**Publicaciones en Ciencias y Tecnología**
Vol 8, N$^0$1, Ene–Jul 2014, pp.31–44, ISSN:1856-8890, Depósito Legal:pp200702LA2730

# CÓMPUTO DE ALTO DESEMPEÑO PARA OPERACIONES VECTORIALES EN BLAS-1

[*] José Antonio Muñoz Gómez [**] Abimael Jiménez Pérez
[***] Gustavo Rodríguez Gómez

Recibido: 08/10/2014 Aprobado: 31/01/2015

**Resumen**

La biblioteca de funciones denominada Subprogramas Básicos de Álgebra Lineal (BLAS-1) es considerada el estándar de programación en computación científica. En este trabajo nos enfocamos en el análisis de diversas técnicas de optimización de código para incrementar el desempeño computacional de BLAS-1. En particular abordamos un enfoque combinacional para explorar las posibles formas de codificación empleando la técnica de unroll con diversos niveles de profundidad, programación vectorial de datos con MMX y SSE para procesadores Intel. Empleando las funciones principales de BLAS-1 determinamos numéricamente un incremento computacional, expresado en mega-flops, de hasta 52 % en comparación con la biblioteca optimizada BLAS-1 de ATLAS.

**Palabras clave**: cómputo científico, BLAS-1, técnica de unroll, programación vectorial.

[*] *Departamento de Ingeniería, Universidad de Guadalajara, Jalisco, México, Doctor, jose.munoz@cucsur.udg.mx*

[**] *Departamento de Electrónica y Computación , Universidad Autónoma de Ciudad Juarez, Ciudad Juárez, México, Doctor, abimael.jimenez@uacj.mx*

[***] *Departamento Ciencias Computacionales, Instituto Nacional de Astrofísica, Óptica y Electrónica, Puebla, México, grodrig@inaoep.mx*

**INCREASED COMPUTATIONAL PERFORMANCE FOR VECTOR OPERATIONS ON BLAS-1**

**Abstract**

The functions library, called Basic Linear Algebra Subprograms (BLAS-1), is considered the programming standard in scientific computing. In this work, we focus on the analysis of various code optimization techniques to increase the computational performance of BLAS-1. In particular, we address a combinational approach to explore possible methods of encoding using unroll technique with different levels of depth, vector data programming with MMX and SSE for Intel processors. Using the main functions of BLAS-1, it was determined numerically a computational increase, expressed in mega-flops, up to 52 % compared to the optimized BLAS-1 ATLAS library.

**Keywords**: Scientific computing, BLAS-1, unroll technique, vector programming.

# Introducción

Las aplicaciones de software requeridas en la solución numérica de ecuaciones en derivadas parciales, problemas de optimización y en diversos campos de las ciencias y las ingenierías, comparten la necesidad de realizar operaciones aritméticas con vectores y matrices (Trefethen y Bau, 1997; Golub y Loan, 1996; Van Loan, 1999; Bouhamidi, Hached, y Jbilou, 2013; Mansour y Gtze, 2013; Yzelman, Roose, y Meerbergen, 2015). Esto se realiza a través de bibliotecas de funciones e interfaces de programación de aplicaciones (APIs) que están diseñadas para reutilizar código y aumentar la portabilidad con el mayor rendimiento posible para una arquitectura de microprocesador específica. La biblioteca de funciones BLAS-1 dedicada a operaciones vectoriales, es considerada el estándar de programación en computación científica (Lawson, Hanson, Kincaid, y Krogh, 1979; Goto y Van De Geijn, 2008; Napoli, Fabregat-Traver, Quintana-Ort, y Bientinesi, 2014). Las tres operaciones clásicas entre vectores son:

$$< x, y >, \qquad \|x\|, \qquad y \leftarrow y + \alpha \cdot x, \qquad x, y \in \mathbb{R}^n$$

lo que corresponde al producto punto, la norma de un vector y la operación denominada *axpy* con $\alpha \in \mathbb{R}$.

