

## Capítulo

# 4

## Introdução à Vetorização em Arquiteturas Paralelas Híbridas

Silvio Stanzani<sup>1</sup>, Raphael Cóbe<sup>1</sup>, Rogério Iope<sup>1</sup>, Igor Freitas<sup>2</sup>

<sup>1</sup>Núcleo de Computação Científica - Universidade Estadual Paulista

<sup>2</sup>Software and Services Group - Intel Brasil

### *Resumo*

*Técnicas de vetorização representam o recurso mais básico para explorar as possibilidades oferecidas por modernas arquiteturas paralelas, compostas por combinações de recursos incluindo processadores de múltiplos núcleos, memória em múltiplos níveis, aceleradores e coprocessadores. O objetivo deste minicurso é apresentar algumas das características oferecidas por compiladores C e C++, bem como extensões dessas linguagens e recursos da versão 4 do OpenMP que permitem explorar vetorização em tais arquiteturas.*

### 4.1.Introdução

A contínua evolução dos microprocessadores nas últimas décadas proporcionou melhorias significativas no desempenho dos computadores, com o desenvolvimento de microarquiteturas cada vez mais complexas, incluindo técnicas de paralelismo em nível de instrução de máquina, aumento dos níveis de memória *cache*, além de consideráveis avanços na frequência do *clock* interno que rege a execução das microinstruções. No entanto, as condições que permitiram tais melhorias começaram a atingir seus limites por volta de 2005, de modo que hoje em dia não se conseguem grandes avanços apenas com aumentos nas frequências de *clock*, ou melhorias na microarquitetura e/ou na hierarquia de *cache*, ou ainda através de incrementos nas taxas de transferências de dados de/para a memória. Para explorar as novas arquiteturas de modo a se obter

melhorias significativas de desempenho torna-se cada vez mais necessário aplicar técnicas de paralelismo explícito, pois o aumento no desempenho dos computadores atuais tem se baseado no desenvolvimento de arquiteturas computacionais paralelas.

As modernas arquiteturas de computadores podem ser compostas por diversos processadores, que por sua vez dispõem de múltiplos núcleos de processamento, além de um sistema de memória organizado em múltiplos níveis. Mais recentemente, tem se popularizado o desenvolvimento de sistemas computacionais compostos por coprocessadores e aceleradores que dispõem de muitos núcleos e grande capacidade de processamento vetorial, e atuam em conjunto com os processadores principais [1]. Tais arquiteturas compostas por recursos heterogêneos são definidas também como arquiteturas paralelas híbridas.

Arquiteturas paralelas híbridas disponibilizam múltiplos níveis de paralelismo, tais como paralelismo no nível de dados (vetorização), paralelismo no nível de *threads* e paralelismo no nível de processos, que podem ser usados de modo combinado. A vetorização é o nível mais básico de paralelismo que pode ser explorado, e consiste em aplicar uma mesma operação em um conjunto de pares de operandos; as instruções de máquina que permitem tais operações são chamadas de instruções vetoriais [2]. A utilização de tais recursos, devidamente combinada com técnicas de acesso eficiente a sistemas de memória de múltiplos níveis, é essencial para melhorar o desempenho de aplicações em tais arquiteturas.

Atualmente, boa parte dos compiladores e linguagens de programação oferecem suporte para uso de recursos de processamento vetorial, basicamente de três formas. A primeira é por meio de parâmetros de otimização do compilador, que permitem utilizar instruções vetoriais de modo automático. A segunda é utilizando extensões das linguagens de programação, que servem para instruir o compilador quanto ao uso de tais recursos de modo mais preciso, e também para forçar o uso de tais instruções. Tais extensões são disponibilizadas pela linguagem C, C++ e Fortran, estão presentes na versão 4.0 da especificação OpenMP (Open Multi-Processing) [3], e também estão presentes em outros *frameworks*, tais como, OpenCL [4] e OpenACC [5]. A terceira é utilizando bibliotecas ou funções que fornecem acesso direto a instruções vetoriais, nos compiladores do pacote *Intel Parallel Studio* [6] é possível utilizar a biblioteca *Intrinsics* [7], o compilador GCC (**GNU Compiler Collection**) também provê acesso ao uso de instruções vetoriais explicitamente [8].

O objetivo deste minicurso é capacitar os estudantes quanto ao uso das unidades de processamento vetorial e técnicas de acesso eficiente a memória, com ênfase em arquiteturas *multicore* e *manycore* da Intel. Nesse sentido, serão abordadas as extensões disponibilizadas pelas linguagens e pelo padrão OpenMP 4.0, incluindo os parâmetros dos compiladores. A estrutura geral do minicurso é detalhada na Seção 4.1.1.

#### 4.1.1. Estrutura do Minicurso

Este documento está estruturado da seguinte forma: a Seção 4.2 apresenta o conceito de arquitetura paralela híbrida, descrevendo como funciona o sistema de memória e as

unidades de processamento vetorial. A Seção 4.3 descreve um processo para explorar o uso de vetorização usando a ferramenta Intel Advisor. A Seção 4.4 descreve técnicas para melhorar o desempenho do acesso à memória. A Seção 4.5 descreve como explorar paralelismo de dados nas unidades de processamento vetorial por meio de parâmetros de compilação e por meio de extensões da linguagem C e OpenMP 4.0. Finalmente, a Seção 4.6 apresenta algumas considerações finais.

## 4.2. Arquiteturas Paralelas Híbridas

Sistemas computacionais paralelos modernos são constituídos de uma combinação de recursos que incluem processadores com múltiplos núcleos, subsistemas de memória que podem apresentar múltiplos níveis de acesso e subsistemas de entrada e saída [9].

Um tipo de arquitetura computacional que tem se popularizado são as arquiteturas paralelas híbridas [1], que são sistemas computacionais paralelos compostos por recursos heterogêneos, que podem ser coprocessadores e/ou aceleradores (gráficos ou de uso geral). Esses recursos podem acrescentar dezenas ou centenas de elementos de processamento extras, acessíveis ao programador, conforme mostrado na Figura 1.

