Tema 3. Vectorización, paralelismo de datos

Computación de Altas Prestaciones

Carlos García Sánchez

20 de septiembre de 2021

- "Computer Architecture: A Quantitative Approach", J.L. Hennessy, D.A. Patterson, Morgan Kaufmann 2011
- "Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition", James Jeffers, James Reinders, Avinash Sodani

Outline

1 Introducción

2 ¿Como vectorizar?

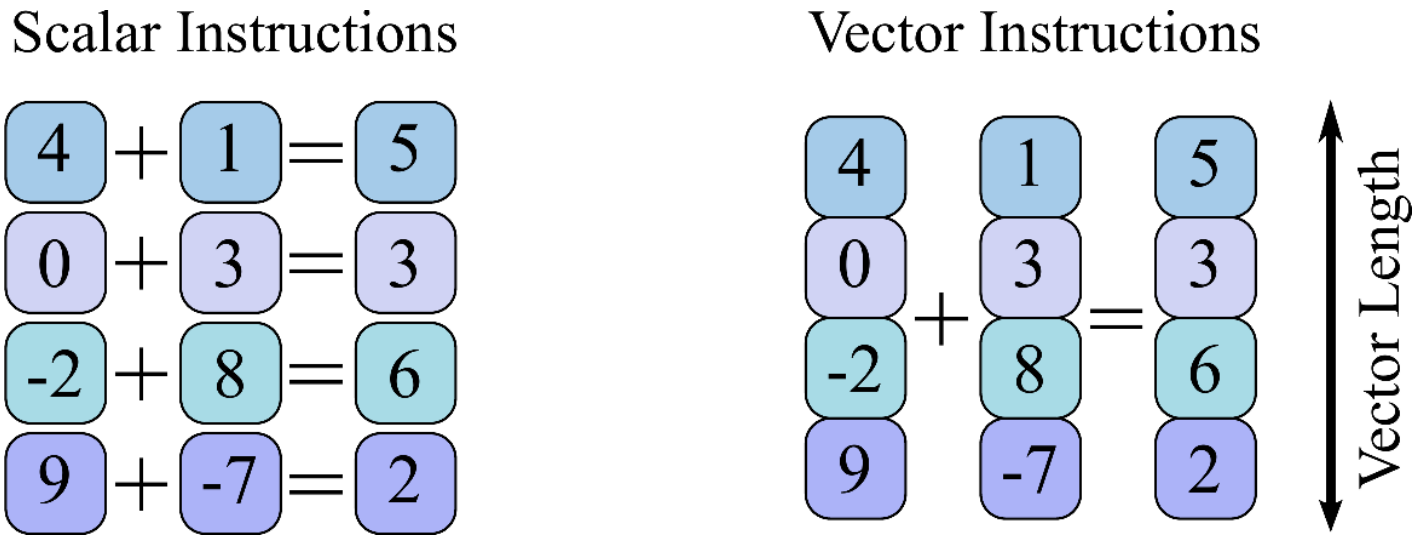
3 Vectorización intrínsecas

4 Vectorización automática

5 Vectorización guiada

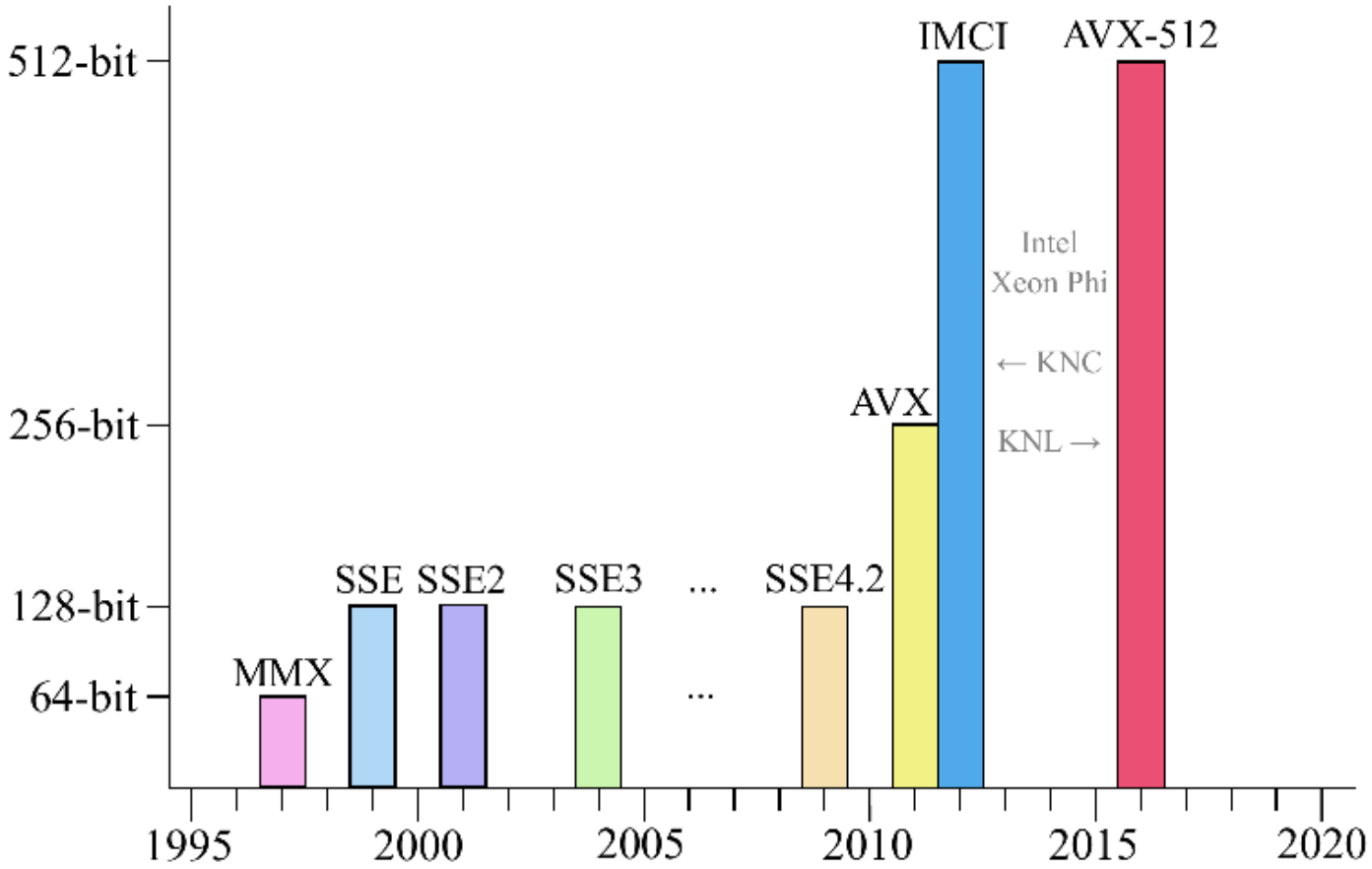
Soporte operaciones vectoriales

- Pequeños vectores = SIMD
- SIMD (Single Instruction Multiple Data)



