PRÁCTICA AC – FASE III

Ejercicios con instrucciones SIMD (SSE)

Realizada por **Pavel Razgovorov** (pr18@alu.ua.es) con N.I.E. Y0888160-Y del **grupo 1** de prácticas (lunes 13:00-15:00)

Índice

Introducción

En esta fase veremos en profundidad el funcionamiento del **set de instrucciones SSE** (y MMX) y su implementación en una serie de algoritmos para comparar su rendimiento con el de la implementación en C estándar. Pero antes, veamos en qué consisten y conozcamos algo de su historia.

Motivación

El uso de datos en **coma flotante**, muy utilizado en **aplicaciones multimedia**, siempre ha sido un aspecto complicado para las arquitecturas de los ordenadores, pues su manejo es difícil e ineficiente. Para esto, los diseñadores siempre han tratado de proveer un tratamiento especial para este tipo de datos, puesto que con los registros de propósito general no se puede trabajar -de manera directa-. Incluso en algunas ocasiones, con tal de evitar ralentizar el funcionamiento de un CPU, se ha utilizado el uso de operaciones en **coma fija**, como es el caso de la videoconsola **PlayStation**, la cual en su primera versión (que supuso un gran avance en los gráficos 3D) no incluía unidad de coma flotante [Vía].

Historia

Los primeros intentos de implementar una unidad de coma flotante dedicada se dieron en los años 80; destacando el conjunto de instrucciones x87 [Vía, Vía], todavía presente en las máquinas actuales. Se caracterizaba por tener 8 registros, del st0 al st7, de tamaño 80 bits y que trabajaban como una pila. Debido al tamaño de los registros, no podían interactuar con los de propósito general, así que las operaciones se realizaban o bien entre ellos mismos o bien a través de la memoria.

Más tarde, en enero 1997, Intel presenta su conjunto de instrucciones **MMX** [Vía, Vía]. Introdujeron 8 nuevos registros de 64 bits, del mm0 al mm7, que en realidad son referencias a los registros de la Unidad de Coma Flotante (Floating Point Unit, FPU) del x87; se diferenciaban en que no tenían la estructura de una pila. Debido a esto, **no era posible usar ambas a la vez**. Este set de instrucciones no tuvo mucho éxito debido a los problemas que presentaba, pero, sin embargo, supuso un importante avance respecto a la arquitectura SIMD ya que introdujo el concepto del **empaquetado**: en cada registro de 64 bits pueden caber, por ejemplo, 2 datos de 32 bits, incluyendo operaciones disponibles para hacerlas de 2 en 2 [Vía]. De esta manera, se podía operar sobre dos datos distintos con una sola instrucción.

Un año después, AMD presenta su propio set de instrucciones, el **3DNow** [Vía], que se suponía mejor y con menos problemas que el MMX. Este set, al igual que su rival, tampoco llegó a tener mucho éxito, aunque esta vez fue debido al poco mercado del que disponía (y dispone) la empresa. Actualmente, AMD ha calificado este set de instrucciones como **obsoleto**.

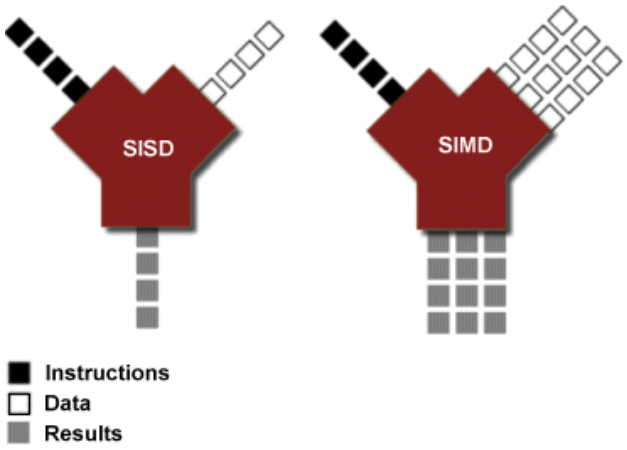
Ya en 1999, Intel vuelve a presentar un nuevo set de instrucciones, el **SSE** [Vía, Vía], que supondría una extensión al set MMX. De nuevo se introducen 8 registros dedicados, del xmm0 al xmm7 (para 64 bits son 16, hasta el xmm15), con las diferencias (ventajas) de que éstos son de 128 bits y que son independientes del x87. Con el paso del tiempo, Intel siguió desarrollando revisiones de este set, lanzando así el SSE2 (2001), SSE3 (2004), SSSE3 (2006) y SSE4 (2006-2007) que se subdivide en SSE4.1 y SSE4.2. Actualmente, es el set de instrucciones SIMD más famoso y extendido, pues cualquier CPU x86-64 moderna dispone de, al menos, SSE y SSE2.

