LABORATORIO 2.

PROGRAMACIÓN UTILIZANDO EXTENSIONES SIMD

El propósito de este laboratorio es adquirir una experiencia mínima en la utilización de instrucciones SIMD de la arquitectura X86 y facilitar la aplicación de las optimizaciones de auto vectorización por parte del compilador. El compilador de trabajo es el GCC puesto que soporta dos conjuntos de funciones intrínsecas para las instrucciones SIMD. Una de ellas es nativa de GCC, mientras que la otra es la definida por Intel para su compilador de C++. Dado que existe mucha más documentación disponible para las funciones intrínsecas de Intel utilizaremos estas últimas.

Tanto Intel cómo AMD proporcionan muy buenos manuales de optimización donde se explica el uso de instrucciones SIMD y las optimizaciones software. Para la realización de esta práctica no será necesario estudiar dichos manuales, pero es interesante que el estudiante los tenga presentes y los utilice como material de consulta complementario. La práctica se puede realizar en cualquier plataforma Linux, incluida la que incorpora dentro de la versión Windows 10 de Microsoft utilizando cualquier tipo de estas extensiones.

Los equipos de los laboratorios tienen procesadores Intel y se pueden utilizar las extensiones AVX y se supone que la práctica se desarrollará utilizando esas extensiones vectoriales.

Trabajo para realizar

1. Lectura del artículo "Crunching Numbers with AVX and AVX2" donde se introduce las funciones intrínsecas de C para programar utilizando extensiones SIMD en los procesadores INTEL.
2. Averigua que extensión vectorial se puede utilizar según las capacidades del procesador donde se vayan a implementar la práctica. En este enlace (https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX_512&text=mm512_red&ig_expand=95,143,123) encontrarás información del conjunto de extensiones de Intel. En el aula virtual tienes ejemplos muy sencillos y básicos de utilización de estas estas extensiones vectoriales.

   Nota: Para utilizar las extensiones vectoriales AVX512, compila con el argumento -mavx512f en lugar de -mavx.

3. **Multiplicación matriz × vector (dgmv).** La operación de multiplicación de una matriz por un vector se puede realizar conforme al código del Listado 5, y es una de las operaciones básicas de álgebra matricial.

   El primer paso para vectorizar esta operación es expandir (*unroll*) el bucle. Se expandirá en función de los elementos que se pueden almacenar en un registro vectorial. El programa trabaja con números de simple precisión. La capacidad de los registros vectoriales depende de la extensión vectorial que se pueda utilizar, siendo de 128 bits, 256 o 512 bits para la extensión AVX, AVX256, AVX512, respectivamente. Esto limita el número de datos que podemos empaquetar en un registro y que podempos procesar simultáneamente. (4, 8 o 16 operados de simple precisión). El código desenrollado se muestra en el Listado 6. Aunque el nivel de desenrollado dependerá de la capacidad de los registros vectoriales. El ejemplo se ha hecho para un desenrrollado de 4, suponiendo que en el registro vectorial se pueden almacenar hasta 4 operandos.

```
// dgmv
// realiza la operación matricial c = c + M*b, donde c y b son
//vectores de n elementos, y M es una matriz de nxn elementos.

static void simple_dgmv( size_t  n , float c[ n ],
   const float M[n][n], const float b[n]) {
 for ( int i = 0; i < n ; i ++)
      for ( int j = 0; j < n ; j ++)
         c[i] += mat_a[i][j]*b[j];
}
```

Listado 5. Operación básica de multiplicación de matriz por un vector.

```
1. // dgmv_unroll
2. // realiza la operación matricial c = c + M*b, donde c y b son
   vectores
3. // de n elementos, y M es una matriz de nxn elementos. El bucle
   está
4. // extendido (unroll) 4 veces.
5. static void dgmv_unroll( size_t n, float c[n],
6.                          const float M[n][n], const float
   vec_b[n]) {
7.    for (int i = 0 ; i<n ; i++)
8.       for (int j = 0 ; j<n ; j += 4)
9.          c[i] += M[i][j+0]*b[j+0]+
10.                  M[i][j+1]*b[j+1]+
11.                  M[i][j+2]*b[j+2]+
12.                  M[i][j+3]*b[j+3];
13.      }
14.
```

Listado 6. Operación acumulativa de matriz por vector con el bucle nivel 4.

4. **Multiplicación matriz x matrix (dgemm).** La forma más obvia de multiplicar dos matrices es la indicada en el código del Listado 7. Para obtener una implementación vectorial se puede empezar por desenrollar el bucle interno tantas veces como sea necesaria en función de la capacidad de los registros vectoriales, lo que permitirá después utilizar instrucciones vectoriales.

```
// dgemm
// realiza la operación matricial C = C + A*B, donde A, B y C son tres
// matrices de elementos de tipo double de dimensiones:
// A de dim1 x dim2
// B de dim2 x dim3
// C de dim1 x dim3
void dgemm( int dim1, int dim2, int dim3,
            double *A, double *B, double *C) {

   for (int i=0; i<dim1; i++)
      for (int j=0; j<dim2; j++)
          for (int k=0; k<dim3; k++)
             //C[i][k] += C[i][k] + A[i][j]*B[j][k];
             *(C+i*dim3+k) += *(A+i*dim2+j) * *(B+j*dim3+k);
}
```

Listado 7. Operación acumulativa de matriz por matriz.