Laboratorio 1: Tiny MC

Álvaro Frías Garay - Ary Lautaro Di Bartolo

Resumen
En el Laboratorio 1 de la cátedra **Computación Paralela** de la Universidad Nacional de Córdoba se busca lograr la optimización secuencial para un problema particular. En este caso en la utilización del método Monte Carlo para la simulación del transporte de fotones. Para conocer más visite: **Monte Carlo Method for Photon Transport**.

En este informe se presentan las características del hardware y software utilizados, el método de medición, las optimizaciones (aplicadas y fallidas) y finalmente la gráfica de scaling para la versión más veloz.

Se concluye que hay muchas optimizaciones disponibles a tener en cuenta y que es muy importante el testeo de cada cambio realizado, ya que muchas veces mejoras que parecen ser evidentes tienen resultados inesperados.

Keywords
Monte Carlo — Optimización Secuencial — C

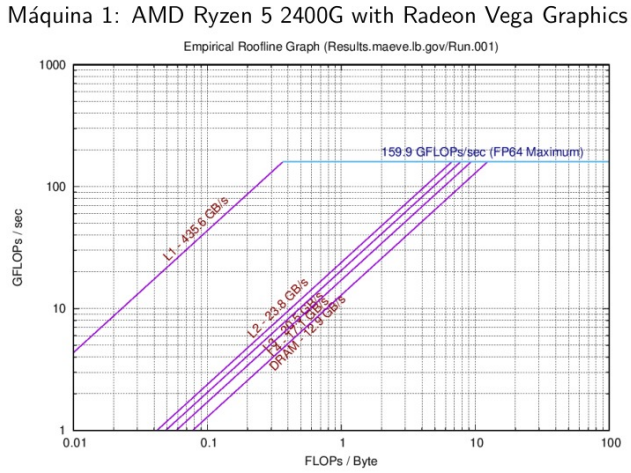

Figura 1

Características del Hardware y Software

Hardware

El procesador de la computadora utilizada para correr este laboratorio es un AMD Ryzen 5 2400G con Radeon Vega Graphics.

Como se puede apreciar en la **Figura 1** la caché L1 tiene un ancho de banda sensiblemente superior al resto de las cachés y tiene un máximo de 159.9 GFLOPs/sec.

Software

El sistema operativo utilizado es Ubuntu 20.04.2 LTS (Focal Fossa). La arquitectura es x86 64. El compilador es gcc (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0.

Mediciones

Cada simulación consistió de una generación de 60 números aleatorios (entre 1 y 32k) los cuales representan la cantidad de fotones a generar para cada corrida. Cada conjunto de flags utilizó los mismos 60 números aleatorios por simulación.

Para cada flag simulada se obtiene la media de fotones por segundo que procesó.

Optimizaciones

Aplicadas

Primero que nada, y sin modificar ningún aspecto del programa, se eligieron un conjunto de flags óptimas y se testearon. Como se puede observar en la **Figura 2**, el conjunto de flags que mejores resultados arroja es: **-o3 -ffast-math -march=native**.

Luego se reemplazó el generador de números aleatorios por una implementación de Mersenne Twister en puro C: **Mersenne Twister**. El resultado se puede observar en la **Figura 3**.

Además se reemplazó la división por la multiplicación del inverso (aunque probablemente ya lo hace **-ffast-math**). Se añadió la flag **-flto** para lograr inlining de MT. La mejora es sustancial, tal y como se ve en la **Figura 4** (columna 3).

