Universidad Nacional de San Agustín

Escuela Profesional de Ciencia de la Computación

Algoritmo Paralelos

Práctica de Séptimo Laboratorio

Alumno:
Eddy René Cáceres Huacarpuma

CUI:
20101934

17 de julio de 2017

Índice general

Capítulo 1

Introducción

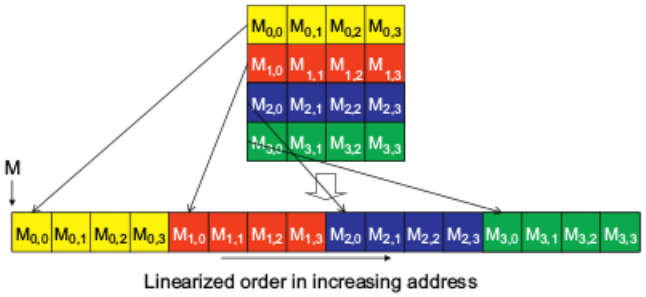
En el presente trabajo se realizó la implementación de los ejemplos presentados en el capítulo 4 del Libro **"Programming Massively Parallel Processors"** titulado **"Memory and data locality "**. El repositorio de los programas es :

https : *//github.com/eddyrene/Laboratorio/tree/master/7to%20Laboratorio*

1.1. Tiled Matrix Multiplication: Performance Considerations

La multiplicación de matrices mejorada usando memoria compartida, es una excelente manera de acelerar la ejecución de tal operación , sin embargo existe otros factores a considerar.

La primera es que al pasar nuestra matriz a un puntero de cuda , pasamos un array de dimensión N x M , donde N es las filas y M las columnas, entonces organizamos los grids de threads que le levantaran en 2D para abstraer la doble dimensionalidad y realizar los cálculos .

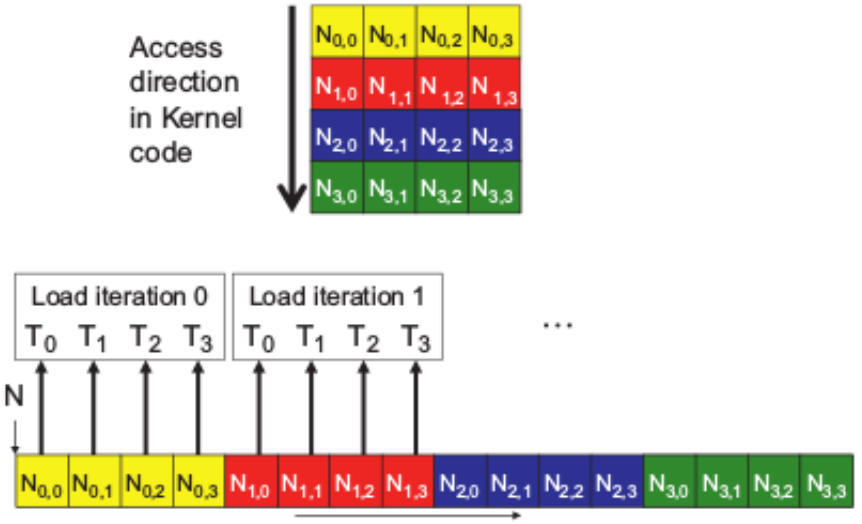


Como sabemos por las leyes de localidad espacial leer datos continuos es mas rápido que los que no lo están en C y C++ la localidad espacial se aplica sobre la misma fila , es decir que los elementos $M_{0,0}$ y $M_{0,1}$ son de más rápido acceso que los los elementos $M_{0,0}$ y $M_{1,0}$, hasta ahora hemos seguido esta convención.

