Introducción a OpenMP e Extensión SSE

Elisardo Antelo

Arquitectura de Computadores

2º GEI

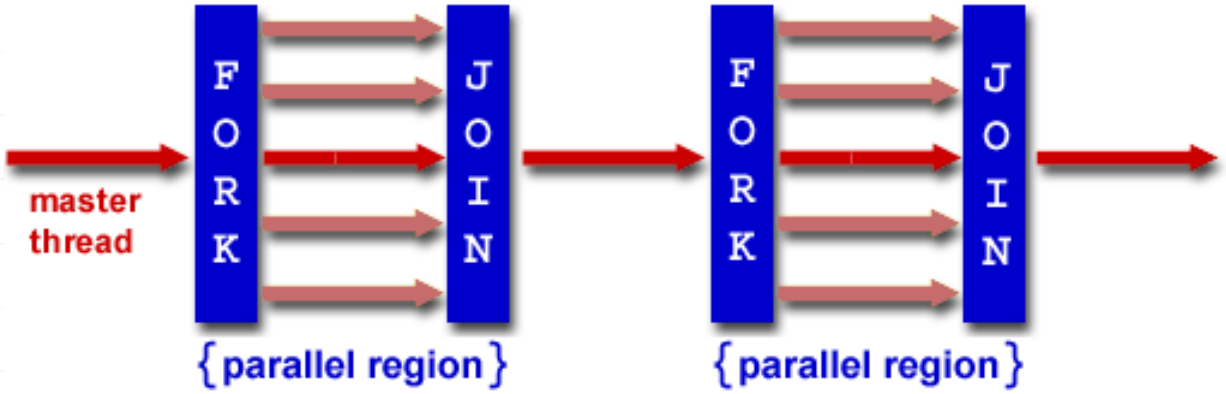
OpenMP

- Estándar industrial para programación en memoria compartida.
- Directivas de compilación, biblioteca de funcións e variables de entorno.
- Soportado para Fortran, C e C++.
- 1998: versión 1.0 para C/C++.
- 2002: versión 2.0 para C/C++.
- 2008: versión 3.0.
- 2013: versión 4.0.

Modelo Fork-Join

- Modelo de paralelismo baseado en fíos.
- Paralelismo explícito: o programador indica as partes do programa paralelizables.
- Modelo fork-join:
  - Fío master inicial executando código serie.
  - Fork: lánzanse fíos para operar en paralelo.
  - Join: terminación síncronizada dos fíos e continuación da execución serie no fío master.

Modelo Fork-Join



Estructura xeral do código

```
#include <omp.h>
main ()  {
int var1, var2, var3;
Serial code
 .
 .
Beginning of parallel section. Fork a team of threads. Specify variable
scoping
#pragma omp parallel private(var1, var2) shared(var3)
 {
 Parallel section executed by all threads
 .
 .
  All threads join master thread and disband
 }
Resume serial code
 .
}
```

Directiva parallel

#pragma omp parallel *[clause ...]*

private *(list)*
shared *(list)*
num_threads(int)

.....

{ Bloque de código }

Directiva parallel

- O código do bloque estructurado é executado por un equipo de fíos.
- O fío master pasa a ser o fío 0.
- Barreira implícita ao final da sección paralela. Só o fío 0 segue a partir deste punto.
- Número de fíos:
  - Opción **num_threads(int).**

Directiva parallel

- **Private (list):** lista de variables privadas a cada fío do equipo. Valor sin inicializar.
- **Shared(list):** lista de variables compartidas entre os fíos.
- Por defecto as variables declaradas antes de entrar en parallel son shared. As variables declaradas dentro da zona parallel son privadas por defecto.