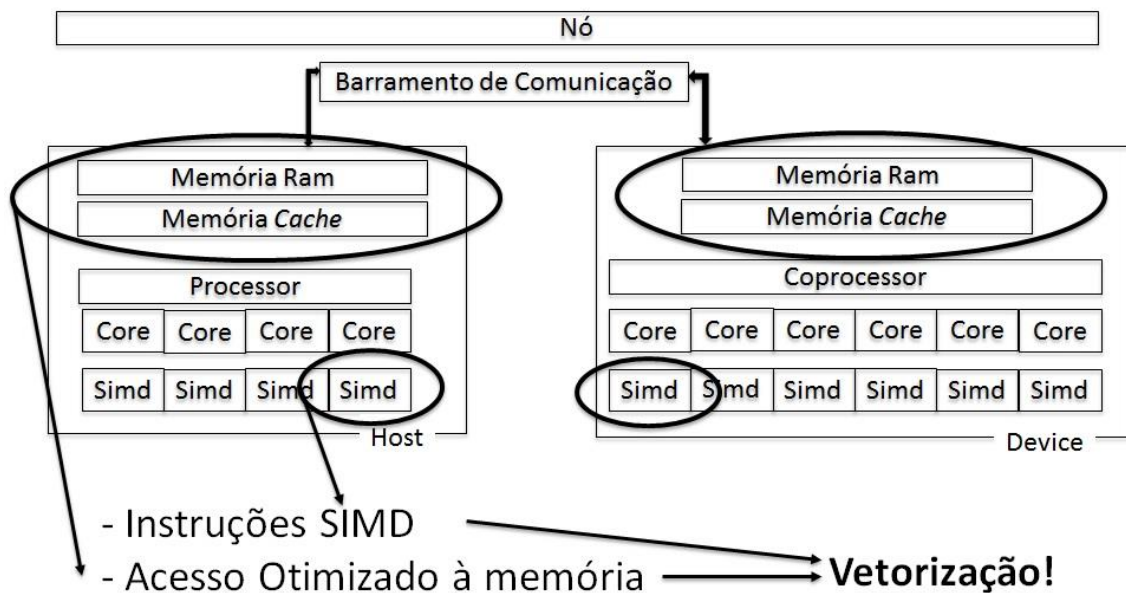


Figura 1. Arquitetura paralela híbrida.

Nas arquiteturas paralelas híbridas dois aspectos principais podem ser considerados quando o objetivo é ganho de desempenho em aplicações. O primeiro é o sistema de memória em múltiplos níveis, que disponibiliza unidades de memória com alto desempenho próximas fisicamente do processador. Essas unidades de memória são

chamadas de memória *cache* [10]. O segundo aspecto são as unidades de processamento vetorial, que permitem explorar o paralelismo de dados, no nível do núcleo de processamento. O conjunto de instruções que permitem o uso dessas unidades de processamento vetorial é chamado SIMD (Single Instruction Multiple Data). Essas instruções possibilitam executar uma mesma operação com diversos pares de operandos de forma paralela [2]. O conjunto de técnicas que exploram esses dois aspectos presentes em um núcleo é chamada vetorização, e é essencial para melhorar o desempenho de aplicações.

A combinação do uso eficiente de sistemas de memória em múltiplos níveis com o uso de unidades de processamento vetorial é essencial para atingir bons níveis de desempenho de vetorização. Nesse sentido, a Seção 4.2.1 descreve o funcionamento do sistema de memória e a Seção 4.2.1 descreve as unidades de processamento vetorial.

#### **4.2.1. Sistema de Memória**

O sistema de memória na maioria dos sistemas computacionais modernos em geral é organizado em múltiplos níveis. Essa organização tem como objetivo permitir explorar aspectos de localidade, que consiste em armazenar fragmentos do programa em diferentes níveis de memória para melhorar o desempenho das aplicações [9]. De modo geral, a hierarquia da memória é dividida nos seguintes níveis:

O sistema de memória das arquiteturas de múltiplos núcleos (*multicore* e *manycore*) da Intel é organizado de acordo com os seguintes níveis:

- Registradores do processador: é a memória interna do processador, que armazena os dados das instruções que serão executadas;
- *Cache*: armazena fragmentos de dados de programas que serão utilizados para a execução das próximas instruções, além de dados temporários frequentemente utilizados pelos programas em execução;
- Memória principal: armazena todos os dados necessários para a execução dos programas em todas as unidades de processamento;
- Memória secundária: são os dispositivos que armazenam dados e programas que podem estar ou não em execução.

Neste documento vamos focar na memória *cache*, que é uma memória especial que apresenta uma velocidade de acesso maior que a da memória principal. Ela funciona como uma área intermediária de armazenamento, utilizada para armazenar conteúdo temporário que poderá ser utilizado pelas próximas instruções que entrarão em execução. Esse tipo de memória apresenta um papel essencial no desempenho da execução das aplicações, pois tem o objetivo de minimizar o impacto da transferência de dados entre o processador e a memória principal [10].

A escolha de quais dados são armazenados na memória *cache* é feita pelo sistema operacional (S.O.), e os critérios para realizar tal escolha, na maioria dos S.O.s, são os seguintes:

- Temporal: define que se um determinado item de dado acabou de ser referenciado, ele provavelmente será referenciado novamente em breve; um exemplo disso são dados usados por todas as iterações de um laço;
- Espacial: define que se um determinado item de dado acabou de ser referenciado, itens de dados próximos a ele também serão referenciados em breve; um exemplo é um laço que realiza a leitura de uma matriz.

Uma métrica de desempenho utilizada para medir a eficiência do S.O. quanto ao uso da memória *cache* é chamada *hit ratio*. Tal métrica é baseada em dois eventos: *cache hit* e *cache miss*. Quando uma instrução em execução faz referência a um item de dado e o encontra na memória *cache*, esse evento é definido como um *hit*; quando o item de dado não é encontrado na *cache*, esse evento é definido como *miss*, conforme mostrado na Figura 2. O *hit ratio* é obtido dividindo a quantidade de eventos do tipo *cache hit* pela quantidade total de operações de busca de dados.

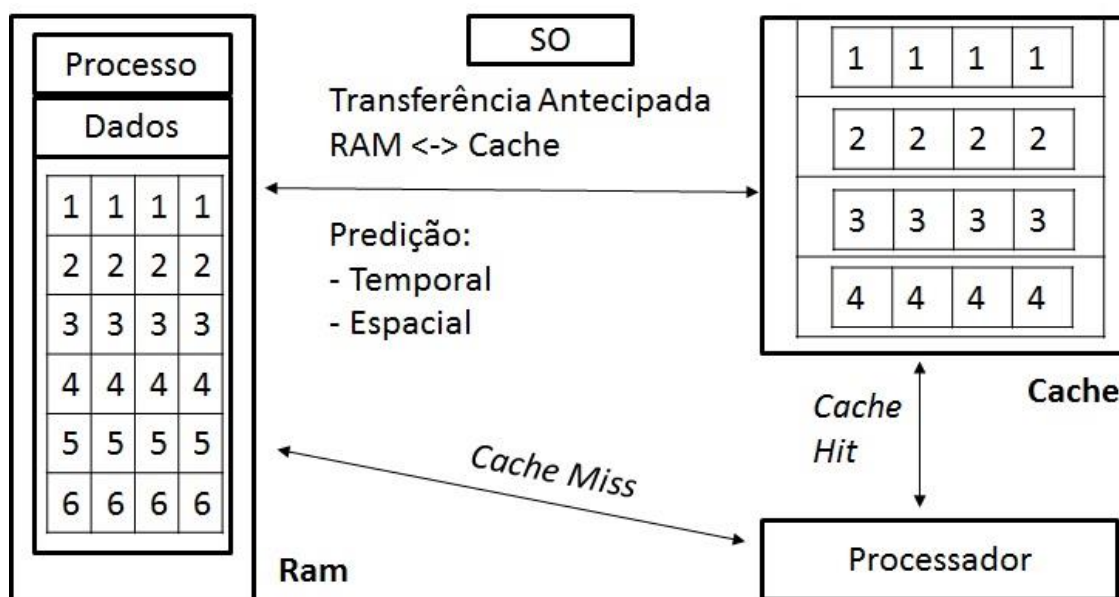


Figura 2. Sistema de memória em múltiplos níveis.

