Hands On Lab: Memoria

Juanjo Costa <jcosta@ac.upc.edu>

Mon, 16 Apr 2023

Contents

Objectius

- Avaluar el rendiment d'un codi (temps d'execució, cicles)
- Detectar el patró d'accés a memoria d'un codi
- Entendre l'impacte de la cache en un codi
- Utilització bàsica de funcions per gestionar memòria dinàmica
- Ús de GPU

Estructura de directoris

En el codi suministrat es troben els següents directoris cadascun amb el codi font necessari per les següents seccions:

- 1_matrix (Avaluació de rendiment)
- 2_access_pattern (Patró d'accés)
- 3_dynamic_memory (Memòria dinàmica)
- tools (Utilitats per mesurar temps i anàlisi)

Avaluació de Rendiment

És important saber avaluar el rendiment dels nostres codis. Saber el que triguem a fer les coses. En el nostre context (videojocs) és imprescindible ja que això defineix el rendiment dels nostres jocs, el nombre de frames que puc mostrar per segon. Quant menys trigui a "calcular" el frame, més frames puc generar.

Mesurar el que triga un tros de codi és relativament simple si tinc una funció *capturarTemps* que funcioni com un cronómetre:

```
...
t1 = capturarTemps();

// Tros de codi a mesurar

t2 = capturarTemps();
temps = t2 - t1; //Temps que trigo a executar el codi
...
```

L'únic problema és que la resolució i el cost d'aquest *capturarTemps* depén del hardware i del sistema operatiu usat. En el cas de Windows(TM), la seva documentació ens en dona una explicació força detallada [TIME].

Per que et donis una idea, al fitxer *measure.c*, trobaràs les rutines *start_counter* i *get_counter* que bàsicament serveixen com a cronòmetre, retornant el nombre de microsegons que han passat entre les dues crides. Aquestes rutines funcionen tant a Windows com a Linux.

Un problema addicional és que el temps que triga un determinat codi no és sempre exactament el mateix degut a la complexitat dels factors que hi intervenen avui dia. El codi a measurar no s'executa directament sobre l'arquitectura, sino que hi ha un sistema operatiu, dispositius que interrompen el flux habitual d'execució, ... per tant, el que fem habitualment és executar el mateix codi diversos cop i fer un anàlisi estadístic sobre els temps resultants per quedar-nos amb el valor més estable.

La rutina *measure_full*, que pots trobar al fitxer *measure.c*, automatitza aquest procés i retorna el mínim temps d'execució d'un codi. En concret, executa el codi varios cops, mesura el seu temps d'execució a cada iteració i guarda aquests valors en una taula fins que la diferencia de temps entre els valors més petits sigui menor d'un 1 per cent (1% o 0.01), que és el factor que hem decidit per considerar una mesura de temps suficientment estable. Aquesta rutina és la que usem per calcular els temps d'execució en els següents exercicis.

Avaluant matrius...

Per començar anem a avaluar un codi molt simple de recorregut de matrius, per veure si triguem el mateix fent un recorregut per files o per columnes. Veurem

que per matrius petites, els dos recorreguts són semblant, però per matrius grans... això ja no és cert.

Al fitxer *matriu.c* tens un codi que recorre una matriu molt petita (8x8). El codi mostra el temps en microsegons que triga fent el recorregut per files i pel recorregut per columnes. Aquest temps és el menor d'executar fins a 10 cops el mateix codi, avaluant que l'error entre els 3 valors més baixos sigui inferior a un 1% (0.01).

Per fer l'avaluació compilarem i executarem l'aplicació diverses vegades, anotant el temps d'execució per cadascun dels recorreguts i anirem incrementant les mides de les files i les columnes de la matriu (sempre amb el mateix valor a totes dues) en potencies de 2 (8, 16, 32, ...).

