Práctica 6: Paralelismo multinivel: vectorización y ejecución multinúcleo
30237 Multiprocesadores - Grado Ingeniería Informática
Esp. en Ingeniería de Computadores

Jesús Alastruey Benedé, Víctor Viñals Yúfera y Rubén Gran Tejero

Área Arquitectura y Tecnología de Computadores
Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

17-mayo-2019

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Resumen

El objetivo principal de esta práctica es combinar paralelismo a nivel vectorial (AVX) con paralelismo a nivel de núcleo (OpenMP). Vamos a paralelizar con directivas OpenMP un bucle sencillo del que generaremos una versión escalar y otra vectorial (AVX). Compararemos las prestaciones de ambos códigos y analizaremos cómo escalan con el número de threads. También realizaremos distintos análisis del código para obtener métricas sencillas que nos permitan caracterizar su ejecución.

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. Inicializa la variable de entorno `CPU`. Se utiliza para organizar los experimentos realizados en distintas máquinas en distintos directorios.

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

---

1. Análisis estático del código

Vamos a realizar un análisis estático del código con la herramienta Intel Architecture Code Analyzer (IACA).

https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

IACA analiza de forma estática pequeños fragmentos de código (kernels) para varias microarquitecturas de Intel (Haswell, Broadwell, Skylake).

Para un código dado, IACA:

- Identifica la asociación entre las instrucciones y los puertos del procesador en condiciones ideales del *front-end*, núcleo de ejecución fuera de orden y jerarquía de memoria.
- Realiza un análisis estático del *throughput* del kernel y muestra su número de ciclos de ejecución.

Puedes encontrar más información acerca de IACA y su uso en el siguiente enlace:

http://stackoverflow.com/questions/26021337/what-is-iaca-and-how-do-i-use-it

1. Observa en el fichero `triad.c` los marcadores `IACA_START` e `IACA_END` que se han insertado para indicar a la herramienta el inicio y el final del código a analizar. El valor de dichos marcadores está en el fichero

   `/usr/local/include/iacaMarks.h`

2. Analiza y comprende el contenido del fichero `comp.iaca.sh`.

3. Compila con soporte IACA la versión escalar (noavx) del programa `triad.c`:

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

4. Observa los ficheros que contienen el ensamblador del binario generado y localizar los marcadores `IACA_START` e `IACA_END`. ¿Cuál es la secuencia de bytes asociada a cada marcador?

   Nota: estos marcadores impiden la vectorización del código. Ver por ejemplo:

   https://gcc.gnu.org/bugzilla/show_bug.cgi?id=63467

5. Ejecuta el binario generado:

   ```
   $ ./triad.noavx.single.gcc.iaca
   ```

   Parece que hay algún problema. Vamos a ver si identificamos la causa de dicho problema. ¿En qué registro se guarda el iterador del bucle externo (`nl`)? ¿Qué registro modifica el código de los marcadores `IACA_START` e `IACA_END`?

6. Analiza y comprende el contenido del fichero `run.iaca.sh`.

7. Ejecuta IACA para realizar el análisis estático del bucle:

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

8. Analiza los informes de *throughput* generados:

   - ¿Cuál es el *throughput* del bucle en ciclos?
   - ¿Cuál es el cuello de botella del bucle?
   - ¿Cuáles son los puertos más utilizados?

9. Analiza la traza generada:

   - Identifica las instrucciones que generan dos uops, y observa la dependencia entre ambas.
   - Identifica las dependencias entre instrucciones

10. **Optativo**. Repetir los apartados anteriores con la versión vectorial (avx) de `triad.c`. Utiliza para compilar con soporte IACA el script `comp.iaca.avx.sh`.

2. Paralelización manual mediante directivas OpenMP

1. Especificar de forma manual el paralelismo existente en el bucle principal del programa. Para ello, inserta las directivas OpenMP correspondientes en torno al bucle principal.

2. Compila el código con el script proporcionado que generará una versión escalar y vectorial del código paralelizado:

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

3. Ejecuta los programas utilizando 1, 2 y 4 threads:

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

4. A partir de los tiempos de ejecución, calcula para cada versión (escalar/vectorial) las aceleraciones (*speedups*) de las ejecuciones con varios threads respecto a la ejecución de este código con un thread. Calcula asimismo, la aceleración de la versión vectorial ejecutada con cuatro threads respecto la versión escalar ejecutada con un thread.

3. Análisis dinámico de la ejecución

3.1. Valgrind

Vamos a utilizar la herramienta `valgrind` para obtener una estimación de varias métricas de ejecución, como por ejemplo, el número de instrucciones ejecutadas, el número fallos de predicción de salto o la tasa de fallos de cache.

1. Reducir el número total de operaciones (FLOPs) que se van a ejecutar. Para ello hay que cambiar el valor de la constante `FLOP_COUNT` en el fichero `triad.c`.
2. Recompilar el código para actualizar los binarios escalar y vectorial:

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

3. Analizar y comprender el contenido del fichero `run.valgrind.sh`.
4. Ejecutar las herramientas `callgrind` y `cachegrind` con 1 thread:

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

5. A partir de los tiempos de ejecución, estima la sobrecarga que introducen ambas herramientas.
6. Observa los ficheros generados por `callgrind` y `callgrind_annotate`.
7. Cuál es la reducción en el número de instrucciones ejecutadas (*dynamic instruction count*) por la función `triad` en la versión `avx` respecto la versión `noavx`?
   Ayuda: ejecuta la siguiente orden en el directorio `valgrind`:

   ```
   $ grep refs callg.outfile.triad.*
   ```

8. **Optativo**. Observa los ficheros generados por `cachegrind` y `cachegrind_annotate`.
9. **Optativo**. Compara las tasas de fallos de la cache de datos de primer nivel (D1) de las versiones `avx` y `noavx`.
   Ayuda: ejecuta la siguiente orden en el directorio `valgrind`:

   ```
   $ grep "D1  miss rate" cacheg.outfile.triad.*
   ```

   ¿A qué se debe dicha diferencia?

3.2. Application Performance Snapshot

Vamos a realizar un análisis de la ejecución del código con la herramienta Intel Application Performance Snapshot (APS):

https://software.intel.com/sites/products/snapshots/application-snapshot/

Según el manual de APS:

> Application Performance Snapshot analyzes your application's CPU and FPU usage, I/O and memory footprint, memory access stalls, and MPI and OpenMP* utilization. After analysis, it displays basic performance enhancement opportunities for systems using Intel® platforms. Use this tool as a first step in application performance analysis to get a simple snapshot of key optimization areas.

1. Restaura el número total de operaciones (FLOPs) original. Para ello hay que cambiar el valor de la constante `FLOP_COUNT` en el fichero `triad.c`.
2. Recompilar el código para actualizar los binarios escalar y vectorial:

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

3. Ejecuta APS para realizar el análisis de la ejecución:

```
$ ./run.aps.omp.sh
```

Analiza los informes generados (ficheros de texto y html):

- ¿Qué ocurre con el uso de CPU al aumentar el número de threads?
- ¿Qué versión tiene un CPI mayor, escalar con un thread o vectorial con un thread?