OpenMP: Vetorização

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente

Programar para Alto Desempenho

- Recursos já disponíveis impactam o desempenho

- Exemplo para o Cálculo do Pi
- Exemplo de operações aritméticas sobre dois vetores
  - Por que estes desempenhos foram obtidos?
  - Quais recursos estão disponíveis?
  - Precisamos conhecer:
    - arquitetura,
    - compilador e
    - modelo de programação.

Processadores (relembrando)

- Processadores SIMD na Taxonomia de Flynn
  - Proposta dos supercomputadores vetoriais (70´s)
  - A mesma instrução processando blocos de dados
  - Paralelismo de dados usualmente sobre vetores

- Contraste às instruções escalares
  - Processam sequencialmente itens de dados únicos
    - Ok, podemos ter paralelismo de baixo nível aqui também, mas nosso foco agora é outro...

- Termos usuais: vetorização ou apenas SIMD
  - Há forte interesse em desempenho com portabilidade

Extensões nos Processadores (relembrando)

- Alguns exemplos da Intel

Instruction Set	Name	Functionality	Year
MMX	Multimedia Extensions	64 bit MMX for packed integers	1997
SSE	Streaming SIMD Extensions	128 bit XMM for floating point.	1999
SSE2	Steaming SIMD Extensions 2	XMM supports doubles and integers	2001
SSE3	Streaming SIMD Extensions 3	Horizontal operations added	2004
SSSE3	Supplemental SSE3	Horizontal and data movement	2006
SSE4.1	Streaming SIMD Extensions 4.1	Extra functionality	2007
SSE4.2	Streaming SIMD Extensions 4.2	Vector string instructions	2008
AVX	Advanced Vector Extensions	256 bit YMM for floating point.	2011
FMA	Fused Multiply Add	Fused multiply add instructions	2011
AVX2	Advanced Vector Extensions 2	YMM supports packed integers	2013
AVX512	Advanced Vector Extensions 512	512 bit ZMM registers	2016

Table 1. Available x86 vector instruction sets

Huber et al. (2017)

- Processador que estou usando (um velho guerreiro!)
  - Processador Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz, 2201 Mhz, 4 Núcleo(s), 8 Processador(es) Lógico(s)
    - AVX 256 bits, 16 registradores YMM

Extensões nos Processadores (relembrando)

- Como explorar esses recursos?
  - *Assembly*, Compilador ou outros modelos de programação
    - Programação *Assembly* é mais custosa e tem problema com portabilidade
    - Compiladores, sozinhos, têm dificuldades de encontrar boas vetorizações

- OMP oferece suporte às instruções SIMD, com **portabilidade** de código
  - Permite indicar onde usar o paralelismo SIMD, o que não é trivial com *Assembly*
  - SIMD é fortemente integrada ao modelo de *thread* do OMP
    - o que traz paralelismo em múltiplos níveis

Extensões nos Processadores (relembrando)

- Instruções SIMD
  - Usam registradores (vetoriais) SIMD com múltiplos elementos
  - Registradores determinam o comprimento do vetor (*vector length*):
    - Determinam quantos dados escalares podem ser operados em paralelo por uma instrução SIMD

