Instrucciones Vectoriales - AVX

Álvaro Moure Sabaté

15 de noviembre de 2021

Índice

1. Función main

Se declaran las matrices dinámicamente con alineamiento de memoria a 16 bytes.

```
#define ALIGN 16
int main( int argc, char *argv[] ) {

  int  n=8,i,k, tb=4;
  clock_t tic,toc;
  int nb=n/tb; //numero de bloques

    /* Reservamos memoria para los datos */
  double *A       = (double *) _mm_malloc(n*n*sizeof(double),ALIGN);
  double *B       = (double *) _mm_malloc(n*n*sizeof(double),ALIGN);
  double *C       = (double *) _mm_malloc(n*n*sizeof(double),ALIGN);

    .
    .
    .

   _mm_free(A);
   _mm_free(B);
   _mm_free(C);
      return 0;
}
```

2. Funciones To_blocked y From_blocked

Se hace uso de las instrucciones vectoriales **_mm256_storeu_pd(double*, __m256d)** y **_mm256_loadu_pd(double*)** para acelerar el proceso de copia de matrices de formato

contiguo a bloques, y viceversa.

```
void To_blocked(double A[], int n, int b) {
   int i, j;
   int i_bar, j_bar;  // index block rows and block cols
   int n_bar = n/b;
   double *a_p, *t_p;
   __m256d tmp;

   double *T = _mm_malloc(n*n*sizeof(double),ALIGN);
   if (T == NULL) {
      fprintf(stderr, "Can't allocate temporary in To_blocked\n");
      exit(-1);
   }

   // for each block in A
   t_p = T;
   for (j_bar = 0; j_bar < n_bar; j_bar++)
      for (i_bar = 0; i_bar < n_bar; i_bar++) {

         // Copy block into contiguous locations in T
         a_p = A + (i_bar*b + j_bar*b*n);
         for (j = 0; j < b; j++, a_p += (n-b))
            for (i = 0; i < b; i+=4) {
                tmp = _mm256_loadu_pd(a_p);
                _mm256_storeu_pd(t_p, tmp);
                a_p+=4; t_p+=4;
            }
      }

   memcpy(A, T, n*n*sizeof(double));

   _mm_free(T);
}  /* To_blocked */

void From_blocked(double C[], int n, int b) {
   int i, j;
   int i_bar, j_bar;  // index blocks of C
   int n_bar = n/b;
   double *c_p, *t_p;
   __m256d tmp;

   double *T = _mm_malloc(n*n*sizeof(double),ALIGN);
   if (T == NULL) {
      fprintf(stderr, "Can't allocate temporary in To_blocked\n");
      exit(-1);
   }

   // for each block of C
   c_p = C;
```

```
   for (j_bar = 0; j_bar < n_bar; j_bar++)
      for (i_bar = 0; i_bar < n_bar; i_bar++) {

         // Copy block into correct locations in T
         t_p = T + (i_bar*b + j_bar*b*n);
         for (j = 0; j < b; j++, t_p += (n-b))
            for (i = 0; i < b; i+=4) {
                tmp = _mm256_loadu_pd(c_p);
                _mm256_storeu_pd(t_p, tmp);
                c_p+=4; t_p+=4;
            }
      }

   memcpy(C, T, n*n*sizeof(double));
   _mm_free(T);
}  /* From_bloc */
```

3. Función Mult_add_avx

Dando por supuesto que el producto de matrices por bloques permite alojar al bloque de la matriz "A" en la caché, el acceso a los elementos de una fila del bloque de "A" no produce un fallo de caché (Matriz almacenada por columnas). Además, la influencia del orden de los bucles no supone una grave penalización, en cuanto a prestaciones se refiere, cuando la matriz entera se encuentra en la caché. Tomando estas dos premisas, dentro del bucle interno, el producto de matrices se convierte en el producto escalar de dos vectores (fila x columna).

```
void Mult_add_avx(double *A, double *B, double *C,int i_bar, int j_bar, int
↪  k_bar, int n_bar, int tb)
{
    int b_sqr=tb*tb;
    double *c_p = C +( i_bar + j_bar*n_bar)*b_sqr;
    double *a_p = A + (i_bar + k_bar*n_bar)*b_sqr;
    double *b_p = B + (k_bar + j_bar*n_bar)*b_sqr;

    __m256d a_vec,tmp,b_vec;
    double suma = 0.0;

   int i, j, k;
    for (j = 0; j < tb; j++){
        for (i = 0; i < tb; i++){
            for (k = 0; k < tb; k+=4){
                a_vec = _mm256_set_pd(a_p[i+(k+3)*tb], a_p[i+(k+2)*tb],
                ↪  a_p[i+(k+1)*tb], a_p[i+(k+0)*tb]);
                b_vec = _mm256_load_pd(&b_p[k+j*tb]);

                tmp = _mm256_mul_pd(a_vec, b_vec);

                tmp = _mm256_hadd_pd(tmp,tmp);
```

```
                suma += tmp[0] + tmp[2];
            }
            c_p[i+j*tb] += suma;
            suma = 0.0;
        }
    }
}
```

La instrucción **_mm256_set_pd()** permite cargar cuatro elementos de la fila "i" de la submatriz de "A". Así mismo, sabiendo que la submatriz de "B" está contigua en memoria y alineada a 16 bytes, la instrucción **_mm256_load_pd()** carga cuatro de la columna "j" de la submatriz de "B". Una vez tenemos cargados los vectores, se multiplican elemento a elemento con la función **_mm256_mul_pd()** y se suman horizontalmente con **_mm256_hadd_pd()**. Esta secuencia se repite tantas veces como sea necesaria hasta que se complete el producto escalar de fila x columna. El resultado de esta operación (acumulado en la variables **suma**) se almacena en la posición $[i, j]$ de la submatriz de "C".

4. Compilación y resultados

Compilador	Flag	Normal	Bloques	AVX
Intel	-O0	94.89	97.43	109.82
	-O1	26.54	17.42	16.28
	-O2	21.12	5.22	15.22
	-O3	22.49	5.11	15.34
GNU	-O0	95.69	98.81	-
	-O1	29.71	25.80	17.50
	-O2	28.90	20.31	13.41
	-O3	19.12	16.50	13.94

Tabla 1: Tiempo de cálculo, en segundos, del producto de matrices con tamaño 3200x3200 y tamaño de bloque 16x16.

Como se puede observar en la Tabla 1, el código que incluye las instrucciones vectoriales y segmentación por bloques supera en todos los casos a la versión de multiplicación de matrices tradicional (excepto en el compilador de Intel cuando no se activa ningún tipo de optimización "-O0"). Así mismo, con el compilador de GNU, la versión de AVX con bloques tiene un mejor rendimiento que la de bloques, para todas las optimizaciones habilitadas. Un caso particular sucede con el compilador de Intel cuando se habilitan las optimizaciones "-O2" y "-O3". Para estos casos la versión con bloques sin AVX experimenta un aumento del rendimiento muy elevado. Posiblemente esto es debido a que el compilador de Intel es capaz de reconocer el producto de matrices y lo sustituye por su código más optimizado.