Informe N1 Laboratorio - Algritmos paralelos

Luigy Machaca

4 de abril de 2017

Los ejercicios corresponden al capitulo 2 [1]:

0.1. Implement in C the simple three-nested-loop version of the matrix product and try to evaluate its performance for a relatively large matrix size.

Realizaremos una multiplicacion de dos matrices; empezando la multiplicación con la filas de la primera matriz y las columnas de la segunda matriz, usaremos 3 buclues, teniendo un tiempo total de n cubo

```
void prodoct3loop(int **a,int **b ,int **r, int tam){
  for(int i=0;i<tam;i++){
    for(int j=0;j<tam;j++){
      for(int k=0;k<tam;k++){
        r[i][j]+=(a[i][k]*b[k][j]);
      }
    }
  }
```

```
};
```

0.2. Implement the blocked version with six nested loops to check whether you can observe a significant gain.

También realizaremos una multiplicación de matrices. Usando la variante de multiplicación por bloques; usando 6 bucles.

```
void product6loop(int **a,int **b ,int **r, int tam,int sizeofblock){
  if(sizeofblock>=tam){
    return;
  }

  for(int i=0 ; i<tam ; i+=sizeofblock ){
    for(int j=0 ; j<tam ; j+=sizeofblock ){
      for(int k=0 ; k<tam ; k+=sizeofblock){
        for(int ii=i ; ii<((i+sizeofblock)>tam?tam:(i+sizeofblock)) ;ii
          for(int jj=j ; jj<((j+sizeofblock)>tam?tam:(j+sizeofblock)) ;
            for(int kk=k ; kk<((k+sizeofblock)>tam?tam:(k+sizeofblock))
;kk++){
              r[ii][kk]+=a[ii][jj]*b[jj][kk];
            }
          }
        }
      }
    }
  }
};
```

0.3. Execute these algorithms step by step to get a good understanding of data movements between the cache and the memory and try to evaluate their respective complexity in term of distant memory access.

```
#include <stdlib.h>
#include <iomanip>
#include <iostream>
#include <ctime>

using namespace std;
#define COTA 5


void start(int **&a,int tam){
       a=new int*[tam];
       for(int i=0;i<tam;i++){
         a[i]=new int[tam];
       }
};

void fill(int **a,int tam,int value=1){
  //srand (time(NULL));
     for(int i=0;i<tam;i++){
       for(int j=0;j<tam;j++){
         a[i][j]=rand() %COTA *value;
       }
     }
};

void show(int **a,int tam){
  cout<<"----------------"<<endl;
```

```
  for(int i=0;i<tam;i++){
    for(int j=0;j<tam;j++){
      cout<<a[i][j]<<"  ";
    }
    cout<<endl;
  }
  cout<<"---------------"<<endl;
};


//matrices cuadradas del mismo tama o
void prodoct3loop(int **a,int **b ,int **r, int tam){
  for(int i=0;i<tam;i++){
    for(int j=0;j<tam;j++){
      for(int k=0;k<tam;k++){
        r[i][j]+=(a[i][k]*b[k][j]);
      }
    }
  }
};



//matrices cuadradas del mismo tama o
void product6loop(int **a,int **b ,int **r, int tam,int sizeofblock){
  if(sizeofblock>=tam){
    return;
  }

  for(int i=0 ; i<tam ; i+=sizeofblock ){
    for(int j=0 ; j<tam ; j+=sizeofblock ){
      for(int k=0 ; k<tam ; k+=sizeofblock){
        for(int ii=i ; ii <((i+sizeofblock)>tam?tam:(i+sizeofblock)) ;ii
          for(int jj=j ; jj <((j+sizeofblock)>tam?tam:(j+sizeofblock)) ;
            for(int kk=k ; kk<((k+sizeofblock)>tam?tam:(k+sizeofblock))
```

```
;kk++){
                    r[ii][kk]+=a[ii][jj]*b[jj][kk];
                }
            }
        }
    }
  }
};


int main(){

  int tamx=1200;

 //---------------prodoct3loop---------------------
  int tam1=tamx;
  int **m1,**m2,**r1;
  start(m1,tam1);
    fill(m1,tam1);
  start(m2,tam1);
    fill(m2,tam1);
  start(r1,tam1);
    fill(r1,tam1,0);

  //show(m1,tam1);
  //show(m2,tam1);
  //show(r1,tam1);

  float t1=clock();
  prodoct3loop(m1,m2,r1,tam1);
  t1=clock()-t1;
  //cout<<setprecision(0)<<fixed;
  cout<<"time 3loops: "<< ((float)t1)/CLOCKS_PER_SEC <<" s"<<endl;
```

```
  //show(r1,tam1);

 //----------------product6loop-----------------------
  int tam2=tamx;
  int sizeofblock=2;
  int **m3,**m4,**r2;
  start(m3,tam2);
    fill(m3,tam2);
  start(m4,tam2);
    fill(m4,tam2);
  start(r2,tam2);
    fill(r2,tam2,0);

  //show(m3,tam2);
  //show(m4,tam2);
  //show(r2,tam2);

  float t2=clock();
  product6loop(m3,m4,r2,tam2,sizeofblock);
  t2=clock()-t2;
  //cout<<setprecision(0)<<fixed;
  cout<<"time 6loops: "<< ((float)t2)/CLOCKS_PER_SEC <<" s"<<endl;

  //show(r2,tam2);

  return 0;
};
```

La verificacion de ambos algoritmos para tener una referencia en la diferencia de tiempo, se utilizó la libreria la función Clock() de la libreria *Time*.