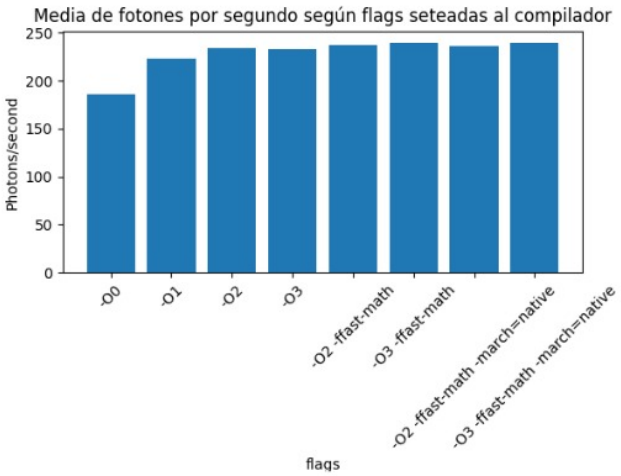

Figura 2

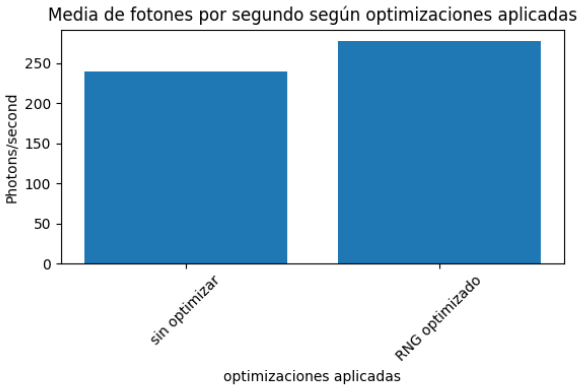

Figura 3

Fallidas

La flag **-funroll-loops** capaz ayudaba al ciclo do-while. Es cuestión de probarlo y es lo que se hizo. De la **Figura 4** (última columna) se desprende que era contraproducente.

Se intentó optimizar el siguiente ciclo do-while:

```
do {
        xi1 = 2.0f * genRand(&r) - 1.0f;
        xi2 = 2.0f * genRand(&r) - 1.0f;
        t = xi1 * xi1 + xi2 * xi2;

} while (1.0f < t);
```

Para hacerlo, primero se debió comprender lo que hace. Este fragmento de código busca dos números aleatorios xi1 y xi2 que estén comprendidos en el intervalo [-1,1] y cuya suma de cuadrados sea menor a 1, si esto no ocurre vuelve a intentar, generando nuevos números aleatorios y repitiendo los cálculos. Para evitar este retrabajo se propuso la siguiente optimización:

```
if ((float)genRand(&r) > 0.5) {
        xi1 = 2.0f * (float)genRand(&r)
        - 1.0f;
        xi2 = ((float)genRand(&r)) * 2
        * sqrtf(1 - xi1 * xi1) - sqrtf
        (1 - xi1 * xi1);
} else {
        xi2 = 2.0f * (float)genRand(&r)
         - 1.0f;
        xi1 = ((float)genRand(&r)) * 2
        * sqrtf(1 - xi2 * xi2) - sqrtf
        (1 - xi2 * xi2);
}
t = xi1 * xi1 + xi2 * xi2;
```

Básicamente es un ciclo que tira una moneda (número aleatorio entre 0 y 1). Si la moneda sale de cierta manera, menor a 0.5 por ejemplo, el resultado del número aleatorio de xi1 acortará el intervalo del número aleatorio xi2. Si sale de otra manera, mayor a 0.5, el resultado del número aleatorio de xi2 será quien defina la amplitud del intervalo del número aleatorio xi1. Así se asegura que todas las veces se generarán números aleatorios xi1 y xi2 que cumplan con la condición de que la suma de sus cuadrados sea menor a 1 sin necesidad de generar retrabajos.

Sin embargo, cuando se procede a testear esta mejora se obtienen resultados peores, como se puede observar en la **Figura 4** (quinta columna).

Scaling

La gráfica de scaling para la versión más optimizada se puede ver en la **Figura 5**.

Se desprende claramente que el aumento en la cantidad de fotones inicialmente tiene una incidencia notoria en el aumento de la performance del problema. Sin embargo, al sobrepasar aproximadamente los 20 mil elementos del problema la performance comienza a incrementarse cada vez más lento con un comportamiento asintótico.

Conclusiones

A la hora de realizar optimizaciones es muy importante realizar testeos constantes. Muchas veces mejoras aparentemente evidentes dan resultados contraproducentes que empeoran la performance de un programa. Con la misma lógica se puede pensar que no hay que dejar ideas sin probar, ya que quizás algo que parece ser peor da mejores resultados. Esto sería en principio correcto, sin embargo, si no se quiere perder el tiempo intentando infinidad de alternativas es importante tener un conocimiento profundo sobre los aspectos que afectan al rendimiento.

