Universidad Nacional de San Agustin

Algoritmos paralelos

Práctica I : Impacto de caché

Escalante Calcina, Judith

3 de abril de 2017

Índice

1. Multiplicación de matrices con 3 bucles anidados

En el siguiente código podemos ver la multiplicación con tres bucles anidados, los cuales son representados por los tres últimos for , estos realizan la multiplicación de dos matrices en este caso la matriz a y la matriz b y almacenan en resultado en la matriz c.

Este algoritmo es muy lento ya que se utilizan muchos caché misses, esto sucede porque la matriz es demasiado grande y por ende requiere demasiada memoria , por lo que sólo una parte se queda en la memoria caché. Si se esta usando LRU (least recently used), al acceder a las columnas de la matriz b en el bucle má interno, se producirán cache misses porque deberá buscar las columnas que no pudieron entrar en la misma línea de caché y porque al llegar al final , las columnas iniciales ya no estarán en la caché debido al LRU.

Esto sucede porque la matriz no cabe totalmente en la memoria caché para ejecutar el algoritmo eficientemente , por lo tanto las últimas columnas , quita las menos usadas recientemente , es decir , las primeras.

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

void mult_3(int n, int m, int p)
{       srand(time(NULL));

        int **c,**b,**a;
        int i1, i2,i,j1,j2,j,k,l;
        c = (int **)malloc(n*sizeof(int*));

        a = (int **)malloc(n*sizeof(int*));
        b = (int **)malloc(m*sizeof(int*));

        for (l=0;l<n;l++)
                a[l] = (int*)malloc(m*sizeof(int));

        for (l=0;l<m;l++)
                b[l] = (int*)malloc(p*sizeof(int));


        for (l=0;l<n;l++)
                c[l] = (int*)malloc(p*sizeof(int));

        for (i1=0;i1<n;++i1){
                for (j1=0;j1<m;++j1){
```

```
                        a[i1][j1]=rand()%20;
                }
        }
        for (i2=0;i2<m;++i2){
                for (j2=0;j2<p;++j2){
                        b[i2][j2]=rand()%20;
                }
        }

        for(i=0;i<n;++i){
                for(j=0;j<p;++j){
                        int sum=0;
                        for (k=0;k<m;++k)
                                sum=sum+a[i][k]*b[k][j];
                        c[i][j]=sum;
                }
        }

        for (l=0;l<n;l++) {
                free(c[l]);
                free(a[l]);
        }
        free(c);
        free(a);

        for (l=0;l<m;l++) {
                free(b[l]);
        }
        free(b);
}
int main(int ac, char **av){
        int n=2000;
        int p=2000;
        int m=2000;

        mult_3(n,m,p);

        exit(0);
}
```

2. Multiplicación de matrices con 6 bucles anidados

En el siguiente código podemos ver la multiplicación con seis bucles anidados, los cuales son representados por los seis últimos for , estos realizan la multiplicación de dos matrices en este caso la matriz a y la matriz b y almacenan en resultado en la matriz c, a diferencia del primer algoritmo este utiliza los tres primeros for para dividir la matriz en partes más pequeñas para que la ejecución sea más eficiente , ya que cada parte cabe en la mamoria caché se reducen en gran magnitud los caché misses.

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
//#include <sys/time.h>

#define min(a,b) \
   ({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a < _b ? _a : _b; })

void mult_6(int n, int m, int p)
{       srand(time(NULL));
        int **c,**a,**b;
        int i1, i2,j1,j2,k1,k2,l;
        float m_=32768.0;
        int t=round(sqrt(m));

        c = (int **)malloc(n*sizeof(int*));
        a = (int **)malloc(n*sizeof(int*));
        b = (int **)malloc(m*sizeof(int*));

        for (l=0;l<n;l++)
                a[l] = (int*)malloc(m*sizeof(int));

        for (l=0;l<m;l++)
                b[l] = (int*)malloc(p*sizeof(int));

        for (l=0;l<n;l++)
                c[l] = (int*)malloc(p*sizeof(int));

        for (i1=0;i1<n;++i1){
                for (j1=0;j1<m;++j1){
                        a[i1][j1]=rand()%20;
                }
```

```
        }
        for (i2=0;i2<m;++i2){
                for (j2=0;j2<p;++j2){
                        b[i2][j2]=rand()%20;
                }
        }

        for (i1=0;i1<n;i1+=t){
                for (j1=0;j1<p;j1+=t){
                        for (k1=0;k1<m;k1+=t){
                                for (i2=i1;i2<min(i1+t,n);++i2){
                                        for (j2=j1;j2<min(j1+t,p);++j2)
                                            {
                                                int sum=0;
                                                for (k2=k1;k2<min(k1+t,
                                                    m);++k2){
                                                        sum=sum+a[i2][
                                                            k2]*b[k2][
                                                            j2];
                                                }
                                                c[i2][j2]=sum;
                                        }
                                }
                        }
                }
        }

        printf("%i\n",c[232][232]);

        for (l=0;l<n;l++) {
                free(c[l]);
                free(a[l]);
        }
        free(c);
        free(a);

        for (l=0;l<m;l++) {
                free(b[l]);
        }
        free(b);
}

int main(int ac, char **av){
        int n=2000;
        int p=2000;
        int m=2000;
        mult_6(n,m,p);
        exit(0);
}
```