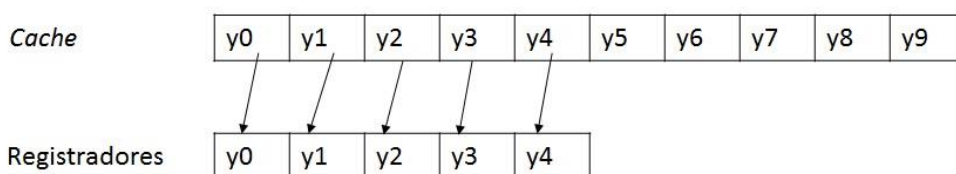
Diversos aspectos da codificação influenciam o *hit ratio* do S.O. durante a execução do programa. Um desses aspectos é o *stride* de um laço, que é o padrão de acesso aos dados em memória, o qual pode ser definido usando três padrões:

- O primeiro é chamado *unit-stride*, e indica que os elementos de uma matriz são referenciados um após o outro.
- O segundo é chamado *constant-stride*, e indica que os elementos de uma matriz, são referenciados em intervalos regulares.

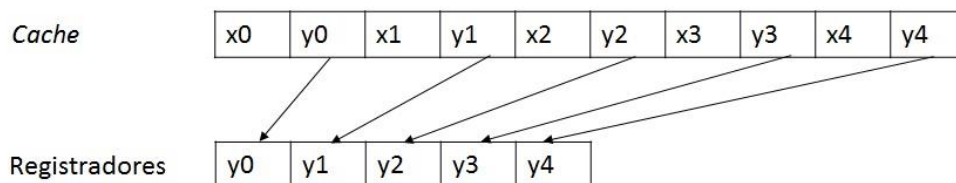
- O terceiro é chamado aleatório, e indica que os dados são referenciados em intervalos desconhecidos.

A Figura 3 ilustra as transferências realizadas entre memória *cache* e registrador da unidade de processamento utilizando os três padrões apresentados. No caso do *unit-stride*, os dados são encontrados sequencialmente na *cache*. O padrão *constant-stride* possui um desempenho inferior ao *unit-stride*, pois parte dos dados carregados na *cache* não são acessados e, portanto, a taxa de *cache hit* diminui. O padrão aleatório apresenta o pior desempenho entre eles pois a predição do S.O. apresenta uma taxa baixa de acerto.

#### **Unit-Stride**



#### **Constant-Stride**



#### **Random-Stride**

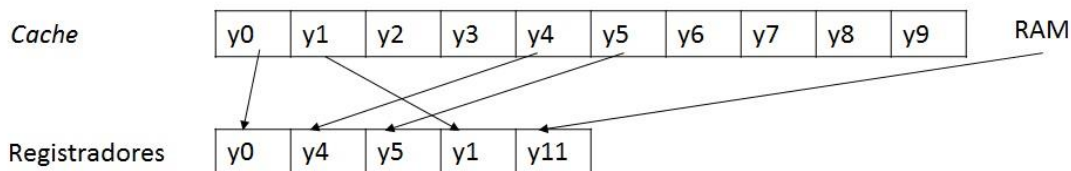


Figura 3. Transferências de dados entre memória *cache* e registradores usando diferentes *Strides*.

A Seção 4.4 apresenta técnicas para definir estruturas de dados que melhoram o *hit ratio* de *cache* do S.O..

### **4.2.2.Unidades de Processamento Vetorial**

A microarquitetura da maioria dos processadores atuais permite executar dois tipos de instruções: escalares ou vetoriais [9]. Instruções escalares aplicam uma operação em um par de operandos, e instruções vetoriais aplicam uma operação em um conjunto de pares de operandos.

A execução de instruções vetoriais é realizada em unidades de processamento vetorial, que possuem um grande número de registradores para armazenar todos os dados que serão utilizados por uma mesma instrução, conforme mostrado na Figura 4.

Nas arquiteturas Intel diversos conjuntos de instruções vetoriais têm sido desenvolvidos, tais como MMX (*Multimedia Extensions*), SSE (*Streaming SIMD Extensions*), AVX (*Advanced Vector Extensions*), AVX-2, AVX512 e IMCI (*Initial Many Core Instructions*).

Instruções Vetoriais (SIMD)								Instruções Escalares	
A7	A6	A5	A4	A3	A2	A1	A0	A	
+								+	
B7	B6	B5	B4	B3	B2	B1	B0	B	
=								=	
A7+B7	A6+B6	A5+B5	A4+B4	A3+B3	A2+B2	A1+B1	A0+B0	A+B	

**Figura 4. Execução de instruções vetoriais e instruções escalares.**

As estruturas de código mais apropriadas para serem executadas em unidades de processamento vetorial são os laços, pois definem uma mesma região de código que é executada para um conjunto de valores. Nesse contexto, um requisito para que um laço seja compilado utilizando instruções vetoriais é que as iterações não possuam dependências entre si. Três estratégias podem ser adotadas para utilizar instruções vetoriais [11]:

- Vetorização automática: é realizada pelo compilador durante o processo de compilação, e consiste em trocar instruções escalares por instruções vetoriais, sempre que houver garantias de que não haverá alterações no resultado final;
- Vetorização guiada: é realizada por meio de extensões das linguagens de programação, que podem ser colocadas pelo desenvolvedor antes de laços específicos para forçar e/ou parametrizar a vetorização desse laço. Nesses casos, o desenvolvedor é responsável por falhas ou erros no resultado final;
- Vetorização explícita: consiste na utilização de recursos da linguagem de programação ou bibliotecas específicas para desenvolver código que será executado nas unidades de processamento vetorial.