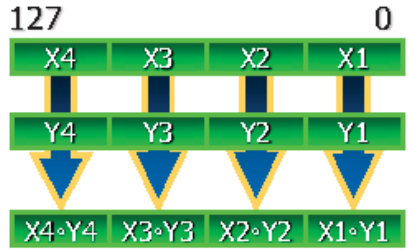
Soporte operaciones vectoriales

- Introducción en el ISA de Intel a partir de 1998



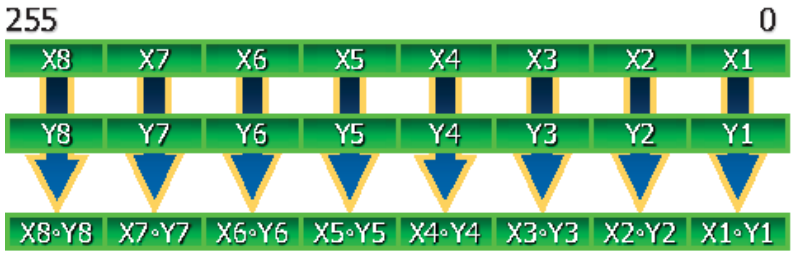
Soporte operaciones vectoriales

- Introducción en el ISA de Intel a partir de 1998



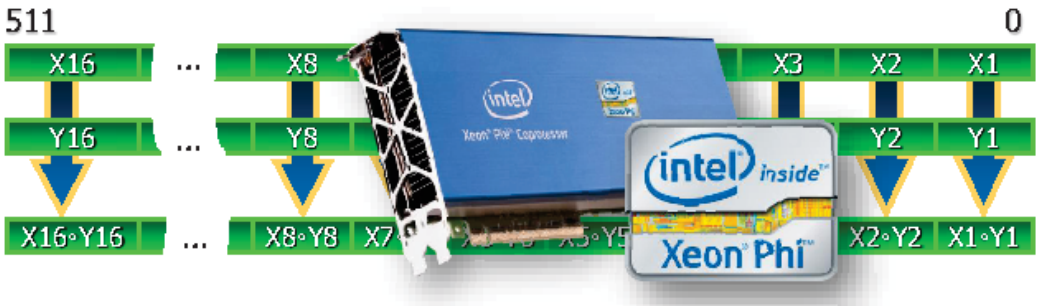
SSE
Vector size: **128 bit**
Data types:
- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

VL: 2, 4, 8, 16



AVX
Vector size: **256 bit**
Data types:
- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

VL: 4, 8, 16, 32



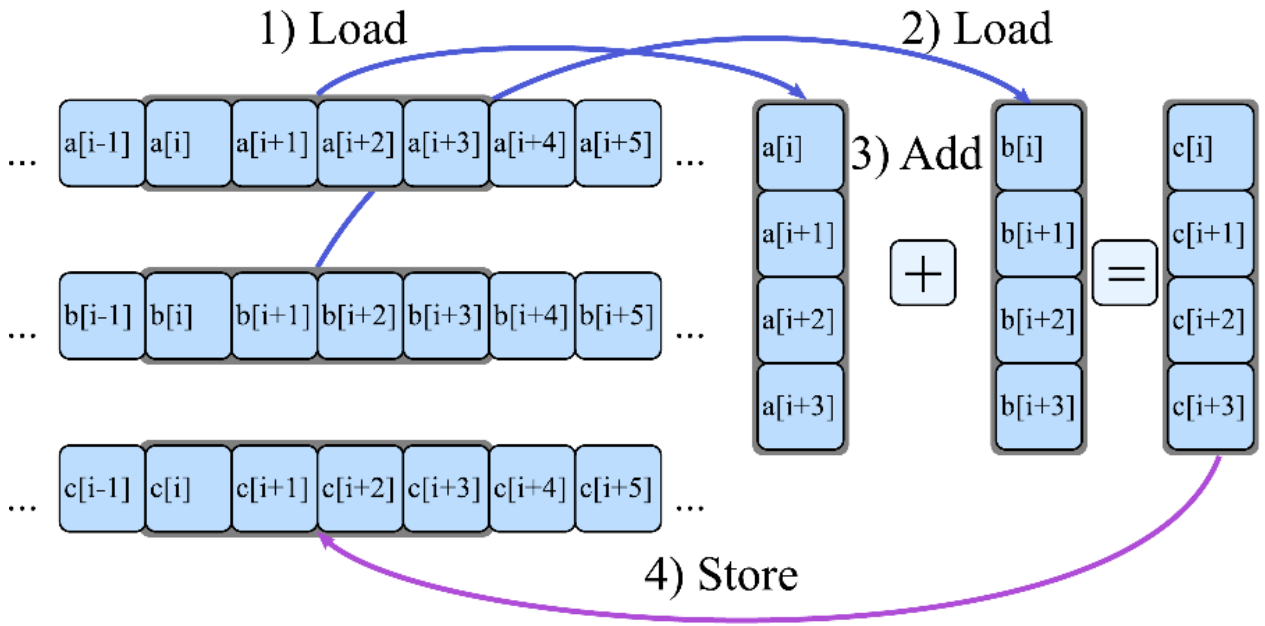
Intel® MIC
Vector size: **512 bit**
Data types:
- 32 bit integer
- 32 and 64 bit float

