Trabajo Práctico 1: Optimización secuencial

Computación paralela

Marzorati Denise
M-6219/7

11 de abril de 2018

Docentes de la materia

Nicolás Wolovick
Carlos Bederián

Características del hardware y del software

- CPU: Intel Core i7-3632QM @ 2.20GHz.
- Memoria: 8GB, DDR3, 2 canales.
- Compilador: GCC 7.1.0.
- Sistema operativo: Ubuntu 16.04.4, x86_64.

Metodología de trabajo

Para comenzar a atacar cada problema lo primero que se hará es realizar algo de profiling utilizando la herramienta perf, para poder identificar a dónde debe dirigirse el mayor esfuerzo en optimizar. Se hará un breve resumen con las optimizaciones realizadas que dieron resultados significativos tras repetir varias veces la ejecución del programa; aquellas que no aparezcan mencionadas es porque no realizan ningún aporte que valga la pena mencionar.

Heat

Para obtener la versión más rápida se ha compilado el código con:
gcc heat.c -o heat -O3 -ffast-math -lm -lgomp.
El código de step pudo optimizarse reescribiendo el flujo del loop, con respecto a una condición dependiente del índice.

Para N = 500 los resultados han sido los siguientes:

- Avg GFLOPS/s: 1.720405
- Avg IPC: 2.33
- Avg cache misses: 0.12%
- Avg time: 2.112038s

Para N = 1000 los resultados han sido los siguientes:

- Avg GFLOPS/s: 0.253729
- Avg IPC: 1.64
- Avg cache misses: 16.29%
- Avg time: 11.610365s

Para N = 1500 los resultados han sido los siguientes:

- Avg GFLOPS/s: 0.023805
- Avg IPC: 1.35
- Avg cache misses: 25.64%
- Avg time: 31.615983s

Tiny_MC

Para obtener la versión más rápida posible se ha compilado el código con:
gcc -Wall -Wextra -std=gnu99 -lm -fopemp -O2
En este caso las optimizaciones manuales no han aportado ninguna mejora significativa.

Para PHOTONS = 32768 los resultados han sido los siguientes:

- Avg IPC: 1.46
- Avg percentage cache misses: 29.23%
- Avg time: 0.233097s
- Avg percentage stalled cycles, frontend: 49.59%

Para PHOTONS = 327680 los resultados han sido los siguientes:

- Avg IPC: 1.44
- Avg percentage cache misses: 42.52%
- Avg time: 1.998486s
- Avg percentage stalled cycles, frontend: 50.05%

Para PHOTONS = 3276800 los resultados han sido los siguientes:

- Avg IPC: 1.47
- Avg percentage cache misses: 16.99%
- Avg time: 18.5439116s
- Avg percentage stalled cycles, frontend: 49.24%

IntegralImage

Para obtener la versión más rápida posible se ha compilado el código con:
gcc integralimage.c -o integral -O2 -Wall -Wextra -std=c99 -lm -lgomp.
Al hacer profiling se pudo observar que la función rand() consumía gran parte del tiempo de computación, por lo que se buscó versiones más rápidas de la función, encontrando así la presentada en el código como fastrand().

Para FRAMES = 300 los resultados han sido los siguientes:

- Avg IPC: 2.55
- Avg percentage cache misses: 18.01%
- Avg time: 1.3503946s
- Avg percentage stalled cycles, frontend: 32.38%

Para FRAMES = 3000 los resultados han sido los siguientes:

- Avg IPC: 2.57
- Avg percentage cache misses: 15.09%
- Avg time: 13.387516s
- Avg percentage stalled cycles, frontend: 32.38%

Para FRAMES = 30000 los resultados han sido los siguientes:

- Avg IPC: 2.58
- Avg percentage cache misses: 14.18%
- Avg time: 132.091406s
- Avg percentage stalled cycles, frontend: 35.32%

Tiny_ising

Para obtener la versión más rápida posible se ha compilado el código con:
gcc -std=gnu99 -Wall -Wextra -O2 -funroll-loops

Al hacer profiling se observó que nuevamente la función de generación de números aleatorios estaba realentizando la computación, por lo que se volvió a usar fastrand(). Otra optimización importante fue la de reescritura de las operaciones de la función update, puesto que esta consumía gran parte de los recursos. Una optimización menor también fue realizada en el código de cycle para no realizar tantas iteraciones con una comparación en cada una de ellas.

