Práctica1: Procesamiento vectorial SIMD

Ejercicio 1: Vectorización automática con el compilador

En este ejercicio se propone el uso de un compilador con vectorización automática para optimizar un programa simple escrito en lenguaje C. Para vectorizar se va a utilizar el compilador gcc instalado en lo equipos del laboratorio o en el ordenador personal de cada estudiante y también el compilador disponible en la página web ***Compiler Explorer*** (https://godbolt.org/) puede ser muy didáctico para comparar códigos.

El punto de partida es un fichero de código denominado ***simple2.c*** (proporcionado en el material y listado en el apéndice I) que contiene un programa escrito en C, en el que se realizan operaciones básicas sobre los elementos de vectores. En el programa se identifican varios bucles cerrados y sin dependencias, que el compilador debería poder vectorizar fácilmente.

El objetivo es probar varias opciones del compilador y estudiar las diferencias en el código ensamblador generado y finalmente comparar el rendimiento de los diferentes códigos vectorizados resultantes con el código no vectorizado.

1.1 Realice los siguiente pasos y documente en la memoria los resultados obtenidos en cada uno de ellos:

- Identifique las instrucciones SIMD admitidas por el microprocesador de la computadora del laboratorio (o su computadora). Indique el modelo de CPU y las instrucciones SIMD que admite y la versión del compilador utilizado.
- Partiendo del código de ejemplo simple2.c, revise el código y compruebe que es un programa simple que realiza una multiplicación-suma básica de matrices muchas veces dentro de un bucle.
- El compilador GCC aplicará automáticamente la vectorización en el nivel de optimización -O3. Pruebe el nivel -O3 e inspeccione el informe de vectorización para verificar si se ha vectorizado algún bucle. Por ejemplo:

```
gcc -O3 -march=native -fopt-info-vec-optimized -o simple2 simple2.c

simple2.c:26:9: note: loop vectorized
simple2.c:19:5: note: loop vectorized
```

Este informe nos muestra que se vectorizaron dos bucles: el bucle que inicializó los valores de los datos y el bucle de cálculo principal. Proporcione la versión de GCC y explique el informe que obtiene.

- Ahora que hemos determinado qué bucles son vectorizables, veamos qué sucede con el informe cuando compilamos con la vectorización totalmente deshabilitada. Llame al ejecutable simple2_no_vec.

```
gcc -O3 -march=native -fno-tree-vectorize -fopt-info-vec-
optimized -o simple2_no_vec simple2.c
```

Aquí, el compilador no muestra ningún informe por una razón bastante obvia, a saber, la inclusión del flag -fno-tree-vectorize. Compruebelo.

- Revise el código ensamblador que genera el compilador para el código vectorizado y no vectorizado.

```
gcc -S -O3 simple2.c
mv simple2.s simple2_o3.s

gcc -S -O3 -fno-tree-vectorize simple2.c
mv simple2.s simple2_o3_native.s

diff simple2_o3.s simple2_o3_native
```

En la memoria se debe explicar las principales diferencias entre los códigos ensambladores centrándonos en las instrucciones SIMD generadas por el compilador y los registros SIMD. Puede probar también a ver las diferencias con página web ***Compiler Explorer*** (https://godbolt.org/)

1.2 Compare el rendimiento de un código compilado con diferentes opciones de compilación en los flag que afectan el rendimiento, al menos pruebe con las combinaciones más signifcativas de los flag: -O3, -O2, -O1, con y sin vectorización `-fno-tree-vectorize`.
Para obtener más detalles sobre las distintas optimizaciones del compilador, consulte https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Para evaluar el rendimiento debe tomar tiempos de una manera adecuada en la ejecución de los ejecutables o preferiblemente incluir en el código *gettimeofday()* antes y después de la sección de código que sea objeto de comparación.

Se debe tomar como referencia el código generado con los flags que implican un mayor tiempo de ejecución y para para otras opciones de los flag, mida los tiempos de ejecución y vaya calculando las aceleraciones obtenidas. Visualice los resultados en una tabla o en una gráfica que facilite las comparaciones y comente los resultados más significativos.

1.3. El compilador con el flag `-march` permite crear código para arquitecturas diferentes de la nativa, máquina en la que se ejecuta el compilador(`-march=native`). Esto nos permite ver como sería el código generado en arquitecturas muy diferente, aunque no lo podamos ejecutar, este proceso se denomina crosscompilación.

