Multiplicación de matrices utilizando la división por bloques

Gabriel Aparicio Tony

April 3, 2017

1 Introducción

Debido al gran nivel de integración y el diseño de arquitectura a gran escala, la capacidad de cálculo de los microprocesadores ha crecido considerablemente en los últimos años. Desafortunadamente, el incremento en la velocidad del procesador no ha ido a la par con el incremento en velocidad de memoria. Para reconocer completamente el potencial de los procesadores, la jerarquía de memoria debe ser usada eficientemente.

Mientras que los caches de datos han demostrado ser efectivos en aplicaciones de propósito general para reducir esta diferencia de velocidad, su efectividad en código de tipo numérico no ha sido establecida.

Una característica distintiva de las aplicaciones numéricas es que tienden a operar en datasets muy grandes. La memoria cache probablemente solo pueda almacenar una pequeña parte de una matriz; por lo que si los datos son reutilizados, al momento de serlo existe la posibilidad de que hayan sido desplazados de la memoria.

2 Optimización por bloques

La optimización por bloques realiza los cálculos en submatrices. Puede ser aplicado para múltiples niveles de jerarquía de memoria, incluyendo la memoria virtual, caches, registros vector y registros escalares. Por ejemplo, cuando se aplica un algoritmo que utiliza bloques en los niveles de cache y registros, se llega a observar que la multiplicación de dos matrices puede incrementarse hasta un factor 4.3, en una máquina con un desempeño relativamente alto en el subsistema de memoria.

3 Código fuente

3.1 Utilizando 3 bucles

```
void mult_matrix(int m1[][m_size],int m2[][m_size],int r[][m_size],
    int n)
{
     for(int i=0;i<n;i++)
     {
          for(int j=0;j<n;j++)
          {
               r[i][j]=0;
               for(int k=0;k<n;k++)
               {
                    r[i][j]+=m1[i][k]*m2[k][j];
               }
          }
     }
}
```

3.2 Utilizando 6 bucles

```
void block_mult(int m1[][m_size],int m2[][m_size],int r[][m_size],
    int n,int block_num)
{
    int block_size = n/block_num;

    for(int ii=0;ii<block_num;ii++)
    {
        for(int jj=0;jj<block_num;jj++)
        {
            for(int kk=0;kk<block_num;kk++)
            {
                for(int i=ii*block_size;i<(ii+1)*block_size;i++)
                {
                    for(int j=jj*block_size;j<(jj+1)*block_size;j
                        ++)
                    {
                        for(int k=kk*block_size;k<(kk+1)*block_size
                            ;k++)
                        {
                            r[i][j] += m1[i][k]*m2[k][j];
                        }
                    }
                }
            }
        }
    }
}
```

3.3 Código para medir el tiempo de ejecución

```
int main()
{

    int m1[m_size][m_size];
    fill_matrix(m1,m_size,2);
    int m2[m_size][m_size];
    fill_matrix(m2,m_size,2);
    int r[m_size][m_size];
    fill_matrix(r,m_size,0);

    auto _start = chrono::system_clock::now();

    //mult_matrix(m1,m2,r,m_size);
    //block_mult(m1,m2,r,m_size,2);

    auto _end = chrono::system_clock::now();

    auto elapsed = chrono::duration_cast<chrono::microseconds>(_end
        - _start);
    cout<<"Execution␣time:␣"<<elapsed.count()<<endl;
        return 0;
}
```

N	Medición 1	Medición 2	Medición 3
200	71129	74448	70344
400	603133	615750	593598
600	2141432	2125741	2142713

Table 1: Tiempo de ejecución en la multiplicación normal de matrices (3 loops)

N	Medición 1	Medición 2	Medición 3
200	70986	69184	71681
400	560100	561386	564184
600	1923947	1904520	1934584

Table 2: Tiempo de ejecución multiplicación por bloques (6 loops)

Método	I1 Misses	D1 Misses	LL Misses
Normal	705	223 761 054	106 036
Por bloques	707	14 789 052	117 214

Table 3: Comparación en función del número de cache misses

4 Experimentos

4.1 Tiempo de ejecución

En las tablas 1 y 2 se muestran los experimentos hechos con matrices cuadradas NxN para 3 valores distintos de N, el tiempo se muestra en microsegundos, donde se puede apreciar que para tamaños relativamente pequeños de N la ganancia obtenida en tiempo de ejecución es prácticamente imperceptible, mientras que a partir de ciertos valores altos de N notamos la diferencia que se consigue al utilizar el algoritmo de multiplicación por bloques.