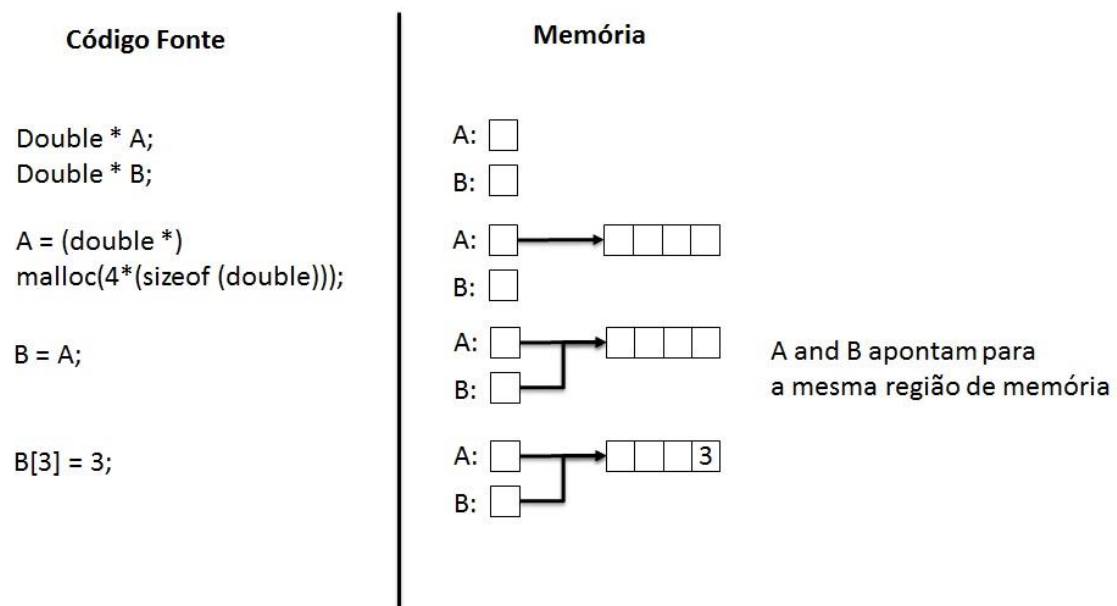
Um exemplo do uso de unidades de processamento vetorial pode ser verificado no laço apresentado no Código 1. Nesse código, a vetorização pode ser aplicada carregando um intervalo de elementos das três matrizes presentes nessa operação, como por exemplo 4 elementos por vez ( $A[0..3]$ ,  $B[0..3]$  e  $C[0..3]$ ), nos registradores vetoriais e aplicar a operação de adição em todos os elementos.

```
for (i=0; i<n; i++)  
    A[i]= B[i]+ C[i];
```

**Código 1. Exemplo de um laço sem dependências entre iterações.**

Alguns dos aspectos que fazem com que a vetorização automática não seja realizada são:

- Dependências assumidas: o compilador avalia para cada laço do código se as iterações são independentes entre si. Nos casos em que o compilador não consegue garantir que as iterações são independentes, o compilador assume que o laço possui dependências;
- Vetorização ineficiente: nos laços em que o compilador consegue garantir que as iterações são independentes, é feita uma análise de estimativa de ganho de desempenho; caso a estimativa de ganho seja baixa o compilador não utiliza instruções vetoriais;
- Laços com número incerto de iterações;
- Ponteiros que potencialmente podem acessar uma mesma região de memória (*aliasing*): essa situação é ilustrada na Figura 5, e pode ocorrer quando um laço está dentro de uma função que manipula ponteiros passados como argumento por exemplo. Nessas situações o compilador não é capaz de garantir que os diferentes ponteiros vão apontar para posições distintas da memória e, portanto, não vetoriza automaticamente.



**Figura 5. Duas variáveis apontando para a mesma região de memória.**



O Código 2 apresenta dois exemplos de laços que não podem ser vetorizados automaticamente. O primeiro laço apresenta uma dependência: a iteração atual atualiza o valor de uma posição da matriz A ( $A[i]$ ), que depende do valor calculado na iteração anterior ( $A[i-1]$ ). No segundo laço os índices dos vetores A e B que serão utilizados na operação são obtidos a partir de valores de outras variáveis; essa referência indireta ao índice da matriz que será atualizado apresenta baixo desempenho quando vetorizado, e em geral o compilador utiliza instruções escalares.

```
for (i=1; i<n; i++)  
    A[i]= A[i-1]+ B[i]*randV;  
  
for (i=1; i<n; i++)  
    A[B[i]]= B[C[i]]+randV;
```

**Código 2.** Exemplos de laços que não são vetorizados automaticamente.

### 4.3. Fluxo Iterativo de Vetorização

Uma metodologia para facilitar a otimização de desempenho considerando diferentes estratégias de paralelização pode ser implementada por meio da utilização de ferramentas de perfilamento [12], que têm como objetivo realizar medições quanto ao tempo de execução de cada linha de código, e quanto à eficiência de uso dos recursos de *hardware*. Tais informações são essenciais para descobrir as oportunidades de paralelismo, permitem avaliar quais estratégias podem ser adotadas para aproveitar tais oportunidades.

A ferramenta de perfilamento Intel Advisor provê suporte à vetorização por meio das seguintes análises [13], [14],[15]:

- *Survey target*: medição do tempo de execução e informação sobre vetorização de cada laço, tais como, conjunto de instruções usado e estimativa de ganho de desempenho;
- *Find trip count*: medição da quantidade de iterações de cada laço;
- *Check dependencies analysis*: verificação quanto à existência de dependências entre iterações de um laço;
- *Check memory access patterns*: análise do padrão de acesso a memória realizado por um laço, utilizando *strides* como parâmetro;

Com base nas análises realizadas pelo Intel Advisor é proposto um fluxo, apresentado na Figura 6, para vetorizar uma aplicação. O objetivo desse fluxo é de modo iterativo identificar as possibilidades de vetorização, aplicar otimizações e avaliar os resultados de ganho de desempenho obtidos.

#### 4.4. Otimização de Acesso à Memória

Um aspecto importante para explorar a vetorização é melhorar o desempenho da transferência de dados entre a memória principal e as unidades de processamento vetorial. Conforme mostrado na Seção 4.2.1, nas arquiteturas *multicore* e *manycore* da Intel o recurso que possibilita melhorar o desempenho dessas transferências é a memória *cache*. Nessa seção será apresentado um modelo para definir estruturas de dados que favoreçam o uso eficiente da memória *cache*.