Los últimos avances en este campo son los del set de instrucciones **AVX** [Vía], presentado en **2008** (aunque no implementado hasta 2011) en los que se introducen otros 16 nuevos registros de 256 bits, que pasan a llamarse ymm0-ymm15. Futuras novedades están previstas, ya que recientemente se ha lanzado **AVX2**, el cual introduce operaciones interesantes como la multiplicación-y-suma fusionada [Vía] (útil para, por ejemplo, multiplicación de matrices); también se espera para dentro de poco el lanzamiento de **AVX-512**, que incluirá 32 nuevos registros de 512 bits (del zmm0 al zmm31).

También podemos encontrar interesantes alternativas para otras arquitecturas que no sean la x86, como puede ser **Altivec** [Vía] para **PowerPC**, **NEON** [Vía] para **ARM** o incluso **MSA** [Vía] para **MIPS**.

¿Cómo implementa SSE la arquitectura SIMD?

Como todos sabemos, la arquitectura SIMD trata de realizar una sola operación (una sola instrucción) en un conjunto múltiple de datos (a diferencia del tradicional SISD, que solo opera con un dato por instrucción).



SSE logra esto gracias a los **datos "empaquetados"**. Esto es, que puede guardar dentro de su registro de 128 bits varios datos a la vez:

- 2 datos de 64 bits
- 4 datos de 32 bits
- 8 datos de 16 bits
- 16 datos de 8 bits

Disponiendo después de instrucciones específicas para operar los datos según su distribución.

La mayoría de compiladores modernos pueden utilizar estas instrucciones, pero no siempre son capaces de hacerlo y muchas veces se implementan mediante ensamblador.

Testeo de algoritmos

A continuación, analizaremos una serie de algoritmos implementados en SSE y resolveremos las cuestiones que se nos plantean:

Bubble Sort

¿Qué hace el programa?

Es una implementación del famoso algoritmo del **ordenamiento de burbuja** con números decimales (double) hecho con instrucciones SSE.

¿Se utilizan instrucciones MMX o SSE?

Las únicas instrucciones utilizadas son las del ensamblador x86 estándar y las de **SSE**; las MMX no aparecen (ni siquiera se utilizan sus registros dedicados)

Comenta el funcionamiento de la función sortDoublesSSE (Int32 byteCount, double* values)

Es una función que recibe el array de números y la cantidad de bytes que ocupa el mismo. Con estos datos, se va recorriendo hasta llegar a su límite y se van ordenando de 4 en 4*.

Para ello, se copian los números de dos en dos (en un registro de 128 bits caben 2 datos de 64 bits) en los registros 0-1 y 2-3. Los números se copian en dos registros para, a continuación, cambiar de orden los números en uno de ellos con la finalidad de hacer de manera correcta y eficiente la comparación. También se copia el valor del registro xmm0 en xmm4 para guardar el valor que tiene ya que después se necesitará ese registro.

A continuación, se llaman a un par de funciones de mínimo y máximo que lo que realizan es dejar los valores más bajos en el registro xmm0 y los mayores en xmm1 (por eso se copió auxiliarmente el registro xmm0 al xmm4).

