Universidad de Costa Rica
Escuela de Ciencias de la Computación
Programación Paralela y Concurrente
Prof. Esteban Rodríguez Betancourt

Instrucciones SIMD en C

Esta práctica tiene como objetivo que usted aprenda a utilizar instrucciones SIMD (Single Instruction, Multiple Data) en C utilizando intrinsics y a explorar la autovectorización con gcc en Linux.

Para cada ejercicio, cree un archivo diferente. Para facilitar la compilación, puede crear un archivo Makefile. En todos los archivos, documente de qué forma debe ser compilado.

Ejercicio 1: Introducción a SIMD

Cree un programa en C que sume dos vectores de enteros utilizando intrinsics SIMD. Investigue el uso de las instrucciones _mm_loadu_si128, _mm_add_epi32 y _mm_storeu_si128. Use el siguiente código como base:

```
#include <stdio.h>
#include <stdint.h>
#include <immintrin.h>

int main() {
    int32_t a[4] = {1, 2, 3, 4};
    int32_t b[4] = {5, 6, 7, 8};
    int32_t result[4];

    __m128i va = ________((__m128i*)a);
    __m128i vb = ________((__m128i*)b);
    __m128i vresult = ________(va, vb);

    ________((__m128i*)result, vresult);

    for (size_t i = 0; i < 4; i++) {
        printf("%d ", result[i]);
    }
    return 0;
}
```

Ejercicio 2: Resta de vectores

Copie el programa anterior y modifíquelo para restar dos vectores de enteros. Investigue el uso de _mm_sub_epi32.

Universidad de Costa Rica
Escuela de Ciencias de la Computación
Programación Paralela y Concurrente
Prof. Esteban Rodríguez Betancourt

Ejercicio 3: Producto punto

Calcule el producto punto de dos vectores de enteros usando SIMD. Investigue el uso de _mm_mullo_epi32. La reducción final de cada elemento de 32 bits deben hacerla con código en C típico, ya que no existe una instrucción _mm_reduce_add_epi32 (la más parecida requiere AVX512).

Ejercicio 4: Carga y almacenamiento alineados

Instrucciones:

1. Use como base el código del ejercicio 1
2. Cambie _mm_loadu_si128 por _mm_load_si128
3. Cambie _mm_storeu_si128 por _mm_store_si128
4. Asegúrese de que los datos estén alineados a 16 bits. Para ello utilice __attribute__((aligned(16)))