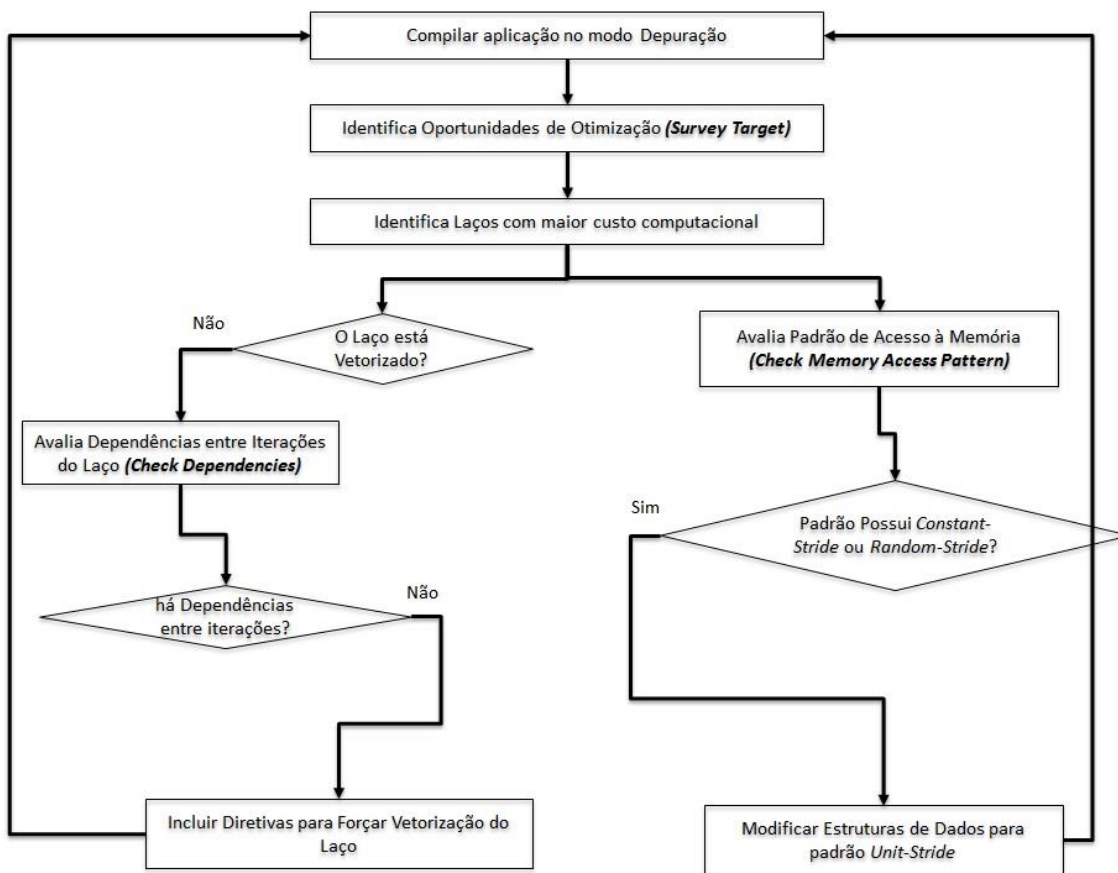


Figura 6. Fluxo iterativo de vetorização usando Intel Advisor.

As estruturas de dados em um programa podem ser declaradas de acordo com dois modelos: AOS ( *Array of Structures* ) ou SOA ( *Structure of Arrays* ). O modelo AOS é definido como uma estrutura declarada como uma matriz composta por variáveis simples. O modelo SOA é definido como uma única estrutura composta por vetores.

A Tabela 1 apresenta a implementação de uma estrutura para representar coordenadas geográficas em três dimensões, utilizando três variáveis do tipo *float* (x,y e z). O código à esquerda mostra a implementação seguindo o modelo AOS, e o código à direita mostra a implementação seguindo o modelo SOA.

<pre>struct coordinate {     float x, y, z; } crd[N];</pre>	<pre>struct coordinate {     float x[N], y[N], z[N]; } crd;</pre>
---	---

**Tabela 1. Duas implementações de uma mesma estrutura de dados usando modelo AOS (esq.) e SOA (dir.).**

O armazenamento dos dados de um vetor de estruturas declarado usando o modelo AOS é feito colocando uma estrutura completa ao lado da outra. Nesse contexto, um laço que manipula dados dessa estrutura segue o padrão *constant-stride*, pois os dados de x, y e z estão intercalados na memória *cache*. No modelo SOA o armazenamento de cada vetor da estrutura é feito em espaços contíguos de memória *cache*, o que favorece que os laços que manipulam dados dessa estrutura sigam o modelo *unit-stride* [16].

O Código 3 a seguir mostra a implementação da estrutura de coordenadas descrita na Tabela 1 seguindo o modelo AOS e SOA, e dois laços que executam o mesmo conjunto de instruções; um deles recebe como entrada a estrutura que representa coordenada implementada de acordo com o modelo AOS, e o outro laço a mesma estrutura implementada de acordo com o modelo SOA.

```
struct coordinate {
    float x, y, z;
} aosobj[60000];

struct coordinate2 {
    float x[60000], y[60000], z[60000];
} soaobj;

...
for(i=0; i<60000; i++) {
    aosobj[i].x=i*j- randV;
    aosobj[i].y=i+j* randV;
    aosobj[i].z=i-j+ randV;
}
for(i=0; i<60000; i++) {
    soaobj.x[i]=i*j- randV;
    soaobj.y[i]=i+j* randV;
    soaobj.z[i]=i-j+ randV;
}



...
```

**Código 3. Fragmento de código que implementa um laço que manipula dados de estruturas AOS e SOA.**

Ao avaliar esse código com o *check memory access patterns* foi verificado que o padrão de acesso feito no primeiro laço (AOS) é *constant stride*, enquanto no segundo laço (SOA) o padrão de acesso é *unit-stride* (Figura 8). O impacto no desempenho pode ser verificado na Figura 7: o laço *unit-stride* apresenta menor tempo de execução e maior ganho estimado de desempenho.

Loops	Vector Issues	Self Time	Total Time	Loop Type	Why No Vectorization?	Vectorized Loops		
						Vec...	Efficiency	Gain Estimate
[loop in __libc_start_main]		0.000s	18.700s	Scalar				
[loop in main at vec.c:30]	1 Data type conversions present	0.000s	18.700s	Scalar	inner loop was al...			
[loop in main at vec.c:33]	2 Data type conversions present	6.919s	6.919s	Vectorized (Body)		SSE2	~82%	1.65x
[loop in main at vec.c:38]	2 High vector register pressure	5.906s	5.906s	Vectorized (Body)		SSE2	~70%	2.80x
[loop in main at vec.c:47]	2 High vector register pressure	5.875s	5.875s	Vectorized Versions		SSE2	~78%	3.14x

Figura 7. Relatório gerado após a análise *survey target*.