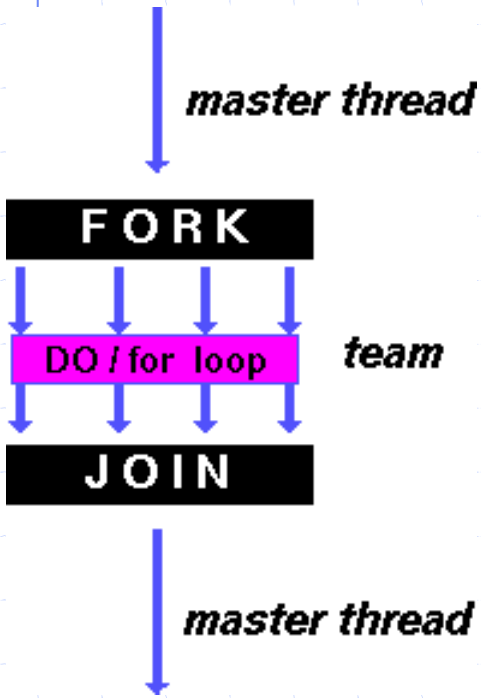
Exemplo de parallel

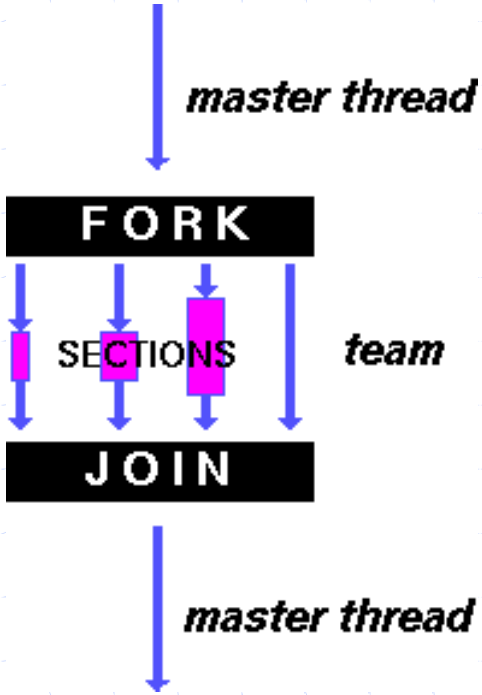
```
#include <omp.h>
#define k 8
main ()  {
int nthreads, tid;
/* arranca conxunto de fíos con copias privadas das variables */
#pragma omp parallel private(nthreads, tid) num_threads(k)
{
        /* Obter e imprimir o rango do fío*/
        tid = omp_get_thread_num();
        printf("Ola dende o fío = %d\n", tid);
        /* código para o fío master */
        if (tid == 0)     {    nthreads = omp_get_num_threads();
                printf("Número de fíos= %d\n", nthreads);
        }
 }  /* sincronización final (join) e terminación */ }
```

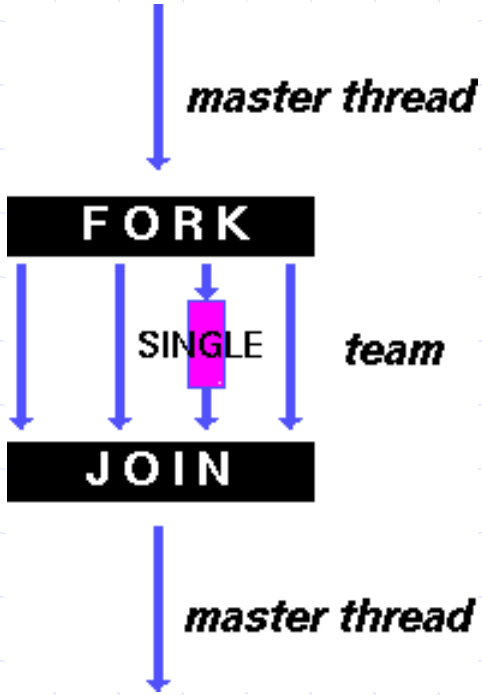
Construccións para división de traballo (Work sharing)

- Construccións:
  - **for**: reparte iteracións dun blucle entre o equipo de fíos (paralelismo de datos).
  - **Sections**: conxunto discreto de seccións a repartir entre o equipo de fíos (paralelismo funcional).
  - **Single**: serializa unha sección de código (execución única por parte dun só fío).
- Barreira implícita ao final da construcción.

Construcciónes para work sharing







Construcción for

#pragma omp for

Bucle for a paralelizar

Os fíos executan concorrentemente difierentes iteracións do bucle. É esencial ter en conta este feito para garantir resultados correctos cando unha iteración do bucle depende de resultados de iteracións anteriores.