Cuando analizaremos que sucede cuando realizamos la operación con la convención mencionada. Para seguir esta convención resalta que el índice se forma de la manera :

$$N[k * Width + Col]$$

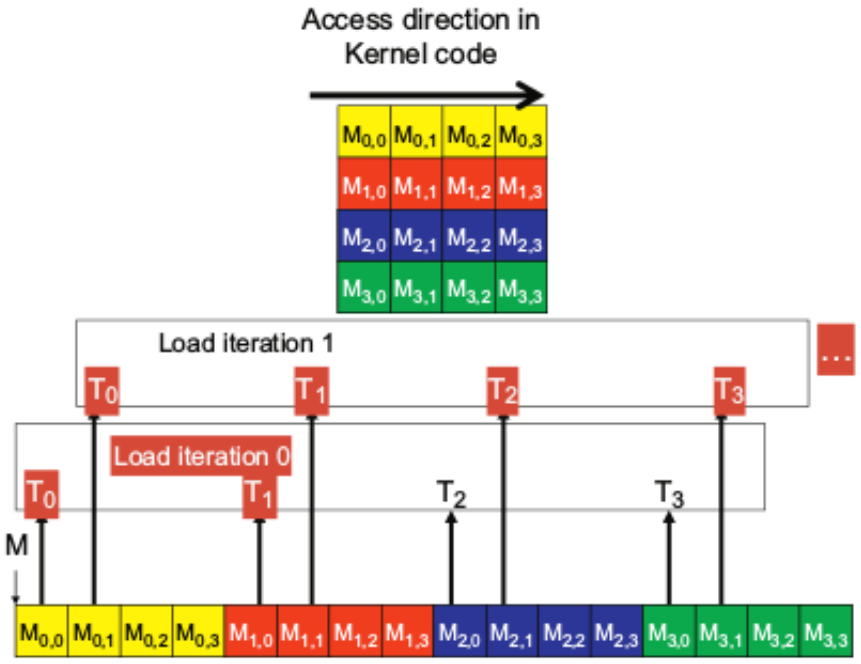
que nos permite el recorrido de cada thread, cada uno sobre una fila , para una apreciación más clara, ver gráfico de abajo .



En la primera iteración cada thread, se posiciona en el primer elemento de cada fila, para la segunda iteración lo harán sobre el segundo elemento , y así hasta terminar.

Ahora analizaremos el recorrido a través de columnas. La imagen de abajo describe la posición que tomará cada thread según la iteración .

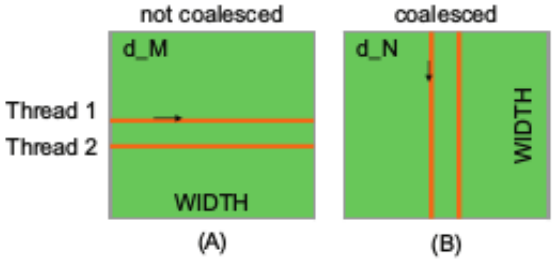
$$M[Row * Width + k]$$



Después de haber visto esto la pregunta es cual de los 2 operaciones es más rápida. Para responder rápidamente a esta pregunta ejecutamos las operaciones de en un principio sudatoria de matrices sobre filas y columnas, obteniendo la tabla de abajo , donde claramente se aprecia que la operación de una columna sobre una fila es más rápida.

	Sumatoria de Matrices		
	2000x2000	6000x6000	11000x11000
1 thread x 1 posicion	21,33	590,66	2207,07
1 thread x 1 Fila	21,46	591,18	2221,03
1 thread x 1 Columna	3,66	33,01	111,58