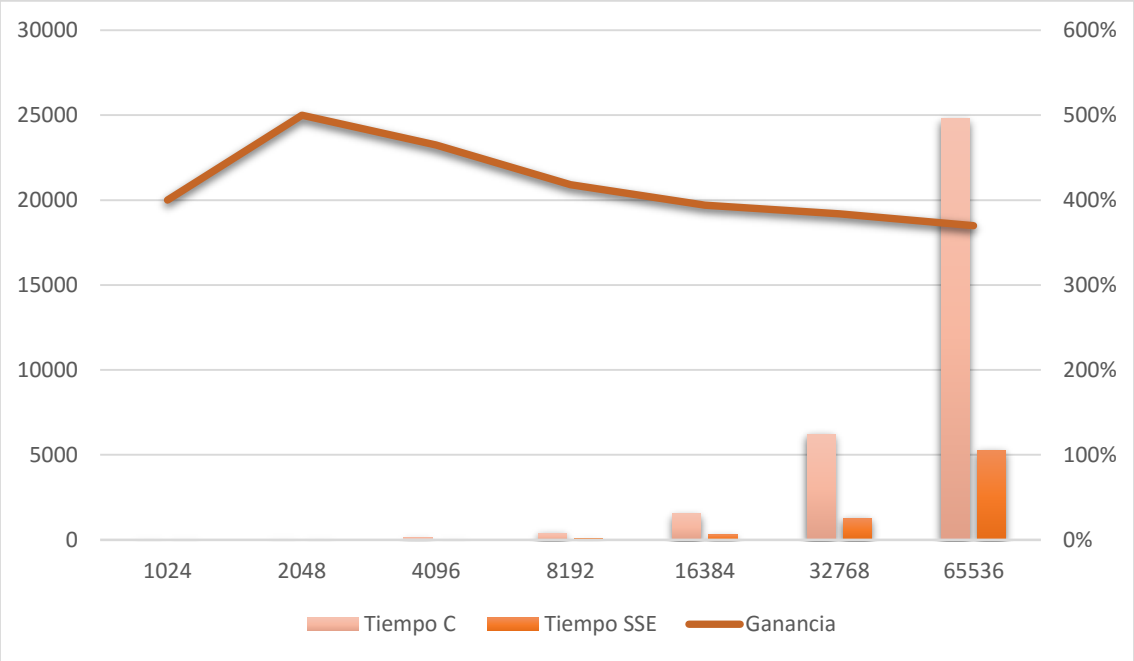
Ahora, una vez tenemos los números ordenados a nivel de paquete, se ordenan a nivel de registro, es decir: ordenamos de 2 en 2 cada registro de forma particular. Esto se hace comparando con los valores del registro con el del registro copia que tiene los números cruzados. Si los 64 bits superiores del registro (el primer número) son menores que los 64 bits superiores del registro copia (el segundo numero), los 64 bits superiores del registro copia se llenan de unos. Lo mismo ocurre, pero al contrario con los 64 bits inferiores de ambos. Después, se copian los bits de signo (el 63 y el 127) hasta un registro de propósito general, el cual tendrá el valor 2 si el primer número era menor que el segundo o 1 si es el caso contrario. En caso de 2, se ordenan los valores volviendo a utilizar la función de cambiar los de orden mencionada anteriormente.

Por último, se devuelven los valores a las posiciones del array que les corresponden, pero esta vez ya ordenados.

*El hecho de que ordene los números de 4 en 4, hace que el algoritmo no funcione para arrays de tamaño 2 o 3.

¿Qué ganancia obtenemos con el algoritmo MMX/SSE con respecto al algoritmo secuencial? Realiza una batería de pruebas y muéstralo utilizando gráficas explicativas.