Construcción for: exemplo

```
#include <omp.h>
#define n 1000
#define k 8
main () {
        int i;
        float a[n], b[n], c[n];
        for (i=0; i < N; i++)
                a[i] = b[i] = i * 1.0;

        #pragma omp parallel shared(a,b,c) private(i) num_threads (k)
        {
                #pragma omp for
                        for (i=0; i < n; i++)
                                c[i] = a[i] + b[i];

        }
}
```

Exemplo problemático

```
#include <omp.h>
#define k 8
main () {
int i, n;
float a[100], b[100], result, my_result,p_result[k];
n = 100;
result = 0.0;
for (i=0; i < n; i++) {
a[i] = i * 1.0; b[i] = i * 2.0;
}
#pragma omp parallel  private(i,my_result) num_threads(k)
{
        #pragma omp for
        for (i=0; i < n; i++) // bucle for a paralelizar
                my_result = my_result + (a[i] * b[i]);
        p_result[omp_get_thread_num()]=my_result;
}
for(i=0;i<k;i++) result=result+p_result[i]; // non se paraleliza
        printf("Final result= %f\n",result);
}
```

Sections

#pragma omp sections *[clause ...]*

{
#pragma omp section
{bloque estructurado}
#pragma omp section
{bloque estructurado}
} (/* Barreira implícita */

Exemplo: Sections

```
#include <omp.h>
#define N 1000
main (){
        int i;
        float a[N], b[N], c[N];
        for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
        #pragma omp parallel shared(a,b,c) private(i)
        {
                #pragma omp sections
                {
                #pragma omp section
                        for (i=0; i < N/2; i++) c[i] = a[i] + b[i];
                #pragma omp section
                        for (i=N/2; i < N; i++) c[i] = a[i] + b[i];
}       }       }
```

Single

#pragma omp single *[clause ...]*

{bloque estructurado}

Execución do bloque estructurado nun só fío

Exemplo con single

```
#include <omp.h>
#define n 1000
#define k 8
main () {
        int i;
        int a[n], b[n], c[n],part_count[k],count=0; int my_count=0;
        for (i=0; i < N; i++)
                {a[i] = rand(); b[i]= rand();}

        #pragma omp parallel shared(a,b,c) private(i,my_count) num_threads (k)
        {
                #pragma omp for
                        for (i=0; i < n; i++){
                                c[i] = a[i] + b[i];
                                if (c[i]==0) my_count++;}
                part_count[omp_get_thread_num()]=my_count++;
                 #pragma omp single
                        { for(i=0;i<k;i++) count=count+part_count[k];
                                printf("count=%d \n",count);}
        }
}
```

Directivas de sincronización

- Directiva Master.
- Directiva Critical.
- Directiva Barrier.
- Directiva Atomic.

Master

- Indica que o bloque estructurado só debe ser executado polo fío master.
- Non existe barreira implícita. O resto de fíos ingoran o bloque.

```
#pragma omp master
        structured_block
```

Critical
(serializa o código, utilizar con precaución)

- Especifica unha zona do código (sección crítica) que debe ser executada únicamente por un fío cada vez.
- Pódese asignar un nome á sección crítica. As seccións co mesmo nome actúan como unha soa sección crítica.

#pragma omp critical *[name]*
{bloque estructurado}

Exemplo de Critical

```
#include <omp.h>
#define k 8
main()
{
        int x,y;
        x = 100; y=100;
        #pragma omp parallel shared(x,y) num_threads (8)
                {
                        #pragma omp critical
                        {
                                x = x + 1;
                                y = y - 1;
                        }
                }
}
```

Barrier e Atomic

- Barreira de sincronización do equipo de fíos.

```
#pragma omp barrier
```

- Implementación eficiente dunha sección crítica simple (asignación do resultado dunha operación aritmética a unha variable). Ningunha escritura por parte doutro fío interfire na evaluación da expresión

```
#pragma omp atomic
        expression_statement
```

Exemplo con Atomic (pode que non sexa eficiente!)

```
#include <omp.h>
#define k 8
main()
{
        int x[100];
        #pragma omp parallel  shared(x) num_threads (k)
                {
                        #pragma omp for
                        for (i=0; i < 100; i++){
                                #pragma omp atomic
                                        x[rand()] += i;
                        }

                }
}
```

Funcións

- **int omp_get_num_threads(void):** devolve o número de fíos executando unha rexión paralela.
- **int omp_get_thread_num(void):** número natural que identifica o fío no equipo.

Funcións

- **int omp_get_num_procs(void)**: número de procesadores dispoñibles.
- **omp_set_num_threads(int n_threads)**: axusta o número de threads que deben executarse en paralelo.

Funcións para locks

- **void omp_init_lock(omp_lock_t *lock)**: inicializa un lock asociado cunha variable tipo lock.
- **void omp_destroy_lock(omp_lock_t *lock)**: disocia o lock da variable tipo lock.
- **void omp_set_lock(omp_lock_t *lock)**: bloquea o thread até que o lock está libre, e toma posesión do lock.
- **void omp_unset_lock(omp_lock_t *lock)**: libera o lock.

