Computación Paralela 2012 — Universidad Nacional de Córdoba
Práctico 1 - Arquitecturas de CPU Modernas — FaMAF

Carlos Bederián, Nicolás Wolovick

Objetivos:

- Refrescar conocimientos de C
- Obtener métricas diversas de un programa
- Poner en práctica lo aprendido en el teórico sobre ILP y memoria

Ejercicio 1. Examine la configuración del hardware, usando los comandos `cat /proc/cpuinfo` y `lstopo`. Investigue la organización de las unidades de ejecución de la arquitectura del procesador y estime el ancho de banda de memoria y los GFLOPS máximos teóricos del procesador.

Ejercicio 2. Implemente la función `float avg(const float * a, size_t n)` que calcula el promedio de un vector. Córrala repetidamente sobre un arreglo de valores aleatorios en un programa para distintos tamaños que sean potencia de 2 y estime GFLOPS y MBPS. Utilice `time(1)` y `gettimeofday(2)`[1] para calcular el tiempo transcurrido. Pruebe con otro compilador y opciones de compilación. Compare el assembly generado.

Ejercicio 3. Devise e implemente un programa que, realizando operaciones de manera repetitiva y evitando problemas de dependencias, corra a la mayor cantidad de GFLOPS posible[2]. Analice el assembly generado y compare con el resto del curso.

Ejercicio 4. Lea el programa `cache.c` que mide tiempos de acceso a memoria para distintos tamaños de paso. Ejecútelo y compare los resultados con la jerarquía de memoria reportada por `lstopo`. Ejecute varias instancias a la vez para utilizar todas las caches y canales del controlador de memoria del procesador.

Ejercicio 5. Desarrolle e implemente un dos programas que recorran un bloque de 256MB de la forma más lenta posible, primero forzando sólo cache misses y luego forzando tanto cache misses como TLB misses. Mida estos contadores usando `perf stat`[3].

Ejercicio 6. Implemente las funciones:

```
void init(size_t n, float * a, float * b, float * c)
void mmulf(const float * a, const float * b, size_t n, float * c)
```

Donde `a`, `b` y `c` son arreglos de $\texttt{n}^2$ elementos que representan matrices $\texttt{n} \times \texttt{n}$, de modo que:

- `init` inicializa los buffers de $\texttt{n}^2$ elementos `a` y `b` con números aleatorios entre `-1.0f` y `1.0f`, y `c` con `0.0f`[4].
- `mmulf` calcula el producto entre `a` y `b` y lo devuelve en `c`.

Luego cree un programa que pida memoria para 3 arreglos de `SIZE*SIZE` elementos[5], inicialícelos y calcule el producto. Una vez completo:

[1] `timersub` es muy útil para restar `struct timeval`

[2] Tenga en cuenta que si no se utiliza(n) el(los) resultado(s), un compilador agresivo puede eliminar el loop completo

[3] `perf list` muestra los contadores disponibles para medir

[4] La representación de `0.0f` es 4 bytes con ceros, o sea que puede usar `memset`.

[5] Donde `SIZE` es una definicíon provista a la hora de compilar con el parámetro `-DSIZE=`x.

1. Calcule la cantidad de operaciones de punto flotante realizadas por `mmulf` e infiera los GFLOPS obtenidos.
2. Calcule la cantidad de bytes transferidos por `mmulf` e infiera los GBps obtenidos.
3. Compare los valores anteriores contra la performance pico del hardware en el que corre. Está limitado por el ancho de banda o por el procesador?
4. Experimente con distintos compiladores y opciones de compilación[6]. Analice el assembly generado por cada uno y cómo afecta la performance.
5. Permute el orden de los loops de `mmulf`. ¿Qué pasa? ¿Por qué? Corra las distintas permutaciones usando `perf stat` y analice los contadores de caché y TLB misses.

Repita lo anterior para distintos valores de `SIZE`. Grafique los tiempos de ejecución y analice los resultados comparando el tamaño ocupado por las matrices con la caché disponible para el proceso.

[6]Por ejemplo: `gcc -O0`, `gcc -O3 -march=native -funroll-loops`, `icc -fast`.