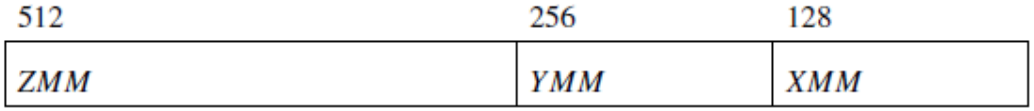
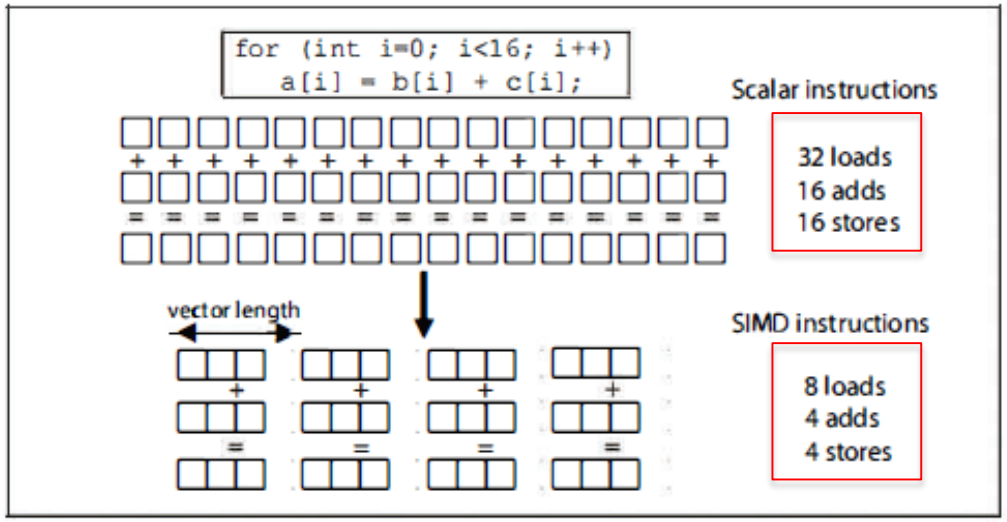


Figure 1. Layout of x86 vector registers

Huber et al. (2017)

  - Tempos das instruções SIMD são tão rápidos quanto as respectivas instruções escalares, mas operam sobre dados diferentes em paralelo



No exemplo temos:

Regs de 128 bits (ou 16 Bytes)

Dados de 04 Bytes

Processamos 04 pares por iter

(*vector length* == 04)

(Pas et al., 2017)

Extensões nos Processadores (relembrando)

- Reforçando o conceito de comprimento do vetor (***vector lenght***)
  - Número de elementos que podem ser usados em um registrador vetorial

Register	long double	double	float	long	int	short	char
64 bit MM	–	–	–	1	2	4	8
128 bit XMM	–	2	4	2	4	8	16
256 bit YMM	–	4	8	4	8	16	32
512 bit ZMM	–	8	16	8	16	32	64

Table 2. Vector lengths for different vector registers and data types

Huber et al. (2017)

Paralelismo SIMD

- Compilador deve identificar onde otimizar o código com segurança
- Desafios para automação desse processo
  - Informações imprecisas sobre a dependência de dados
  - *Layout* de dados e alinhamento
  - Execução condicional
  - Empacotamento e desempacotamento de dados escalares em vetores (*gathering* e *scattering*)
  - Chamadas para funções
  - Número de iterações que nem sempre são múltiplos de comprimento do vetor

- Desafios em C/C++, por exemplo:
  - diferentes ponteiros para a mesma posição de memória causam dependências implícitas

OpenMP & vetorização

- Os primeiros passos na vetorização com OMP:
  - ***Construtor simd*** marca laços *for* (C) como loops SIMD
    - Iterações *for* executadas concorrentemente por **uma *thread***
  - Loop de comprimento ***n*** possui iterações lógicas 0 a n-1, considerando sua execução sequencial
  - SIMD *chunk* (bloco): conjunto de iterações executadas concorrentemente pela instrução SIMD
    - Comprimento do vetor (*vector length*) é o número de iterações do bloco SIMD
  - SIMD *lane*: mecanismo que uma instrução SIMD usa para processar um elemento dos dados

OMP & vetorização – Construtor SIMD

#pragma omp simd [clause[,[clause]...] new-line
for-loops

Cláusulas: *private (list)*
lastprivate (list)
reduction (reduction-identifier: list)
collapse (n)
simdlen(length)
safelen(length)
linear(list[:linear-step])

- Quando uma *thread* OMP encontra um ***construtor simd***, as iterações do loop associado ao construtor podem ser executadas concorrentemente usando as SIMD *lanes* que estão disponíveis para a *thread*