3. Análisis con cachegrind de valgrind

Esta herramienta simula la interacción con la caché de una máquina.

Cuando ejecutamos el comando *valgrind -tool -cachegrind ./mul3* , me apareció una advertencia.

warning: L3 cache found , using its data for he LL simulation

Esto quiere decir que mi máquina tiene tres niveles de cáche, pero Cachegrind sólo simula dos niveles , en esta oportunidad simulará el primer y el último nivel, Además , hay dos cachés de primer nivel : I1 y D1 , que son las instrucciones y los datos respectivamente .

En las siguientes imagenes presentaré las estadísticas de caché del los algoritmos previamente descritos.

```
==17420== D   refs:      112,301,087,283  (112,224,706,528 rd   + 76,380,755 wr)
==17420== D1  misses:      9,515,517,698  (  9,511,009,233 rd   +  4,508,465 wr)
==17420== LLd misses:      8,051,477,189  (  8,046,969,167 rd   +  4,508,022 wr)
==17420== D1  miss rate:             8.4% (            8.4%     +        5.9%  )
==17420== LLd miss rate:             7.1% (            7.1%     +        5.9%  )
```

Figura 1: Estadísticas de caché del algoritmo con tres bucles

```
==21676== D   refs:      173,375,516,169  (156,030,609,388 rd   + 17,344,906,781
 wr)
==21676== D1  misses:         45,341,490  (     30,911,091 rd   +     14,430,399
 wr)
==21676== LLd misses:         17,036,259  (     16,276,310 rd   +        759,949
 wr)
==21676== D1  miss rate:             0.0% (            0.0%     +            0.0
%  )
==21676== LLd miss rate:             0.0% (            0.0%     +            0.0
%  )
```

Figura 2: Estadísticas de caché del algoritmo con seis bucles

En cada imagen , el primer cloque corresponde a la caché de instrucciones , y el segundo ,bloque a los datos ; cada uno registra el número de lecturas a caché (linea 1), número de *misses* en las cachés de primer y último nivel (2 y 3 líneas), así como

el porcentaje de *misses* en cada nivel (4 y 5 líneas). Adicionalmente , el segundo bloque muestra por separado la información mencionada para la lectura y escritura en memoria. Finalmente , el último bloque muestra los accesos a memoria y los *misses* totales en la caché de último nivel , así como el porcentaje general de *misses*.

Lo primero que salta a la vista es que el algoritmo de 6 bucles anidados tiene menos *misses* en general que el algoritmo de 3 bucles anidados principalmente el la caché de último nivel, esto sucede porque es la caché que tiene mayor influencia cuando el programa está corriendo .

