

Producto Interno de un Vector Usando Instrucciones AVX Empacadas

Curso: IC3101 Arquitectura de Computadoras

Semestre: II – 2025

Profesor: Jorge Vargas Calvo

Estudiantes: Justin Lacayo Picado, Nayelith Sojo
Mendez, Federick Fernandez Calderon y Gabriel
Solano Salazar

Fecha de entrega: 4/12/2025

Contenido

[1. Introducción](#)

[2. Fundamento Matemático](#)

[3. Diseño del Algoritmo Paralelo \(SIMD con AVX\)](#)

[4. Descripción Paso a Paso del Algoritmo](#)

[5. Código Fuente Completo \(Ensamblador\)](#)

[6. Pruebas y Validación](#)

[7. Conclusiones](#)

1. Introducción

Este proyecto implementa el producto interno entre dos vectores de seis elementos utilizando instrucciones AVX en ensamblador x86-64. Se emplea paralelismo de datos mediante registros YMM de 256 bits para realizar multiplicaciones y sumas de manera vectorizada, optimizando el rendimiento y practicando el uso de operaciones de punto flotante en formato IEEE 754. El programa fue desarrollado en MASM/ML64 y demuestra técnicas de reducción horizontal y manejo eficiente del estado AVX/SSE.

2. Fundamento Matemático

Dado dos vectores:

$$A = (a_1, a_2, a_3, a_4, a_5, a_6)$$

$$B = (b_1, b_2, b_3, b_4, b_5, b_6)$$

El producto interno se define como:

$$P = \sum_{i=1}^6 a_i b_i$$

Este cálculo requiere seis multiplicaciones y cinco sumas.

Para aprovechar paralelismo AVX, se divide el vector en:

- **Bloque 1:** elementos 0 a 3 (4 valores → encajan en un YMM)
- **Bloque 2:** elementos 4 y 5 (2 valores)

Cada bloque se procesa en paralelo utilizando vmulps.

3. Diseño del Algoritmo Paralelo (SIMD con AVX)

Para calcular el producto interno con el mínimo de instrucciones ensamblador, se diseñó el siguiente algoritmo:

Bloque A (5 elementos)

Este bloque carga en dos partes los vectores. La parte 1 carga los primeros 4 elementos. Utiliza la multiplicación empacada para multiplicar los primeros 4 elementos. Carga la parte 2 para multiplicarlo igual que en la parte 1.

Bloque B (7 elementos)

La subsección reducción 1 y 2 busca sumar todos los elementos obtenidos del bloque A. Utilizando suma vectorial con la parte 1 y parte 2 que se obtuvieron de la multiplicación empacada.

Bloque C (3 elementos) Resultado final

En este bloque se guardan los resultados y se indica el final del código.

Este algoritmo utiliza solamente instrucciones AVX empacadas, cumpliendo la restricción del proyecto.

4. Descripción Paso a Paso del Algoritmo

Paso	Descripción	Instrucciones AVX
1	Cargar primeros 4 elementos del punto 1	<code>vmovupd ymm0, YMMWORD PTR [Punto1]</code>
2	Cargar primeros 4 elementos del punto 2	<code>vmovupd ymm2, YMMWORD PTR [Punto2]</code>
3	Multiplica los 4 primeros elementos de ambos vectores	<code>vmulpd ymm0, ymm0, ymm2</code>
4	Carga los 2 elementos faltantes del punto 1	<code>vmovupd ymm1, YMMWORD PTR [Punto1 + 32]</code>
5	Carga los 2 elementos faltantes del punto 2	<code>vmovupd ymm3, YMMWORD PTR [Punto2 + 32]</code>
6	Multiplica los 2 elementos faltantes De ambos vectores	<code>vmulpd ymm1, ymm1, ymm3</code>
7	Separa los elementos en pares para sumarlos en paralelo	<code>vextractf128 xmm5, ymm0, 1</code>
8	Realiza la suma en parejas de los 4 elementos resultantes del paso 3	<code>vaddpd xmm4, xmm0, xmm5</code>
9	Mueve la tercera pareja	<code>vmovhlps xmm5, xmm4, xmm4</code>
10	Suma la pareja resultante del paso 8	<code>vaddsd xmm4, xmm4, xmm5</code>
11	Mueve un elemento de la pareja de paso 9	<code>vmovhlps xmm6, xmm1, xmm1</code>
12	Suma la pareja de paso 9 (parte 2 de vectores)	<code>vaddsd xmm5, xmm1, xmm6</code>
13	Suma final de todos los elementos	<code>vaddsd xmm0, xmm4, xmm5</code>
14	Guarda el resultado	<code>vmovsd QWORD PTR [Producto], xmm0</code>

