# Práctica 2: Limitaciones a la Vectorización: Alineamiento, Solapamiento (Aliasing), Accesos a Memoria No Secuenciales (Stride), Condicionales. 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

2-marzo-2017

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

## Resumen

*El objetivo de esta práctica es identificar las limitaciones existentes a la hora de vectorizar código en una plataforma x86 y aprender a superarlas. Analizaremos cómo afecta al proceso de vectorización el alineamiento de las variables en memoria, su solapamiento, los accesos a memoria no secuenciales (con stride) y la presencia de sentencias condicionales. También analizaremos su impacto en el rendimiento.*

## Ficheros de trabajo

Se proporciona un paquete `.tar.gz` en el que se encuentran los siguientes ficheros:

- `p2.md`: fichero de texto en formato *markdown* con el guión de la práctica.
- `axpy_align.c`: programa en C con distintas versiones del siguiente bucle:

```
for (int i = 0; i < LEN - 1; i++)
    y[i] = alpha*x[i] + y[i];
```

  Con este programa analizaremos el efecto de la alineación de vectores en la vectorización y en el rendimiento.
- `axpy_alias.c`: programa en C con distintas versiones del bucle anterior. Con este programa analizaremos el efecto del solapamiento de vectores en la vectorización y en el rendimiento.
- `precision.h`: fichero donde se define el tipo de dato `real` (`float` o `double`) y el número de elementos procesados por cada instrucción vectorial para distintas extensiones vectoriales.
- `dummy.c`: 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 los ficheros fuente.
- `run.sh`: script que ejecuta las compilaciones de los programas de trabajo.

---

## Consideraciones previas

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. En el guión de la práctica 1 se explica cómo descubrir qué máquinas de un laboratorio están accesibles de forma remota.
2. Inicializar la variable de entorno CPU. Se utiliza para organizar los experimentos realizados en distintas máquinas en distintos directorios.
   a. Editar el fichero `init_cpuname.sh` y, si es necesario, cambiar el nombre de la CPU
   b. Ejecutar
      `$ source ./initcpuname.sh`

---

## Parte 1. Efecto del alineamiento de los vectores en memoria

En esta parte vamos a trabajar con el fichero `axpy_align.c`.

La función `axpy_align_v1()` calcula el kernel AXPY. Todos los vectores están alineados con el tamaño de AVX, es decir, su dirección inicial es múltiplo de 32 bytes (256 bits).

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

La función `axpy_align_v2()` hace el mismo cálculo pero con vectores **NO** alineados (los vectores se procesan desde el elemento con índice 1):

```
for (int i = 0; i < LEN; i++) {
    y[i+1] = alpha*x[i+1] + y[i+1];
}
```

1. Compila con gcc el programa `axpy_align.c`:

   `$ ./comp.sh -f axpy_align.c`

   Observa el informe del compilador. ¿Ha vectorizado los bucles en `axpy_align_v1()` y `axpy_align_v2()`? Si el compilador ha aplicado alguna transformación, indícala.
2. Analiza el fichero que contiene el ensamblador del código vectorial y busca las instrucciones correspondientes a los bucles en `axpy_align_v1()` y `axpy_align_v2()`. ¿Qué diferencias hay?
3. Las funciones `axpy_align_v4()` y `axpy_align_v5()` implementan con instrínsecos los bucles de las funciones `axpy_align_v1()` y `axpy_align_v2()` respectivamente. En el primer caso los accesos a memoria son alineados y en el segundo son no alineados.

   Observa de nuevo el informe del compilador. ¿Ha vectorizado los bucles en `axpy_align_v4()` y `axpy_align_v5()`?

   Analiza el fichero que contiene el ensamblador del código vectorial y busca las instrucciones correspondientes al bucle en `axpy_align_v5()`. Indica qué instrucciones se usan para:

   - leer los vectores de memoria

- escribir el vector resultado en memoria

4. Ejecuta el programa `axpy_align`:

   ```
   $ ./run.sh -f axpy_align.c
   ```

   Comenta brevemente los tiempos de ejecución obtenidos.