Ejercicio 5: Multiplicación de matrices 2x2

Realice la multiplicación de dos matrices 2x2 utilizando intrinsics SIMD. Utilice el siguiente código:

```
#include <stdio.h>
#include <stdint.h>
#include <immintrin.h>

int main() {
    int32_t A[2][2] = {{1, 2}, {3, 4}};
    int32_t B[2][2] = {{5, 6}, {7, 8}};
    int32_t C[2][2];

    __m128i row1 = _mm_set_epi32(0, 0, A[0][1], A[0][0]);
    __m128i row2 = _mm_set_epi32(0, 0, A[1][1], A[1][0]);
    __m128i col1 = _mm_set_epi32(0, 0, B[1][0], B[0][0]);
    __m128i col2 = _mm_set_epi32(0, 0, B[1][1], B[0][1]);

    __m128i mul1 = _mm_mullo_epi32(row1, col1);
    __m128i mul2 = _mm_mullo_epi32(row1, col2);
    __m128i mul3 = _mm_mullo_epi32(row2, col1);
    __m128i mul4 = _mm_mullo_epi32(row2, col2);
```

```
    C[0][0] = _mm_extract_epi32(mul1, 0) + _mm_extract_epi32(mul1, 1);
    C[0][1] = _mm_extract_epi32(mul2, 0) + _mm_extract_epi32(mul2, 1);
    C[1][0] = _mm_extract_epi32(mul3, 0) + _mm_extract_epi32(mul3, 1);
    C[1][1] = _mm_extract_epi32(mul4, 0) + _mm_extract_epi32(mul4, 1);

    for (size_t i = 0; i < 2; i++) {
        for (size_t j = 0; j < 2; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Ejercicio 6: Mayor elemento de un array

Cree dos funciones que calculen el mayor elemento de un array de enteros de 32 bits. Ambas funciones deben recibir un array de int32_t (importen limits.h) y un size_t con el tamaño del array. Una impleméntela usando instrucciones SIMD (con registros de 128 bits) y otra impleméntela usando un for tradicional. Para la versión SIMD, considere usar _mm_set1_epi32 para setear el valor inicial de los registros en INT32_MIN (importe limits.h), _mm_load_si128 para cargar datos del array y _mm_max_epi32 para obtener los elementos máximos uno a uno. Luego usen _mm_storeu_si128 para guardar el registro en un array normal (en la pila). Finalmente, usen programación tradicional para obtener el mayor elemento tanto del array final que crearon con _mm_storeu_si128 como de los elementos restantes que no pudieron procesar con _mm_max_epi32.

Paso 0

1	2	3	4	5	6	7	8	9	10
-infty	-infty	-infty	-infty						

Paso 1

1	2	3	4	5	6	7	8	9	10
1	2	3	4						

Universidad de Costa Rica
Escuela de Ciencias de la Computación
Programación Paralela y Concurrente
Prof. Esteban Rodríguez Betancourt

Paso 2

1	2	3	4	5	6	7	8	9	10
				1	2	3	4		

Paso 4

1	2	3	4	5	6	7	8	9	10
				5	6	7	8		

Paso 5

Buscar el mayor entre lo que quedó al final en el registro (5, 6, 7, 8) y lo que no se procesó con SIMD (9 y 10).

Compile el programa y luego decompílelo usando objdump -d. Observe las diferencias en el código generado. ¿El compilador logró optimizar la versión “tradicional”?

Ejercicio 7: Multiplicación escalar

Implemente la multiplicación por un escalar. Cree tanto una versión que use SIMD como una tradicional. Ambas funciones deben recibir un array de int32_t (importen limits.h) y un size_t con el tamaño del array. Utilicen _mm_loadu_si128 para cargar los datos, _mm_set1_epi32 para guardar el escalar en cada posición del registro, _mm_mullo_epi32 para realizar la multiplicación y finalmente _mm_storeu_si128 para guardar el resultado al array.

Compile el programa y luego decompílelo usando objdump -d. Observe las diferencias en el código generado. ¿El compilador logró optimizar la versión “tradicional”?

Ejercicio 8: Autovectorización

Compile el siguiente código con las siguientes opciones en gcc:

- -O0 (o cero)
- -O3 -ftree-vectorize -fopt-info-vec-optimized

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

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

int main() {
    int a[4] = {1, 2, 3, 4};
    int b[4] = {5, 6, 7, 8};
    int result[4];

    add_vectors(a, b, result, 4);

    for (int i = 0; i < 4; i++) {
        printf("%d ", result[i]);
    }
    return 0;
}
```

Guarde los dos ejecutables. Luego, decompílelos usando objdump -d y reporte las diferencias.

Si compila los ejercicios 6 y 7 con los flags de optimización sugeridos en este ejercicio: ¿el compilador logra optimizar mejor la versión “tradicional”? Reporten los resultados.

Ejercicio 9: Reducción paralela

Considere el siguiente código en C para realizar reducción paralela:

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

int main() {
    int a[4] = {1, 2, 3, 4};
    __m128i va = _mm_loadu_si128((__m128i*)a);
    va = _mm_hadd_epi32(va, va);
    va = _mm_hadd_epi32(va, va);

    int sum = _mm_extract_epi32(va, 0);

    printf("Suma total: %d\n", sum);
```

```
    return 0;
}
```

Investigue el funcionamiento de la función _mm_hadd_epi32. Una vez que entienda cómo funciona, implemente una función que sume todos los elementos de un array. Igual que en el resto de la práctica, considere arrays de int32_t y reciba el tamaño en un size_t. Sume los elementos de 4 en 4 usando _mm_add_epi32, luego sume el resultado acumulado en el registro SSE usando la suma parcial del código anterior. Finalmente, sume los elementos que no pudo procesar usando SIMD.