UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Organización de Computadores

Laboratorio #3

Integrantes: Néstor Mora
Cristian Espinoza
Profesores: Erika Salas
Nicolás Hidalho
Ayudante: Felipe Garay

Lunes, 08 de Diciembre de 2014

Índice

Referencias **14**

1. Introducción

El presente informe, detalla la elaboración del Laboratorio 3 del ramo Organización de Computadores, el cual consiste en implementar de forma escalar dos algoritmos de funciones matemáticas pedidas (en lenguaje c); una vez realizado esto, se deben implementar de forma vectorial estas dos funciones además de otros dos programas entregados de forma escalar.

Una vez terminada la implementacion de las funciones (tanto escalar como vectorialmente), se realiza un análisis en los tiempos de ejecución de las funciones, comparando las funciones escalares con sus correspondientes funciones vectoriales, analizando los tiempos obtenidos y el porqué se obtienen estos valores.

El objetivo principal es analizar los tiempos de ejecución de las funciones escalares y vectoriales.

Los objetivos específicos son la implementación de ambas funciones escalares y de las cuatro funciones vectoriales, manejar los conceptos de básicos de vectores y cómo se deben utilizar para la elaboración de un programa.

El documento contiene un marco teórico con los conceptos necesarios para poder trabajar, además de la descripción de cómo se realizaron los programas y el análisis de éstos.

2. Marco Teórico

2.1. SIMD

Single **I**nstruction, **M**ultiple **D**ata (una instrucción, múltiples datos), es una técnica empleada para conseguir paralelismo a nivel de datos. Los repertorios SIMD consisten en instrucciones que aplican una misma operación sobre un conjunto más o menos grande de datos. (*SIMD*, 2014)

2.2. SSE

Streaming **S**IMD, **E**xtensions, es una extensión al grupo de instrucciones MMX, las instrucciones SSE son especialmente adecuadas para decodificación de MPEG2, que es el códec utilizado normalmente en los DVD, procesamiento de gráficos tridimensionales y software de reconocimiento de voz.(*SSE*, 2013)

2.3. SSE2

Streaming **S**IMD, **E**xtensions **2**, es uno de los conjuntos de instrucciones de la arquitectura IA-32 SIMD. Sigue el mismo modelo que las utilizadas en los predecesores SSE y MMX manteniendo compatibilidad con esas extensiones, pero amplía su modelo con soporte para paquetes de valores flotantes de precisión doble y para paquetes de enteros de 128 bits. (*SSE2*, 2014)

2.4. SSE3

Streaming **S**IMD, **E**xtensions **3**, es la tercera generación de las instrucciones SSE para la arquitectura IA-32. SSE3 añade 13 nuevas instrucciones a SSE2. (*SSE3*, 2013)

2.5. Make

El comando de linux make ayuda a compilar los programas. Presenta muchas ventajas para programas grandes, en los que hay muchos ficheros fuente (muchos .c y muchos .h) repartidos por varios directorios (*Conceptos básicos de make y los Makefile*, 2007)

2.6. Getopt

Analiza los argumentos de la línea de órdenes. Sus argumentos argc y argv son el número y el vector de argumentos como los pasados a la función main() cuando se ejecuta el programa. (*GETOPT*, 1998)

2.7. Biblioteca

La biblioteca, o también mal conocida como librería (del ingles library) nos permite el uso de funciones en un programa sin la necesidad de escribir su código en nuestro programa, únicamente llamando a la biblioteca donde está contenida. (*Manual para para crear tu propia biblioteca en C/C++*, 2012)

3. Desarrollo

Lo primero que se realizó fue una investigación acerca de los conceptos necesarios para realizar el laboratorio, tales como SIMD y todas las funciones y definiciones asociadas a este concepto; ademas de repasar contenidos como Make, Getopt, entre otros; que si bien se han comprendido y utilizado, es necesario retomar esa información para una buena elaboración del laboratorio.

Después de comprender todos los conceptos, se procedió a elaborar los algoritmos escalares solicitados en las funciones 1 y 2.

3.1. Función 1 - Escalar

$$\sum \sqrt{V_i}^{V_i}$$

Lo primero que se realizó, fue la implementación de la sumatoria utilizando un ciclo for para ésta, al tener que realizarse también un exponente, se decidió hacerlo con un ciclo for, ya que se debe realizar tantas veces como el número que se esté tomando.