Tam. Array	Tiempo C	Tiempo SSE	Ganancia
1024	5	1	400 %
2048	24	4	500 %
4096	96	17	465 %
8192	383	74	418 %
16384	1550	314	394 %
32768	6204	1282	384 %
65536	24791	5271	370 %



En la recogida de datos hemos observado que entre ambos algoritmos apenas hay diferencia hasta que llegamos con un tamaño de problema considerable, 1024 elementos. Durante sus potencias de 2 sucesivas, el porcentaje de ganancia se ha mantenido sobre la línea del **400%**, es decir, 5 veces más rápido. También hay que ver que, en valor absoluto, en el problema de tamaño 65536 hay una **diferencia** de casi **20 segundos**, que es muy considerable.

Data Transfer

¿Qué hace el programa?

Copia los datos de un array de enteros a otro utilizando SSE para poder copiar varios datos a la vez.

Explica las siguientes instrucciones: shufpd, cmpltpd, movmskpd

- SHUFPD: **Shuffle Packed Double**. Dados dos registros, se intercambian los valores de éstos en formato [Double, Double]. Si ambos registros resultan ser el mismo, se intercambian los 2 valores entre sí (es el usado en el Bubble Sort SSE).
- CMPLTPD: **Compare Less Than Packed Double**. Compara si un registro [Double, Double] es menor que otro, dejando en el registro de destino 64 unos (true) o 64 ceros (false) en función de si es menor o no.
- MOVMSKPD: **Move Mask Packed Double**. Copia en los dos bits de menor valor de un registro de propósito general los bits de signo de cada uno de los dos Double que hay en el registro.

Estas instrucciones pueden encontrarse aquí.

Explica el funcionamiento de la siguiente función: int DataTransferOptimised (int* piDst, int* piSrc, unsigned long SizeInBytes)

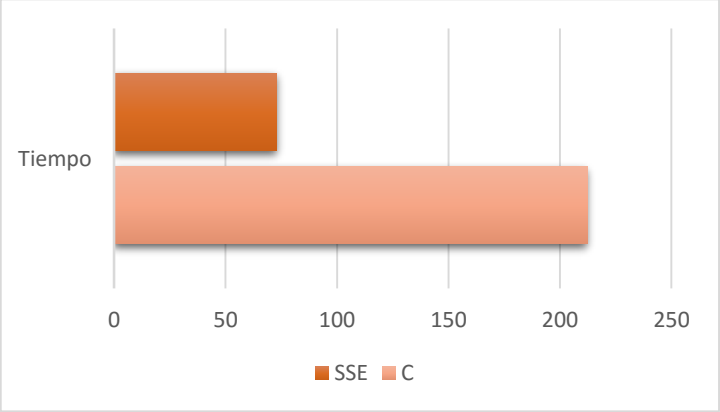
La función recibe un array de enteros como fuente y otro como destino, así como el tamaño del array de ambos.

En la parte de SSE se va recorriendo, del revés, ambos array de enteros hasta llegar al límite de éstos. Lo que se hace es copiar de 4 en 4 los datos del array fuente al registro xmm1 y seguidamente copiar desde el registro al array destino. En pocas palabras, un bucle con una simple asignación.

¿Qué ganancia obtenemos con el algoritmo optimizado mediante extensiones SIMD con respecto al algoritmo secuencial? Realiza una batería de pruebas y muéstralo utilizando gráficas explicativas

Gráficas, las justas. Realizamos la prueba 5 veces y obtenemos su media:

Tiempo C	212,2
Tiempo SSE	73
Ganancia	190 %



Con tan solo un par de líneas de código se consigue una mejora sustancial.

Inner Product

¿Qué hace el programa?

Genera dos vectores de tipo char, short, int, float y double, los llena de valores aleatorios y realiza un **producto escalar** (el sumatorio de multiplicar cada posición) de ambos vectores.

Comenta la siguiente función: float sse3_inner(const float* a, const float* b, unsigned int size)

Recibe ambos arrays a los que se le deberá de aplicar el producto y el tamaño de éstos.