VL: 8, 16

Nociones básicas

vectorAdd.c

```
// Compute vector sum: C = A+B
   for (i = 0; i < n; i++)
      c[i] = a[i] + b[i];
```



Tres aproximaciones

- Vectorización automática

vectorAdd_auto.c

```
double a[vec_width], b[vec_width], c[vec_width];
//...
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

- Vectorización explícita (intrínsecas)

vectorAdd_instrisic.c

```
double a[8], b[8], c[8];
//...
__m512d A_v = _mm512_load_pd(a);
__m512d B_v = _mm512_load_pd(b);
__m512d C_v = _mm512_add_pd(A_v,B_v);
_mm512_store_pd(c, C_v);
```

- Vectorización guiada (#pragmas)

Ejemplo *axpy*

- SSE en un procesador

axpy_instrisic.c

```
#include <xmmintrin.h>

void axpy_C(float *c, float *a, float *b, float cte, int n)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] + cte*b[i];
}

void axpy_intrinsic(float *c, float *a, float *b, float cte, int n)
{
    int i;
    __m128 sse_c, sse_a, sse_b, sse_cte;
    sse_cte = _mm_set1_ps(cte);

    for (i=0; i<n; i+=4){
        sse_a = _mm_load_ps(&a[i]);
        sse_b = _mm_load_ps(&a[i]);
        sse_b = _mm_mul_ps(sse_b, sse_cte);
        sse_c = _mm_add_ps(sse_a, sse_b);
        _mm_store_ps(&c[i], sse_c);
    }
}
```

Vectorización intrínsecas

- Información en los manuales de Intel[1]
- o en la Guía de Intrínsecas[2]

[1] The Intel® 64 and IA-32 Architectures Software Developer Manuals
`https://software.intel.com/en-us/articles/intel-sdm`

[2] The Intel® Intrinsics Guide:
`https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

Vectorización intrínsecas (SSE)

- Para usar las extensiones SSE hay que añadir las cabeceras
  - Donde se encuentran los prototipos de las funciones

headers.c

```
#include <xmmintrin.h> //(for SSE)
#include <emmintrin.h> //(for SSE2)
#include <pmmintrin.h> //(for SSE3)
#include <smmintrin.h> //(for SSE4)
```

Vectorización intrínsecas (SSE)

- Se pueden usar diferentes tipos de datos (registros vectoriales XMM)
  - Datos empaquetados de flotantes de simple precisión ___m128
  - Datos empaquetados de flotantes de doble precisión ___m128d
  - Datos empaquetados de enteros de 32 bits ___m128i

example.c