Para L = 128 los resultados han sido los siguientes:

- Avg IPC: 2.21
- Avg percentage cache misses: 24.063%
- Avg time: 0.423495s
- Avg percentage stalled cycles, frontend: 25.91%

Para L = 512 los resultados han sido los siguientes:

- Avg IPC: 2.27
- Avg percentage cache misses: 7.53%
- Avg time: 6.174049s
- Avg percentage stalled cycles, frontend: 24.23%

Para L = 1024 los resultados han sido los siguientes:

- Avg IPC: 2.28
- Avg percentage cache misses: 52.105%
- Avg time: 24.324567s
- Avg percentage stalled cycles, frontend: 23.95%

Hornschunk

Para obtener la versión más rápida posible se ha compilado el código con:
gcc -Wall -Wextra -O3 -lpng -ljpeg -ltiff -lm -lgomp
Al hacer profiling se observó que la mayor parte del tiempo de computación se utilizaba en la función compute_bar, por lo que los esfuerzos se dirigieron a hacer un poco de unrolling en loop de dicha función.
Para niters = 1000 los resultados han sido los siguientes:

- Avg IPC: 1.84
- Avg percentage cache misses: 48.66%
- Avg time: 3.822993s
- Avg percentage stalled cycles, frontend: 39.34%

Para niters = 10000 los resultados han sido los siguientes:

- Avg IPC: 1.89
- Avg percentage cache misses: 48.58%
- Avg time: 38.928659s
- Avg percentage stalled cycles, frontend: 38.28%

Para niters = 30000 los resultados han sido los siguientes:

- Avg IPC: 1.89
- Avg percentage cache misses: 48.58%
- Avg time: 38.928659s
- Avg percentage stalled cycles, frontend: 38.28%

Navierstrokes

Para obtener la versión más rápida posible se ha compilado el código con:
gcc -Wall -Wextra -Wno-unused-parameter -lm
En este caso ni siquiera se utilizaron optimizaciones del compilador porque las mismas causaron un aumento en el porcentaje de cache misses y stalled cycles. Tampoco se pudo idear ninguna mejora manual para optimizar el programa.

Para este problema se usó otra metodología de medición al ser un programa que puede ejecutarse indefinidamente: se hicieron varias repeticiones de t segundos cada una.

Para t = 10 los resultados han sido los siguientes:

- Avg IPC: 2.24
- Avg percentage cache misses: 13.96%
- Avg percentage stalled cycles, frontend: 42.01%

Para t = 60 los resultados han sido los siguientes:

- Avg IPC: 2.28
- Avg percentage cache misses: 11.89%
- Avg percentage stalled cycles, frontend: 41.49%

Para t = 180 los resultados han sido los siguientes:

- Avg IPC: 2.20
- Avg percentage cache misses: 20.39%
- Avg percentage stalled cycles, frontend: 43.26%

Tinny_manna

Para obtener la versión más rápida posible se ha compilado el código con:
gcc -Wall -Wextra -std=c++0x -O3 -funroll-loops
Compilando de este modo se logra hacer que la pila evolucione de tal manera que no haya más actividad.

Al hacer profiling se observó que nuevamente la función de generación de números aleatorios estaba realentizando la computación, por lo que se volvió a usar fastrand(). Si bien no marcó una diferencia contundente, la opción -funroll-loops ayudó a disminuir un par de segundos de ejecución.

Para N = 3276 los resultados han sido los siguientes:

- Avg IPC: 2.20
- Avg percentage cache misses: 33.85%
- Avg time: 1.534134s
- Avg percentage stalled cycles, frontend: 24.89%

Para N = 15000 los resultados han sido los siguientes:

- Avg IPC: 2.51
- Avg percentage cache misses: 6.61%
- Avg time: 28.087699s
- Avg percentage stalled cycles, frontend: 19.98%

Para N = 32768 los resultados han sido los siguientes:

- Avg IPC: 2.35
- Avg percentage cache misses: 0.07%
- Avg time: 30.209312s
- Avg percentage stalled cycles, frontend: 24.61%