Inicializa el registro xmm0, que actuará de acumulador, a 0. Seguidamente, realiza un bucle* (en C) para recorrer los arrays que debe de multiplicar. Para ello, copia los valores de ambos arrays en un registro (cada uno), los **multiplica** y los **suma**.

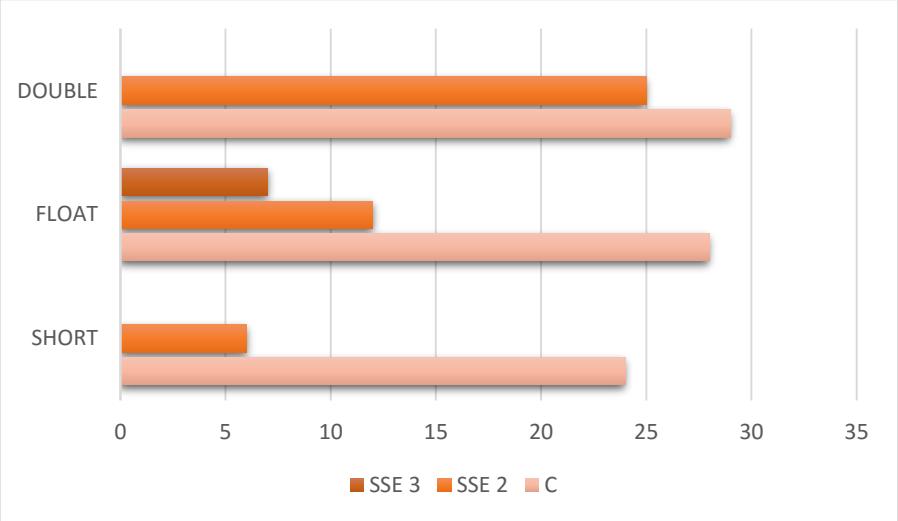
El problema está en que ahora el producto escalar está separado en 4 flotantes que tiene el registro xmm0 acumulados. Así que se hace una "**suma horizontal**" para sumar todos los 4 valores que contiene el registro y dejarlo en uno solo. Por último, deja el resultado en la variable que va a devolver.

*DE NUEVO, EL ALGORITMO REALIZARÁ EL PRODUCTO ESCALAR SÓLO SI EL TAMAÑO DEL ARRAY ES UN PRODUCTO DE 4

¿Qué ganancia obtenemos con el algoritmo optimizado mediante extensiones SIMD con respecto al algoritmo secuencial? Realiza una batería de pruebas y muéstralo utilizando gráficas explicativas

Para que las pruebas sean más fidedignas, aumentamos el tamaño del array en una decena.

SHORT C	SHORT SSE	FLOAT C	FLOATS SSE	FLOAT SSE3	DOUBLE C	DOUBLE SSE
24	6	28	12	7	29	25



- Ganancia short: 300%
- Ganancia float: 133% con SSE 2, 300% con SSE 3
- Ganancia double: 16%

Obviamente, como en un registro SSE caben más datos short que double, el **paralelismo** es mayor, así como su ganancia.