Cuadro 1: .

n/seconds	3loops	6loops
300	*0.197584*	0.16815
600	*1.31013*	1.31083
900	*4.94984*	4.42283
1200	13.783	12.5805

Dando los siguientes resultados. TABLA 1

0.4. Execute these two versions of the code with valgring and kcachegrind to get a precise evaluation of their performance in term of cache misses.

VALGRIND es un conjunto de herramientas utilizadas para anlisis dinmico de programas escritos en C/C++.
Haciendo lo siguiente; por cada instrucción que ejecuta el programa, Valgrind aade una serie de instrucciones adicionales para analizar el comportamiento del programa.
La tcnica principal que utiliza es conocida como shadow memory (se abre en nueva ventana). Dando como resultado un conjunto de bits a cada porcin de memoria, que refleja si los datos correspondientes son accesibles, si han sido correctamente inicializados, etc.

Para lo cual utilizaremos el siguiente comando

```
valgrind --tool=cachegrind ./ejecutable
```

Obteniendo los siguientes resultados. Figura [1,2]

Valgrind posee varias herramientas una de ellas, *cachegrind* para verificar si hay presencia de *cache miss*. Si vemos en la figura[1,2], se divide en tres partes,

- Primera parte: muestra información acerca del cache en el primer nivel del cache.

Figura 1: *3matrix-800*

```
luigy@luigy-pc:~/Desktop/paralelos2017/algoritmo-paralelos-csunsa/lab1$ valgrind --tool=cachegrind ./3matrix_800
==19776== Cachegrind, a cache and branch-prediction profiler
==19776== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==19776== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==19776== Command: ./3matrix_800
==19776==
--19776-- warning: L3 cache found, using its data for the LL simulation.
time: 121.754 ms
==19776==
==19776== I   refs:      26,274,043,115
==19776== I1  misses:             1,905
==19776== LLi misses:             1,788
==19776== I1  miss rate:           0.00%
==19776== LLi miss rate:           0.00%
==19776==
==19776== D   refs:      11,842,216,691  (11,313,938,553 rd   + 528,278,138 wr)
==19776== D1  misses:       609,299,165  (   609,173,592 rd   +     125,573 wr)
==19776== LLd misses:           130,172  (         7,869 rd   +     122,303 wr)
==19776== D1  miss rate:            5.1% (           5.4%     +         0.0%  )
==19776== LLd miss rate:            0.0% (           0.0%     +         0.0%  )
==19776==
==19776== LL refs:          609,301,070  (   609,175,497 rd   +     125,573 wr)
==19776== LL misses:            131,960  (         9,657 rd   +     122,303 wr)
==19776== LL miss rate:             0.0% (           0.0%     +         0.0%  )
luigy@luigy-pc:~/Desktop/paralelos2017/algoritmo-paralelos-csunsa/lab1$
```

Figura 2: *6matrix-800*

```
luigy@luigy-pc:~/Desktop/paralelos2017/algoritmo-paralelos-csunsa/lab1$ valgrind --tool=cachegrind ./6matrix_800
==19653== Cachegrind, a cache and branch-prediction profiler
==19653== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==19653== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==19653== Command: ./6matrix_800
==19653==
--19653-- warning: L3 cache found, using its data for the LL simulation.
timess: 183.364 s
==19653==
==19653== I   refs:      37,533,880,047
==19653== I1  misses:             1,907
==19653== LLi misses:             1,793
==19653== I1  miss rate:           0.00%
==19653== LLi miss rate:           0.00%
==19653==
==19653== D   refs:      17,727,494,739  (16,751,696,990 rd   + 975,797,749 wr)
==19653== D1  misses:        16,341,165  (    16,215,592 rd   +     125,573 wr)
==19653== LLd misses:           130,172  (         7,869 rd   +     122,303 wr)
==19653== D1  miss rate:            0.1% (           0.1%     +         0.0%  )
==19653== LLd miss rate:            0.0% (           0.0%     +         0.0%  )
==19653==
==19653== LL refs:           16,343,072  (    16,217,499 rd   +     125,573 wr)
==19653== LL misses:            131,965  (         9,662 rd   +     122,303 wr)
==19653== LL miss rate:             0.0% (           0.0%     +         0.0%  )
luigy@luigy-pc:~/Desktop/paralelos2017/algoritmo-paralelos-csunsa/lab1$
```

- Segunda parte: muestra los cache misses del ultimo nivel(L3 o L4 , según el ordenador)
- La tercera parte muestra el numero de acceso a memoria y el porcentaje de las solicitudes *(read and write)* **atendidas por la cache**.

El rendiminento de cada algoritmo, tiene una diferencia de **121.754** y **183.364** s es muy importante si tenemos en cuenta que la cach L2 suele ser 10 veces ms rpida que la memoria RAM.

En conclusión, dado el primer algoritmo que consta de 3 bucles, que a primera vista tendria que ser más rapido que el 6 buclues. Pero siendo ejecutado dichos bucles del segundo algoritmo en cache L1 y L2, claramente son mucho mas rapidos

Referencias

[1] Peter Pacheco , *An Introduction to Parallel Programming*, 1st Edition, 2011.