5. La función `axpy_align_v6()` es igual que `axpy_align_v4()` excepto en que los vectores se procesan desde el elemento con índice 1.

   Quita el comentario en la siguiente línea del programa principal:

   ```
   // axpy_align_v6();
   ```

   Recompila y ejecuta:

   ```
   $ ./comp.sh -f axpy_align.c
   $ ./run.sh -f axpy_align.c
   ```

   ¿Qué ocurre? ¿Cuál crees que es la causa?

6. (OPTATIVO) La función `axpy_align_v3a()` hace el mismo cálculo que `axpy_align_v1()` pero con el vector `y[]` **NO** alineado (se procesa desde el elemento con índice 1):

   ```
   for (int i = 0; i < LEN; i++) {
       y[i+1] = alpha*x[i] + y[i+1];
   }
   ```

   Quita el comentario en la siguiente línea del programa principal:

   ```
   // axpy_align_v3a();
   ```

   Elimina asimismo las directivas `#if 0 ... #endif` en torno a la función `axpy_align_v3a()`.

   Recompila y ejecuta:

   ```
   $ ./comp.sh -f axpy_align.c
   $ ./run.sh -f axpy_align.c
   ```

   Observa el informe del compilador.
   ¿Ha vectorizado el bucle en `axpy_align_v3a()`?
   Si el compilador ha aplicado alguna transformación, indícala.

   Analiza el fichero que contiene el ensamblador del código vectorial y busca la instrucción correspondientes al bucle en `axpy_align_v3a()` que se usan para escribir el vector resultado en memoria.

   Comenta brevemente el tiempo de ejecución obtenido.

   Escribe una variante `axpy_align_v3b()` que fuerce el alineamiento de y[1].

## Parte 2. Efecto del solapamiento de las variables en memoria

En esta parte vamos a trabajar con el fichero `axpy_alias.c`.

La función `axpy_alias_v1()` calcula el kernel ZAXPY (z = ax + y). Las direcciones de los vectores origen y destino son parámetros de la función.

```
for (int i = 0; i < LEN; i++) {
    vz[i] = alpha*vx[i] + vy[i];
}
```

1. Compila con gcc el programa `axpy_alias.c`:

   ```
   $ ./comp.sh -f axpy_alias.c
   ```

   Observa el informe del compilador.
   ¿Ha vectorizado el bucle en `axpy_alias_v1()`? Indica las transformaciones realizadas por el compilador.

Analiza el fichero que contiene el ensamblador del código vectorial AVX e identifica **TODOS** los bloques de código correspondientes al bucle. Ayuda: ten presente las transformaciones realizadas por el compilador.

2. La función `axpy_alias_v2()` es una versión de `axpy_alias_v1()` en la que se han declarado como `restrict` los punteros que se pasan como parámetros:

   ```
   int axpy_alias_v2(real * restrict vx, real * restrict vy, real * restrict vz)
   ```

   Busca en el actual estándar de C el significado de la palabra clave `restrict` y explica su efecto en esta función.

   Repite las tareas del punto anterior (análisis del informe del compilador y del código ensamblador) con la función `axpy_alias_v2()`.

3. La función `axpy_alias_v3()` es una versión de `axpy_alias_v1()` en la que se ha insertado antes del bucle la siguiente línea:

   ```
   #pragma GCC ivdep
   ```

   Busca en la documentación de `gcc` el significado del citado pragma y explica su efecto en esta función.

   Repetir las tareas del punto 7 (análisis del informe del compilador y del código ensamblador) con la función `axpy_alias_v3()`.

4. La función `axpy_alias_v4()` es una versión de `axpy_alias_v2()` en la que el bucle trabaja con las siguientes variables locales:

   ```
   real *xx = __builtin_assume_aligned(vx, ARRAY_ALIGNMENT);
   real *yy = __builtin_assume_aligned(vy, ARRAY_ALIGNMENT);
   real *zz = __builtin_assume_aligned(vz, ARRAY_ALIGNMENT);
   ```

   Busca en la documentación de gcc el significado de `__builtin_assume_aligned()` y explica su efecto en esta función.