Funcións para locks

- **int omp_test_lock(omp_lock_t *lock)**: intenta facerse co control de lock. Si non está libre, a rutina non bloquea e devolve o control (neste caso devolve un cero).
- O lock foi tomado polo fío se devolve valor distinto de cero.

Extensións Vectoriais

- Permiten procesamento vectorial explícito (SIMD).
- Requiren declaracións de tipos de datos apropiados e funcións específicas (intrinsics).
- Maior esforzo de programación, pero ganancia en velocidade potencial.
- Nesta práctica utilizaremos extensións SSE3 con operacións sobre 128 bits.

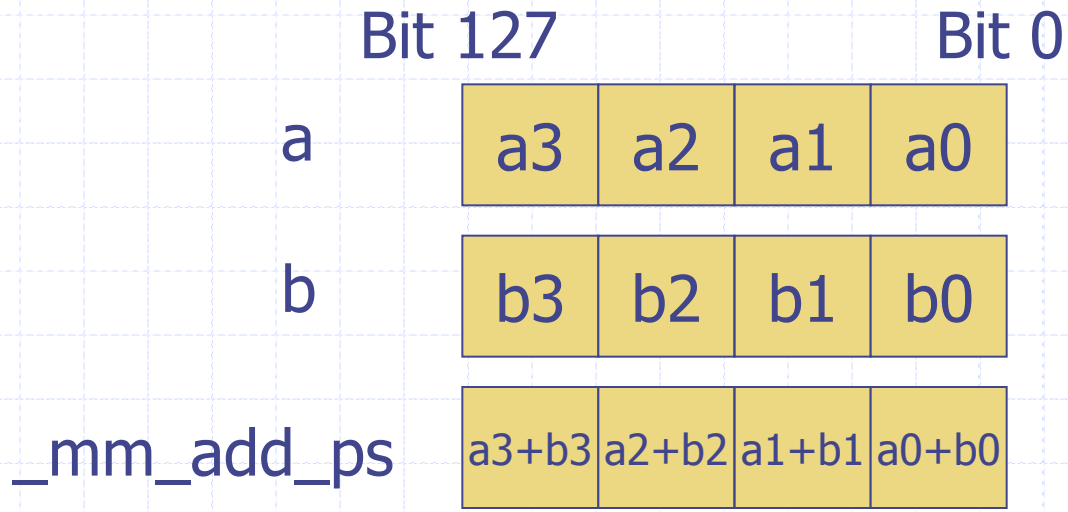
Tipos de datos

- __m128: tipo de dato para designar aos rexistros SIMD de 128 bits, que pode almacenar un total de 4 operandos de punto flotante en simple precision (float).

Exemplo: __m128 a; // a é un rexistro de 128 bits

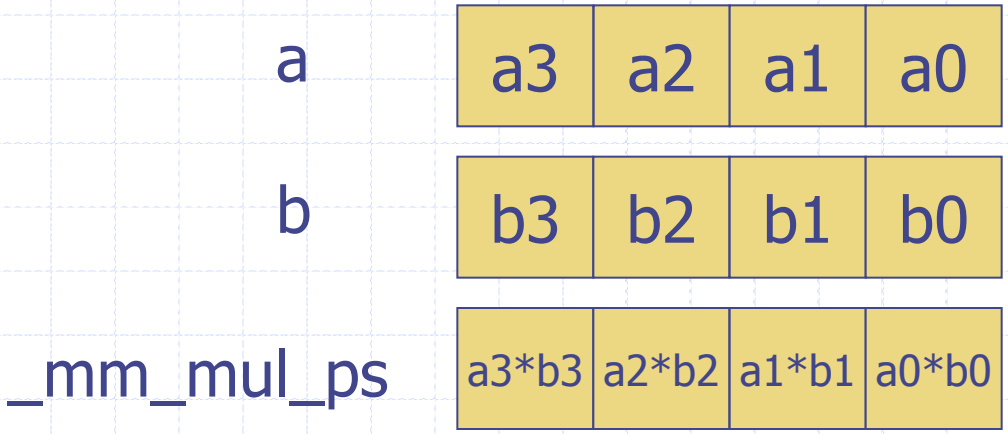
Funcións

- __m128 _mm_add_ps(__m128 a, ___m128 b)
  - Suma dous rexistros do tipo __m128, facendo a suma vectorial dos catro floats que compoñen cada rexistro.
  - Devolve un tipo de dato __m128 cos catro floats do resultado. _mm_sub_ps para restar.