- Fes una taula amb la mida de la matriu i els temps d'execució pels dos recorreguts.
- Es similar el temps d'execució quan es recorre per files de quan es recorre per columnes? (Assegurat de fer un parell d'execucions per estar-ne segurs)
- A la vista de la taula, què en pots extreure? Hi ha algun canvi en els temps d'execucció entre els 2 recorreguts? Quant ocupa 1 fila? Per què creus que passa?
- Comprova els nivells de cache del teu processador [CPUDB] amb els resultats que t'han sortit a la taula.

Patrons d'accés i impacte de la cache

Les estructures que usem per guardar informació (com les matrius) s'han de linearitzar per guardar-se a memoria (com varem veure a teoria). I per tant el patró d'accés a aquestes estructures afecta al rendiment (tal i com hem vist a l'apartat anterior). Anem a veure de quina manera exactament.

En el següent exercici presentem 3 programes: *mm-ijk.c, mm-jki.c i mm-kij.c.* Corresponen a 3 de les 6 formes ijk del producte de matrius. Per exemple, en el programa *mm-ijk.c* es mesura la següent porció de codi (ijk vol dir que el bucle més extern és el bucle controlat per la variable 'i' i el més intern és el controlat per 'k'):

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Els 3 programes realitzen, de forma diferent, la mateixa operació: el producte de matrius: C = A * B.

Hi ha una macro que defineix la mida de les matrius, per defecte de 256x256 floats:

```
#ifndef N
#define N 256  /* Dimensión por defecto */
#endif
```

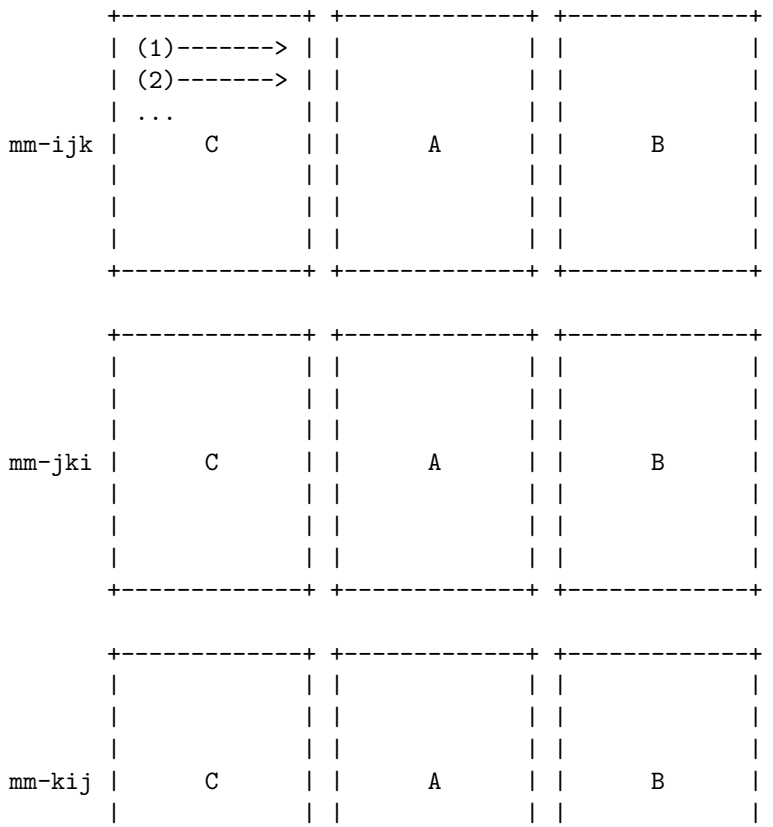
Si aquesta mida es N=6 (o més petit), mostra el contingut de la matriu resultat C, de forma que puguis comprovar que els 3 programes fan el mateix. Si és més gran, només fa el producte i mostra el temps d'execució en milisegons.

Pel següents exercicis, tria 1 dels casos.