```
#include <xmmintrin.h>
int main ( ) {
...
__m128 A, B, C; /* three packed s.p. variables */
...
}
```

Vectorización intrínsecas (SSE)

- Las instruciones intrísecas pueden operar con diferentes datos y tiene el formato:
  - *_mm_instruction_suffix(...)*
- Donde el sufijo puede tener diferente forma:
  - *ss* para un único dato flotante
  - *ps* para un vector empaquetado de *floats*
  - *sd* para un único flotante *double precision*
  - *pd* para un vector empaquetado de *doubles*
  - *si#* para un vector enteros (8, 16, 32, 64, 128 bits)
  - *su#* para un vector enteros sin signo (8, 16, 32, 64, 128 bits)

Vectorización intrínsecas (SSE)

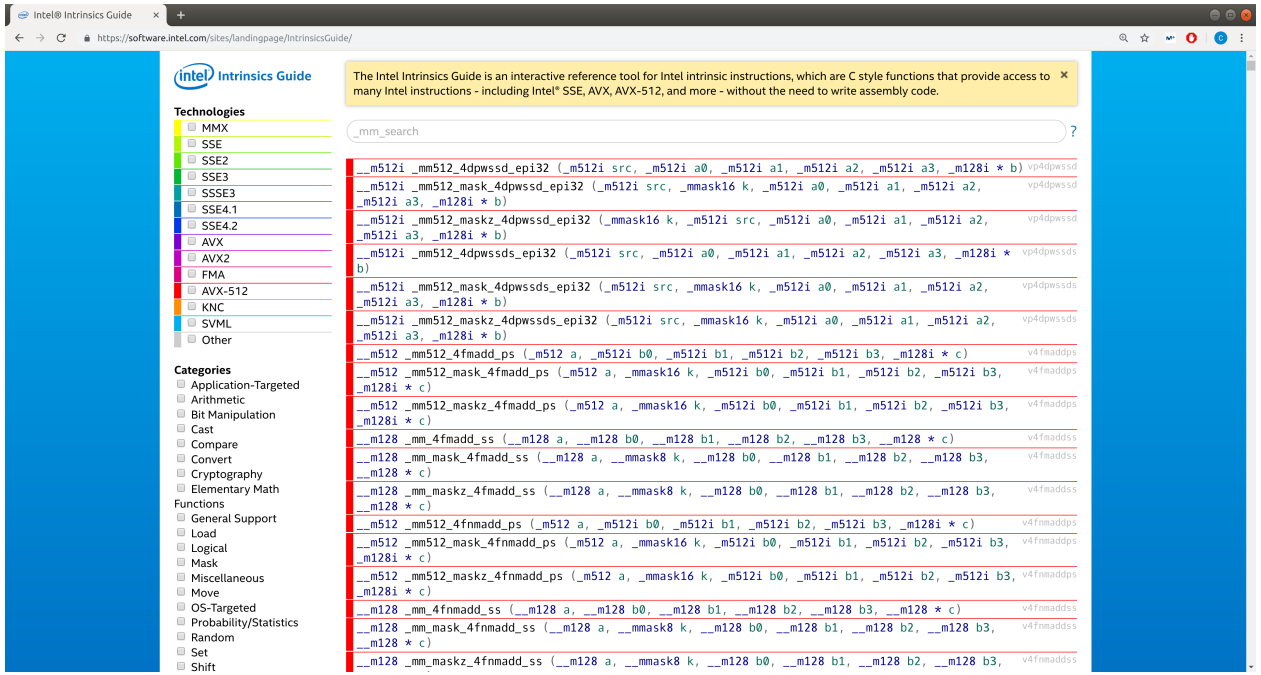
- Ejemplo de un load de 4 *flotantes* alineados (a 16-byte)
- Ejemplo de suma de dos vectores de *floats*

example.c

```
#include <xmmintrin.h>
int main ( ) {
...
float a[4]={1.0,2.0,3.0,4.0};//a must be 16-byte aligned
__m128 x = _mm_load_ps(a);
__m128 a, b;
__m128 c = _mm_add_ps(a, b);
...
}
```

Instrucciones Intrínsecas

- Guía online [3]



[3] https://software.intel.com/sites/landingpage/IntrinsicsGuide

Vectorización automática

- Activar el flag *-qopt-report* para conocer los bucles vectorizados
- Mejor rendimiento si los loads son alineados:
  - Como con instrínsecas: *_mm_load_XX* mejor que *_mm_loadu_XX*

example_autovec.c

```
#include <cstdio>
int main(){
    const int n=1024;
    int A[n] __attribute__((aligned(64)));
    int B[n] __attribute__((aligned(64)));

    // ...

    // This loop will be auto-vectorized
    for (int i = 0; i < n; i++)
        A[i] = A[i] + B[i];
}
```

Vectorización automática

example_autovec.c

```
11: // This loop will be auto-vectorized
12: for (int i = 0; i < n; i++)
13:    A[i] = A[i] + B[i];
```

Terminal #1