3.2. Función 2 - Escalar

$$\sum V_i * V_{i+1} \forall i < |V|$$

Para la implementación de esta ecuación, fue necesaria la implementación de una función *módulo*, la cual se encarga de obtener el módulo del arreglo. Una vez obtenido esta

pudimos implementar la sumatoria hasta el número correspondiente.

Luego de implementar ambas funciones de forma escalar, fue necesaria la implementación de las mismas funciones de forma vectorial. Para esto, es necesario comprender como funcionan los vectores y como p̈asarëstas funciones a vectores.

3.3. Función 1 - Vectorial

Se decidió crear funciones auxiliares que realizaran las acciones necesarias:

- **parte entera:** calcula parte entera para evtar problemas en la implementacion de la funcion pow
- **compare:** funcion que compara uno a uno los elementos en un vector
- **pow:** eleva el arreglo a con exponencial b $\quad \forall\ a, b:\ a \in R, b \in Z^+$
- **pow2:** eleva el arreglo a con exponencial b, sin expeciones pero no de forma paralela

Pero se decidió no utilizar *pow* debido a que es mas propenso a fallar y entrega valores menos precisos (en comparación a pow2). Además se utiliza la función *sumar* donde se realiza el ciclo for que obtiene la raiz cuadrada y además utiliza las otras funciones.

3.3.1. Implementación

Para trabajarlo de manera vectorial se trabaja con el tipo de dato *__m128* representada por 4 flotantes de 32 bit, de esta manera se pudieron realizar las operaciones de raíz cuadrada y sumas de forma paralela, reduciendo la cantidad operaciones realizadas por el procesador para poder reducir el tiempo de respuesta. Se intento implementar una forma de realizar la operación potencia en SIMD pero debido al tipo de datos y las pocas operaciones implementadas en SSE, SSE2 y SSE3 el cálculo perdía mucha precicion haciendo que el

cálculo posterior se viera muy afectado, debido a esto se optó por implementar una función pow2 que obtenia los 4 datos y los procesaba con la funcion powf() de la libreria estándar math.h de forma secuencial pero con menor perdida de presición.

3.4. Función 2 - Vectorial

Se decidió crear funciones auxiliares que realizaran las acciones necesarias:

- **módulo:** se utiliza para obtener el valor del módulo del arreglo
- **compare:** funcion que compara uno a uno los elementos en un vector
- **sumar:** fes la encargada de realizar el ciclo para realizar la sumatoria, además de invocar las funciones ya mencionadas

3.4.1. Implementación

En la segunda función fue mas sencillo implementar operaciones SIMD, por ejemplo, para el cálculo del módulo se ocupó la función _mm_mul_ps multiplicando el dato por si mismo, consiguiendo el cuadrado, luego estos cuadrados se sumaron de forma paralela con la función _mm_sum_ps a un vector acumulador, finalmente se sumaron los cuatro valores contenidos en el vector acumulador, y se calculó la raiz cuadrada, haciendo que el cálculo del módulo reduciera la cantidad iteraciones 4 veces. Una vez obtenido el módulo el cálculo de la sumatoria se realizó de la misma manera anterior con la función _mm_sum_ps y luego sumando los cuatro valores del vector.

3.5. Programa 1 - Vectorial

Aquí fue un trabajo de transformar las funciones entregadas a funciones que trabajen con vectores:

- **compare:** funcion que compara uno a uno los elementos en un vector
- **pow2:** eleva el arreglo a con exponencial b, sin expeciones pero no de forma paralela
- **calcular:** es la encargada de realizar los ciclos for, *calcular* la raiz cuadrada e invocar las otras funciones necesarias para la realización del programa

3.5.1. Implementación

Esta función fue un poco mas compleja de implementar en SIMD ya que contiene 2 ciclos, entonces se tuvo que generar 2 vectores acumuladores, al finalizar el primer ciclo, se sumó lo acumulado en el primer vector y con este se generó un nuevo vector que contenía 4 veces el dato acumulado para ser operado en el segundo ciclo, una ves concluido el segundo ciclo se pudo obtener el valor del resultado al sumar todos los valores en el vector.

3.6. Programa 2 - Vectorial

Al igual que programa 1, lo que se realizó fue "pasar.ª vectores el programa entregado; en esta ocasión *calcular* es la función encargada de realizar toda la acción.