- Aplicam-se as mesmas restrições do ***construtor for***

OMP & vetorização – Construtor SIMD

- Vetores devem ter ponteiros para diferentes áreas de memória. Resultado indefinido se usar *aliases*.
- Variáveis: ***i*** é privada a cada SIMD *lane* e ***a[], b[]*** e ***c[]*** são compartilhadas.
- O vector *length* é escolhido pelo compilador em função da arquitetura

```
1 void simd_loop(double *a, double *b, double *c, int n)
2 {
3   int i;
4
5   #pragma omp simd
6   for (i=0; i<n; i++)
7      a[i] = b[i] + c[i];
8 }
```

(Pas et al., 2017)

OMP & vetorização – Construtor SIMD

- Variáveis ***t1*** e ***t2*** são privadas
  - Não são iniciadas e não deve haver suposições sobre seu valor inicial
  - Conteúdos das variáveis privadas não são acessados após a finalização do bloco do ***construtor simd***

- Para o ***construtor simd***, ***i*** é ***lastprivate***

```
 1 void simd_loop_private(double *a, double *b, double *c, int n)
 2 {
 3   int i;
 4   double t1, t2;
 5
 6   #pragma omp simd private(t1, t2)
 7   for (i=0; i<n; i++)
 8   {
 9     t1 = func1(b[i], c[i]);
10     t2 = func2(b[i], c[i]);
11     a[i] = t1 + t2;
12   }
13 }
```

(Pas et al., 2017)

OMP & vetorização – Construtor SIMD

- ***Reduction*** tem o mesmo comportamento já previsto para outros construtores
- ***Collapse*** (idem) O número em ***collapse*** determina quantos loops que são unidos em um único espaço de iteração
  - Todas as variáveis das iterações loops com ***collapse*** são ***lastprivate***
  - ***Collapse*** aumenta a complexidade para gerar o código SIMD (usar com cautela)

```
1 void simd_loop_collapse(double *r, double *b, double *c,
2                         int n, int m)
3 {
4   int i, j;
5   double t1;
6
7   t1 = 0.0;
8   #pragma omp simd reduction(+:t1) collapse(2)
9   for (i = 0; i<n; i++)
10      for (j = 0; j<m; j++)
11          t1 += func1(b[i], c[j]);
12  *r = t1;
13 }
```

(Pas et al., 2017)

OMP & vetorização – Construtor SIMD

- Cláusula ***simdlen***
  - O valor desta cláusula sugere ao compilador o número de iterações para executar concorrentemente, afetando o *vector length*
    - Compilador é livre para usar o *vector length* que desejar
  - ***Simdlen*** permite guiar o compilador, supondo que o programador pode ter mais informações que apenas o código
  - No exemplo, 32 Bytes equivalem a 256 bits (2^5Bytes * 2^3 bits)
    - Está sugerindo um *vector length* de 8 posições (32 Bytes / 4 bytes do *sizeof()*)

```
1 unsigned int F(unsigned int *x, int n, unsigned int mask)
2 {
3    #pragma omp simd simdlen(32/sizeof(unsigned int))
4    for (int i=0; i<n; i++) {
5       x[i] &= mask;
6    } // End of simd region
7 }
```

(Pas et al., 2017)

OMP & vetorização – Construtor SIMD

- Cláusula ***safelen***
  - Limita o *vector length* no ***construtor simd***
  - Útil para dependências entre loops
    - uma iteração depende de iterações prévias
- No primeiro exemplo abaixo o valor de *safelen* seria de 08 iterações

```
1 void dep_loop(float *a, float c, int n)
2 {
3   for (int i=8; i<n; i++)
4     a[i] = a[i-8] * c;
5 }
```

(Pas et al., 2017)

- *Com safelen* o *vector length* ainda é determinado pelo compilador, mas não ultrapassaria o limite determinado por *safelen*
- *safelen* permite resultados corretos; *simdlen* é uma preferência

