TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica
Área Académica de Ingeniería en Computadores
CE 4302 — Arquitectura de Computadores II

Taller 3:
Extensión SIMD: SSEx

Fecha de asignación:	19 abril 2023	Fecha de entrega:	26 abril 2023
Grupo:	2 personas	Profesor:	Luis Barboza Artavia

Integrantes

Carlos Adrian Araya Ramirez	2018319701
Michael Shakime Richards Sparks	2018170667

1. Investigación

Para comprender mejor extensiones SIMD, realice una pequeña búsqueda para responder las siguientes preguntas:

1. ¿En qué consiste un extensión SIMD llamada SSE?

Según menciona el manual de arquitectura de software [1], SSE (Streaming SIMD Extensions) es una extensión SIMD para los procesadores Intel que se introdujo por primera vez en 1999 con los procesadores Pentium III. Básicamente es una herramienta que ayuda a los procesadores Intel a realizar operaciones simultáneas en múltiples elementos de datos en paralelo, esto se logra a través de un conjunto de instrucciones que pueden manejar datos en formatos de punto flotante y enteros de 128 bits.

Actualmente, existen varias versiones: SSE2, SSE3, SSSE3, SSE4, SSE4.1 y SSE4.2, cada una incorpora instrucciones y características para mejorar aún más el rendimiento del procesamiento en paralelo.

2. ¿Cuáles tipos de datos son soportados por este tipo de instrucciones?

En la sección "4. Data Types" de [1], se describen los diferentes formatos de datos que pueden ser manejados por las instrucciones SSE y SSE2. En particular, SSE permite manejar datos en formatos de punto flotante de 32 y 64 bits, enteros de hasta 64 bits, y también datos empaquetados de 128 bits. Los datos empaquetados se refieren a datos enteros que están agrupados en paquetes de

128 bits, lo que significa que se pueden procesar múltiples elementos de datos al mismo tiempo con una sola instrucción.

Los datos empaquetados también pueden contener combinaciones de enteros y datos de coma flotante. SSE2, por su parte, amplía la capacidad para manejar datos punto flotante de precisión doble (aún más grandes), con 128 bits por registro.

3. ¿Cómo se realiza la compilación de un código fuente en C que utilice el set SSEx de Intel?

Primero se debe incluir en el código fuente la librería de cabecera "xmmintrin.h", que es la que contiene las definiciones de las funciones y tipos de datos SSE. Luego para la compilación como tal depende del compilador:

- Para compilar con SSE en el compilador GCC de Linux, se pueden agregar flags de compilación específicas, como "-msse" o "-msse2", según el conjunto de instrucciones que se desea utilizar.
- Para compilar con SSE en el compilador Visual C++ de Windows, se pueden agregar opciones de compilación específicas, como "/arch:SSE" o "/arch:SSE2".

4. ¿Qué importancia tienen la definición de variables y el alineamiento de memoria al trabajar con un set SIMD vectorial, como SSE?

En SSE el correcto uso de variables y el alineamiento de memoria son fundamentales para lograr un buen rendimiento y eficiencia al trabajar con conjuntos SIMD vectoriales, debido a que estos conjuntos de instrucciones están diseñados para operar en paralelo en múltiples elementos de datos, de forma tal que para simplificar el proceso, los datos deben estar alineados de manera adecuada en la memoria para que los accesos a memoria sean eficientes y no se produzcan interrupciones en el rendimiento [2].

2. Análisis

El código fuente `helloWorld.c` muestra un ejemplo de cómo utilizar el set de instrucciones que se encuentra en el siguiente enlace, por lo que debe realizar lo siguiente:

1. Explicar las variables `oddVector`, `evenVector` y los diversos `data` en términos de cómo se definieron (instrucción utilizada), el tipo de dato que representan y por qué la cantidad de argumentos.

En el código proporcionado se definen dos variables, **oddVector** y **evenVector**, que son de tipo **__m128i**, el cual es un tipo de datos específico de SSE que representa un vector de 128 bits con elementos enteros de 32 bits [3].

Ambos vectores se utilizan en la operación de sustracción de vectorial que se realiza con la función **_mm_sub_epi32**, el resultado se almacena en la variable **result** que de igual forma es un vector.