La biblioteca BLAS-1 fue desarrollada hace más de tres décadas y ha tenido una gran aceptación. Sin embargo, el desempeño computacional que ofrece es bajo debido a que fué diseñada para ser portable. Para incrementar el rendimiento se ha desarrollado recientemente un enfoque basado en búsqueda de parámetros óptimos para determinar los esquemas de codificación con el mayor rendimiento, expresado en millones de operaciones de coma flotante por segundo (MFLOPS). Esto ha sido desarrollado en los APIs de ATLAS (Whaley y Dongarra, 1997) y recientemente en OpenBlas (Wang, Zhang, Zhang, y Yi, 2013). Ambos son esquemas de código libre llamados auto-ajustables y tienen como énfasis el nivel 2 y 3 de BLAS. En ambos casos, se requiere de una gran experiencia para poder compilar e instalar las funciones y hasta donde tenemos conocimiento, no es posible determinar cuál es el mejor esquema de codificación.

En este trabajo, analizamos diversas técnicas empleadas en computación de alto rendimiendo para incrementar el desempeño computacional en la bliblioteca de funciones BLAS-1. En particular, se realiza un enfoque combinacional en donde fusionamos la técnica de loop unrolling, el procesamiento vectorial de datos con la tecnología MMX y el enfoque de SSE. Con base en las funciones principales de BLAS-1, se determina numéricamente una mejora en el rango de 1.2x a 2.1x en comparación con la biblioteca optimizada de ATLAS. Aunado a ello, se esclarece el esquema de codificación el cual permitirá emplear este enfoque en una gran variedad de aplicaciones en computación científica.

En la siguiente sección se describe la técnica de loop unrolling y se muestra en detalle su codificación. En la sección 3, se aborda la programación paralela a nivel de datos empleando los registros vectoriales de Intel mediante las instrucciones explícitas para registros XMM y las funciones implícitas con SSE. Posteriormente, en la sección 4 se analiza la fusión combinacional de las técnicas desarrolladas en las secciones anteriores, con ello se determina la configuración con el mejor desempeño expresado en MFLOPS. Finalmente, se enuncian las conclusiones del presente trabajo y se indica el trabajo a futuro.

## Loop unrolling

Los ciclos de procesamiento son los responsables de la mayoría del tiempo de ejecución en diversas aplicaciones científicas, que se caracterizan por un consumo mayor al 90 % de su tiempo de ejecución en uno o pocos ciclos de procesamiento (Hennessy y Patterson, 2003). La compilación con técnicas automáticas de unroll es comúnmente usada en programas científicos, y se pueden obtener grandes beneficios en tiempo de procesamiento (Aiken y Nicolau, 1987; Davidson y Jinturkar, 1995). Sin embargo, no existe un compilador que incremente el desempeño computacional para cualquier tipo de programa científico considerando la arquitectura del microprocesador. Es por ello la necesidad de realizar codificaciones y mejoras no automáticas para incrementar el desempeño computacional. En este trabajo, ejemplificaremos las diversas técnicas computacionales para incrementar el desempeño, empleando exclusivamente el producto punto entre un par de vectores. Esto permitirá concentrarnos en una explicación directa, dejando al lector que consulte la biblioteca de funciones disponibles para comprender a mayor profundidad la forma de aplicarla.

La técnica de loop unrolling consiste en disminuir el número de iteraciones de un ciclo mediante un aumento de operaciones en el mismo ciclo. La idea radica en desdoblar el ciclo para tratar de incrementar el desempeño computacional mediante el procesamiento paralelo a nivel de máquina subyacente en la arquitectura pipeline de los procesadores (Davidson y Jinturkar, 1995). Para ejemplificar la técnica de unroll, consideremos el producto punto entre un par de vectores

$$< x, y >:= \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n. \qquad (1)$$

La profundidad de este esquema es cero, debido a que no existe unroll. El método de unroll de longitud dos se puede expresar como

$$< x, y >:= \sum_{i=1}^{n} x_i y_i + x_{i+1} y_{i+1}, \quad \Delta i = 2. \qquad (2)$$