Matrix Vector Mult

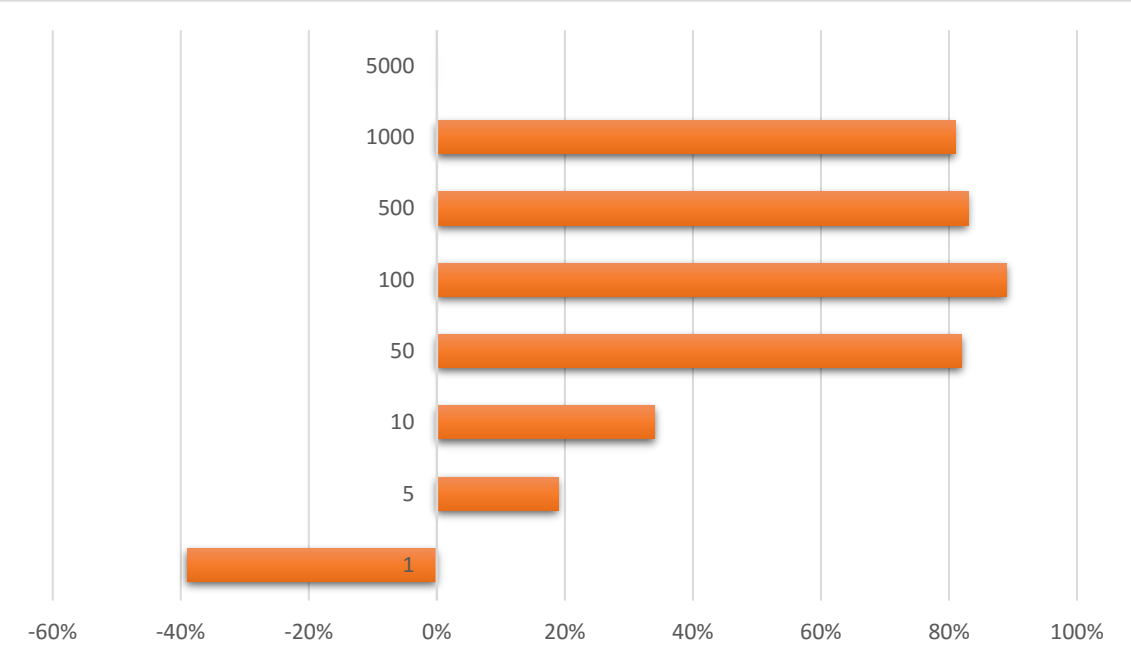
¿Qué hace el programa?

Multiplica una matriz n*n y n*1, devolviendo como resultado una matriz n*1, siendo los datos de las matrices de tipo float.

¿Qué ganancia obtenemos con el algoritmo optimizado mediante extensiones SIMD con respecto al algoritmo secuencial? Realiza una batería de pruebas y muéstralo utilizando gráficas explicativas

Esta vez, en vez de contar los milisegundos, contaremos los ciclos de reloj para una mayor precisión. Lo haremos con una media de 5 muestras.

Tam. Array	Ciclos C	Ciclos SSE	Diferencia	Ganancia
1	85422	118674	-33252	-39%
5	109120	88028	21092	19%
10	175613	115366	60247	34%
50	2254796	411821	1842975	82%
100	9104986	1027122	8077864	89%
500	218975637	36319646	182655991	83%
1000	894181515	170663524	723517991	81%
5000	22245686804	22709979732	-464292928	-2.08%



Podemos observar que, **misteriosamente**, cuando el tamaño de la matriz es muy grande o muy pequeño, no hay ganancia, sino todo lo contrario: ésta **disminuye**. Esto puede deberse a que, aunque los datos se operen con mayor paralelismo, la implementación tiene muchas más instrucciones que ejecutar, debido a que la implementación SISD es mucho más sencilla que la SIMD.

Calificaciones prácticas de grupo

El próximo día que tengamos clase, explico más detalladamente el progreso de cada uno (ahora no tengo tiempo). Aunque los roles de los equipos hayan rotado, sigo estando al mando, y bien contento que estoy con ello. Lo más destacable de esta fase ha sido, primero, que mi compañero Miguel Sánchez haya sido capaz, con mi ayuda, de aprender el lenguaje ensamblador a un buen nivel, habiendo podido a aspirar hasta a formar parte activa del desarrollo (sobre todo la del algoritmo de contraste en ASM) y, segundo, el desarrollo del proyecto en sí: en la práctica grupal aparecen reflejados los resultados, pero desde aquí puedo decir que es simplemente BESTIAL. Me siento bien al saber que he sido capaz de gestionar a los integrantes de mi grupo y poder sacar el máximo partido de ellos y llevar el desarrollo de esta práctica a otro nivel.