Estas variables se crean utilizando **_mm_set_epi32**, la cual es una función de SSE que establece un vector de 128 bits a partir de 4 enteros empaquetados de 32 bits con los valores proporcionados. Internamente se distribuye así [3]:

- dst[31:0] := e0
- dst[63:32] := e1
- dst[95:64] := e2
- dst[127:96] := e3

```c
int main()
{
  int data;
  printf("Probando SSE \n");

  __m128i oddVector = _mm_set_epi32(1, 5, 9, 13);
  __m128i evenVector = _mm_set_epi32(12, 14, 16, 18);

  __m128i result = _mm_sub_epi32(evenVector, oddVector);

  printf("Result ********************* \n");

  data = 0;

  data = _mm_extract_epi32(result, 0);
  printf("%d \t", data);

  data = _mm_extract_epi32(result, 1);
  printf("%d \t", data);

  data = _mm_extract_epi32(result, 2);
  printf("%d \t", data);

  data = _mm_extract_epi32(result, 3);
  printf("%d \t", data);

  printf("\n");
  return 1;
}
```

La variable **data** de tipo **int** se define para almacenar temporalmente los valores de los enteros empaquetados en **result**. Esto se logra con la función **_mm_extract_epi32** que permite extraer cada uno de los cuatro elementos del vector utilizando el índice.

2. ¿Qué sucede si las variables `data` se imprimieran con un ciclo y no uno por uno? ¿Por qué ocurre eso?

Al utilizar la función **_mm_extract_epi32** se puede acceder directamente a los elementos del registro SSE sin necesidad de desplazamiento de bits, por lo que se facilita la extracción de los valores de manera precisa.

En cambio, si se utiliza un ciclo para extraer los elementos del vector, es necesario realizar un desplazamiento de bits para acceder a cada elemento del vector SSE, porque no funciona como un arreglo. Asimismo, como cada variable **data** solo puede almacenar un entero de 32 bits, solo se puede imprimir un elemento por iteración del ciclo.

3. Compile el código fuente y adjunte una captura de pantalla con el resultado de la ejecución.

```
jey@jey:~/Documents/Arquitectura2/Taller3/Entregable$ make helloworld
make: Warning: File 'helloWorld.c' has modification time 28107 s in the future
gcc -msse -msse4.1 -lrt -lm -w -o bin/helloworld helloWorld.c
./bin/helloworld
Probando SSE
Result **********************
5       7       9       11
```

3. Ejercicios prácticos

1. Realice un programa en C que busque el elemento mayor en cada columna de una matriz y lo guarde en un vector. La matriz corresponde a 4x3 y está compuesta por enteros (32 bits). Para este programa se desea que todos los números sean positivos para realizar la comparación. El usuario ingresa todos los números como parámetros y debe imprimir las tres filas, así como el resultado de cada mayor.

```c
#include <emmintrin.h>
#include <immintrin.h>
#include <xmmintrin.h>
#include <smmintrin.h>

#include <stdio.h>
#include "colors.h"

// gcc -o problema2 problema2.c

int main()
{
  // variable for prints
  int data;

  // user's inputs
  int input[12];
  bold_yellow();
  printf("\n\nIngrese numeros positivos de la matriz: \n");
  for (int i = 0; i < 12; i++)
  {
    scanf("%d", &input[i]);
  }

  // matrix declaration with user's inputs
  __m128i row0 = _mm_set_epi32(input[3], input[2], input[1], input[0]);
  __m128i row1 = _mm_set_epi32(input[7], input[6], input[5], input[4]);
  __m128i row2 = _mm_set_epi32(input[11], input[10], input[9], input[8]);

  // max_values
  __m128i max_first_two_rows = _mm_max_epi16(row0, row1);
  __m128i max_values = _mm_max_epi16(max_first_two_rows, row2);

  // print matrix
  bold_blue();
  printf("\nLa matriz ingresada fue:\n");
  __m128i matriz[3] = {row0, row1, row2};

  for (int i = 0; i < 3; i++)
  {
    // Convert the pointer to matriz[i] to a pointer to int using the expression (int *)&matriz[i].
    // Then create a row pointer that points to the first 4 bytes of row 'i'.
    int *row = (int *)&matriz[i];
    // 32-bit right shift to extract each int
    printf("%d \t%d \t%d \t%d \t\n", row[0], row[1] >> 32, row[2] >> 32, row[3] >> 32);
  }
  printf("\n");

  // print max_values
  bold_green();
  printf("El valor mayor ingresado de cada columna es:\n");
  // Convert the pointer to matriz[i] to a pointer to int using the expression (int *)&matriz[i].
  // Then create a row pointer that points to the first 4 bytes of row 'i'.
  int *row = (int *)&max_values;
  printf("%d \t%d \t%d \t%d \t\n\n", row[0], row[1] >> 32, row[2] >> 32, row[3] >> 32);
  default_color();
  return 0;
}
```