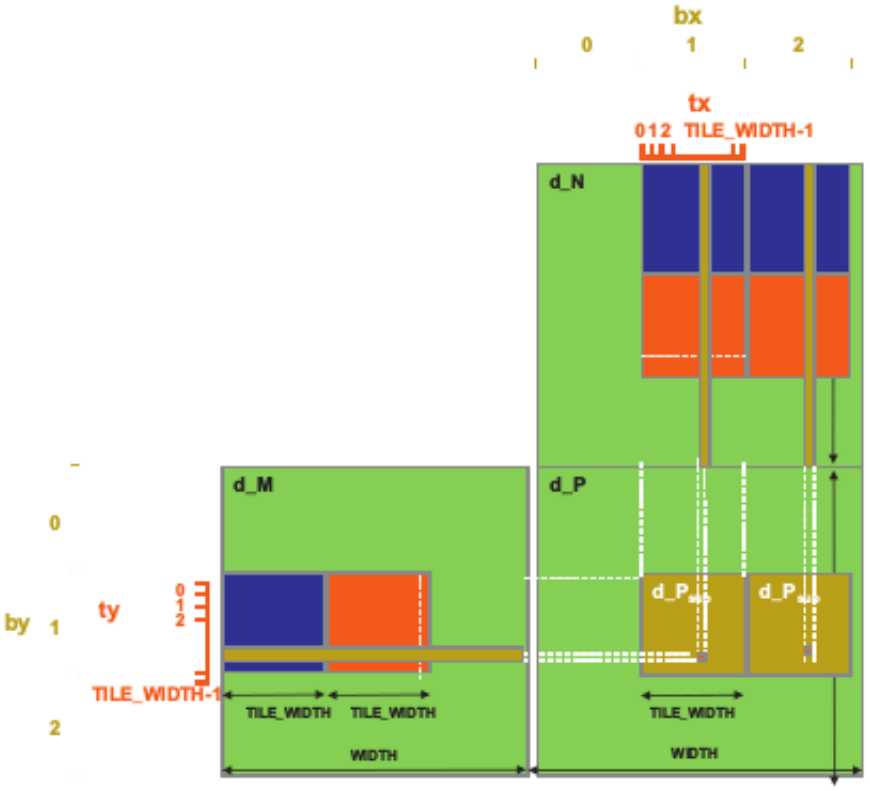
La explicación de estos resultados es que al momento de leer los datos de memoria principal ocurre la misma resultado que ya se mencione en RAM. osea leer datos continuos de un array es mas rápido que los que no lo están, cuando se divide la data para ser leída por filas, estamos cargando los datos de forma discontinua separadas por un espacio *with* para cada thread, lo que no permite una devolución rápida, por otro lado cuando realizamos el recorrido por columnas, cada thread se ubica sobre la posición de la columna, y cuando se consulta a Memoria Global la devolución se hace en ráfaga y es mas rápida.



1.1.1. Granularidad de Threads

Una de las cosas a tener en cuenta, es que mientras más procesamiento haga cada thread tenemos mayor rendimiento. En el problema de multiplicación de matrices recordemos que cada thread se encarga del cálculo de un nuevo valor, a continuación veremos como modificar el kernel para que se cada thread ahora realice el cálculo de al menos dos valores.

Como se aprecia en la imagen debajo, para calcular cada posición, para calcular $d_P(5,5)$ y $d_P(5,5+TILE_WIDTH)$ se usa dos veces los tiles asignados en la matriz M, que usando memoria compartida se reduce el tiempo de mejora, sin embargo lo que haremos es calcular de una vez los dos valores.



Ahora necesitamos 2 índices diferentes para manejar las columnas de los tiles continuos, para así obtener $d_P(i,j)$ y $d_P(i, j + TILE_W IDTH)$,pero observemos que no debe haber solapamiento osea una vez calculado los valores de esos tiles ya no se debe repetir alguno. Esto se logra usando el recorrido para el índice de la columna 1. Debido que cada Blockid.x es la enumeración de los bloques que estamos usando y cada bloque es del mismo orden de memoria compartida.

Col1 = TILE_WIDTH* (2*blockIdx.x) + threadIdx.x;

y para el segundo índice solo añadimos

Col2 = TILE_WIDTH* (2*blockIdx.x+1) + threadIdx.x;

En la ejecución se evaluarán :

- blockIdx.x = 0;
  Col1 = 32* (2 *0) + 0 = 0
  Col2 = 32* (2 *0 +1) + 0 = 32
- blockIdx.x = 1