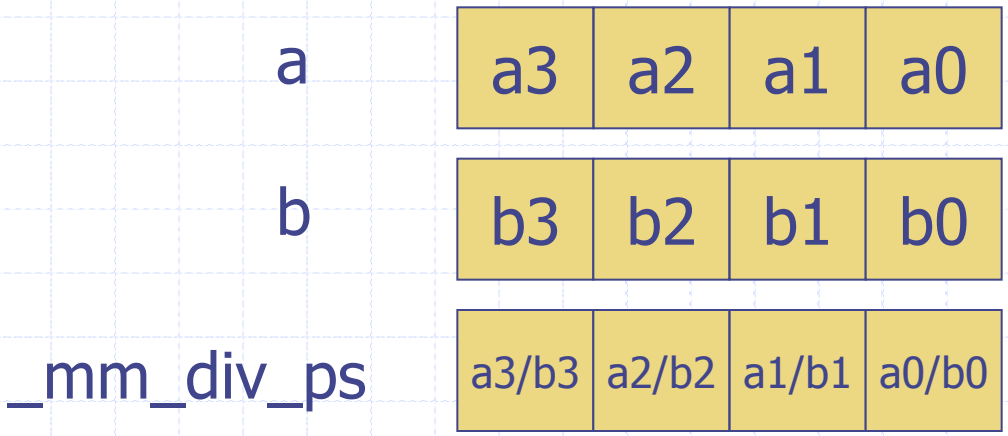
Funcións

- __m128 _mm_mul_ps(__m128 a, ___m128 b)
  - multiplica dous rexistros do tipo __m128, facendo a multiplicación vectorial dos catro floats que compoñen cada rexistro.
  - Devolve un tipo de dato __m128 cos catro floats do resultado.



Funcións

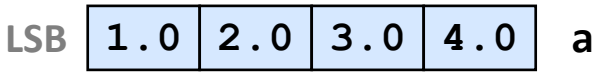
- __m128 _mm_div_ps(__m128 a, ___m128 b)
  - Divide dous rexistros do tipo __m128, facendo a división vectorial.
  - Devolve un tipo de dato __m128 cos catro floats do resultado.

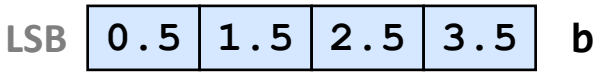


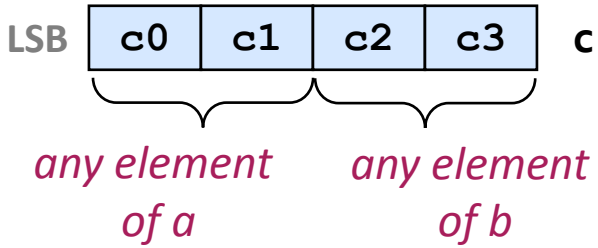
Shuffles

```
c = _mm_shuffle_ps(a, b, _MM_SHUFFLE(l, k, j, i));
```

helper macro to create mask








```
c0 = ai
c1 = aj
c2 = bk
c3 = bl
i,j,k,l in {0,1,2,3}
```