De manera análoga, empleando la técnica de unroll de longitud

cuatro en el producto punto se obtiene

$$< x, y >:= \sum_{i=1}^{n} x_i y_i + x_{i+1} y_{i+1} + x_{i+2} y_{i+2} + x_{i+3} y_{i+3}, \quad \Delta i = 4, \tag{3}$$

donde $\Delta i$ representa el factor de incremento sobre el índice del ciclo y corresponde al valor de profundidad del unroll. Por cuestiones de claridad en la exposición, consideraremos que la longitud del vector $n$ tiene división exacta con respecto al incremento $\Delta i$. Cabe señalar que desde el punto de vista algebraico el producto punto se puede expresar de distintas maneras (1-3). Sin embargo, cada implementación conlleva a rendimientos computaciones diferentes. De hecho, la manera más común de codificar $< x, y >$ es la aplicación directa de la ecuación (1). A esta forma la llamaremos canónica y con ella se obtiene el peor rendimiento computacional.

Aunado a la técnica de unroll podemos agregar otro grado de libertad empleando variables temporales en el acumulador de la sumatoria. Esto se ejemplifica en el siguiente fragmento de programa.

```
function dot_4
   variables
     i, s1=0, s2=0;
   for(i=0; i<n; i+=4)
       s1 += x[i+0]*y[i+0] + x[i+1]*y[i+1];
       s2 += x[i+2]*y[i+2] + x[i+3]*y[i+3];
   end
   result = s1 + s2;
end
```

Este nuevo grado de libertad incrementa el número de posibles codificaciones para el producto punto, empleando solamente la técnica de unroll. Así mismo, tiene un impacto en el desempeño computacional. En particular se codificaron de 13 maneras distintas la operación $< x, y >$, el grado de profundidad empleado fue unroll=$\{0, 2, 4, 6, 8, 10\}$. A partir del unroll-4 se agregaron las variables temporales de tal forma que el número de variables auxiliares fuése divisible con el incremento $\Delta i$. Cabe mencionar que el empleo de variables auxiliares

modifica ligeramente el valor de la sumatoria final. Este resultado es conocido y está fuera del alcance del presente trabajo. En el libro de Higham se puede encontrar un análisis detallado al respecto (Higham, 2002).

## Programación con MMX & SSE

La demanda de recursos en las aplicaciones multimedia y de comunicaciones fomentó el desarrollo de nuevas tecnologías que permitieran cubrir la creciente demanda. En 1999 Intel incorporó la tecnología MMX en sus procesadores. Esta nueva tecnología estaba destinada principalmente al procesamiento vectorial de datos con operaciones principalmente aritméticas sobre vectores. La nueva tecnología fue construida bajo el esquema SIMD (Single Instruction, Multiple Data) o una instrucción para múltiples datos (Mittal, Peleg, y Weiser, 1997). Se aplica una misma instrucción/operación sobre un conjunto de datos; esto permite conseguir paralelismo a nivel de datos mediante el procesamiento vectorial. Conforme a lo reportado por la empresa Intel, los núcleos de procesamiento escritos con MMX presentan un incremento en la velocidad de ejecución en el rango [1.5x, 2.0x].

La tecnología MMX inicialmente contaba con ocho registros estáticos de 64 bits de longitud, principalmente para procesamiento de números enteros. Posteriormente se extendió con la adición de Streaming SIMD Extensions, incorporando ocho nuevos registros capaces de trabajar con números en punto flotante nombrados XMM, los cuales pueden almacenar cuatro números IEEE 754 en cada registro de 128 bits (Inc., 2012). De esta manera cargamos cuatro elementos de cada vector y los procesamos de forma paralela. Este comportamiento es análogo a procesar con unroll-4 (U-4) en paralelo a nivel de datos.

Para ejemplificar el uso de los registros XMM en la programación vectorial, se muestra a continuación la implementación de $< x, y >$. Las funciones intrínsecas utilizadas están en las líneas 8, 11, 15-17. La declaración de las variables de 128 bits se realiza en la línea cuatro.

```
1 void dot_mmx(float *px, float *py, int n, float *result)
2 {
3    int    i;
```

```
4    __m128 sum,  *x,  *y;
5
6    x   = (__m128*) px;
7    y   = (__m128*) py;
8    sum = _mm_setzero_ps();
9
10   for( i = 0; i < n; i += 4 ) {
11        sum = _mm_add_ps(sum, _mm_mul_ps(*x, *y));
12        px++;  py++;
13    }
14
15   sum = _mm_hadd_ps(sum, sum);
16   sum = _mm_hadd_ps(sum, sum);
17   _mm_store_ss(result,sum);
18 }
```

Para esclarecer el uso de las funciones intrínsecas, analizaremos la línea 11 en donde se realiza las siguientes operaciones

$$t_0 = x_0 * y_0 \qquad t_1 = x_1 * y_1 \qquad t_2 = x_2 * y_2 \qquad t_3 = x_3 * y_3$$
$$s_0 = s_0 + t_0 \qquad s_1 = s_1 + t_1 \qquad s_2 = s_2 + t_2 \qquad s_3 = s_3 + t_3$$

como se observa, se tienen dos operaciones vectoriales sobre registros de 128 bits que operan cuatro números en punto flotante de manera paralela. De manera análoga, en las líneas 15 y 16 se realizan operaciones vectoriales. Como se aprecia en la línea 10 el incremento es $\Delta i = 4$. En este sentido, es posible fusionar la técnica de unroll con XMM, esto nos permitirá realizar incrementos con $\Delta i = 8, 12$. Esto es analizado en la sección de resultados. A pesar de las ventajas de la programación vectorial, su uso no es muy difundido debido en parte a la nula portabilidad cuando migramos el programa a otras arquitecturas de computadora.

Es conocido que el uso de funciones intrínsecas para vectorizar códigos nos limita a una arquitectura en particular; los juegos de instrucciones de cada procesador son distintos. Sin embargo, para compensar esta situación el compilador GCC provee de una interface la cual permite escribir código vectorial independiente de la arquitectura subyacente, mediante la especificación de atributos en las variables

(Chisnall, 2007). En la línea 2 del código siguiente se establece la definición del vector con tamaño 4 de números tipo flotante bajo la norma IEEE-754. El tamaño del vector se restringe a que éste debe ser un entero y una potencia de dos. Las variables creadas a partir de este tipo de dato (línea 3 del código) pueden ser utilizadas como cualquier variable escalar en operaciones (especialmente aritméticas), por ejemplo la suma y multiplicación de la línea 11, aun cuando estas variables aparentan ser escalares existirán situaciones en las cuales se deba recurrir a las funciones intrínsecas.[1] Cabe señalar, que ésto es una extensión de GCC y no forma parte del estándar ANSI C, por lo cual es posible que no sea soportado por otros compiladores.

```
1  const int VECTOR_SIZE = 4;
2  typedef float vec __attribute__
         ((vector_size (sizeof(float) * VECTOR_SIZE)));
3
4  void axpy_sse(float *x, float *y,  int n, float alpha)
5  {
6   vec *xv = (vec *)x;
7   vec *yv = (vec *)y;
8   vec valpha = {alpha};
9   n /= VECTOR_SIZE;
10  for(int i = 0; i < n; ++i) {
11   *yv += valpha * *xv;
12   xv++;
13   yv++;
14   }
15 }
```

## Resultados Numéricos

El objetivo de esta sección es determinar el esquema de codificación con el mejor desempeño computacional para las funciones principales de BLAS-1. Los resultados obtenidos son comparados con

[1] http://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

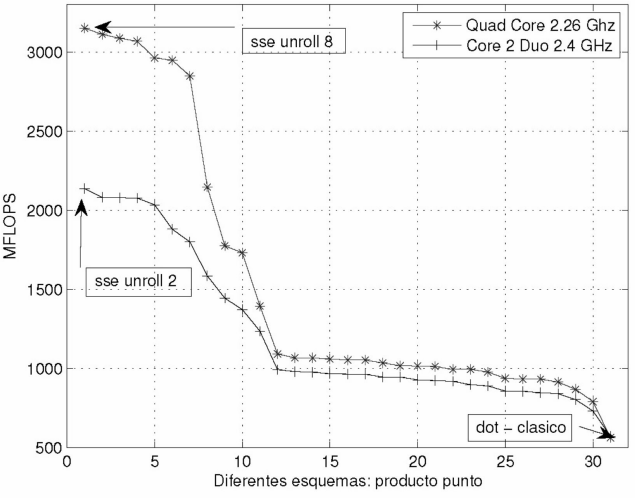


Figura 1: Producto punto codificado en 32 formas distintas con: unroll + xmm + sse.

la biblioteca de funciones de ATLAS. El programa fué escrito en C y compilado con GCC versión 4.5.

Como primera prueba investigamos el efecto que tiene en el desempeño computacional los esquemas de codificación. Para ello, implementamos de 32 formas distintas el producto punto, empleando las diversas técnicas de unroll + XMM + SSE. En la figura 1, se muestran los resultados obtenidos, empleando un vector de longitud $2^{23}$. Se observa que el producto punto implementado de forma clásica produce el peor rendimiento. Además se comparte que empleando la técnica de SSE con un esquema de unroll se producce el mejor desempeño. Por último, se aprecian tres zonas que corresponden a la técnica con SSE, posteriormente con MMX y por último el enfoque de unroll. Las funciones codificadas están disponibles escribiendo al primer autor del trabajo.

Es conocido que el desempeño computacional es afectado por las banderas de compilación utilizadas (Whaley y Dongarra, 1997). En este segundo experimento se analiza este comportamiento con los

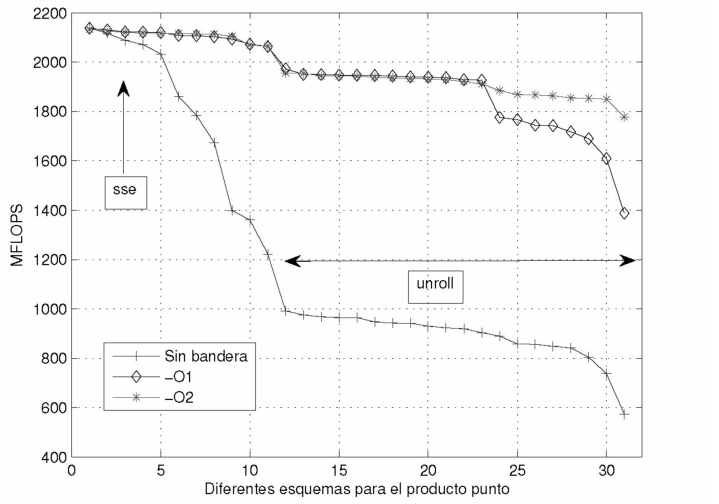


Figura 2: Producto punto compilado con diferentes banderas de compilación.

mismos datos anteriores empleando compilación sin bandera y con las opciones -O1, -O2. En la figura 2, se muestran los resultados obtenidos. El uso de banderas de compilación mejora en gran medida el rendimiento computacional. Sin embargo, cuando empleamos el enfoque son SSE el rendimiento no se ve afectado drásticamente. Este resultado es prometedor debido a que podemos emplear este esquema de codificación sin la necesidad de realizar busquedas exhaustivas para determinar la mejor combinación de banderas de compilación y codificación de los algoritmos.

Como tercer experimento determinaremos el rendimiento computacional de las principales funciones de BLAS-1. Las funciones más conocidas en esta categoría son: asum, axpy, copy, dot, nrm2. Debido a que la norma vectorial (norm2) tiene como núcleo el producto punto (dot) omitiremos su análisis. Adicionalmente, la función copy no contiene procesamiento en coma flotante y por lo tanto tambin es omitida. En la figura 3, se muestra la comparación de nuestro enfoque vs. ATLAS. Con la función axpy se obtuvo un rendimiento de


Figura 3: Comparación de rendimiento en diversas funciones de BLAS-1.

820 MFLOPS con ATLAS y 1250 con nuestro enfoque, obteniendo una ganancia del 52 %. Esto se logró con la función axpy-sse.c. Cabe mencionar que se implementaron de 24 formas distintas la función axpy y el resultado obtenido es un promedio estadístico. Con el producto punto se obtuvo una ganancia del 19 % el cual se logró con la combinación de SSE con unroll-2. Como se observa en la figura 3 con la función asum no se obtuvo una mejora significativa. Aunado a la ganancia obtenida en desempeño computacional, se logró determinar explícitamente el mejor esquema de codificación, el cual no es posible determinar cuando trabajamos con el API de ATLAS.

## Conclusiones

En el presente trabajo analizamos diversas técnicas computacionales de alto rendimiendo para incrementar el desempeño computacional en la bliblioteca de funciones BLAS-1. En particular, se realizó un enfoque combinacional en donde fusionamos la técnica de

loop unrolling, el procesamiento vectorial de datos con los registros XMM y el enfoque de SSE.

Con base en las funciones principales de BLAS-1, determinamos numéricamente una mejora del 52 % para la función axpy y un incremento del 19 % para el producto punto. Aunado a ello, se esclareció el esquema de codificación el cual permitirá emplear este enfoque en una gran variedad de aplicaciones en computación científica. Actualmente estamos desarrollando una biblioteca de funciones en la categoría BLAS-2, donde el objetivo es obtener una herramienta visual para determinar la función con el máximo rendimiento para un problema con dimensión establecida.

## Referencias

Aiken, A., y Nicolau, A. (1987). *Loop quantization: An analysis and algorithm*. Department of Computer Science, Cornell University.

Bouhamidi, A., Hached, M., y Jbilou, K. (2013). A meshless method for the numerical computation of the solution of steady burgers-type equations. *Applied Numerical Mathematics*, *74*(0), 95 - 110.

Chisnall, D. (2007, marzo). Programming with gcc. *InformIT Article is provided courtesy of Prentice Hall Professional*.

Davidson, J. W., y Jinturkar, S. (1995). Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. En *In proceedings of the 28th annual international symposium on microarchitecture* (pp. 125–132). IEEE Computer Society.

Golub, G. H., y Loan, C. F. V. (1996). *Matrix computations* (3rd ed.). The Johns Hopkins University Press.

Goto, K., y Van De Geijn, R. (2008). High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, *35*(1), 4:1–4:14.

Hennessy, J. L., y Patterson, D. A. (2003). *Computer architecture: A quantitative approach* (3.ª ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Higham, N. J. (2002). *Accuracy and stability of numerical algorithms (2. ed.)*. SIAM.

Inc., I. (2012, abril). Intel®64 and ia-32 architectures optimization reference manual (Vol. A) [Manual de software informático].

Lawson, C. L., Hanson, R. J., Kincaid, D. R., y Krogh, F. T. (1979, septiembre). Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, *5*(3), 308–323.

Mansour, A., y Gtze, J. (2013). Utilizing robustness of krylov subspace methods in reducing the effort of sparse matrix vector multiplication. *Procedia Computer Science*, *18*(0), 2406 - 2409. (2013 International Conference on Computational Science)

Mittal, M., Peleg, A., y Weiser, U. (1997). Mmx technology architecture overview. (Q3).

Napoli, E. D., Fabregat-Traver, D., Quintana-Ort, G., y Bientinesi, P. (2014). Towards an efficient use of the {BLAS} library for multilinear tensor contractions. *Applied Mathematics and Computation*, *235*(0), 454 - 468.

Trefethen, L. N., y Bau, D. (1997). *Numerical linear algebra*. SIAM.

Van Loan, C. F. (1999). *Introduction to scientific computing*. Prentice-Hall.

Wang, Q., Zhang, X., Zhang, Y., y Yi, Q. (2013). Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. En W. Gropp y S. Matsuoka (Eds.), *Sc* (p. 25). ACM.

Whaley, R. C., y Dongarra, J. J. (1997). *Automatically tuned linear algebra software* (Inf. Téc.). Knoxville, TN, USA: University of Tennessee.

Yzelman, A.-J. N., Roose, D., y Meerbergen, K. (2015). Chapter 27 - sparse matrix-vector multiplication: Parallelization and vectorization. En J. R. Jeffers (Ed.), *High performance parallelism pearls* (p. 457 - 476). Boston: Morgan Kaufmann.