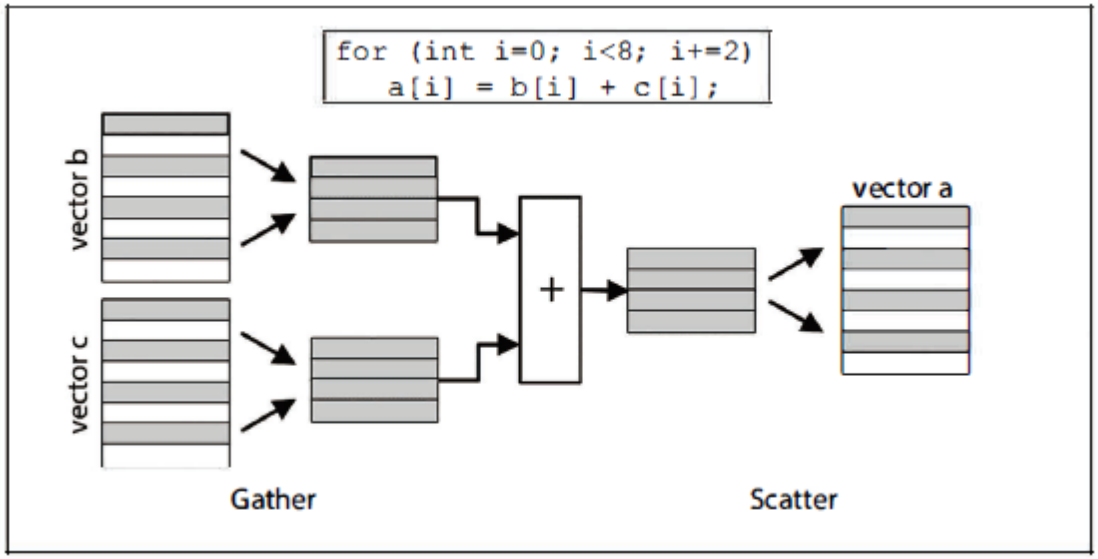
```
1 void simd_loop_safelen(double *a, double *b, double *c, int n,
2                         int offset)
3 {
4   int i;
5   #pragma omp simd safelen(16)
6   for (i=offset; i<n; i++)
7     a[i] = b[i-offset] + c[i];
8 }
```

(Pas et al., 2017)

OMP & vetorização – Construtor SIMD

- Cláusula ***linear***:
  - Indica comportamento linear de uma variável privada em um loop
  - Compilador pode determinar a maneira mais eficiente para empacotar e desempacotar dados nos vetores
  - Determina como os dados escalares são acessados pelas SIMD *lanes*
- Cláusula linear pode ter:
  - Acesso consecutivo (*stride == 1*)
    - é o melhor caso
  - Acesso com deslocamento regular (*stride > 1*)
    - considera um deslocamento fixo entre os dados
  - Acessos com deslocamentos irregulares
    - requerem que dados escalares sejam acessados um-a-um
      - (des)montar o vetor de dados
- Mais fácil de ver a utilidade dela com a diretiva ***declare simd***
  - especifica comportamento linear de parâmetros de funções
  - esta análise é complicada para o compilador, pela necessidade de verificar internamente as funções

OMP & vetorização – Construtor SIMD



(Pas et al., 2017)

```
1 void simd_loop_linear(double *a, double *b, double *c, int n,
2                        int offset)
3 {
4   int i, j = 0;
5
6   #pragma omp simd linear(j:1)
7   for (i=offset; i<n; i+=2)
8     a[i] = b[j++] + c[i];
9 }
```

(Pas et al., 2017)

