

Tema 4.3 Programación mediante directivas OpenMP: Vectorización Computación de Altas Prestaciones

Carlos García Sánchez

10 de octubre de 2022

- ‘Using OpenMP : portable shared memory parallel programming’, Barbara Chapman, et all. 2008
- “OpenMP 5.0”, <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>



Outline

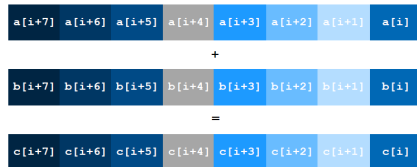
- 1 Visión general
- 2 OpenMP
- 3 Evolución OpenMP



Terminología

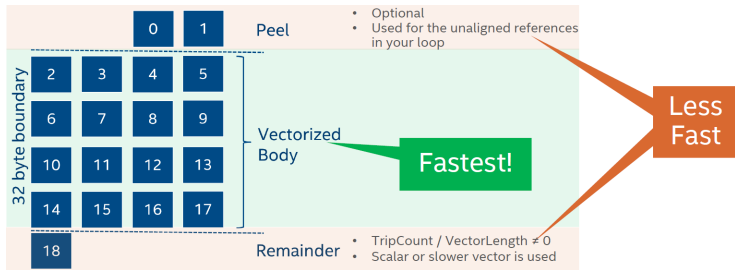
- Concepto **SIMD**: explotación de paralelismo de datos
- Alternativas
 - Explotación mediante invocación de llamada a librerías
 - De forma automática vs guiada
 - De forma explícita con **instrínsecas**

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



Peel, main & remainder

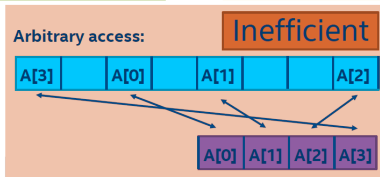
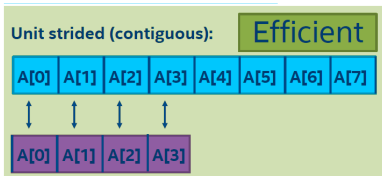
- Un bucle vectorizado consta de las siguiente partes
 - Peel loop (opcional): se puede usar para alinear datos de memoria
 - Cuerpo del bucle principal **vectorizado**
 - Parte final o remanente: iteraciones no es divisible por VL



Factores degradan rendimiento

■ Patrones de acceso a memoria

- $stride = 1$ vs $stride \neq 1$ vs $stride = random$
- SoA vs AoS



Aproximaciones

■ Vectorización automática

vectorAdd_auto.c

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

■ Construcción OpenMP SIMD (#pragmas)

vectorAdd_omp.c

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

■ Construcción OpenMP SIMD con funciones

vectorAdd_func_omp.c

```
#pragma omp declare simd  
float ef(float a, float b) {  
    return a + b;  
}  
  
...  
#pragma omp simd  
for (int i = 0; i < N; ++i)  
    A[i] = ef(B[i], C[i]);  
...
```



Vectorización automática ineficiente

- Vectorización automático no funciona como “se espera”
 - El compilador prioriza la **ejecución correcta** (compilador conservador)
 - Las heurísticas del compilador estima vectorización ineficiente
 - Dependencias de datos (*punteros* pueden inhibir)
 - Alineamiento de datos
 - Llamadas a funciones en el bloque del bucle
 - Flujo de ejecución complejo: condicionales, saltos...
 - Mezclado de tipos de datos
 - $\text{stride} \neq 1$
 - Accesos a memoria no uniformes



Vectorización automática ineficiente

■ Data Layout: SoA vs AoS

■ Instrucciones de gather/scatter

Array-of-Structs (AoS)

x	y	z	x	y	z
x	y	z	x	y	z
x	y	z	x	y	z

- Pros:
Good locality of
{x, y, z},
1 memory stream
- Cons:
Potential for gather/scatter

Struct-of-Arrays (SoA)

x	x	x	x	x	x
y	y	y	y	y	y
z	z	z	z	z	z

- Pros:
Contiguous load/store
- Cons:
Poor locality of
{x, y, z},
3 memory streams

Hybrid (AoSoA)

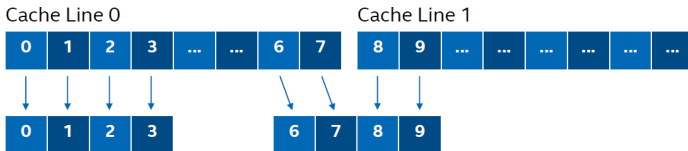
x	x	y	y	z	z
x	x	y	y	z	z
x	x	y	y	z	z

- Pros:
Contiguous load/store,
1 memory stream
- Cons:
Not a "normal" layout



Vectorización automática ineficiente

■ Alineamiento de memoria



Aligned Load

- Address is aligned
- One cache line
- One instruction

Unaligned Load

- Address is not aligned
- Potentially multiple cache lines
- Potentially multiple instructions