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Site Name
 [loop in main at vec.c:33]	No information available	0% / 100% / 0%	All const strides	loop_site_7
 [loop in main at vec.c:38]	No information available	100% / 0% / 0%	All unit strides	loop_site_4

Memory Access Patterns Report

Dependencies Report






ID		Stride	Type	Source	Site Name	Nested Function	Modules	Variations
P1		6	Constant stride	vec.c:34	loop_site_7		libc.so.6; vec	aosobj
<pre>32 randV=randV*0.11; 33 for(i=0; i&lt;60000; i++) { 34   aosobj[i].x=i*j- randV; 35   aosobj[i].y=i+j* randV; 36   aosobj[i].z=i-j+ randV; 37 }</pre>								
P2		6	Constant stride	vec.c:35	loop_site_7		libc.so.6; vec	aosobj
<pre>33 for(i=0; i&lt;60000; i++) { 34   aosobj[i].x=i*j- randV; 35   aosobj[i].y=i+j* randV; 36   aosobj[i].z=i-j+ randV; 37 }</pre>								
P3		6	Constant stride	vec.c:36	loop_site_7		libc.so.6; vec	aosobj
<pre>34   aosobj[i].x=i*j- randV; 35   aosobj[i].y=i+j* randV; 36   aosobj[i].z=i-j+ randV; 37 } 38 for(i=0; i&lt;60000; i++) {</pre>								
P4			Parallel site information	vec.c:33	loop_site_7		vec	
<pre>31 randV=rand(); 32 randV=randV*0.11; 33 for(i=0; i&lt;60000; i++) { 34   aosobj[i].x=i*j- randV; 35   aosobj[i].y=i+j* randV;</pre>								

Figura 8. Relatório mostrando a distribuição de *strides* de cada laço.

## 4.5. Explorando Paralelismo de Dados nas Unidades de Processamento Vetorial

Esta seção descreve alguns recursos para explorar o paralelismo de dados nas unidades de processamento vetorial. A seção 4.5.1 descreve como utilizar vetorização automática nos compiladores Intel e GCC. Na seção 4.5.2 serão mostrados recursos para permitir a vetorização por meio da desambiguação de ponteiro. Na seção 4.5.3 serão apresentadas as extensões presentes nos compiladores Intel e GCC para forçar vetorização e na seção 4.5.4, os recursos da especificação OpenMP 4.0 para prover suporte a vetorização guiada.

### 4.5.1. Vetorização Automática

A vetorização automática é um dos passos de otimização que são realizados pela maioria dos compiladores C, C++ e Fortran. Esta seção descreve como aplicar essa otimização utilizando compiladores Intel e GCC.

Nos compiladores citados, Intel e GCC, esse passo de otimização é configurado utilizando o parâmetro **-O<nível>**. O parâmetro nível pode variar de 0 a 3 e define quais os tipos de otimizações que devem ser aplicadas.

A Figura 9 mostra como compilar um programa, apresentado no Código 4, ativando a opção de vetorização automática no compilador Intel e GCC usando opção **-O3**.

```
icc autovec.c -O3 -o autovec
gcc autovec.c -O3 -o autovecGCC
```

Figura 9. Comando para compilação de um programa em C usando icc e gcc.

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

int main(){
    const int n=90000;
    int i, j, randV;
    int A[n];
    int B[n];

    for (j=0; j<45000; j++) {
        srand(time(NULL));
        randV=rand();
```

```
for (i=0; i<n; i++)  
    A[i]+=B[i]*randV;  
  
}  
  
}
```

**Código 4.** Exemplo de código que pode ser vetorizado automaticamente.

É possível avaliar quais regiões de código foram vetorizadas por meio do relatório de otimização, que indica quais regiões de código foram vetorizadas e quais não foram vetorizadas. Para os laços que não foram vetorizados, é indicado o motivo por não terem sido vetorizados.

No GCC é necessário utilizar o parâmetro `-ftree-vectorizer-verbose=1` para gerar o relatório de otimização. A Figura 10 mostra como compilar o código gerando o relatório de otimização usando GCC, e a Figura 11 apresenta um fragmento do relatório de otimização gerado a partir da compilação do programa descrito no Código 4.

```
gcc autovec.c -O3 -ftree-vectorizer-verbose=1 -o autovecGCC
```

**Figura 10.** Comando para compilação usando GCC gerando relatório de otimização.

```
Analyzing loop at autovec.c:10  
  
autovec.c:10: note: not vectorized: loop contains function calls or data references that  
cannot be analyzed  
autovec.c:10: note: bad data references.  
Analyzing loop at autovec.c:14  
  
autovec.c:14: note: Unknown misalignment, is_packed = 0  
autovec.c:14: note: Unknown misalignment, is_packed = 0  
autovec.c:14: note: virtual phi. skip.  
  
Vectorizing loop at autovec.c:14  
  
autovec.c:14: note: virtual phi. skip.  
autovec.c:4: note: vectorized 1 loops in function.
```

**Figura 11.** Fragmento do relatório de vetorização gerado pelo GCC.

No compilador Intel é necessário utilizar o parâmetro `-vec-report5` para gerar o relatório de otimização. A Figura 12 mostra como compilar o código gerando o relatório de otimização usando `icc`, e a Figura 13 apresenta um fragmento do relatório de otimização gerado a partir da compilação do programa descrito no Código 4.

```
icc autovec.c -o autovec -O3 -vec-report5
cat autovec.optrpt
```

**Figura 12. Comando para compilação usando ICC gerando relatório de otimização.**

```
LOOP BEGIN at autovec.c(10,5)
  remark #15542: loop was not vectorized: inner loop was already vectorized

  LOOP BEGIN at autovec.c(14,9)
    remark #15388: vectorization support: reference B has aligned access [
autovec.c(15,18) ]
    remark #15388: vectorization support: reference A has aligned access [
autovec.c(15,13) ]
    remark #15305: vectorization support: vector length 4
    remark #15399: vectorization support: unroll factor set to 4
    remark #15309: vectorization support: normalized vectorization overhead 2.667
    remark #15300: LOOP WAS VECTORIZED
    remark #15449: unmasked aligned unit stride stores: 2
    remark #15475: --- begin vector loop cost summary ---
    remark #15476: scalar loop cost: 4
    remark #15477: vector loop cost: 1.500
    remark #15478: estimated potential speedup: 2.660
    remark #15488: --- end vector loop cost summary ---
  LOOP END
LOOP END
```

**Figura 13. Fragmento do relatório de otimização gerado pelo icc.**

#### 4.5.2. Desambiguação de Ponteiro

Um dos fatores que impede que o compilador realize a vetorização automática é a possibilidade de que ponteiros passados como parâmetros de função apontem para a

mesma região de memória. Um exemplo dessa ambiguidade pode ser observado no Código 5; nesse código a função **add\_floats** recebe cinco parâmetros como ponteiro. Nesse contexto o risco de ambiguidade ocorre, pois a função **add\_floats** pode ser chamada de outros arquivos que são compilados separadamente; nesses casos o compilador identifica que há um risco de ambiguidade.