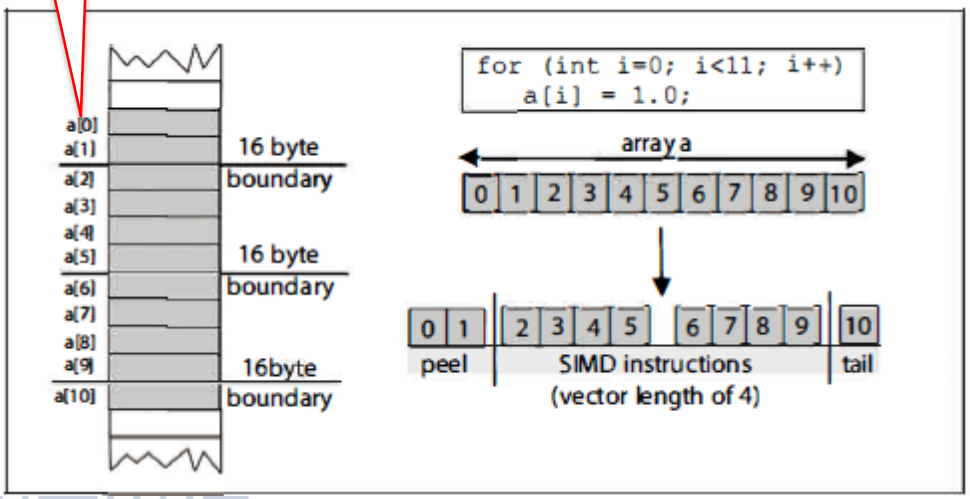
OMP & vetorização – Construtor SIMD

- Alinhamento de vetores são necessários
  - Dados não alinhados na memória, em relação ao tamanho dos elementos em bytes, causam perda de desempenho
    - *Loads* e *stores* acessam blocos de dados na memória para otimizar uso de recursos como barramentos, caches, registradores, ...
    - Alocações dos dados na memória podem (devem) considerar o alinhamento dos dados
    - Evitam perdas de desempenho, porém, podem gerar fragmentação de memória, aumentando seu consumo
- Por isso, início dos vetores alinham-se aos limites dos registradores
  - ***Loop peeling*** antecipa primeiras iterações para fora do loop e as demais ocorrem em paralelo, em blocos de dados que são múltiplos do *vector length*
  - Final do vetor (*tail*) é processado depois, se necessário
  - Pré e pós processamento tornam execução mais lenta
    - mas são necessários
  - Necessidade comumente identificada *on-the-fly*
  - Cláusula ***aligned*** no OpenMP indica o alinhamento ao compilador

OMP & vetorização – Construtor SIMD

- Cláusula ***aligned***
  - Alinhamento de 16 bytes (128 bits) com um vetor de *floats* de 4 Bytes
    - Informa ao compilador que o alinhamento de ***x*** deve ocorrer em 128 bits
    - Facilita operações de *load / store* para carga e leitura dos registradores

a[0] não inicia em um byte múltiplo de 16, mas sim em um múltiplo de 08.



(Pas et al., 2017)

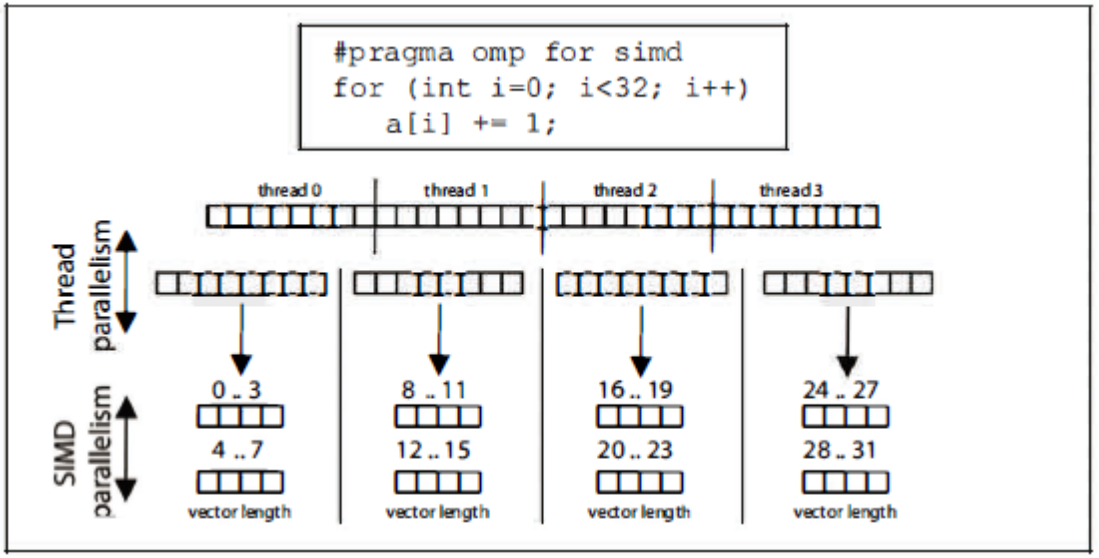
```
1 void f_aligned(float *x, float scale, int n)
2 {
3   int i;
4
5   #pragma omp simd aligned(x:16)
6   for (i=0; i<n; i++)
7     x[i] = x[i]*scale;
8 }
```