```
carlos@posets:$ icpc autovec.cc -qopt-report
carlos@posets:$ cat autovec.optrpt
...
LOOP BEGIN at autovec.cc(12,3)
remark #15399: vectorization support:
unroll factor set to 2 [autovec.cc(12,3)]
remark #15300: LOOP WAS VECTORIZED
[autovec.cc(12,3)]
LOOP END
...
carlos@posets:$ ./a.out
0 0 0
1 2 1
2 4 2
3 6 3
4 8 4
...
```

Limitaciones de la vectorización automática

- Únicamente válido para los bucles **internos**
- Deben conocerse el número de interaciones: **n**
  - Normalmente iteraciones consecutivas: **i++** o **i–**
- Sin dependencias entre iteraciones
  - Los punteros suele inhibir vectorización automática
  - Se puede indicar anti-aliasing de punteros (vectorización guiada)
- En caso de *vectorizar* funciones se debe especificar

Compilador de Intel

- Opción *-x[code]* para especificar la arquitectura del procesador
- Opción *-ax[code]* para multiarquitectura

code	Target Architecture
MIC-AVX512	Intel Xeon Phi (KNL)
CORE-AVX512	Intel Xeon Proc. AVX512
CORE-AVX2	Intel Xeon Proc. E3/E5/E7 v3, v4
AVX	Intel Xeon Proc. E3/E5 and E3/E5/E7 v2
SSE4.2	Intel Xeon Proc. 55XX, 56XX, 75XX y E7
host	where is compiled

Compilador de Intel

-qopt-report

- n=0: No diagnostic information
- n=1: Loops successfully vectorized
- n=2: Loops not vectorized - and the reason why not
- n=3: Adds dependency Information
- n=4: Reports only non-vectorized loops
- n=5: Reports only non-vectorized loops and adds dependency info

Vectorización automática

- A veces la vectorización automática puede ser complicada
  - **Punteros**, variables que pueden ser constantes....

example_autovec.c

```
for (int i = ii; i < ii + tileSize; i++) { // Auto-vectorized

    // 'Newtons law of universal gravity
    const float dx = particle.x[j] - particle.x[i]; // x[j] is a const
    const float dy = particle.y[j] - particle.y[i]; // x[i] -> vector
    const float dz = particle.z[j] - particle.z[i];
    const float rr = 1.0f/sqrtf(dx*dx + dy*dy + dz*dz + softening);
    const float drPowerN32 = rr*rr*rr;

    // Calculate the net force
    Fx[i-ii] += dx * drPowerN32;
    Fy[i-ii] += dy * drPowerN32;
    Fz[i-ii] += dz * drPowerN32;
}
```

Vectorizar bucles

#pragma omp simd

- Usada para *forzar* la vectorización de bucles:
  - Se permiten bucles con funciones (explicitamente definidas como SIMD)
  - Bucles internos
  - **Compilador inhibe vectorización**
    - Longitud vector constante
    - Reducciones
    - Accesos a memoria potencialmente dependientes
- Para más info, ver referencia [4]

[4] #pragma simd: `https://software.intel.com/node/524530`

Vectorizar con #pragma omp simd

example_guided.c

```
const int N=128, T=4;
float A[N*N], B[N*N], C[T*T];
for (int jj = 0; jj < N; jj+=T) // Tile in j
  for (int ii = 0; ii < N; ii+=T) // and tile in i
#pragma omp simd
    // Vectorize outer loop
    for (int k = 0; k < N; ++k) // long loop, vectorize it
      for (int i = 0; i < T; i++) { // Loop between ii and ii+T
        // Instead of a loop between jj and jj+T, unrolling that loop:
        C[0*T + i] += A[(jj+0)*N + k]*B[(ii+i)*N + k];
        C[1*T + i] += A[(jj+1)*N + k]*B[(ii+i)*N + k];
        C[2*T + i] += A[(jj+2)*N + k]*B[(ii+i)*N + k];
        C[3*T + i] += A[(jj+3)*N + k]*B[(ii+i)*N + k];
      }
```

Directivas de vectorización

#pragma omp simd

- *#pragma vector always*
- *#pragma vector aligned | unaligned*
- *__assume_aligned* variable
- *#pragma vector nontemporal | temporal*
- *#pragma novector*
- *#pragma ivdep*
- *restrict* para calificar variable y *-restrict* como argumento en compilación
- *#pragma loop count*

Directiva #pragma omp simd

- Puede usarse en un bucle o para indicar que se desea vectorizar una función sencilla

example_omp_simd.c