- Es important saber l'ordre d'accés als elements de memòria d'un codi qualsevol. Per exemple, en el codi anterior la matriu C s'accedeix mitjançant C[i][j] i per tant segons l'ordre dels bucles 'i' i 'j', s'estan accedint als elements :

  ```
  C[0][0], C[0][1], C[0][2], ... C[0][N],
  C[1][0], C[1][1], C[1][2], ... C[1][N],
  ...
  ```

Es a dir, s'accedeix a tots els elements de la 1a fila, després a tots els de la 2a, i així per totes les files. Dibuixa, pel cas triat, l'ordre en el que es recorren les matrius A, B i C (com a exemple, ja es mostra l'accés de la matriu C):



```
        |             | |             | |             |
        |             | |             | |             |
        +-------------+ +-------------+ +-------------+
```

- Apunta el temps d'execució per la versió triada:

```
       | Temps execució (en ms)
     N | mm-ijk | mm-jki | mm-kij
-------+--------+--------+--------
   256 |        |        |
   512 |        |        |
  1024 |        |        |
```

- Supossant que cada element de la matriu ocupa 4 bytes, i que la mida de pàgina del nostre sistema és de 4Kbytes calcula pel cas triat el nombre de pàgines de memòria virtual que s'usen a l'executar completament el bucle més intern 1 cop. Per exemple, en el cas mm-ijk, per cada execució de tot el bucle 'k' la matriu C només accedeix a 1 element (4bytes), mentres que la matriu A accedeix a tota 1 fila (elements consecutius) i la matriu B a tota 1 columna (elements no consecutius).

```
       |             mm-ijk             |
     N | matriu A | matriu B | matriu C |
-------+----------+----------+----------+
   256 |          |          |    1     |
   512 |          |          |          |
  1024 |          |          |          |
-------+----------+----------+----------+

       |             mm-jki             |
     N | matriu A | matriu B | matriu C |
-------+----------+----------+----------+
   256 |          |          |          |
   512 |          |          |          |
  1024 |          |          |          |
-------+----------+----------+----------+

       |             mm-kij             |
     N | matriu A | matriu B | matriu C |
-------+----------+----------+----------+
   256 |          |          |          |
   512 |          |          |          |
  1024 |          |          |          |
-------+----------+----------+----------+
```

Memòria dinàmica

Les rutines de *malloc* i *free* de la llibreria de C permeten a un programa d'usuari demanar memoria dinàmica al sistema operatiu. El sistema operatiu assigna nou espai al procés, però degut a que la granularitat de la crida (demana bytes) i la granularitat del SO (reserva i mapeja pàgines de 4096 bytes) no coincideixen, pot passar que l'usuari pugui executar codi no legal.

El fitxer *mem.c* conté un codi que reserva un vector de 10 caracters i inicialitza les seves posicions. A continuació intenta accedir a posicions fora d'aquest vector.

- Executa el codi, observa el resultat i intenta deduir què està pasant.

El fitxer *mem2.c* mostra els efectes perniciosos d'usar memòria que no hem reservat explícitament.

- Executa el codi, observa el resultat i intenta deduir què està pasant.

GPU fun

Avui dia, tot i que la cache és important, en el context dels videojocs, la càrrega de feina se l'emporta la GPU i per tant cal tenir-ho present. A [GPU] el Keith O'Conor (programador i CTO de [Romero Games]) dona una molt bona explicació de l'importància de les GPUs (i prou referencies per estar entretinguts tot el temps que volgueu)

References

..[TIME]: https://docs.microsoft.com/es-es/windows/desktop/SysInfo/acquiring-high-resolution-time-stamps "Acquiring high-resolution time stamps"

..[CSAPP]: http://csapp.cs.cmu.edu/2e/home.html "Computer Systems: A Programmer's Perspective. Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University, 3rEdition"

..[MOUNT]: http://csapp.cs.cmu.edu/3e/mountain.tar "Memory Mountain source code"

..[CPUDB]: https://www.techpowerup.com/cpudb/ "CPU Database"

..[GPU]: http://www.fragmentbuffer.com/gpu-performance-for-game-artists/ "GPU performance for game artists. Keith O'Conor"

..[Romero Games]: https://www.romerogames.ie/