```
#include "func.h"

void add_floats(double * a, double * b, double * c, double * d, double * e, int n){
    int i,j;

    for (j=1; j<n-1; j++){
        for (i=1; i<n-1; i++){
            a[i] =e[j-1] + (c[i] * d[j+1]) ;
        }
    }
}
```

**Código 5. Exemplo de função com possível ambiguidade de ponteiro.**

Quando o desenvolvedor consegue garantir que não há risco de ambiguidades, é possível instruir o compilador a ignorar ambiguidades de ponteiros usando dois mecanismos: **fargument-noalias** e **restrict**.

O argumento **fargument-noalias** indica que nenhum ponteiro passado como parâmetro de nenhuma função possui ambiguidade. Um exemplo do uso desse argumento é mostrado na Figura 14.

```
icc func.c -c -o func.o -O3 -vec-report5 -fargument-noalias
```

**Figura 14. Compilação usando parâmetro fargument-noalias.**

O argumento **restrict** também pode ser utilizado para indicar ao compilador que não há ambiguidades, porém essa indicação deve ser feita para cada variável de cada função. Nesse sentido, é necessário passar o argumento **-restrict** para o compilador, e para cada variável incluir a palavra reservada **restrict** entre **\*** e o nome da variável. Um exemplo do uso desse parâmetro é mostrado no Código 6. A compilação deve incluir o parâmetro **-restrict**.



```
...
void add_floats(double * restrict a, double * restrict b, double * restrict c, double *
restrict d, double * restrict e, int n){
...

```

**Código 6. Exemplo de eliminação de ambiguidades usando parâmetro restrict.**

### 4.5.3. Extensões de Compilador para Vetorização

A maioria dos compiladores C, C++ e Fortran disponibiliza extensões que proveem suporte à vetorização, que também são denominadas diretivas de vetorização. Tais diretivas são utilizadas para forçar o compilador a parametrizar a vetorização de um laço. Nessa seção focaremos em três diretivas: **ivdep**, **simd** e **\_\_declspec(vector)**.

A diretiva **ivdep** tem como objetivo instruir o compilador a ignorar dependências de vetor assumidas. O Código 7 mostra um exemplo de uso dessa diretiva.

Sintaxe:

```
#pragma ivdep
Declaração de laço
```

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

**Código 7. Exemplo de uso da diretiva ivdep.**

A diretiva **simd** é similar à diretiva **ivdep**, porém implementa métodos mais agressivos de otimização e possui argumentos para controlar a vetorização. Um exemplo do uso dessa diretiva é apresentado no Código 8.

Sintaxe:

```
#pragma simd [Cláusulas]
Declaração de laço
```

Algumas cláusulas da diretiva **simd** são descritas a seguir:

- **vectorlength(nl[,...])** : Assume que a vetorização é segura para um certo tamanho de vetor;

- `vectorlengthfor(dtype)` : Assume que a vetorização é segura para um certo tipo de dados;
- `linear (variável[:passo-linear],...)` : define que a variável é incrementada a cada passo do laço, e que o valor do incremento atribuído à variável a cada passo é igual ao valor passado no parâmetro `passo-linear`;
- `[no]assert`: gera mensagens de avisos ou de erros quando a vetorização falha.

```
#pragma simd linear(m:1)
for (i=0; i<20; i++) {
    a[i] = a[i]+m;
    m++;
}
```

**Código 8. Exemplo de uso da diretiva SIMD.**

A diretiva **`__declspec(vector)`** tem como objetivo instruir o compilador a utilizar instruções vetoriais para compilar uma função, que é chamada a partir do corpo de um laço. É recomendado utilizar essa diretiva em funções com poucas linhas de código. O Código 9 mostra um exemplo de uso dessa diretiva.

Sintaxe:

```
__declspec(vector)
```

*Definição ou declaração de função*

```
__declspec(vector)
void v_add2(float c, float a, float b)
{
    c = a + b;
}

void v_addMain(float *c, float *a, float *b, int N)
{
    int i;
    for (i = 0; i < N; i++)
        v_add2(c[i], b[i], a[i]);
}
```

**Código 9. Exemplo de uso de uma diretiva para prover suporte à vetorização de função.**

#### 4.5.4. Diretivas do OpenMP 4 para Vetorização

A especificação do OpenMP a partir da versão 4.0 passou a disponibilizar as seguintes diretivas para prover suporte à vetorização: **omp simd** e **omp declare simd**.

A diretiva **omp simd** define que o laço marcado deve ser compilado usando instruções vetoriais, e funciona de modo similar a diretiva **pragma simd** apresentada na Seção 4.5.3.

Sintaxe:

```
#pragma omp simd [Cláusulas ]  
Laço
```

A seguir são descritas as cláusulas que podem ser usadas em conjunto com as diretiva **omp simd**:

- **safelen (tamanho)** : o parâmetro tamanho define o número máximo de iterações do laço que podem ser executadas concorrentemente, sem quebrar uma dependência entre as iterações;
- **aligned(variável[:alinhamento])** : define que a variável foi previamente alinhada em memória, e o parâmetro alinhamento define o tamanho do bloco de alinhamento;
- **collapse (n)** : especifica o número de laços aninhados (parâmetro **n**) que serão agrupados em um único laço.

Além dessas, podem ser utilizadas também as cláusulas descritas na Seção 4.5.3. Um exemplo do uso dessa diretiva é mostrado no Código 10.

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

**Código 10. Exemplo de uso da diretiva omp simd.**

A diretiva **omp declare simd** possui o mesmo objetivo da diretiva **\_\_declspec(vector)** descrita na Seção 4.5.3, porém permite utilizar cláusulas para controlar o processo de vetorização.

Sintaxe:

```
#pragma omp declare simd [Cláusulas]
```

Definição ou declaração de função

A seguir são descritas algumas cláusulas que podem ser usadas em conjunto com a diretiva **omp declare simd**:

- **uniform**(variável): define que o valor da variável se mantém constante para todas as chamadas concorrentes da função;
- **simdlen**(tamanho): O parâmetro tamanho define a quantidade máxima de elementos de cada argumento da função, que podem ser carregados por instruções SIMD;
- **inbranch**: define que a função pode ser chamada a partir de estruturas condicionais no laço;
- **notinbranch**: define que a função nunca é chamada a partir de estruturas condicionais no laço.

Além dessas, podem ser utilizadas também as cláusulas descritas na diretiva **omp simd**: *aligned*, *simdlen*, *linear*, *reduction* e *uniform*.

O Código 11 mostra como utilizar essa diretiva em um código que realizar interpolação de matrizes [17, p. 22].