3.6.1. Implementación

En esta ocación la implementación de forma vectorial fue un poco mas compleja ya que no existían funciones que operasen directamente un vector de enteros(*int*), razon por la cual se trabajo con el tipo de dato *__m128i* que es representado por dos *long long* de 64 bit, y su función de carga de datos _mm_setr_epi32 a la cual ingresan 4 enteros de 32 bit, lo complejo viene al momento de obtener los datos de este vector, ya que no existe función implementada en las librerías SSE que obtenga los datos enteros, para esto se ocupa un casteo de punteros, se declara un puntero del tipo *int32_t* que apunta a la direccion de memoria del vector que es casteada al tipo *int32_t**, de esta manera el puntero de tamaño entero posee un

tamaño de 128 bit, lo que vendría a ser un arreglo de enteros de tamaño 4, de esta manera se puede acceder facilmente a los datos almacenados en el vector.

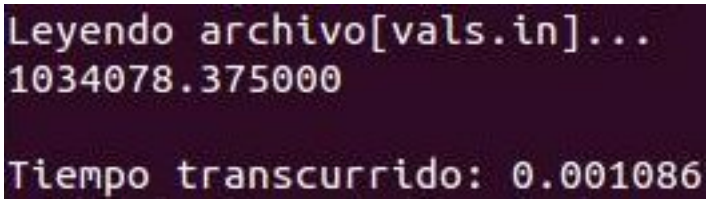
De este modo se pudo realizar la operación XOR de forma SIMD mejorando el rendimiendo de la funciòn.

4. Discusiones

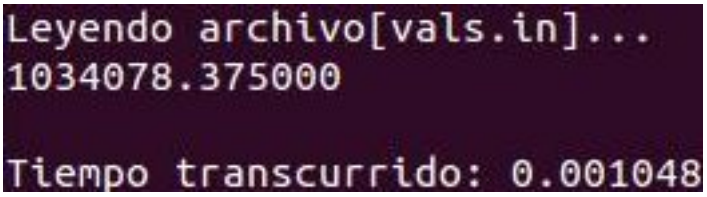
Primero se mostrarán los valores obtenidos de cada una de las funciones (y los programas), tanto escalares como vectoriales, para después realizar un análisis de estos resultados.

4.1. Función 1

Al ejecutar la Función 1, obtenemos dos valores, uno para la forma escalar:



Y un valor para la forma vectorial



Se puede observar que ambas funciones entregan el mismo valor pero en tiempos diferentes. En este caso, la forma vectorial demora un poco menos que la forma escalar.

4.2. Función 2

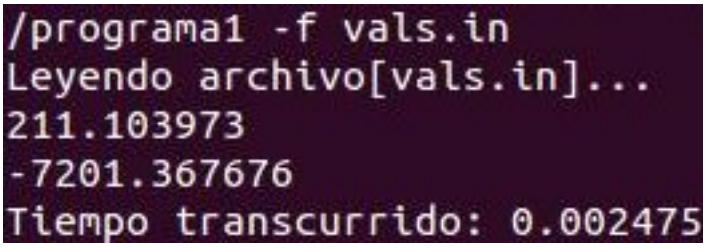
Al ejecutar la Función 2, obtenemos dos valores, uno para la forma escalar:

Y un valor para la forma vectorial

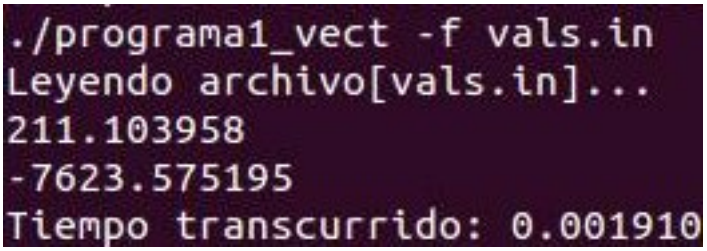
Se puede observar que ambas funciones entregan un valor distinto (y en tiempos diferentes). En este caso, la forma vectorial demora un poco menos que la forma escalar.