Vectorización ineficiente

- Alineamiento de memoria
 - `_mm_malloc(bytes, 64)` Alineamiento a 64bits de la línea de cache
 - 2 Formas de informar al compilador: mediante pragma o guiando al compilador automático

align_memory.c

```
// SIMD pragma
#pragma omp simd aligned(array)
for (i = 0; i < N; i++) {
    array[i] = ...
}

...
// Inform to compiler that array is aligned
__assume_aligned(array, 64);
for (i = 0; i < N; i++) {
    array[i] = ...
}
```



Vectorización automática ineficiente

■ Alineamiento de memoria

■ Unaligned access:

```
void mult(int N, double* a, double* b, double* c)
{
    int i;
    #pragma omp simd
    for (i = 0; i < N; i++)
        c[i] = a[i] * b[i];
}
```

```
LOOP BEGIN at mult.c(5,3)
<Peeled loop for vectorization>
remark #25015: Estimate of max trip count of loop=3
LOOP END

LOOP BEGIN at mult.c(5,3)
remark #15388: vectorization support: reference c[i] has aligned access [mult.c(6,5)]
remark #15389: vectorization support: reference a[i] has unaligned access [mult.c(6,12)]
remark #15389: vectorization support: reference b[i] has unaligned access [mult.c(6,19)]
remark #15381: vectorization support: unaligned access used inside loop body
...
remark #15449: unmasked aligned unit stride stores: 1
remark #15450: unmasked unaligned unit stride loads: 2
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 8
remark #15477: vector cost: 1.750
remark #15478: estimated potential speedup: 3.890
remark #15488: --- end vector cost summary ---
LOOP END
...
```

■ Aligned access

```
void mult(int N, double* a, double* b, double* c)
{
    int i;
    #pragma omp simd aligned(a,b,c)
    for (i = 0; i < N; i++)
        c[i] = a[i] * b[i];
}
```

```
LOOP BEGIN at mult.c(5,3)
remark #15388: vectorization support: reference c[i] has aligned access [mult.c(6,5)]
remark #15388: vectorization support: reference a[i] has aligned access [mult.c(6,12)]
remark #15388: vectorization support: reference b[i] has aligned access [mult.c(6,19)]
...
remark #15448: unmasked aligned unit stride loads: 2
remark #15449: unmasked aligned unit stride stores: 1
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 8
remark #15477: vector cost: 1.250
remark #15478: estimated potential speedup: 5.260
remark #15488: --- end vector cost summary ---
LOOP END
...
```



Vectorización explícita

- Responsabilidad del **compilador**
 - Permitir que el programador declare que el código puede y debe ejecutarse en SIMD
 - Generar el código que indicó el programador
- Responsabilidades del **programador**
 - Corrección (p. ej. ausencia dependencias, accesos a memoria no válidos)
 - Eficiencia (p. ej. alineación, orden de bucle, máscaras)



Vectorización explícita: ejemplo

reduction.c

```
float sum = 0.0f;
float *p = a;
int step = 4;
#pragma omp simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
    sum += *p;
    p += step;
}
```


- Los dos operadores `+=` tienen significados diferentes
 - El programador debe ser capaz de expresarlos de manera diferente
 - El compilador tiene que generar código diferente
 - ¿Significados de variables `i`, `p` y `step`?



Vectorización explícita: ejemplo

mandel.c

```
#pragma omp declare simd simdlen(16)
uint32_t mandel(fcomplex c)
{
    uint32_t count = 1; fcomplex z = c;
    for (int32_t i = 0; i < max_iter; i += 1) {
        z = z * z + c; int t = cabsf(z) < 2.0f;
        count += t;
        if (!t) { break; }
    }
    return count;
}
```



- La función **mandel()** se llama desde un bucle de X/Y puntos
 - Queremos vectorizar el bucle externo
 - El compilador genera función **vectorial** sobre N valores de c



Antes de OpenMP 5.1 (SIMD)

- Los programadores solían guiar al compilador... o utilizar funciones específicas del vendor
 - Modelos de programación: Intel Cilk Plus
 - Directivas, como `#pragma vector`
 - O a bajo nivel con intrínsecas: `_mm_add_pd()`

guided.c

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```



Construcción SIMD

- Paralelismo de datos con `#pragma omp simd`
[`clause`[[`,`],...]
 - *private*(*var-list*): para valores no inicializados
 - *reduction*(*op*:*var-list*): operación de reducción



Clausulas de la construcción SIMD




safoelen(length): iteraciones en las que no se rompe la dependencia

- *linear(list[:linear-step])*: relación de variable con iterador
 $x_i = x_{orig} + i * linear - step$
- *aligned(list[:alignment])*: especifica el alineamiento
- *collapse(n)*: n bucles “fusionados”



Cooperación con contrucciones OpenMP

- Paralelización y vectorización de bucles anidados
- `#pragma omp for simd [clause[,] clause],...`



```
void ssum(int n, double *a, double *b, double *c) {  
    #pragma omp for simd  
    for (int k=0; k<n; k++)  
        c[k] = a[k] + b[k];  
}
```