(Pas et al., 2017)

OMP & vetorização – Composição for simd

#pragma omp for simd [clause[[,] clause] ...] new-line

- Permite dois níveis de paralelismo pela composição dos construtores
  - *threads* e vetorização
- Blocos de iterações são distribuídos para *threads* disponíveis
  - usa o escalonamento indicado para o ***construtor for***
- Depois o bloco de iterações é distribuído às SIMD *lanes* da *thread*
  - Usando o ***construtor simd***



(Pas et al., 2017)

OMP & vetorização – Composição for simd

- O número de *threads* e o escalonamento afetam o desempenho
  - *Trade-off*: mais *threads* diminuem o nr de iterações para o loop SIMD
    - Blocos de iterações para as *threads* deveriam considerar o *vector length*
- No exemplo:
  - ***func_1()***: o tamanho do bloco de iterações das *threads* é 5
  - ***func_2()***: o tamanho do bloco de iterações das *threads* é *ceiling(chunk_sz/simd_len) * simd_len*, i.e., múltiplos de *vector length*

```
 1 void func_1(float *a, float *b, int n)
 2 {
 3   #pragma omp for simd schedule(static, 5)
 4   for (int k=0; k<n; k++)
 5   {
 6     // do some work on a and b
 7   }
 8 }
 9
10 void func_2(float *a, float *b, int n)
11 {
12   #pragma omp for simd schedule(simd:static, 5)
13   for (int k=0; k<n; k++)
14   {
15     // do some work on a and b
16   }
17 }
```

(Pas et al., 2017)

OMP & vetorização – Composição for simd

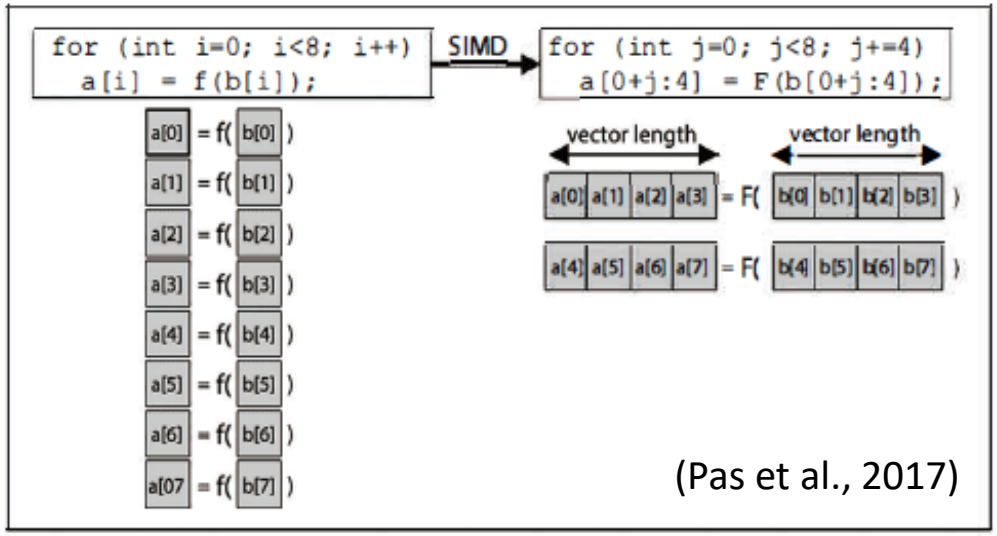
- Outro exemplo:

```
 1 double compute_pi(int n)
 2 {
 3   const double dH = 1.0 / (double) n;
 4   double dX, dSum = 0.0;
 5
 6   #pragma omp parallel for simd private(dX) \
 7                         reduction(+:dSum) schedule(simd:static)
 8   for (int i=0; i<n; i++) {
 9     dX = dH * ((double) i + 0.5);
10     dSum += (4.0 / (1.0 + dX * dX));
11   }
12   // End parallel for simd region
13
14   return dH * dSum;
15 }
```