→ blackboard

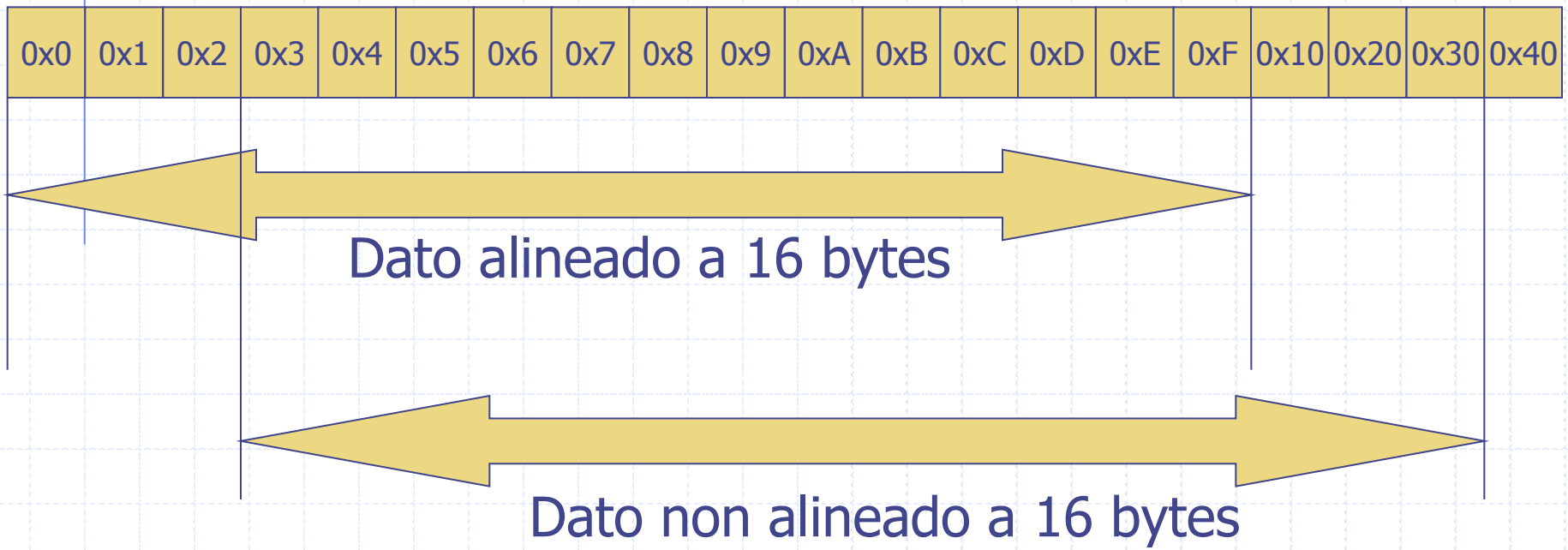
Funcións

- __m128 _mm_setzero_ps()
  - Devolve un tipo de dato __m128 con todos os compoñentes a cero.
- __m128 _mm_set_ps(float z, float y, float x, float w)
  - Devolve un tipo de dato __m128 co valor dos catro floats indicado (á orde é zyxw).
- __m128 _mm_setr_ps(float z, float y, float x, float w)
  - Semellante ao anterior, pero cambiando a orde (wxyz).
- __m128 _mm_set1_ps(float w)
  - Devolve un __m128 co mesmo valor para os catro floats: w.

Funcións

- __m128 _mm_load_ps(float *p)
  - Devolve un tipo de dato __m128 cos valores (p[3],p[2],p[1],p[0]) ou de xeito equivalente (*(p+3),*(p+2),*(p+1),*(p)), con p[i] float.
  - Indicamos o enderezo de comenzo para cargar catro floats que están emprazados de xeito consecutivo en memoria.
  - Os enderezos deben estar alineados a 16 bytes (enderezo mod 16 =0).
- __m128 _mm_loadu_ps(float *p)
  - Equivalente á anterior pero se necesidade de enderezos alineados a 16 bytes (pode ser máis lento!)

Enderezos alineados



Enderezos alineados

- Alinear datos cun atributo na declaración:
  - float k[8] __attribute__((aligned(16)));
- Reserva dinámica con enderezo alineado:
  - float *p; p=_mm_malloc(8*sizeof(float),16);

    .....

    _mm_free(p);

Funcións

- void _mm_store_ps(float *p,__m128 a)
  - Almacena o contido dun rexistro tipo __m128 con contido (a3,a2,a1,a0), nas posicións de memoria indicadas polo punteiro, de tal xeito que p[0]=a0, p[1]=a1, p[2]=a2, e p[3]=a3.
  - O punteiro debe estar alineado a 16 bytes.
- void _mm_storeu_ps(float *p,__m128 a)
  - Igual que a anterior, pero sen que punteiro teña que estar alineado (pode ser máis lento).
- void _mm_storer_ps(float *p,__m128 a)
  - Igual que _mm_store_ps, pero de tal xeito que p[0]=a3, p[1]=a2, p[2]=a1 e p[3]=a0.

Compilación

- #include <pmmintrin.h>
- #include <omp.h>
- gcc –fopenmp –msse3….
  - fopenmp: para compilar con directivas OpenMP.
  - -msse3 para as extensións simd sse.

Exemplo SSE

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

int main(){
      int i;
      float k[4] __attribute__((aligned(16)))={0.5,0.25,0.125,0.0625};
      float s[4] __attribute__((aligned(16)));
      float r=2.0;
      __m128 A,B;
      A=_mm_set1_ps(r);
      B=_mm_load_ps(&k[0]);
      _mm_store_ps(&s[0],_mm_mul_ps(A,B));
      for(i=0;i<4;i++)
            printf("\n s[%d]=%f \n",i,s[i]);
}
```