAM como uma memória regular

MCDRAM exposta como um nó NUMA
- Gerenciada via software
- Mesmo espaço de endereçamento

Memória é alocada na DDR por padrão.

Funções de bibliotecas e **numactl** para alocar dados na MCDRAM.



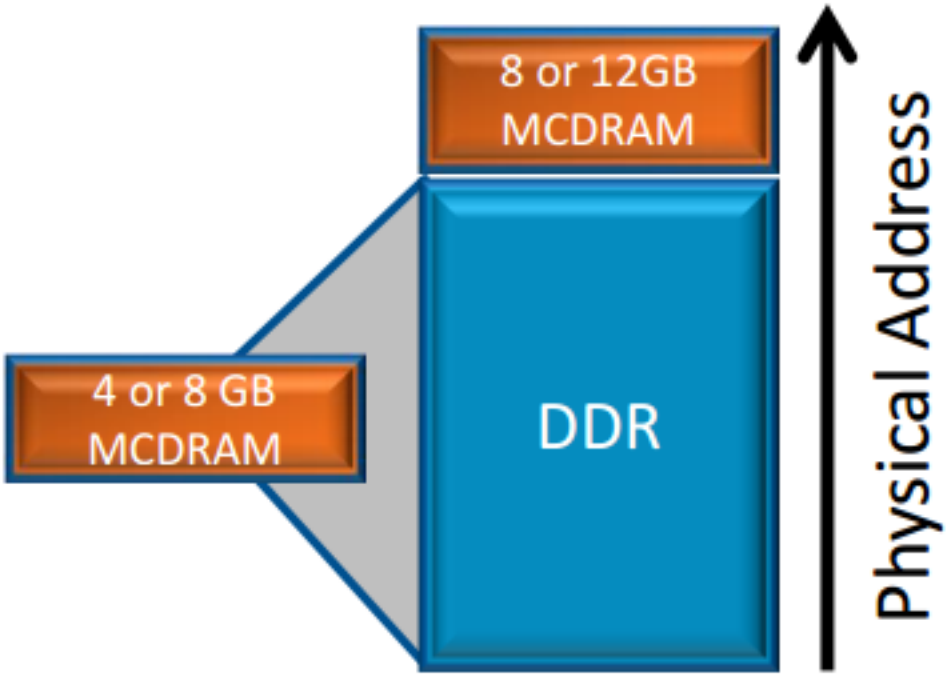



# INTEL XEON PHI
## MODO HÍBRIDO

Parte cache, parte flat

25% ou 50% cache

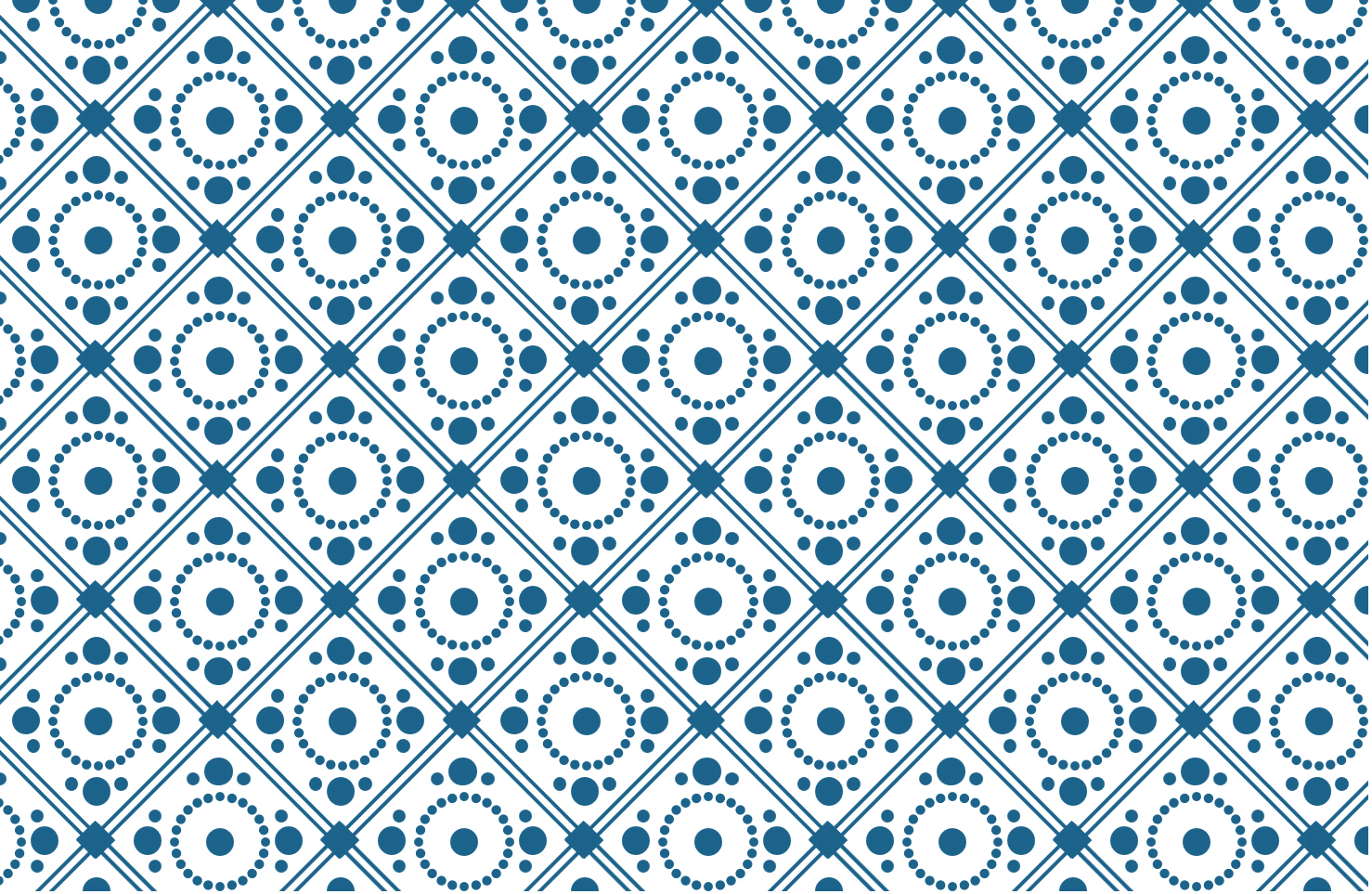
Benefícios de ambos os modos

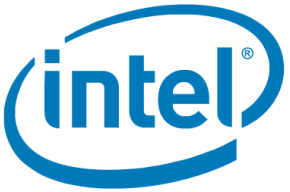


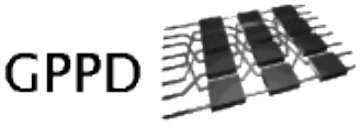
# EXERCÍCIO 4
# MM – XEON PHI FLAT MODE

cd matrixMultiplication/ make ./matrixMultiplication.exec <elementos>

```
Executar na DDR
numactl --membind=0 ./matrixMultiplication.exec 4096

Executar na MCDRAM
numactl --membind=1 ./matrixMultiplication.exec 4096
```









# **INTEL MODERN CODE:** INTRODUÇÃO À PROGRAMAÇÃO VETORIAL E PARALELA PARA O PROCESSADOR INTEL XEON PHI KNIGHTS LANDING


Matheus S. Serpa, Vinícius G. Pinto, Philippe O. A. Navaux
**Contato:** msserpa@inf.ufrgs.br

INTEL MODERN CODE PARTNER