4.2 Valgrind y Kcachegrind

En la tabla 3 se muestra una comparación entre el número de cache misses que se obtiene al aplicar el método clásico para multiplicar matrices (3 loops) y el número de cache misses resultado de aplicar el algoritmo por bloques (6 loops). En este caso se considera una matriz NxN, para un $N = 600$, vemos que la diferencia es más que notoria.

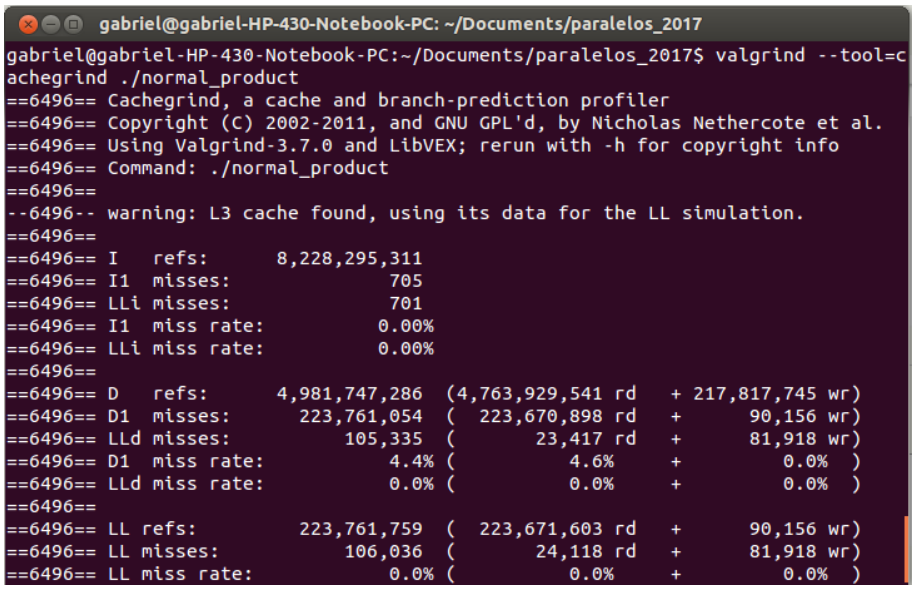

Figure 1: Valgrind - Mutiplicación normal

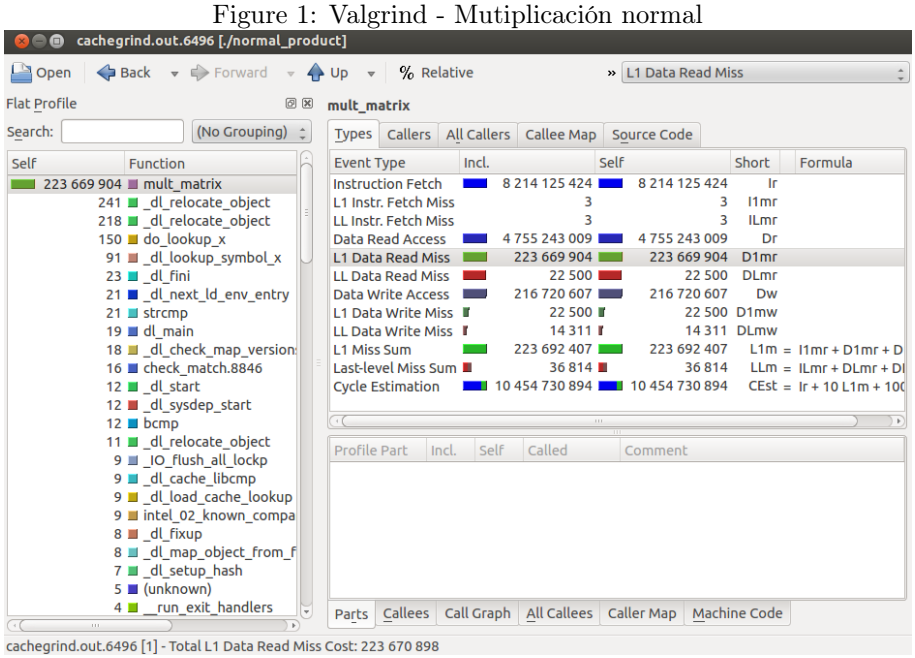

Figure 2: Kcachegrind - Mutiplicación normal

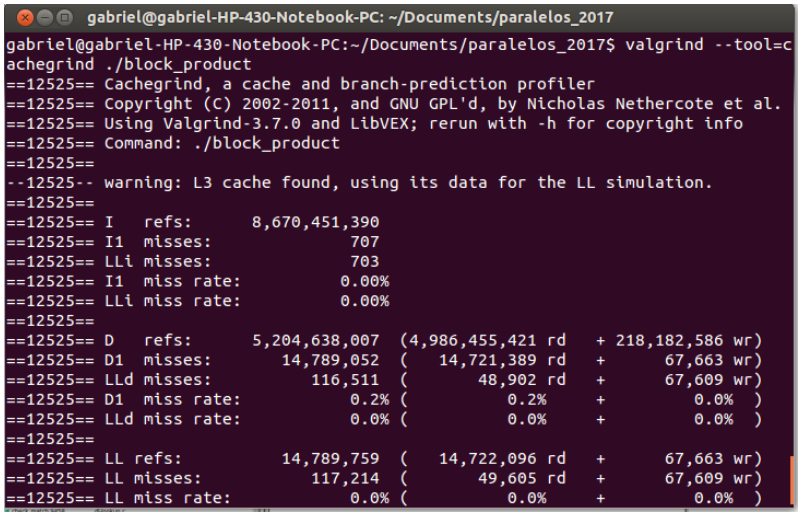

Figure 3: Valgrind - Mutiplicación por bloques

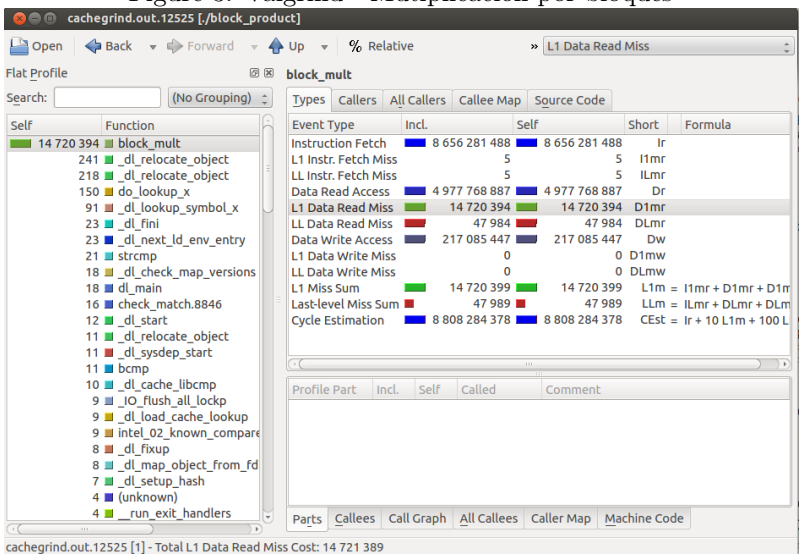

Figure 4: Kcachegrind - Mutiplicación por bloques

5 Conclusiones

El uso de algoritmos por bloques deriva en una mejora del tiempo de ejecución de una gran cantidad de algoritmos, en este caso específico la multiplicación de matrices, debido a que en lugar de realizar cálculos de elemento a elemento, divide las matrices en submatrices de menor tamaño con el objetivo de reducir el número de cache misses. Esto es necesario debido a que en una matriz NxN, el tamaño de N posiblemente sea más grande que un bloque cache, por lo que se incurre en una considerable cantidad de cache misses, para reducir esta cantidad se divide cada matriz en porciones más pequeñas que puedan residir en la memoria cache por una mayor cantidad de tiempo, resultando en un menor número de cache misses.