4.3. Programa 1

Al ejecutar el Programa 1, obtenemos dos valores, uno para la forma escalar:



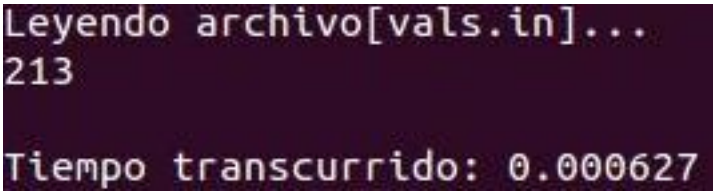
Y un valor para la forma vectorial



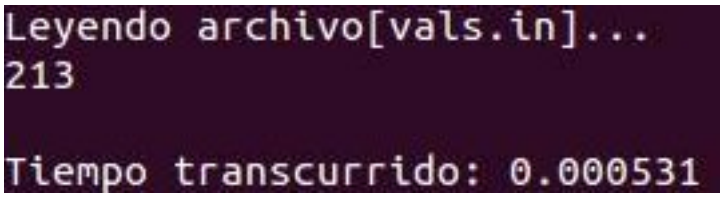
Se puede observar que ambas funciones entregan un valor distinto (y en tiempos diferentes). En este caso, la forma vectorial demora un poco menos que la forma escalar.

4.4. Programa 2

Al ejecutar el Programa 2, obtenemos dos valores, uno para la forma escalar:



Y un valor para la forma vectorial



Se puede observar que ambas funciones entregan el mismo valor pero en tiempos diferentes. En este caso, la forma vectorial demora un poco menos que la forma escalar.

4.5. Análisis de los Resultados

Al analizar la diferencia en los tiempos de ejecución de las funciones (y programas) escalares con respecto a los vectoriales; se puede observar que en su mayoría, los tiempos de ejecución de las funciones vectoriales es menor; a excepción de la *Funcion2*, la cual, se demora más tiempo en su forma vectorial que escalar; suponemos que esto sucede debido al constamte paso de datos de vector a array, lo cual hace que el tiempo de ejecución se eleve; aunque esto se podría variar al poser una lista mayor de datos que analizar, ya que los pasos de vector a array (y viceversa) se realizan al principio y al final de cada ejecución, entonces, al poseer más elementos, este cambio se notaría menos que la ejecución de todas las funciones de su forma escalar.

Además, hay funciones donde se entrega un valor distinto en su forma escalar que vectorial, esto se le atribuye a la presición de el vector es mayor que la presición de la funcion escalar; ésto sucede ya que se va perdiendo presición entre más grande es el *float*, entonces, como en la función vectorial el tamaño del *float* es un cuarto, la presición perdida también es menor, haciendo que el dato sea más preciso al *"perder menos información"*.

5. Conclusión

Para la elaboración del laboratorio 3, los primeros problemas que aparecieron fue el desconocimiento sobre *SIMD*, lo cual demoró la elaboración de las funciones vectoriales; ya que estabamos acostumbrados a las funciones escalares y no vectoriales.

Los objetivos específicos lograron cumplirse, ya que se logró implementar las funciones escalares y las funciones vectoriales; y para ésto era necesario el manejo de los conceptos de vectores y como se utilizan para la implementación de un programa.

El objetivo general también se logra cumplir, ya que se realiza el análisis de los datos, concluyendo que si bien al momento de implementar una función, el hacerlo de forma vectorial cuesta un poco más, esta función demora menos tiempo en ejecutarse que su forma escalar.

Esto se debe, ya que las funciones implementadas de forma vectorial, realizan algunas operaciones en forma paralela, lo cual disminuye la cantidad de veces que necesita pasar por el procesador, reduciendo el tiempo de ejecución.

Referencias

Conceptos básicos de make y los makefile. (2007, Febrero). Descargado de `http://www.chuidiang.com/clinux/herramientas/makefile.php`

Getopt. (1998, Mayo). Descargado de `http://es.tldp.org/Paginas-manual/man-pages-es-1.28/man`

Manual para para crear tu propia biblioteca en c/c++. (2012, Noviembre). Descargado de `http://cypascal.blogspot.com/2012/11/crea-tu-propia-biblioteca-en-cc.html`

Simd. (2014, Febrero). Descargado de `http://es.wikipedia.org/wiki/SIMD`

Sse. (2013, Junio). Descargado de `http://es.wikipedia.org/wiki/SSE`

Sse2. (2014, Agosto). Descargado de `http://es.wikipedia.org/wiki/SSE2`

Sse3. (2013, Marzo). Descargado de `http://es.wikipedia.org/wiki/SSE3`