(Pas et al., 2017)

OMP & vetorização – Funções SIMD

- Funções chamadas em um loop SIMD dificultam a geração de instruções SIMD
  - Elas devem ter cópias SIMD da versão que executará na porção escalar da CPU
  - Compilador deve gerar uma ou mais versões especiais da função com parâmetros e instruções SIMD
- No exemplo, a função escalar ***f()*** é modificada e renomeada para ***F()*** alterando argumentos de entrada e retornando um vetor inteiro.



OMP & vetorização – Funções SIMD

#pragma omp declare simd [clause[[,] clause] ...] new-line

Declaração da função

Cláusulas: ***simdlen (length)***

linear (list[:linear-step])

aligned (list[:alignment])

uniform (argument-list)

inbranch

notinbranch

- A ***declare simd*** permite que a função seja definida em um arquivo e chamada em outro
  - Compilador pode gerar novas versões SIMD mesmo sem ter analisado ainda as chamadas

OMP & vetorização – Funções SIMD

```
 1 #pragma omp declare simd
 2 double my_func(double b, double c)
 3 {
 4   double r;
 5   r = b + c;
 6   return r;
 7 }
 8
 9 void simd_loop_function(double *a, double *b, double *c, int n)
10 {
11   int i;
12   #pragma omp simd
13   for (i=0; i<n; i += 2)
14   {
15     a[i] = my_func(b[i], c[i]);
16   }
17   // End simd region
18 }
```

(Pas et al., 2017)

OMP & vetorização – Funções SIMD

- ***Cláusula uniform***
  - Lista deve ter variáveis que são parâmetros da função SIMD
  - Indica que todas as ***SIMD lanes*** têm o mesmo valor da variável
- ***Cláusula linear***
  - Não é uma cláusula de compartilhamento de dados como ocorre com o ***simd***
  - Determina como espalhar os dados do vetor nas *SIMD lanes*, considerando o deslocamento delas na memória e o *linear-step*

```
 1 #include <math.h>
 2 #pragma omp declare simd uniform(ptr, scale) linear(idx:1)
 3 double cosScaled(double *ptr, double scale, int idx)
 4 {
 5   return (cos(ptr[idx]) * scale);
 6 }
 7
 8 void simd_loop_uniform_linear(double *a, double *b, double c,
 9                                int n)
10 {
11   int i;
12
13   #pragma omp simd
14   for (int i=0; i<n; i++) {
15     a[i] = cosScaled(b, c, i);
16   }
17   // End simd region
18 }
```

(Pas et al., 2017)

OMP & vetorização – Funções SIMD

- Cláusulas ***uniform*** e ***linear***
  - Outro exemplo:

```
1 #pragma omp declare simd uniform(x, y, d1, i, a) linear(j)
2 void saxpy_2d(float *x, float *y, float a, int d1, int i, int j)
3 {
4   y[(d1*i)+j] = a*x[(d1*i)+j] + y[(d1*i)+j];
5 }
```

(Pas et al., 2017)

- Cláusula ***simdlen***
  - Especifica um comprimento do vetor (*vector length*) para a função chamada dentro do loop SIMD. Não é uma sugestão apenas neste caso, diferente do que ocorre com o construtor loop SIMD

```
 1 #pragma omp declare simd simdlen(16)
 2 char F(char x, char y, unsigned char mask)
 3 {
 4    return (x + y) & mask;
 5 }
 6
 7 void img_mask(char *img1, char *img2, int n, unsigned char *m)
 8 {
 9   #pragma omp simd simdlen(16)
10   for (int i=0; i<n; i++) {
11     img1[i] = F(img1[i], img2[i], m[i]);
12   } // End simd region
13 }
```