- Encuentre que opciones son posibles para el flag march. El propio compilador te da esa información si pones un valor indefinido

```
gcc -O3 -S -march=xxx simple2.c
```

- Realice crosscompilaciones para arquitecturas SIMD que corresponda a arquitecturas equivalentes a SSE, AVX y AVX512.

En la memoria se debe explicar las principales diferencias entre los códigos ensambladores para las diferentes arquitecturas, centrándonos en las instrucciones SIMD generadas por el compilador y los registros SIMD utilizados.

Ejercicio 2: Uso de intrinsics para vectorizar manualmente.

Este ejercicio introducirá el uso de vectores intrínsecos en el ejemplo simp. Trabajaremos con el programa simple2.c del ejercicio anterior. Nuestro objetivo es vectorizar el código manualmente. Realice los siguientes pasos:

- Obtenga una copia del código de ejemplo simple2.c y cámbiele el nombre a simple2_intrinsics.c. Recuerde que es un programa simple que realiza una suma múltiple básica de matrices muchas veces dentro de un bucle.

- El bucle a vectorizar bucle objetivo es el bucle interior del segundo bucle. Antes del segundo ciclo, declaramos dos variables __m256d: una para almacenar el valor constante 1.0001 (el valor m) y la segunda para almacenar las sumas parciales:

```
__m256d mm = {1.0001, 1.0001, 1.0001, 1.0001};
__m256d sum = {0.0, 0.0, 0.0, 0.0}; // to hold partial sums
```

- Ahora, debemos cambiar el valor de incremento del índice i de i++ a i += 4 porque queremos operar 4 dobles en paralelo (vector de 256 bits).

```
for (i=0; i < ARRAY_SIZE; i += 4) {
```

- Dentro del bucle, primero tenemos que cargar los 4 doubles de los arrays de entrada a vectores y luego, calcular c += m*a+b. Para ello, debemos dividir la operación en dos operaciones: tmp = m*a+b y luego c += tmp:

```
// Load arrays
__m256d va = _mm256_load_pd(&a[i]);
__m256d vb = _mm256_load_pd(&b[i]);

// Compute m*a+b
__m256d tmp = _mm256_fmadd_pd (mm, va, vb);
// Accumulate results
sum = _mm256_add_pd (tmp, sum);
```

- Ahora, sum contiene las sumas de todos los productos en cuatro partes y queremos un resultado escalar. Hay dos opciones. La más simple es realizar la suma vectorial con operadores aritméticos

```
for (i = 0; i < 4; i++) {
  c += sum[i];
}
```

La otra opción es realizar la suma vectorial con intrínsecos:

```
// Get sum[2], sum[3]
__m128d xmm = _mm256_extractf128_pd (sum, 1);

// Extend to 256 bits: sum[2], sum[3], 0, 0
__m256d ymm =  _mm256_castpd128_pd256(xmm);

// Perform sum[0]+sum[1], sum[2]+sum[3], sum[2]+sum[3], 0+0
sum = _mm256_hadd_pd (sum, ymm);
```

```
// Perform sum[0]+sum[1]+sum[2]+sum[3]…
sum = _mm256_hadd_pd (sum, sum);
c = sum[0];
```

Consulte la Guía de intrínsecos de Intel (https://software.intel.com/sites/landingpage/IntrinsicsGuide/) para leer la descripción y comprender el funcionamiento de los intrínsecos utilizados anteriormente.

Para comprobar que la vectorización no altera el resultado, umprima el valor final (variable c) de esta versión y compárelo con el programa simple2.c original para asegurarse de que el programa simple2_intrinsics.c no cambie la salida. Agregue el código necesario a ambos programas. Cambie el valor de NUMBER_OF_TRIALS a 1 para que sea más fácil comparar los resultados.

2.1 Realice la vectorización del primer bucle (de inicialización) utilizando intrínsecos. Consejo: cree vectores iniciales.

2.2 Obtenga el tiempo de ejecución para diferentes valores de NUMBER_OF_TRIALS: de 100.000 a 1.000.000 en pasos de 100.000. Dibuje el resultado en un gráfico que compare los resultados con y sin vectorización. Explique los resultados.

2.3 Calcule la aceleración (Tiempo no vectorizado/ Tiempo vectorizado) obtenida y represéntela gráficamente.

> ***Nota1:*** Use *gettimeofday()* antes y después del segundo bucle para obtener el tiempo de ejecución.
>
> ***Nota2:*** Es importante que en la memoria explique con detalle como funciona la implementación/algoritmo. Se valorará utilizar en las explicaciones referencias al código, aportando gráficas, esquemas, capturas, imágenes, etc. Como por ejemplo hacer explicaciones con esquemas similares a los utilizados en las presentaciones ppt de introducción a esta práctica.