   Repetir las tareas del punto 7 (análisis del informe del compilador y del código ensamblador) con la función `axpy_alias_v4()`.

5. Ejecutar el programa:

   ```
   $ ./run.sh -f axpy_alias.c
   ```

   Comenta brevemente los tiempos de ejecución obtenidos. Relaciona los resultados con las características de cada ejecución.

## Parte 3. Efecto de los accesos no secuenciales (stride) a memoria

En esta sección vamos a trabajar con el fichero `axpy_stride.c`. La función `axpy_stride_v1()` calcula AXPY(S=2), es decir, el kernel AXPY para **uno de cada dos elementos**:

```
for (int i = 0; i < LEN; i+=2) {
    y[i] = alpha*x[i] + y[i];
}
```

La función `axpy_stride_v2()` hace el mismo cálculo pero las direcciones de los vectores son parámetros de la función.

1. Compila con `gcc` el programa `axpy_stride.c`:

   ```
   $ ./comp.sh -f axpy_stride.c
   ```

   Observar el informe del compilador. ¿Ha vectorizado los bucles en `axpy_stride_v1()` y `axpy_stride_v2()`?

2. Para este apartado se facilita el informe de compilación y el código ensamblador generados por `icc` (por si no tenéis disponible una versión reciente del mismo). Observa el informe generado por el compilador `icc`. ¿Ha vectorizado los bucles en `axpy_stride_v1()` y `axpy_stride_v2()`?

   Analiza el fichero que contiene el ensamblador del código vectorial y echa un vistazo a las instrucciones correspondientes al bucle.

¿Cuántas instrucciones vectoriales corresponden al cuerpo del bucle? Ayuda: utiliza las etiquetas al final de cada línea para identificarlas.

(OPTATIVO) Detalla las operaciones realizadas por las instrucciones vectoriales del bucle con stride.

3. Ejecuta los programas generados por `gcc` e `icc` (éste último es facilitado por si no tenéis disponible una versión reciente de `icc`):

   ```
   $ ./run.sh -f axpy_stride.c
   ```

   Calcula la aceleración (*speedup*) de la versión `icc` sobre la `gcc`.
   Comenta muy brevemente los tiempos de ejecución obtenidos.

## Parte 4. Efecto de las sentencias condicionales en el cuerpo del bucle

En esta sección vamos a trabajar con el fichero `cond.c`. La función `cond_vec()` contiene una sentencia condicional en el cuerpo del bucle:

```
if (y[i] < (real) (1.0/1023))
    z[i] = y[i];
else
    z[i] = x[i];
```

La función `cond_esc()` realiza el mismo cálculo, pero se ha inhibido la vectorización con la directiva `__attribute__((optimize("no-tree-vectorize")))`.

1. Compila con gcc el programa `cond.c`:

   ```
   $ ./comp.sh -f cond.c
   ```

   Observar el informe del compilador. ¿Ha vectorizado el bucle en `cond_vec()`?

2. Analiza el fichero que contiene el ensamblador y echa un vistazo a las instrucciones correspondientes al bucle vectorizado.
   ¿Cuántas instrucciones vectoriales corresponden al cuerpo del bucle?
   Detalla las operaciones realizadas por las instrucciones vectoriales del bucle.

3. Ejecuta el programa generado:

   ```
   $ ./run.sh -f cond.c
   ```

   Calcula la aceleración (*speedup*) de la versión vectorial sobre la escalar.

## Bibliografía


- Estándar C11 (documento WG14 N1570 con fecha 12-04-2011, es la última versión pública disponible de C11). Fecha de consulta: 6-marzo-2016. Disponible en:
  http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf
- Auto-vectorization with gcc 4.7. Fecha de consulta: 6-marzo-2016. Disponible en:
  http://locklessinc.com/articles/vectorize/