

Práctica 2: Limitaciones a la Vectorización en x86: 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

Febrero-2025



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.

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 remota. En el enunciado 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 en directorios diferentes los experimentos realizados con distintos procesadores. Para ello hay que 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`. Analizaremos el efecto de la alineación de vectores en la vectorización y en el rendimiento.

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

```
for (unsigned 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, ya que se procesan desde el elemento con índice 1:

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

1. Compila con `gcc` el fichero `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 informe del compilador indica que ha aplicado alguna transformación, reséñala.

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_v1_intr()` y `axpy_align_v2_intr()` implementan con intrínsecos 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_v1_intr()` y `axpy_align_v2_intr()`?

Analiza el fichero que contiene el ensamblador del código vectorial y busca las instrucciones correspondientes al bucle en `axpy_align_v1_intr()` y `axpy_align_v2_intr()`. ¿Qué diferencias hay entre las versiones `v1` y `v1_intr`? ¿Y entre las versiones `v2` y `v2_intr`?

Nota: para generaciones de procesadores anteriores a Haswell, el compilador puede dividir cada acceso a memoria de 32 bytes (256 bits) en dos accesos de 16 bytes (128 bits). Ver más detalles en [1].

4. Observa en el siguiente enlace el ensamblador del bucle en `axpy_align_v2()` generado por las versiones 7.2 y 13.2 de `gcc`:

<https://godbolt.org/z/rxzT3dTWM>

Compara las dos versiones de código e identifica las diferencias.

Si en tu sistema está disponible la versión 7.2 de `gcc`, puedes compilar y ejecutar con las siguientes órdenes:

```
$ ./comp.sh -f axpy_align.c -c gcc-7
$ ./run.sh -f axpy_align.c -c gcc-7
```

5. Ejecuta las distintas funciones del programa `axpy_align`:

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

Comenta brevemente los tiempos de ejecución obtenidos.

6. La función `axpy_align_v1_intru()` es igual que `axpy_align_v1_intr()` excepto en que el vector `x[]` se procesa desde el elemento con índice 1. Quita el comentario en la siguiente línea del programa principal:

```
// axpy_align_v1_intru();
```

Recompila y ejecuta:

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

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

Para obtener más información de lo que ha ocurrido, ejecuta el binario con `gdb`:

```
$ cd $CPU
$ gdb axpy_align.1k.single.avx2.gcc
(gdb) run
```

`gdb` nos mostrará la línea de código que ha provocado el error. Para ver la última instrucción ejecutada:

```
(gdb) layout asm
```

Parte 2. Efecto del solapamiento de las variables en memoria

En esta parte vamos a trabajar con el fichero `axpy_alias.c`. Analizaremos el efecto del solapamiento de vectores en la vectorización y en el rendimiento.

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

```
for (unsigned 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.

Analiza el informe del compilador y el código ensamblador de 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.

Analiza el informe del compilador y el código ensamblador de 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.

Analiza el informe del compilador y el código ensamblador de 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 código ejecutado.

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_vec()` calcula `axpy(S=2)`, es decir, el kernel AXPY para **uno de cada dos elementos**:

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

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

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

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

Observa el informe del compilador. ¿Ha vectorizado el bucle en `axpy_stride_vec()`?

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 interno? Ayuda: utiliza las etiquetas al final de cada línea para identificarlas.

OPTATIVO. Detalla las operaciones realizadas por las instrucciones vectoriales del bucle interno en `axpy_stride_vec()`.

2. Compila con `icx` el programa `axpy_stride.c`:

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

Este compilador está solamente disponible en lab000. Para usarlo hay que editar el fichero oculto `.software` que está en vuestro `$HOME` y añadir la palabra clave `intel`. Este cambio tendrá efecto en los terminales que se abran a partir de ese momento. Para verificar que funciona correctamente:

```
$ icx -v
Intel(R) oneAPI DPC++/C++ Compiler 2023.2.0 (2023.2.0.20230622)
```

Observa el informe generado por el compilador. ¿Ha vectorizado el bucle en `axpy_stride_vec()`?

En caso afirmativo, analiza el fichero que contiene el ensamblador del código vectorial y echa un vistazo a las instrucciones correspondientes al bucle.

Copia el binario generado por `icx` al directorio donde se alojan los binarios que has generado durante la sesión.

3. Ejecuta los programas generados por `gcc` e `icx`:

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

Calcula las aceleraciones (*speedup*) de las versiones vectoriales sobre las escalares.

Calcula las aceleraciones (*speedup*) de las versiones `icx` sobre las `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] < umbral)
    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 interno? 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.

Referencias

[1] Why doesn't gcc resolve `__mm256_loadu_pd` as single `vmovupd`? <https://stackoverflow.com/questions/52626726/why-doesnt-gcc-resolve-mm256-loadu-pd-as-single-vmovupd>

Bibliografía

- Estándar C11. Fecha de consulta: 20-febrero-2024. Disponible en:
<https://www.iso.org/standard/57853.html>
- Auto-vectorization with gcc 4.7. Fecha de consulta: 20-febrero-2024. Disponible en:
<http://locklessinc.com/articles/vectorize/>
- Auto-vectorization in GCC. Fecha de consulta: 20-febrero-2024. Disponible en:
<https://gcc.gnu.org/projects/tree-ssa/vectorization.html>