5. Código Fuente Completo (Ensamblador)

```
option casemap:none
includelib kernel32.lib
ExitProcess PROTO
.data
; =====
; 1. DEFINICIÓN EXPLÍCITA DE DATOS
; =====

; Vector 1 completo de 8 elementos (usamos solo los primeros 6)
Punto1 REAL8 2.0, -1.0, 4.0, 4.0, 6.0, 6.0, 0.0, 0.0

; Vector 2 completo de 8 elementos (usamos solo los primeros 6)
Punto2 REAL8 0.4, 5.0, 1.5, -2.0, 2.5, 3.0, 0.0, 0.0

; Guardamos el resultado
Producto REAL8 ?

.code
main PROC
; =====
; A. Carga y Multiplicación
; =====

; Cargar la Parte 1 (x1..x4 * y1..y4)
vmovupd ymm0, YMMWORD PTR [Punto1]
vmovupd ymm2, YMMWORD PTR [Punto2]
vmulpd ymm0, ymm0, ymm2 ; ymm0 = (p1, p2, p3, p4)
; Cargar la Parte 2 (x5, x6, 0, 0 * y5, y6, 0, 0)
vmovupd ymm1, YMMWORD PTR [Punto1 + 32]
vmovupd ymm3, YMMWORD PTR [Punto2 + 32]
vmulpd ymm1, ymm1, ymm3 ; ymm1 = (p5, p6, 0, 0)

; =====
; B. Reducción Horizontal y Suma (45.5)
; =====

; 1. Reducción de YMM0 (p1..p4) a XMM4[0]

; Extraer carril superior de ymm0 (p3, p4) a xmm5
vextractf128 xmm5, ymm0, 1 ; xmm5 = [p3, p4]

; Sumar carriles: xmm4 = [p1, p2] + [p3, p4]
vaddpd xmm4, xmm0, xmm5 ; xmm4 = [p1+p3, p2+p4]

; Mover el segundo double a xmm5[0]
vmovhlps xmm5, xmm4, xmm4 ; xmm5[0] = p2+p4

; Suma final de la Parte 1
vaddsd xmm4, xmm4, xmm5
; -----

; 2. Reducción de YMM1 (p5, p6, 0, 0)

; Mover p6 a xmm6[0]
; vshufpd xmm6, xmm1, xmm1, 01b
vmovhlps xmm6, xmm1, xmm1

; Suma final de la Parte 2: p5 + p6
vaddsd xmm5, xmm1, xmm6
; -----

; 3. Suma Total: XMM4[0] + XMM5[0]
vaddsd xmm0, xmm4, xmm5
; =====
; C. Salir
; =====

; Guardar resultado
vmovsd QWORD PTR [Producto], xmm0
vzeroupper
mov ecx, 0
call ExitProcess
main ENDP
END
```

6. Pruebas y Validación

Para validar el funcionamiento del programa, se utilizan valores conocidos donde el producto interno es fácil de verificar manualmente.

$$A = (2.0, -1.0, 4.0, 4.0, 6.0, 6.0)$$

$$B = (0.4, 5.0, 1.5, -2.0, 2.5, 3.0)$$

$$P = 2.0 \cdot 0.4 + -1.0 \cdot 5.0 + 4.0 \cdot 1.5 + 4.0 \cdot -2.0 + 2.5 \cdot 6.0 + 6.0 \cdot 3.0 = 0.8 - 5 + 6 - 8 + 15 + 18 = 26.8$$

Resultado esperado = 26.800000

Resultado del Bloque A:

```
YMM0 = C020000000000000-4018000000000000-C014000000000000-3FE99999999999A
YMM1 = 0000000000000000-0000000000000000-4032000000000000-402E000000000000
```

Lo cual genera los siguientes valores al pasarlos de **Double precision** a decimal:

$$YMM0 = (-8, 6, -5, 0.8) = (4.0 \cdot -2.0, 4.0 \cdot 1.5, -1.0 \cdot 5.0, 2.0 \cdot 0.4)$$

$$YMM1 = (0, 0, 18, 15) = (0, 0, 6.0 \cdot 3.0, 2.5 \cdot 6.0)$$

Resultado de sumar la parte 1 en el Bloque B:

```
YMM4 = 0000000000000000-0000000000000000-C02A000000000000-C018CCCCCCCCCCCD
```

$$YMM4: (0, 0, -13, -6.2) = (0, 0, -13, -8 + 6 - 5 + 0.8).$$

Donde el **-6.2** es el número que nos interesa.

Resultado de sumar la parte 2 en el Bloque B:

```
YMM5 = 0000000000000000-0000000000000000-4032000000000000-4040800000000000
```

$$YMM5 = (0, 0, 18, 33) = (0, 0, 18, 18 + 15)$$

Donde el **33** es el número que nos interesa

El resultado final de sumar la parte 1 y 2 es:

```
YMM0 = 0000000000000000-0000000000000000-C02A000000000000-403ACCCCCCCCCCD
```

$$YMM0 = (0, 0, -13, 26.800000000000001) = (0, 0, -13, 33 + -6.2)$$

Donde el resultado final es **26.800000000000001**

7. Conclusión

El proyecto permitió implementar de manera eficiente el producto interno de dos vectores utilizando instrucciones AVX empacadas, demostrando las ventajas del paralelismo de datos en arquitecturas modernas. Durante el desarrollo se profundizó en la aritmética de punto flotante bajo el estándar IEEE-754 y en la organización interna de los registros vectoriales YMM, lo que fortaleció la comprensión del modelo SIMD.

El uso de instrucciones AVX redujo la cantidad de operaciones necesarias y mejoró el rendimiento en comparación con una implementación escalar tradicional. La correspondencia entre el diseño planteado y la solución implementada fue clara y coherente, evidenciando un adecuado aprovechamiento de las capacidades del conjunto de instrucciones SIMD en x86-64.