```
// Define function in one file (e.g., library), use in another
#pragma omp declare simd
float my_simple_add(float x1, float x2){
   return x1 + x2;
}

...
   // May be in a separate file
   #pragma omp simd
   for (int i = 0; i < N, ++i) {
      output[i] = my_simple_add(inputa[i], inputb[i]);
}
```

Dependencias de datos

- Dependencias reales: **vectorización imposible**

example_simd.c

```
for (int i = 1; i < n; i++)
  a[i] += a[i-1]; // dependence on the previous element
```

- Vectorización segura

example_simd.c

```
for (int i = 0; i < n-1; i++)
  a[i] += a[i+1]; // no dependence on the previous element
```

- Puede ser segura la vectorización

example_simd.c

```
for (int i = 16; i < n; i++)
  a[i] += a[i-16]; // no dependence if vector length <=16
```

Dependencias de datos

- Dependencias que pueden aparecer porque no se dispone de información en tiempo de compilación
  - Como se sabe que el espacio de memoria de punteros **a** y **b** no apunta a la misma región de memoria?
- Si *a* y *b* no solapan (*aliased*) la vectorización es segura
- ... pero si *a* y *b* solapasen (ej: b=a-1), se requiriría hacerlo de forma secuencial

example_ambigous.c

```
void AmbiguousFunction(int n, int *a, int *b) {
   for (int i = 0; i < n; i++)
      a[i] = b[i];
}
```

Dependencias de datos (#pragma ivdep)

- Informa al compilador que el bucle no tiene dependencias
  - Para prevenir la desambiguación de punteros (no hay solapamiento)

example_ambigous.c

```
void AmbiguousFunction(int n, int *a, int *b) {
#pragma ivdep
   for (int i = 0; i < n; i++)
      a[i] = b[i];
}
```

Terminal #1

```
user@host% icpc -c example_ambigous.c -qopt-report -qopt-report-phase:vec
user@host% cat vdep.optrpt
...
LOOP BEGIN at example_ambigous.c(4,1)
remark #25228: LOOP WAS VECTORIZED
LOOP END
...
```

Dependencias de datos (restrict)

- Para prevenir la desambiguación de punteros (no hay solapamiento)

example_ambigous.c

```
void AmbiguousFunction(int n, int *restrict a, int *restrict b) {
   for (int i = 0; i < n; i++)
      a[i] = b[i];
}
```

Terminal #1

```
user@host% icpc -c example_ambigous.c -qopt-report -qopt-report-phase:vec -restrict
user@host% cat vdep.optrpt
...
LOOP BEGIN at example_ambigous.c(4,1)
remark #25228: LOOP WAS VECTORIZED
LOOP END
...
```

Strip mining

- Troceado del bucle
  - Técnica de programación que convierte un bucle en dos bucles anidados
- Se utiliza para exponer facilitar la vectorización

strip_mining.c

```
void original(...) {
   for (int i = 0; i < n; i++)
   // ... do work
}


const int STRIP=1024;
const int nPrime = n - n%STRIP;
void strip_mined(...) {
   for (int ii=0; ii<nPrime; ii+=STRIP)
      for (int i = ii; i < ii+STRIP; i++)
      // ... do work

   for (int i=nPrime; i<n; i++)
     // ... do leftover work
}
```

Alineamiento de datos

- El alineamiento de datos es una metodología para crear datos alineados en las palabras de memoria
- Tiene el objetivo de **incrementar el rendimiento**
  - *load* y *store* se realizan en a cada palabra
- En memoria estática con el atributo *aligned*
- En memoria dinámica sustituyendo malloc por *_mm_malloc* y free por *_mm_free*

```
float buf_static[1000] __attribute__((aligned(64)));
float *buf_dynamic = (float*) _mm_malloc(buffer_size, 64);
```

Alineamiento de datos

- El alineamiento de datos es una metodología para crear datos alineados en las palabras de memoria
- Tiene el objetivo de **incrementar el rendimiento**
  - *load* y *store* se realizan en a cada palabra
- En memoria estática con el atributo *aligned*
- En memoria dinámica sustituyendo malloc por *_mm_malloc* y free por *_mm_free*

