Programación Paralela y Distribuida- Informe 01

César Bragagnini

Mayo 2016

1 Introducción

El siguiente trabajo muestra la eficiencia de distintos programas para resolver la multiplicación de matrices, se elegio matrices cuadradas NxN por comodidad, asi mismo los valores de N son multiplos de 32, el valor de 32 sera usado para generar los bloques de matrices de 32x32(submatrices).

2 Multiplicación de matrices

En los siguientes programas usan variables temporales a excepción del programa Base,

2.1 Programa Base - Naive Multiplication

El siguiente codigo muestra la multiplicación de 2 matrices cuadradras de la forma clásica pero es la menos óptima.

```
//...
#define N 3200

int A[N][N], B[N][N], C[N][N];

void multAB(){
  for(int i = 0; i < N; i++)
    for(int j = 0; j < N; j++){
      for(int k = 0; k < N; k++)
        A[i][j] += B[i][k] * C[k][j];
  }
}
//...
```

2.2 Programa A - SAXPY Operation

El siguiente código muestra la multiplicación de 2 matrices cuadradras basado en una operación SAXPY(multiplicación de un escalar por vector matriz), donde se busca mantener la fila del resultado y la fila de una matriz en la misma memoria, logrando mayor localidad(temporal and space locality). Se usa una

variable temporal para mantenerla en memoria mayor tiempo que un acceso a algún B[i][j].

```
//...
#define N 3200

int A[N][N], B[N][N], C[N][N], b;

void multAB(){
  for(int i = 0; i < N; i++)
    for(int k = 0; k < N; k++){
      b = B[i][k];
      for(int j = 0; j < N; j++)
        // SAPXY Operation
        // A = b * X + Y
        A[i][j] += b * C[k][j];
    }
}
//...
```

2.3 Programa B - Blocked Matrix(6-for) with SAXPY Operation

El siguiente código muestra la multiplicación de 2 matrices cuadradras basado en multiplicación de bloques de matrices con una operación SAXPY(multiplicación de un escalar por vector matriz), donde se busca mantener la fila del resultado y la fila de una matriz en la misma memoria, logrando mayor localidad(temporal and space locality). Se usa una variable temporal para mantenerla en memoria mayor tiempo que un acceso a algún B[i][j]. No se usa arreglos temporales para cada submatriz.

```
//...
#define N 3200
#define BLOCK_SIZE 32

int A[N][N], B[N][N], C[N][N], b;

void multAB(){
  for(int i = 0; i < N; i+= BLOCK_SIZE )
    for(int k = 0; k < N; k += BLOCK_SIZE)
      for(int j = 0; j < N; j += BLOCK_SIZE)
        for(int i2 = i; i2 < i + BLOCK_SIZE; i2++)
    for(int k2 = k; k2 < k + BLOCK_SIZE; k2++){
      b = B[i2][k2];
      for(int j2 = j; j2 < j + BLOCK_SIZE; j2++)
        // SAPXY Operation
        // A = b * X + Y     WITH
        // MULTIPLICATION LITTLE SUBMATRIX
        // BECAUSE STAY IN MEMORY FOR MUCH TIME
        // THAN LARGE MATRIX
        A[i2][j2] += b * C[k2][j2];
    }
}
//...
```

2.4 Programa C - Blocked Matrix(6-for) with Temporal Arrays and SAXPY Operation

El siguiente código muestra la multiplicación de 2 matrices cuadradras basado en multiplicación de bloques de matrices con una operación SAXPY(multiplicación de un escalar por vector matriz), donde se busca mantener la fila del resultado y la fila de una matriz en la misma memoria, logrando mayor localidad(temporal and space locality). Se usa una variable temporal para mantenerla en memoria mayor tiempo que un acceso a algún B[i][j]. Se usa arreglos temporales para cada submatriz permitiendo una mayor localidad.

```
//...
#define N 3200
#define BLOCK_SIZE 32

int A[N][N], B[N][N], C[N][N];
int i, i2, j, j2, k, k2;
int *rres, *rmul1, *rmul2;

void multAB(){
  for (i = 0; i < N; i += BLOCK_SIZE)
    for (j = 0; j < N; j += BLOCK_SIZE)
      for (k = 0; k < N; k += BLOCK_SIZE)
        for (i2 = 0, rres = &A[i][j], rmul1 = &B[i][k]; i2 <
    BLOCK_SIZE; ++i2, rres += N, rmul1 += N){
    for (k2 = 0, rmul2 = &C[k][j]; k2 < BLOCK_SIZE; ++k2, rmul2 +=
    N){
      for (j2 = 0; j2 < BLOCK_SIZE; j2 += 2){
         rres[j2] += rmul1[k2] * rmul2[j2];
      }
          }
        }
}
//...
```