(Pas et al., 2017)

OMP & vetorização – Funções SIMD

- Cláusula ***aligned***
  - indica o alinhamento dos ponteiros passados como argumentos à função SIMD. As variantes desta função usam este alinhamento

```
1 #pragma omp declare simd linear(src,dst) \
2                     aligned(src,dst:16) simdlen(32)
3 void copy32x8(char *dst, char *src)
4 {
5    *dst = *src;
6 }
7
8 #pragma omp declare simd uniform(x,y) linear(i) \
9                          aligned(x,y:64) simdlen(16)
10 float saxpy(float a, float *x, float *y, int i)
11 {
12   return a * ((x[i]) + (y[i]));
13 }
```

(Pas et al., 2017)

OMP & vetorização – Funções SIMD

- Cláusulas ***inbranch*** e ***notinbranch***
  - Determinam possibilidade ou não de chamadas condicionais às funções SIMD
- Cláusula ***inbranch***
  - função SIMD sempre é chamada de dentro de um desvio condicional em um loop SIMD
- Cláusula ***notinbranch***
  - função SIMD nunca é chamada de dentro de um desvio condicional em um loop SIMD

```
 1 #pragma omp declare simd inbranch
 2 float do_mult(float x)
 3 {
 4   return (-2.0*x);
 5 }
 6
 7 #pragma omp declare simd notinbranch
 8 extern float do_pow(float);
 9
10 void simd_loop_with_branch(float *a, float *b, int n)
11 {
12   #pragma omp simd
13   for (int i=0; i<n; i++) {
14     if (a[i] < 0.0 )
15       b[i] = do_mult(a[i]);
16
17     b[i] = do_pow(b[i]);
18   } /* --- end simd region --- */
19 }
```

(Pas et al., 2017)

OMP & vetorização – Funções SIMD

- **Múltiplas versões de uma função SIMD**
  - Múltiplas e consecutivas diretivas ***declare simd***
    - Incluídas antes de uma função
    - Diferentes cláusulas indicam a geração de diferentes versões da função SIMD, uma para cada diretiva

```
1 #pragma omp declare simd linear(pixel) uniform(mask) inbranch
2 #pragma omp declare simd linear(pixel) notinbranch
3 #pragma omp declare simd
4 extern void compute_pixel(char *pixel, char mask);
```

(Pas et al., 2017)

Considerações Finais

- O desempenho de aplicações OMP é sensível ao uso otimizado dos recursos deste modelo de programação
  - Desempenho final pode variar muito a depender dos recursos usados
- Aprender o uso correto permite otimizar códigos com portabilidade e mais facilidade, quando comparado ao uso de *Assembly*
- OMP explora instruções SIMD nos processadores atuais
  - Ganhos de desempenho podem ser substanciais
- Instruções SIMD podem ser usadas em uma *thread* apenas
  - espera-se o uso em várias threads para explorar mais níveis de paralelismo

Considerações Finais

- Evitar desvios condicionais complexos
  - Estruturas condicionais deveriam ser evitadas
- Alinhar dados com o comprimento do vetor da arquitetura
- Indicar o comprimento do vetor com dependências
- Usar ***schedule(simd:static)*** em ***for simd*** para balancear o paralelismo de *threads* e instruções SIMD

Referências

Pas, Ruud van der.; Stotzer, Eric; Terboven, Christian. ***Using OpenMP-the next step: affinity, accelerators, tasking and SIMD***. The MIT Press, Cambridge, MA, 2017. ISBN 9780262534789

Huber, Joseph; Hernandez, Oscar; Lopez, Graham; Effective Vectorization with OpenMP 4.5. Report, Oak Ridge National Laboratory, 2017. ORNL/TM-2016/391

OpenMP, Especificações e Guias de Referência OpenMP. Disponíveis em: https://www.openmp.org/specifications/ Último acesso em 20/08/2020.

OpenMP: Vetorização

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente