

# Práctica 1: Fundamentos de Vectorización en x86: Extensiones Vectoriales, Vectorización Automática y Manual 30237 Multiprocesadores - Grado Ingeniería Informática Esp. en Ingeniería de Computadores

Jesús Alastruey Benedé y Víctor Viñals Yúfera  
Área Arquitectura y Tecnología de Computadores  
Departamento de Informática e Ingeniería de Sistemas  
Escuela de Ingeniería y Arquitectura  
Universidad de Zaragoza

Enero-2025

## Resumen

*El objetivo de esta práctica es familiarizarse con las extensiones vectoriales AVX y AVX-512 de Intel. Analizaremos las instrucciones SIMD generadas por el compilador al vectorizar de forma automática un bucle sencillo. También estudiaremos el código generado al vectorizar un bucle de forma manual mediante intrínsecos. Por último, ejecutaremos las versiones escalar y vectorial del bucle y compararemos su rendimiento.*

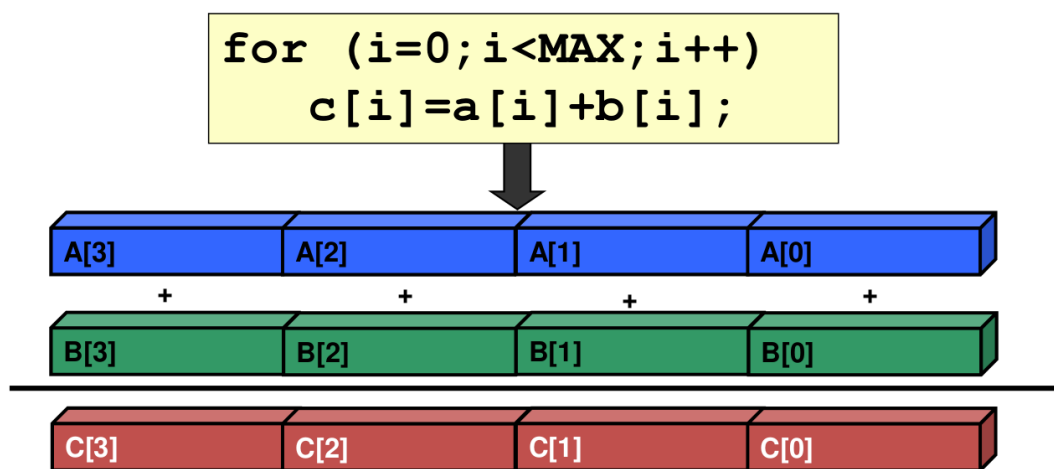


Figure 1: Operación vectorial<sup>1</sup>

<sup>1</sup>Stephen Blair-Chappell (Intel Compiler Labs). The significance of SIMD, SSE and AVX for Robust HPC Development.



Departamento de  
Informática e Ingeniería  
de Sistemas  
Universidad Zaragoza



Escuela de  
Ingeniería y Arquitectura  
Universidad Zaragoza

## Ficheros de trabajo

- p1.md y p1.pdf: ficheros en formatos *markdown* y *pdf* con el enunciado de la práctica.
- axpy.c: código C con dos versiones de un bucle, una **vectorizable** de forma automática (compilador):

```
for (unsigned int i = 0; i < LEN; i++)
    y[i] = alpha*x[i] + y[i];
```

y otra **vectorizada** de forma manual (programador) mediante intrínsecos SSE:

```
valpha = _mm_set1_ps(alpha); //valpha = _mm_load1_ps(&alpha);
for (unsigned int i = 0; i < LEN; i+= SSE_LEN)
{
    vX = _mm_load_ps(&x[i]);
    vY = _mm_load_ps(&y[i]);
    vaX = _mm_mul_ps(valpha, vX);
    vY = _mm_add_ps(vaX, vY);
    _mm_store_ps(&y[i], vY);
}
```

- **precision.h**: fichero donde se define
  - el tipo de dato **real**: **float** o **double**
  - el número de elementos procesados por cada instrucción vectorial para las extensiones vectoriales SSE, AVX y AVX-512
- **dummy.c**: contiene una función cuyo objetivo es forzar al compilador a generar código que ejecute el bucle de trabajo un número especificado de repeticiones.
- **init\_cpuname.sh**: script que inicializa la variable CPU (**export CPU=cpu\_model**), utilizada para organizar los resultados de los experimentos.
- **comp.sh**: script que compila versiones escalares y vectoriales del programa **axpy.c**. Soporta versiones recientes de los compiladores **gcc**, **clang** e **icc** (intel C compiler).  
Nota: compiladores disponibles en los sistemas del DIIS:
  - **gcc 13.2.1**, **gcc 11.2.0**
- **run.sh**: script que ejecuta las compilaciones escalar y vectorial del programa de trabajo.
- **sde.sh**: script para ejecutar las compilaciones escalar y vectorial del programa de trabajo con la herramienta Intel SDE.
- **comp.run.all.len.sh**: script para compilar y ejecutar las versiones escalares y vectoriales del programa **axpy.c** con distintos tipos de datos y longitudes de los vectores de trabajo.

## Trabajo previo

### 1. Requerimientos hardware y software:

- CPU con soporte de la extensión vectorial AVX
- SO Linux

Los equipos del laboratorio L0.04 y L1.02 cumplen los requisitos indicados. Puede trabajarse en dichos equipos de forma presencial y también de forma remota si hay alguno arrancado con Linux. Para saber qué máquinas de un laboratorio están accesibles de forma remota, ejecutar la siguiente orden en **hendrix**:

```
$ rcmds -f lab102 -s -- uptime
```

Puede cambiarse el nombre de laboratorio cuyos equipos se quieren inspeccionar. En la web de la asignatura se proporciona otro script que genera una lista de las direcciones IP y SO de las máquinas que están arrancadas en el L1.02.

### 2. Identificar la plataforma de trabajo (lab004, lab102, equipo propio):

- Características de la plataforma de trabajo: CPU, sistema operativo, versión del compilador ...
- Detallar qué extensiones vectoriales soporta la CPU. Ayuda: consulta el fichero **/proc/cpuinfo**.

En caso de trabajar en un equipo del DIIS, para usar la reciente versión 13.2.1 del compilador **gcc** hay que editar el fichero oculto **software** que está en vuestro **\$HOME** y añadir la palabra clave **gcc**. Este cambio tendrá efecto en los terminales que se abran a partir de ese momento. Para verificar la versión de **gcc**:

```
$ gcc -v
[...]
gcc versión 13.2.1 (GCC)
```

Nota: conviene señalar que cuando se hace login en Linux con el sistema gráfico por defecto, normalmente los terminales que se ejecutan no utilizan un `login shell` por defecto. Esto significa que muchas variables de entorno necesarias no se inicializan, por ejemplo, la que controla la versión de `gcc`. Esta incidencia se soluciona cambiando las preferencias del terminal para que por defecto utilice un `login shell`.

En `gnome-terminal`:

```
Menu->Edit->Preferences->Unnamed->Command:  
[x] Run command as a login shell
```

3. Inicializar la variable de entorno CPU. Se utiliza para organizar en directorios diferentes los experimentos realizados con distintos procesadores. Para ello hay que ejecutar:

```
$ source ./init_cpuname.sh
```

Esta orden ejecuta el script en el shell existente, lo que permite que la variable creada por el script esté disponible después de que el script finalice su ejecución. Si se invoca el script directamente

```
$ ./init_cpuname.sh
```

se ejecuta en otro shell, por lo que la variable de entorno no estará inicializada en el shell de trabajo.

---

## Parte 1. Vectorización automática

En esta parte vamos a estudiar la capacidad para vectorizar bucles del compilador `gcc`. También analizaremos el rendimiento del código vectorizado.

1. Analiza y comprende el contenido de los ficheros `axy.c` y `comp.sh`. AXPY es un conocido kernel que calcula un producto vector-escalar y suma el resultado a otro vector. Hemos escogido un tamaño del vector que permite su almacenamiento en cache. De esta forma, los elementos de los vectores se leerán de memoria cache en lugar de memoria principal. `comp.sh` es un fichero para automatizar las tareas de compilación. Observa con detenimiento las opciones de compilación que especifican las extensiones vectoriales a utilizar.
2. Compila con `gcc` las distintas versiones escalares y vectoriales (`avx`, `avx+fma`, `avx512`) del programa `axy.c`:

```
$ ./comp.sh
```

Observa los informes del compilador que se han generado en el directorio `reports`, en especial la información correspondiente al bucle interno en la función `axy()`. ¿Ha vectorizado el bucle en `axy()`?

3. Analiza los ficheros que contienen el ensamblador de los siguientes códigos escalares y vectoriales: `esc.avx`, `vec.avx`, `vec.avxfma` y `vec.avx512`. Busca las instrucciones correspondientes al cuerpo del bucle en la función `axy()`:

```
y[i] = alpha*x[i] + y[i]
```

Indica qué instrucciones se usan para:

- leer los vectores `x[]` e `y[]` de memoria
- multiplicar y sumar
- escribir el vector resultado `y[]` en memoria

¿Cuántos elementos de cada vector se procesan en una iteración del bucle en ensamblador (`esc.avx`, `vec.avx` y `vec.avx512`)?

Sabiendo el tipo de dato procesado y su tamaño, ¿cuántos bytes de cada vector se procesan en una iteración del bucle en ensamblador (`esc.avx`, `vec.avx` y `vec.avx512`)?

¿Hay alguna diferencia entre las instrucciones AVX y AVX-512?

4. ¿Cuántas instrucciones se ejecutan en el bucle interno (`esc.avx`, `vec.avx`, `vec.avxfma` y `vec.avx512`)?

```
for (unsigned int i = 0; i < LEN; i++)  
    y[i] = alpha*x[i] + y[i];
```

Por ejemplo, para la versión `esc.avx`:

$$ICOUNT = N \cdot LEN = 6 \cdot 1024 = 6144 \text{ instrucciones}$$

siendo  $N$  el número de instrucciones del cuerpo del bucle (6) y  $LEN$  el número de elementos del vector (número iteraciones del bucle escalar, 1024).

Calcula la reducción en el número de instrucciones respecto la versión `esc.avx`.

versión	icount	reducción(%)	reducción(factor)
esc.avx	6144	0	1.0
vec.avx			
vec.avxfma			
vec.avx512			

5. Ejecuta los programas compilados anteriormente:

```
$ ./run.sh
```

Observa los ficheros de salida que se han generado en el directorio **results**. La columna **Time** muestra el tiempo medio de ejecución del bucle interno (kernel `axpy`), no el tiempo de todas sus ejecuciones. La columna **TPE** (tiempo por elemento) muestra el tiempo medio de ejecución por elemento (tiempo medio de ejecución dividido por la longitud de los vectores). Esta métrica permite comparar los rendimientos al ejecutar `axpy` con vectores de distinto tamaño.

¿Qué ocurre al ejecutar la versión `vec.avx512`?

```
$ ./axpy.1k.single.vec.avx512.gcc
```

Para obtener más información de lo que ha ocurrido, carga en `gdb` el binario y el fichero `core` generado:

```
$ gdb axpy.1k.single.vec.avx512.gcc core.xyz
```

Sustituye `xyz` por el identificador del proceso que ha generado el fichero `core`.

`gdb` nos mostrará la línea de código que ha provocado el error.

En caso de que no se haya generado fichero `core`, habilita su creación y vuelve a ejecutar:

```
$ ulimit -c unlimited
$ ./axpy.1k.single.vec.avx512.gcc
```

Para ver la última instrucción ejecutada:

```
$ (gdb) layout asm
```

6. A partir de los tiempos de ejecución obtenidos en el punto anterior, calcula las siguientes métricas para todas las versiones ejecutadas:

- Aceleraciones (*speedups*) de las versiones vectoriales sobre sus escalares (`vec.avx` y `vec.avxfma` respecto `esc.avx`).
- Rendimiento (*R*) en GFLOPS.
- Rendimiento pico ( $R_{\text{pico}}$ ) teórico de un núcleo (*core*), en GFLOPS. Para las versiones escalares, considerar que las unidades funcionales trabajan en modo escalar. Considerar asimismo la capacidad FMA de las unidades funcionales solamente para las versiones compiladas con soporte FMA.
- Velocidad de ejecución de instrucciones ( $V_I$ ), en Ginstrucciones por segundo (GIPS).

versión	tiempo(ns)	speed-up	R(GFLOPS)	$R_{\text{pico}}$ (GFLOPS)	$V_I$ (GIPS)
esc.avx		1.0			
vec.avx					
vec.avxfma					

Nota: GFLOPS =  $10^9$  FLOPS. GIPS =  $10^9$  IPS.

Comenta brevemente la tabla de resultados obtenidos (tiempo de ejecución, *speedup*, rendimiento, velocidad).

¿La velocidad de ejecución de instrucciones es un buen indicador de rendimiento?

¿Son consistentes los resultados (*checksums*) de las distintas versiones?

7. **Optativo.** Para obtener estadísticas de ejecución, ejecuta con `perf` los programas compilados:

```
$ ./run.sh -s
```

8. **Optativo.** Usa la herramienta Intel Software Development Emulator (Intel SDE) [1] para obtener el número total de instrucciones ejecutadas por el bucle en la función `axpy()`. Para ello, recompila el fichero `axpy.c` especificando un valor más reducido para la constante `NTIMES`:

```
$ ./comp.sh -T 10
```

Y ejecuta:

```
$ ./sde.sh
```

Este script ejecuta bajo SDE las versiones `esc.avx`, `vec.avx`, `vec.avxfma` y `vec.avx512` y genera informes con la cuenta de instrucciones correspondiente al bucle en la función `axpy()`. Por ejemplo, para la versión `esc.avx`:

```
BLOCK: 0   PC: 0000000000400720   ICOUNT: 96636764160   EXECUTIONS: 16106127360
#BYTES: 36   %: 99.7   cumltv%: 99.7   FN: axpy
```

El número total de instrucciones ejecutadas (`ICOUNT`) es 61440. Se puede comprobar que coincide con el obtenido mediante la siguiente fórmula:

$$ICOUNT = N \cdot LEN \cdot NTIMES = 6 \cdot 1024 \cdot 10 = 61440$$

siendo  $N$  el número de instrucciones del cuerpo del bucle (6),  $LEN$  las iteraciones del bucle interno (1024) y  $NTIMES$  las iteraciones del bucle externo (10).

El número de ejecuciones del cuerpo del bucle interno (`EXECUTIONS`) es 10240, que coincide con el producto  $LEN \cdot NTIMES = 1024 \cdot 10$ .

El número de instrucciones de una ejecución del bucle interno se puede obtener dividiendo el número total de instrucciones ejecutadas por el número de veces que se ha ejecutado el bucle interno:

$$ICOUNT_{bucle} = \frac{ICOUNT}{NTIMES} = \frac{61440}{10} = 6144$$

Nota: verifica que las siguientes líneas de código estén comentadas para acelerar la ejecución:

```
axpy_intr_SSE();
axpy_intr_AVX();
```

Tras terminar este apartado, recompila con el valor original de `FLOP_COUNT`:

```
$ ./comp.sh
```

## Parte 2. Vectorización manual mediante intrínsecos

1. Observa el informe del compilador correspondiente a la compilación con soporte AVX. ¿Hay alguna indicación de que haya vectorizado el bucle en `axpy_intr_SSE()`?
2. Escribe una nueva versión del bucle, `axpy_intr_AVX()`, vectorizando de forma manual con intrínsecos AVX. Está el esqueleto de la función. El siguiente enlace puede ser de utilidad:

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

3. Quita los comentarios de las llamadas a las funciones `axpy_intr_SSE()` y `axpy_intr_AVX()`. Recompila y ejecuta el código:

```
$ ./comp.sh
$ ./run.sh
```

Analiza el fichero que contiene el ensamblador de la versión AVX y busca las instrucciones correspondientes al bucle en `axpy_intr_AVX()`.

¿Hay alguna diferencia con las instrucciones correspondientes al bucle en `axpy()`?

¿Hay diferencia en el rendimiento de las funciones `axpy()` y `axpy_intr_AVX()` (binario AVX)?

¿Hay diferencia en el rendimiento de las versiones `vec.avx` y `vec.avxfma` correspondientes al bucle `axpy_intr_AVX()`? Compara las instrucciones generadas por el compilador para ambas versiones.

## Apartados optativos

1. El flag `-march=native` genera flags de compilación para la máquina donde se está compilando. Para saber los flags que se generan:

```
$ gcc -march=native -E -v - </dev/null 2>&1 | grep cc1
```

Para compilar y ejecutar las versiones escalar y vectorial native:

```
$ ./comp.sh -n
$ ./run.sh -n
```

Compara el rendimiento de las versiones native (esc.native y vec.native) con el de las versiones avx (esc.avx y vec.avx).

2. Repite los puntos anteriores con vectores de precisión doble (**double**). El tipo de dato **real** se puede seleccionar como **float** o **double** mediante la variable **p** en el fichero **comp.sh**.
3. Repite los puntos anteriores con el compilador **icc**. Se recomienda que utilicéis una versión reciente, que podéis conseguir en el siguiente enlace:

<https://software.intel.com/en-us/qualify-for-free-software/student>

Es probable que haya que modificar los *flags* de compilación del fichero **comp.sh**.

4. Analiza el rendimiento y las aceleraciones del código procesando otros tamaños de vectores. Para ello, puedes ayudarte del script **comp.run.all.len.sh**.
5. Repetir la parte 1 en **berlin** (2xAMD EPYC 7313P), **pilgor** (2xHuawei Kunpeng 920-4826) o en otro sistema que disponga de un procesador con extensiones vectoriales.

## Referencias

[1] Intel® Software Development Emulator. <https://software.intel.com/en-us/articles/intel-software-development-emulator>

## Bibliografía relacionada

- Stephen Blair-Chappell (Intel Compiler Labs). The significance of SIMD, SSE and AVX for Robust HPC Development.
- How do I achieve the theoretical maximum of 4 FLOPs per cycle?  
<http://stackoverflow.com/questions/8389648/how-do-i-achieve-the-theoretical-maximum-of-4-flops-per-cycle>
- Obtaining peak bandwidth on Haswell in the L1 cache: only getting 62%.  
<http://stackoverflow.com/questions/25899395/obtaining-peak-bandwidth-on-haswell-in-the-l1-cache-only-getting-62>
- Do FMA always produce the same result as a mul then add instruction?  
<http://stackoverflow.com/questions/29086377/do-fma-fused-multiply-add-instructions-always-produce-the-same-result-as-a-mul>