2.5 Programa D - Blocked Matrix(5-for)

El siguiente código muestra la multiplicación de 2 matrices cuadradras basado en multiplicación de bloques de matrices sin una operación SAXPY(multiplicación de un escalar por vector matriz).

```
//...
#define N 3200
#define BLOCK_SIZE 32

int A[N][N], B[N][N], C[N][N], sum;

void multAB(){
  for(int k = 0; k < N; k+= BLOCK_SIZE )
    for(int j = 0; j < N; j += BLOCK_SIZE)
      for(int i2 = 0; i2 < N; i2++)
        for(int j2 = j; j2 < j + BLOCK_SIZE; j2++){
    sum = A[i2][j2];
    for(int k2 = k; k2 < k + BLOCK_SIZE; k2++)
```

```
14          sum += B[i2][k2] * C[k2][j2];
15      A[i2][j2] = sum;
16          }
17  }
18  //...
```

3 Resultados

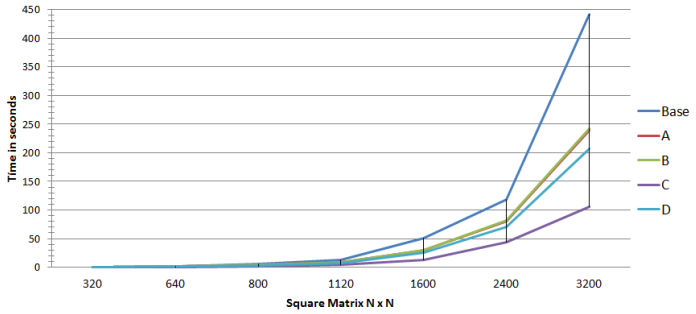
Las pruebas fueron realizadas en una Procesador Intel Core 2 Quad CPU Q6600-2.40GHz 4, y con un compilador GCC 4.8.4 sobre Ubuntu 14.04-64bits, para contabilizar las cache misses se uso KCacheGrind- ValGrind-3.10.1. Se escogío valores de BLOCK_ SIZE = 32 debido a que 64 es el tamaño de linea de la cache 1, se considero para mantener 2 filas de 32. Para revisar este valor se uso el comando: getconf LEVEL1_DCACHE_LINESIZE

3.1 Programas vs Tiempo de ejecución en segundos

La siguiente figura muestra la relacion entre el N y la ejecución de los programas. Se ve una clara diferencia con el programa C que es el mas optimizado.

N\Programa					
	BASE	A	B	C	D
320	0.269084	0.204447	0.21085	0.107158	0.18282
640	2.144752	1.606287	1.641447	0.845934	1.426748
800	5.269604	3.700415	3.788976	1.646598	3.196964
1120	12.166131	8.25645	8.435457	4.497312	7.125952
1600	50.788528	29.940838	30.367004	13.186578	25.649763
2400	118.959145	80.750648	82.251534	44.332302	70.269028
3200	441.403442	239.125046	242.491959	105.91864	206.226852

La siguiente figura muestra la relación entre el N y la ejecución de los programas. Se ve una clara diferencia con el programa C que es el mas optimizado.



3.2 Conteo de Cache misses

Se uso para una matriz cuadrada de 2400x2400 por ser uno de los mas representativos y con mayor uso de memoria. Solo se tomo en cuenta la función MultAB. La siguiente figura muesta el conteo de cache miss para L1, LL.

Programa\N	2400	
	L1	LL
BASE	14279299500	864713715
A	86472000	86472000
B	54360000	27720000
C	54360001	27714603
D	54360000	27720000

4 Conclusiones

- Las mejoras a cada codigo presentadas no son nada dificiles, aun mas su uso pasa desapercibida si no se conoce el manejo de memoria cache, memoria por parte del procesador.
- El uso de variables temporales permite mantener en memoria mayor tiempo posible.
- La eleccion del bloque debe ser de acuerdo al tamaño de la linea del cache.
- Es probable que en el futuro exista hardware que realize esta mejoras de manera desapercibida para el programador.

5 Referencias

1. https://software.intel.com/en-us/articles/putting-your-data-and-code-in-order-optimiza
2. https://devblogs.nvidia.com/parallelforall/six-ways-saxpy/
3. https://software.intel.com/sites/products/vcsource/files/GEMM.pdf
4. http://www.cs.berkeley.edu/~knight/cs267/hw1.html
5. http://stackoverflow.com/questions/1907557/optimized-matrix-multiplication-in-c
6. http://stackoverflow.com/questions/16115770/block-matrix-multiplication
7. http://csapp.cs.cmu.edu/2e/waside/waside-blocking.pdf
8. http://assoc.tumblr.com/post/409759537/cache-misses-with-valgrind-and-kcachegrind