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

En este proyecto se implementó el producto interno de dos vectores de longitud seis en $\mathbb{R}^6$ utilizando instrucciones AVX empacadas del conjunto SIMD de procesadores x86-64.
El objetivo principal fue aplicar paralelismo de datos para mejorar el desempeño y practicar el uso de instrucciones de punto flotante bajo el formato IEEE 754 de precisión doble.

El programa fue desarrollado completamente en lenguaje ensamblador x86-64 (NASM/ML64) utilizando registros YMM de 256 bits.

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	vmovupd ymm0, YMMWORD PTR [Punto1]
2	Cargar primeros 4 elementos del punto 2	vmovupd ymm2, YMMWORD PTR [Punto2]
3	Multiplica los 4 primeros elementos de ambos vectores	vmulpd ymm0, ymm0, ymm2
4	Carga los 2 elementos faltantes del punto 1	vmovupd ymm1, YMMWORD PTR [Punto1 + 32]
5	Carga los 2 elementos faltantes del punto 2	vmovupd ymm3, YMMWORD PTR [Punto2 + 32]
6	Multiplica los 2 elementos faltantes De ambos vectores	vmulpd ymm1, ymm1, ymm3
7	Separa los elementos en pares para sumarlos en paralelo	vextractf128 xmm5, ymm0, 1
8	Realiza la suma en parejas de los 4 elementos resultantes del paso 3	vaddpd xmm4, xmm0, xmm5
9	Mueve la tercera pareja	vpermilpd xmm5, xmm4, 00000001b
10	Suma la pareja resultante del paso 8	vaddsd xmm4, xmm4, xmm5
11	Mueve un elemento de la pareja de paso 9	vshufpd xmm6, xmm1, xmm1, 00000001b
12	Suma la pareja de paso 9 (parte 2 de vectores)	vaddsd xmm5, xmm1, xmm6
13	Suma final de todos los elementos	vaddsd xmm0, xmm4, xmm5
14	Guarda el resultado	vmovsd QWORD PTR [Producto], xmm0

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]
    vpermilpd xmm5, xmm4, 00000001b         ; 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, 00000001b

    ; 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 · 0.4 + -1.0 ·5.0 + 4.0 · 1.5 + 4.0 · -2.0 + 2.5 · 6.0 + 6.0 ·3.0 = 0.8 – 5 + 6 – 8 + 15 + 18 = 26.8

Resultado esperado = 26.800000

Resultado del Bloque A:

```
YMM0 = C020000000000000-4018000000000000-C014000000000000-3FE999999999999A
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 · -2.0 , 4.0 · 1.5, -1.0 ·5.0, 2.0 · 0.4)

YMM1 =(0, 0, 18, 15) = (0 , 0, 6.0 ·3.0, 2.5 · 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-C02A000000000000-403ACCCCCCCCCCCD
```

YMM0 = (0, 0, -13 ,26.8000000000000001) = (0, 0 ,-13, 33 + -6.2)

Donde el resultado final es **26.8000000000000001**

7. Conclusiones

El proyecto logró implementar un producto interno de forma eficiente utilizando instrucciones AVX empacadas. Además de lograr comprender mejor la aritmética IEEE-754 y la organización de registros vectoriales YMM. Se utilizó el paralelismo SMD para poder reducir el número de instrucciones y aumentar el rendimiento del programa. La correspondencia entre diseño y código fue clara y precisa.