Para visualizar mejor las diferencias entre los dos programas , empleamos el comando *cg_diff file1 file2*

Ir	I1m	ILm	Dr	D1mr	DLmr	Dw	D1mw	DLmw
31.6 %	3.9 %	3.7 %	39 %	-99.7 %	-99.8 %	22608.5 %	220.1 %	-83.1 %

Los porcentajes son de el algoritmo con 6 bucles respecto al algoritmo con 3 bucles , aunque las lecturas al caché de instrucciones aumentaron mucho, los *misses* aumentaron en menos del 4 % , las escrituras a caché aumentaron significantemente , al igual que los *misses* en la caché de primer nivel aún así el porcentaje de *misses* sigue siendo menor que el primer algoritmo.

Con el programa Kcachegrind , se puede observar información más detallada . En el panel derecho están los tipos de eventos , los mismos que antes vimos más la suma de los *misses* de la caché de nivel 1 y la estimación del ciclo ; además incluye el costo inclusivo y por sí mismo de cada función por cada tipo d evento ; en el panel izquierdo están las funciones con sus respectivos aportes al costo del evento actualmente seleccionado , de mayor a menor. como podemos ver en las siguientes imágenes de cada algoritmo .

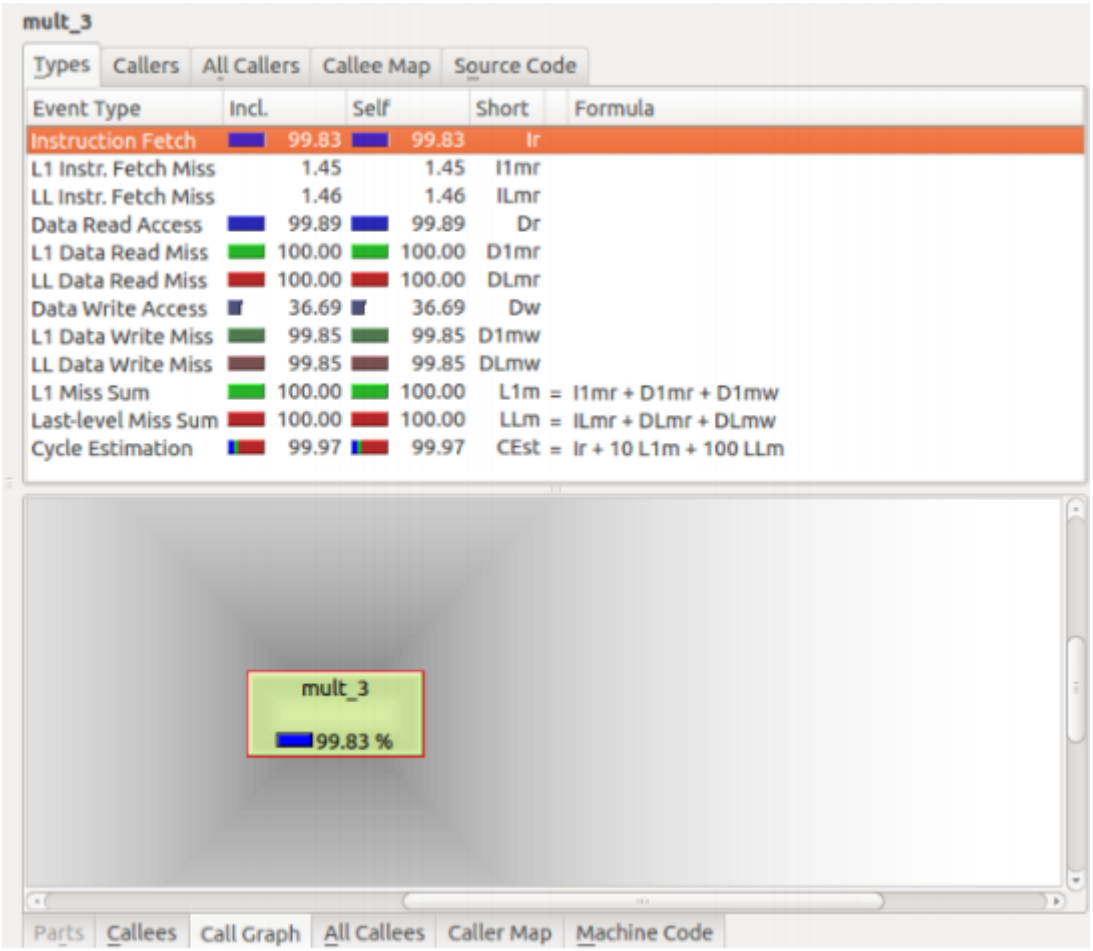

Figura 3: Estadísticas detallada en porcentajes de caché del algoritmo con tres bucles