```
float buf_static[1000] __attribute__((aligned(64)));
float *buf_dynamic = (float*) _mm_malloc(buffer_size, 64);
```

#pragma vector aligned

- El vector *X* y *X2* se define estáticamente alineado
- El vector *a* no se sabe si está alineado

vector_aligned.c

```
__declspec(align(64)) float X[1000], X2[1000];
void foo(float * restrict a, int n, int n1, int n2) {
  int i;
//#pragma vector aligned
  for(i=0;i<n;i++) { // Compiler vectorizes loop with not all aligned
       accesses
    X[i] += a[i] + a[i+n1] + a[i-n1]+ a[i+n2] + a[i-n2];
  }

#pragma vector aligned
  for(i=0;i<n;i++) { // Compiler vectorizes loop with all aligned
       accesses
    X2[i] += a[i] + a[i+n1] + a[i-n1]+ a[i+n2] + a[i-n2];
  }
}
```

Terminal #1

```
user@host% icc -c vector_aligned.c -O3 -qopt-report=5 -xCORE-AVX2
...
```

#pragma vector aligned

Terminal #1

```
user@host% user@host% cat vector_aligned.optrpt
...
LOOP BEGIN at vector_aligned.c(5,3)
   remark #15388: vectorization support: reference X[i] has aligned access   [ vector_aligned.c(6,
   remark #15388: vectorization support: reference X[i] has aligned access   [ vector_aligned.c(6,
   remark #15389: vectorization support: reference a[i] has unaligned access   [ vector_aligned.c(
   remark #15389: vectorization support: reference a[i+n1] has unaligned access   [ vector_aligned
   remark #15389: vectorization support: reference a[i-
n1] has unaligned access   [ vector_aligned.c(6,30) ]
   remark #15389: vectorization support: reference a[i+n2] has unaligned access   [ vector_aligned
   remark #15389: vectorization support: reference a[i-
n2] has unaligned access   [ vector_aligned.c(6,49) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15305: vectorization support: vector length 8
   remark #15399: vectorization support: unroll factor set to 2
   remark #15309: vectorization support: normalized vectorization overhead 0.417
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15450: unmasked unaligned unit stride loads: 5
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 19
   remark #15477: vector cost: 2.250
   remark #15478: estimated potential speedup: 7.910
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=62
LOOP END

....

LOOP BEGIN at vector_aligned.c(10,3)
   remark #15388: vectorization support: reference X2[i] has aligned access   [ vector_aligned.c(1
   remark #15388: vectorization support: reference X2[i] has aligned access   [ vector_aligned.c(1
   remark #15388: vectorization support: reference a[i] has aligned access   [ vector_aligned.c(11
   remark #15388: vectorization support: reference a[i+n1] has aligned access   [ vector_aligned.c
   remark #15388: vectorization support: reference a[i-
n1] has aligned access   [ vector_aligned.c(11,31) ]
   remark #15388: vectorization support: reference a[i+n2] has aligned access   [ vector_aligned.c
   remark #15388: vectorization support: reference a[i-
n2] has aligned access   [ vector_aligned.c(11,50) ]
   remark #15305: vectorization support: vector length 8
   remark #15399: vectorization support: unroll factor set to 2
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 6
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 19
   remark #15477: vector cost: 1.620
   remark #15478: estimated potential speedup: 10.770
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=62
LOOP END
```

Otras pragmas

#pragma loop count (n)

- Informa al compilador de las *n* iteraciones independientes por lo que decide si merece la pena o no generar código vectorial

#pragma vector always

- Fuerza al compilador a vectorizar el bucle

Otras pragmas

#pragma novector

- Fuerza al compilador a no vectorizar el bucle

#pragma vector nontemporal *variable*

- Da una pista al compilador para indicar que los datos no serán reutilizados
  - Hace un *bypass* de la cache para evitar su polución