Ejercicio 3: Tutorial de vectorización manual usando intrinsics

Este tutorial, preparado para su ejecución en GoogleColab es una introducción a la vectorización manual usando intrinsics de Intel:
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

El tutorial es autoexplicativo y en las diferentes secciones se formulan preguntas que deben contestarse e incluirlas en la memoria de la práctica.

El ejemplo que se realiza es la vectorización de un producto escalar de dos vectores con valores en coma flotante de doble precisión. Se realiza un estudio de la aceleración obtenida al vectorizar para diferentes tamaños de vector.

3.1. Realice el tutorial y conteste las preguntas planteadas en la memoria y por claridad incluya junto a la respuesta la pregunta a la que corresponde.

3.2. Reproduzca los resultados del tutorial en un equipo de laborarorio o en su equipo personal, comentando los resultados obtenidos y destacando las principales diferencias observadas.

3.3. Realice una variante del ejemplo del tutorial para un producto escalar de dos vectores con valores en **coma flotante de simple precisión**. Al igual que en el tutorial debe realizar con criterio un estudio de la aceleración obtenida al vectorizar para diferentes tamaños de vector y comparar con resultados l en un equipo de laborarorio o en su equipo personal, comentando los resultados obtenidos y destacando las principales diferencias observadas.

Ejercicio 4: Vectorizar un algoritmo de procesamiento de imagen

El archivo greyScale.c incluye un programa que aplica un algoritmo de procesamiento de imágenes para convertir cualquier imagen a escala de grises. Este algoritmo nos ha sido proporcionado, como prueba de concepto, con la idea de que finalmente procese imágenes de un flujo de vídeo en tiempo real. Recuerde que un vídeo suele estar compuesto por aproximadamente 30 fps (fotogramas por segundo). Es decir, el programa tendría que procesar 30 imágenes en un segundo.

Realice los siguiente pasos y responda las preguntas planteadas de forma razonada:

0. Compila y ejecuta el programa utilizando algunas de las imágenes proporcionadas como argumentos. Examina los resultados que se generaron y analiza brevemente el programa proporcionado.
1. El programa incluye dos bucles. El primer bucle (indicado como Loop 0) itera sobre los argumentos aplicando el algoritmo a cada uno de ellos. El segundo bucle (indicado como Loop 1) computa el algoritmo de escala de grises. ¿Es este bucle óptimo para ser vectorizado? ¿Por qué? Consejo: recomendamos utilizar Compiler Explorer (https://godbolt.org/).
2. Sabemos que el orden de acceso a los datos es importante para permitir la autovectorización. Corrija el código para ayudar al compilador a vectorizar el bucle. Explique sus cambios.
    a. Es imperativo que el programa continúe ejecutando el mismo algoritmo, por lo que solo se deben realizar cambios en el programa que no cambien la salida.
3. Se pide vectorizar el algoritmo de escala de grises utilizando intrínsecos. Complete una tabla con los resultados de tiempo y aceleración en comparación con la versión original y la versión autovectorizada para imágenes de diferentes resoluciones (SD, HD, FHD, UHD-4k, UHD-8k). Debe incluir una columna con los fps a los que procesaría el programa. Discuta los resultados.
    a. Es imperativo que el programa continúe ejecutando el mismo algoritmo, por lo que solo se deben realizar cambios en el programa que no cambien la salida.
    b. Explique cómo traduce el algoritmo serial a una versión vectorizada y los intrínsecos utilizados.

Referencias

- Sergey Slotin, “Algorithms for Modern Hardware”, available online: https://en.algorithmica.org/hpc/. Section SIMD Parallelism.

Apéndice 1 : Programa simple2.c

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

#define ARRAY_SIZE 2048
#define NUMBER_OF_TRIALS 1000000

/*
 * Statically allocate our arrays.  Compilers can
 * align them correctly.
 */
static double a[ARRAY_SIZE], b[ARRAY_SIZE], c;

int main(int argc, char *argv[]) {
    int i,t;

    double m = 1.0001;

    /* loop1: Populate A and B arrays */
    for (i=0; i < ARRAY_SIZE; i++) {
        b[i] = i;
        a[i] = i+1;
    }

    /* loop2 y loop3(inner) Perform an operation a number of times */
    for (t=0; t < NUMBER_OF_TRIALS; t++) {
        for (i=0; i < ARRAY_SIZE; i++) {
            c += m*a[i] + b[i];
        }
    }

    return 0;
}
```