2. Realice un programa en C que realice la multiplicación de un vector de 4 números enteros por una matriz 4x4.

```c
#include <stdio.h>
#include <immintrin.h>
#include <emmintrin.h>
#include "colors.h"

// Prints
void print_results(__m128i vector, __m128i matrix[], __m128i result)
{
  // print matrix
  bold_blue();
  printf("\nLa matrix es:\n");
  for (int i = 0; i < 4; i++)
  {
    // Convert the pointer to matrix[i] to a pointer to int using the expression (int *)&matrix[i].
    // Then create a row pointer that points to the first 4 bytes of row 'i'.
    int *row = (int *)&matrix[i];
    // 32-bit right shift to extract each int
    printf("%d \t%d \t%d \t%d \t\n", row[0], row[1] >> 32, row[2] >> 32, row[3] >> 32);
  }
  printf("\n");

  // print vector
  bold_magenta();
  printf("El vector es:\n");
  int *row = (int *)&vector;
  printf("%d \t%d \t%d \t%d \t\n", row[0], row[1] >> 32, row[2] >> 32, row[3] >> 32);
  printf("\n");

  // print results
  bold_green();
  printf("El vector resultante es:\n");
  // Convert the pointer to matrix[i] to a pointer to int using the expression (int *)&matrix[i].
  // Then create a row pointer that points to the first 4 bytes of row 'i'.
  row = (int *)&result;
  printf("%d \t%d \t%d \t%d \t\n\n", row[0], row[1] >> 32, row[2] >> 32, row[3] >> 32);
  default_color();
}

int main()
{

  // Vector
  __m128i vector = _mm_set_epi32(3, 2, 1, 0);

  // matrix
  __m128i matrix[4] = {
      _mm_set_epi32(3, 2, 1, 0),
      _mm_set_epi32(7, 6, 5, 4),
      _mm_set_epi32(11, 10, 9, 8),
      _mm_set_epi32(15, 14, 13, 12)};

  // Multiply matrix rows with vector
  __m128i prod[4] = {
      _mm_mullo_epi16(vector, matrix[0]),
      _mm_mullo_epi16(vector, matrix[1]),
      _mm_mullo_epi16(vector, matrix[2]),
      _mm_mullo_epi16(vector, matrix[3])};

  // Sum all rows to get result
  __m128i sum01 = _mm_hadd_epi32(prod[0], prod[1]); // add row 0 y row 1
  __m128i sum23 = _mm_hadd_epi32(prod[2], prod[3]); // add row 2 y row 3
  __m128i result = _mm_hadd_epi32(sum01, sum23);    // result

  // Print results
  print_results(vector, matrix, result);

  return 0;
}
```

Resultados:

Ejercicio 1

```
jey@jey:~/Documents/Arquitectura2/Taller3$ make ejercicio1
gcc -msse -lrt -lm -w -o bin/ejercicio1 ejercicio1.c
./bin/ejercicio1


Ingrese numeros positivos de la matriz:
100 200 300 400 900 40 500 1000 700 423 230 603

La matriz ingresada fue:
100     200     300     400
900     40      500     1000
700     423     230     603

El valor mayor ingresado de cada columna es:
900     423     500     1000
```

Ejercicio 2

```
jey@jey:~/Documents/Arquitectura2/Taller3/Entregable$ make ejercicio2
gcc -msse -msse4.1 -lrt -lm -w -o bin/ejercicio2 src/ejercicio2.c
./bin/ejercicio2

La matrix es:
0       1       2       3
4       5       6       7
8       9       10      11
12      13      14      15

El vector es:
0       1       2       3

El vector resultante es:
14      38      62      86
```

4. Referencias

[1] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture, 1st ed.," 2023. https://cdrdv2.intel.com/v1/dl/getContent/671200.

[2] "Memory alignment requirements for SSE and AVX instructions," blog.ngzhian.com. https://blog.ngzhian.com/sse-avx-memory-alignment.html.

[3] "Intel® Intrinsics Guide," Intel. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=SSE_ALL&ig_expand=6358.