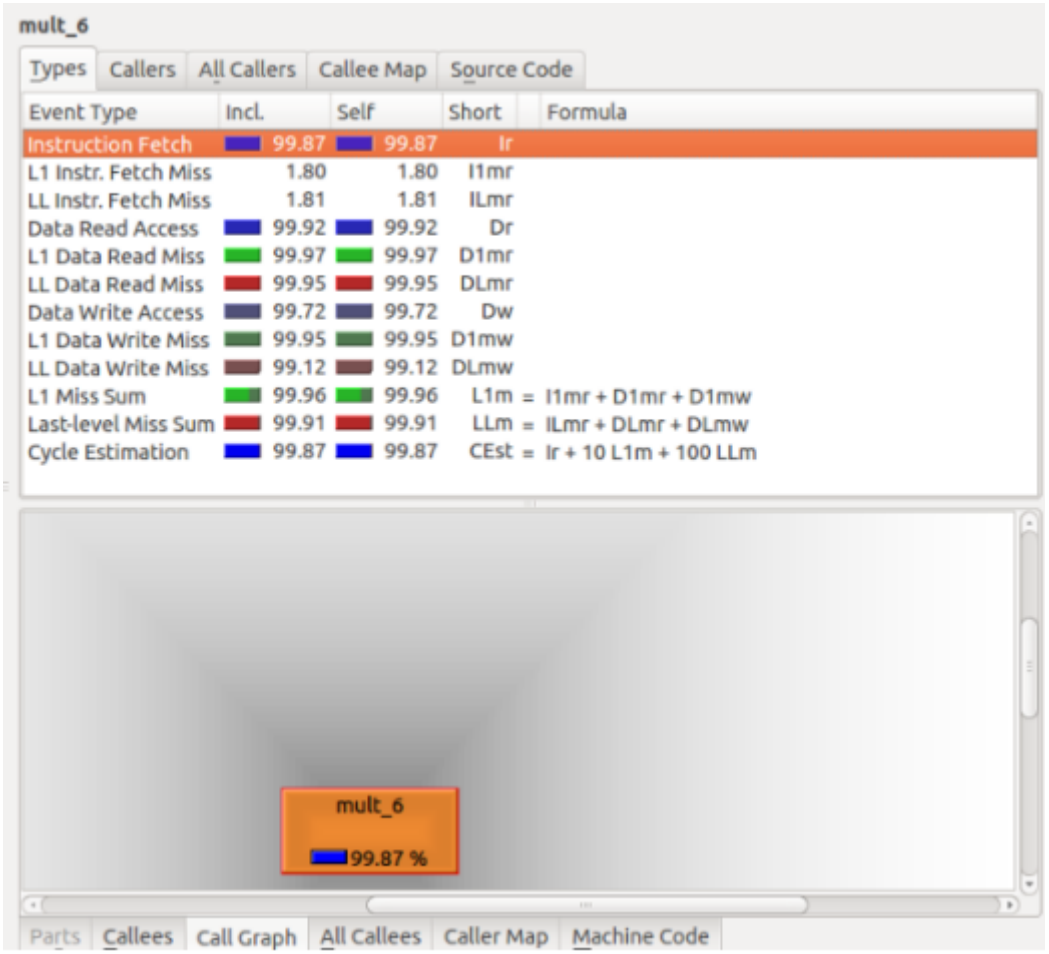

Figura 4: Estadísticas detallada en porcentajes de caché del algoritmo con seis bucles