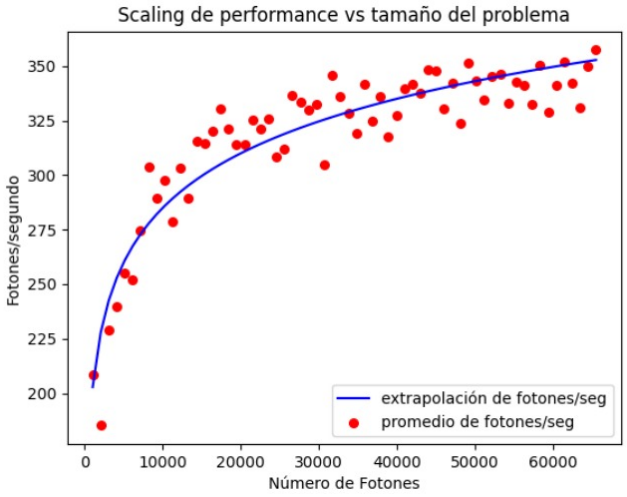

Figura 5

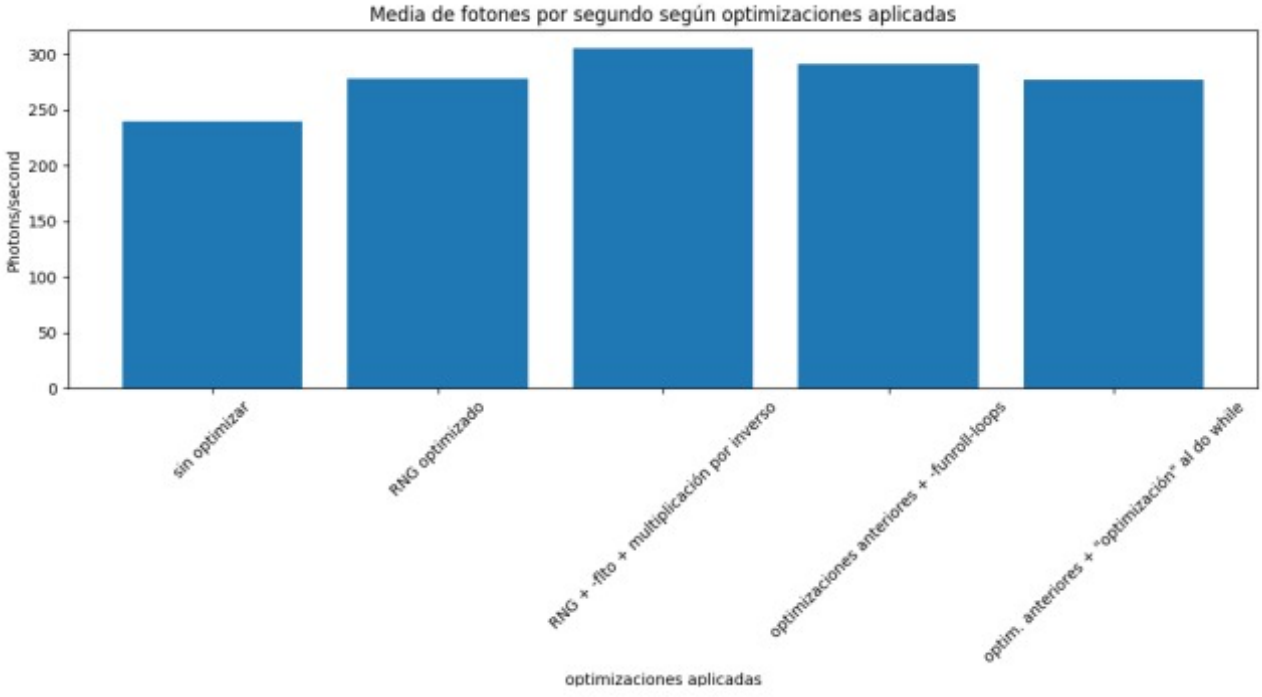

Figura 4