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

23-febrero-2017

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.

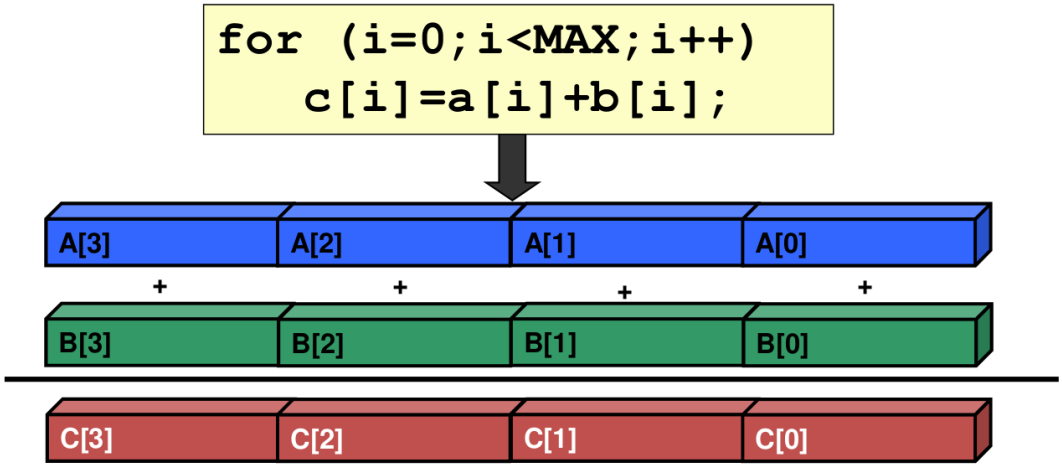

Figure 1: Operación vectorial[1]

[1]Stephen Blair-Chappell (Intel Compiler Labs). The significance of SIMD, SSE and AVX for Robust HPC Development.

Ficheros de trabajo

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

- `p1.md`: fichero de texto en formato *markdown* con el guión de la práctica.
- `axpy.c`: programa en C con dos versiones de un bucle, una **vectorizable** de forma automática (compilador):

```
for (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 (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`: 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` e `icc` (intel C compiler).
  Nota: compiladores disponibles en las maquinas del DIIS:
  - `gcc 6.2.0`, `gcc 5.3.1`, `gcc 4.8.2`
  - `icc 10.1` (sí, es bastante antiguo, por lo que el script no lo soporta)
- `run.sh`: script que ejecuta las compilaciones escalar y vectorial del programa 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):
   a. Características de la plataforma de trabajo: CPU, sistema operativo, versión del compilador ...

b. Detallar qué extensiones vectoriales soporta la CPU. No respondáis con volcados crudos de comandos. Ayuda: consulta el fichero `/proc/cpuinfo`.

En caso de trabajar en un equipo del DIIS, para usar la reciente versión 6.2.0 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 6.2.0 (GCC)
```

Y antes de usar el compilador de intel (`icc`):

```
$ . /usr/local/intel/config.sh  # no dejarse el punto inicial
```

En los equipos del DIIS no es posible trabajar en una misma sesión con `icc` y una versión reciente de `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. La solución consiste en cambiar las preferencias del terminal para que por defecto utilice un `login shell`.

En `gnome-terminal`:

```
Menu->Edit->Profiles->Default->Title and Command:
[x] Run command as a login shell
```

3. 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 ./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. Analizar y comprender el contenido del fichero `comp.sh`. Observar con detenimiento las opciones de compilación que especifican las extensiones vectoriales a utilizar.
2. Compilar con `gcc` las versiones escalares (noavx, noavx512) y vectoriales (avx, avx+fma, avx512) del programa `axpy.c`:

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

   Observa los informes del compilador, en especial la información correspondiente al bucle interno en la función `axpy()`. ¿Ha vectorizado el bucle en `axpy()`?
3. Analiza el fichero que contiene el ensamblador de la versión escalar (noavx) y busca las instrucciones correspondientes al bucle en la función `axpy()`:

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

   Indica qué instrucciones se usan para:

   - leer los vectores de memoria

- multiplicar y sumar
- escribir el vector resultado en memoria

¿Cuántos elementos de cada vector se procesan en cada iteración?
Sabiendo el tipo de dato procesado y su tamaño, ¿cuántos bytes de cada vector se procesan en cada iteración?

4. Analiza los ficheros que contienen el ensamblador de los códigos vectoriales (AVX, AVX+FMA y AVX-512) y busca las instrucciones correspondientes al bucle en la función `axpy()`:

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

   Indica qué instrucciones se usan para:

   - leer los vectores de memoria
   - multiplicar y sumar
   - escribir el vector resultado en memoria

   ¿Hay alguna diferencia entre las instrucciones AVX y AVX-512?
   ¿Cuántos elementos de cada vector se procesan en cada iteración en cada caso (AVX/AVX-512)?
   Sabiendo el tipo de dato procesado y su tamaño, ¿cuántos bytes de cada vector se procesan en cada iteración en cada caso (AVX/AVX-512)?

5. Ejecuta los programas compilados anteriormente:

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

   ¿Qué ocurre al ejecutar la versión AVX-512?

   ```
   $ ./axpy.2k.single.avx512.gcc
   ```

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

   ```
   $ gdb axpy.2k.single.avx512.gcc 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.2k.single.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:

   - Aceleraciones (*speedups*) de las versiones vectoriales sobre sus escalares (AVX y AVX+FMA respecto noavx).
   - Rendimiento de todas las versiones ejecutadas (GFLOPS).

   Comenta brevemente la tabla de resultados obtenidos (tiempo de ejecución, rendimiento, *speedup*).
   ¿Son consistentes los resultados (*checksums*) de las versiones escalar y vectorial?
   ¿Qué ocurre con los resultados (*checksums*) de las versiones AVX+FMA?

Parte 2. Vectorización manual mediante intrínsecos

7. 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()`?

8. Escribir una nueva versión del bucle (`axpy_intr_AVX()`) vectorizando de forma manual con intrínsecos AVX (está el esqueleto de la función).

   Recompila y ejecuta el código:

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

Analiza el fichero que contiene el ensamblador del código 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)?

Apartados optativos

10. 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`.
11. Analiza el rendimiento y las aceleraciones del código procesando otros tamaños de vectores. Para ello, puedes ayudarte del script `comp.run.all.len.sh`.
12. Repite los puntos anteriores con el compilador `icc`. Para este paso 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

    En caso de usar la versión disponible en los laboratorios (10.1) tendrás que modificar los *flags* de compilación del fichero `comp.sh`.

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