```
#pragma omp declare simd
```

```
int FindPosition(double x) {  
    return (int)(log(exp(x*steps)));  
}
```

```
#pragma omp declare simd uniform (vals)
```

```
double Interpolate(double x, const point* vals)  
{  
    int ind = FindPosition(x);  
    ...  
    return res;  
}  
int main ( int argc , char argv [] )  
{  
    ...
```

```

for ( i=0; i <ARRAY_SIZE;++ i ) {
    dst[i] = Interpolate( src[i], vals ) ;
}
...
}

```

**Código 11. Exemplo de vetorização de função usando OpenMP 4.0.**

#### 4.5.5. Avaliação de Dependências entre Iterações de um Laço

Um pré-requisito para utilizar diretivas que forçam a vetorização de laços, é garantir que os laços não apresentem dependências entre as iterações. Uma forma simples de identificar se um laço possui tais dependências é usando a análise *check dependencies* da ferramenta Intel Advisor. Um exemplo do uso dessa análise para suporte à vetorização será mostrado usando como exemplo o programa descrito no Código 12, que apresenta a versão serial de um código que realiza a multiplicação entre duas matrizes bidimensionais (A e B) e armazena o resultado em uma terceira matriz C.

```

void multiply(int msize, int tid, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE
c[][NUM], TYPE t[][NUM])
{
    int i,j,k;
    for(i=0; i<msize; i++) {
        for(k=0; k<msize; k++) {
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}

```

**Código 12. Código sequencial para multiplicação de matrizes.**

O resultado da análise *survey target* dos laços descritos no Código 12 indica que os três laços estão sendo executados de modo escalar. Nesse sentido, o laço mais interno foi selecionado para ser avaliado pelo *check dependencies analysis*. A análise confirma que não há dependências entre as iterações; isso indica que é seguro vetorizar esse laço.

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Site Name
🕒 [loop in multiply0 at multiply.c:1..	No information available	50% / 50% / 0%	Mixed strides	loop_site_42
🕒 [loop in multiply3 at multiply.c:2..	🟢 No dependencies found	No information available	No information available	loop_site_34

Memory Access Patterns Report	Dependencies Report	💡 Recommendations
-------------------------------	---------------------	-------------------

Problems and Messages						
ID	Type	Site Name	Sources	Modules	State	
P1	Parallel site information	loop_site_34	multiply.c	matrix.icc	✓ Not a problem	

Parallel site information: Code Locations							
ID	Instruction Address	Description	Source	Function	Variable references	Module	State
[-] X1	0x40302e	Parallel site	multiply.c:228	multiply3		matrix.icc	✓ Not a problem
	<pre> 226     for(i=0; i&lt;msize; i++) { 227         for(k=0; k&lt;msize; k++) { 228             for(j=0; j&lt;msize; j++) { 229                 c[i][j] = c[i][j] + a[i][k] * b[k][j]; 230             } </pre>						

**Figura 15. Resultado da análise de dependências entre iterações de laços.**

A diretiva **pragma simd** foi colocada no laço mais interno conforme mostrado no Código 13, e a análise *survey target* foi executada novamente. O resultado indica que o laço foi vetorizado pelo compilador.

```

void multiply(int msize, int tid, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE
c[][NUM], TYPE t[][NUM])
{
    int i,j,k;
    for(i=0; i<msize; i++) {
        for(k=0; k<msize; k++) {
            #pragma simd
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}

```

**Código 13. Código vetorizado para multiplicação de matrizes.**

## 4.6. Conclusões

Arquiteturas paralelas híbridas disponibilizam múltiplos níveis de paralelismo que podem ser explorados de modo combinado. Os recursos de processamento vetorial representam o nível mais básico de paralelismo. As técnicas de vetorização representam o conjunto de mecanismos disponibilizados pelos compiladores, extensões das linguagens e até mesmo bibliotecas especializadas como a biblioteca Intel *Intrinsics*[7].

Neste minicurso foram apresentados, de forma introdutória – com exemplos de códigos fontes e aplicações reais - recursos básicos para explorar a vetorização automática oferecida pela maioria dos compiladores C, C+ e Fortran, bem como extensões da linguagem C e funcionalidades do OpenMP para explorar vetorização guiada.

## Referências

- [1] A. Heinecke, M. Klemm, and H.-J. Bungartz, “From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture,” *Comput. Sci. Eng.*, vol. 14, no. 2, pp. 78–83, 2012.
- [2] G. E. Blelloch, *Vector Models for Data-parallel Computing*. Cambridge, MA, USA: MIT Press, 1990.
- [3] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 4.0*. 2013.
- [4] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *IEEE Test*, vol. 12, no. 3, pp. 66–73, May 2010.
- [5] S. Wienke, P. Springer, C. Terboven, and D. an Mey, “OpenACC: First Experiences with Real-world Applications,” in *Proceedings of the 18th International Conference on Parallel Processing*, Berlin, Heidelberg, 2012, pp. 859–870.
- [6] “Intel® Parallel Studio XE | Intel® Software.” [Online]. Available: <https://software.intel.com/en-us/intel-parallel-studio-xe>. [Accessed: 30-Sep-2016].
- [7] “Intel Intrinsics Guide.” [Online]. Available: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/#!=undefined>. [Accessed: 16-Sep-2016].
- [8] “Vector Extensions - Using the GNU Compiler Collection (GCC).” [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/Vector-Extensions.html#Vector-Extensions>. [Accessed: 30-Sep-2016].

- [9] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.
- [10] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 1 edition. Boca Raton, FL: CRC Press, 2010.
- [11] A. Vladimirov, R. Asai, and V. Karpusenko, *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*, 2nd edition. Colfax International, 2015.
- [12] T. Janjusic and K. Kavi, “Hardware and Application Profiling Tools,” in *Advances in Computers*, vol. 92, Elsevier, 2014, pp. 105–160.
- [13] “Intel® Advisor | Intel® Software.” [Online]. Available: <https://software.intel.com/en-us/intel-advisor-xe>. [Accessed: 16-Sep-2016].
- [14] “Get a Helping Hand from the Vectorization Advisor | Intel® Software.” [Online]. Available: <https://software.intel.com/articles/get-a-helping-hand-from-the-vectorization-advisor>. [Accessed: 16-Sep-2016].
- [15] “Guided Code Vectorization with Intel® Advisor XE - Colfax Research.” [Online]. Available: <http://colfaxresearch.com/guided-code-vectorization-with-intel-advisor-xe/>. [Accessed: 16-Sep-2016].
- [16] “Memory Layout Transformations | Intel® Software.” [Online]. Available: <https://software.intel.com/en-us/articles/memory-layout-transformations>. [Accessed: 13-Sep-2016].
- [17] G. M. Raskulinec and E. Fiksman, “Chapter 22 - SIMD Functions Via OpenMP,” in *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, vol. 2, J. Reinders and J. Jeffers, Eds. Boston, MA, USA: Morgan Kaufmann, 2015, pp. 421–440.