Col1 = 32* (2 *1) + 0 = 64
Col2 = 32* (2 *1 +1) + 0 = 96

- blockIdx.x = 2
  Col1 = 32* (2 *2) + 0 = 128
  Col2 = 32* (2 *2 +1) + 0 = 160

Entonces el kernel nos quedaría asi

```
1
2  __global__ void MatrixMulKernelGranularity(int * d_P, int * d_M, int*
       d_N,int Width)
3  {
4    __shared__ int Mds[TILE_WIDTH][TILE_WIDTH];
5    __shared__ int N1ds[TILE_WIDTH][TILE_WIDTH]; // 2 tamano
6    __shared__ int N2ds[TILE_WIDTH][TILE_WIDTH]; // 2 tamano
7
8    int bx = blockIdx.x; int by = blockIdx.y;
9    int tx = threadIdx.x; int ty = threadIdx.y;
10   int Row = by * TILE_WIDTH+ ty;
11   int Col1 = (2*bx) *TILE_WIDTH+ tx;
12   int Col2 = (2*bx+1)*TILE_WIDTH+ tx;
13
14   int Pvalue1 = 0;
15   int Pvalue2 = 0;
16   for (int ph = 0; ph < (Width/TILE_WIDTH); ++ph)
17   {
18     Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
19     N1ds[ty][tx]= d_N[(ph*TILE_WIDTH + ty)*Width + Col1];
20     N2ds[ty][tx]= d_N[(ph*TILE_WIDTH + ty)*Width + Col2];
21     __syncthreads();
22     for (int k = 0; k < TILE_WIDTH;++k)
23     {
24       Pvalue1 += Mds[ty][k] * N1ds[k][tx];
25       Pvalue2 += Mds[ty][k] * N2ds[k][tx];
26     }
27     __syncthreads();
28   }
29   d_P[Row*Width + Col1]= Pvalue1;
30   d_P[Row*Width + Col2]= Pvalue2;
31 }
```

y el número de threads ahora se reduce a la mitad, ya que hemos incrementado el cálculo a dos posicione de bloques adyacentes.

```
1 int gridSize = N/TILE_WIDTH;
2 int THREADS_PER_BLOCK=32;
3 dim3 dimGrid(gridSize/2, gridSize, 1);
4 dim3 dimBlock(THREADS_PER_BLOCK, THREADS_PER_BLOCK, 1);
```

1.2. Experimentos

Uno de los factores determinantes es conocer la arquitectura del device en mi caso trabajo con una Nvidia GEFORCE 930MX.

```
Device 0: "GeForce 930MX"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  ( 3) Multiprocessors, (128) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1020 MHz (1.02 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime Version = 8.0, NumDevs = 1, De
Result = PASS
```

Figura 1.1: Carácteristicas del Dispositivo de Video

Los experimentos realizados sobre diferentes matrices de un orden considerablemente grande son

Tabla de comparación - Multiplicacion de Matrices					
	8000 X 8000	10000 X 10000	12000 X 12000	BlockSize	Tiled Size
Global Memory	42,584	66,605	113,883	16 x 16	16 x 16
Shared Memory	13,756	25,683	44,429	16 x 16	16 x16
Global Memory	31,243	68,641	112,952	32 x 32	32 x 32
Shared Memory	11,612	--	39,569	32 x 32	32 x 32
Granularity S.M.	8,89	17,361	30,236	32 x 32	32 x 32

1.2.1. Capturas de pantalla

- Multiplicación correcta , se muestra los resultados de la multiplicación tiled normal y la multiplicación incrementando la granularidad.

```
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/7toLaboratorio $ ./p2 10
blocks :
1
 threds:
 10
Elapsed time : 0.025344 ms
Printing Matrix C
2850 2895 2940 2985 3030 3075 3120 3165 3210 3255
7350 7495 7640 7785 7930 8075 8220 8365 8510 8655
11850 12095 12340 12585 12830 13075 13320 13565 13810 14055
16350 16695 17040 17385 17730 18075 18420 18765 19110 19455
20850 21295 21740 22185 22630 23075 23520 23965 24410 24855
25350 25895 26440 26985 27530 28075 28620 29165 29710 30255
29850 30495 31140 31785 32430 33075 33720 34365 35010 35655
34350 35095 35840 36585 37330 38075 38820 39565 40310 41055
38850 39695 40540 41385 42230 43075 43920 44765 45610 46455
43350 44295 45240 46185 47130 48075 49020 49965 50910 51855
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/7toLaboratorio $ ./p3 10
Elapsed time : 0.021088 ms
Printing Matrix C
2850 2895 2940 2985 3030 3075 3120 3165 3210 3255
7350 7495 7640 7785 7930 8075 8220 8365 8510 8655
11850 12095 12340 12585 12830 13075 13320 13565 13810 14055
16350 16695 17040 17385 17730 18075 18420 18765 19110 19455
20850 21295 21740 22185 22630 23075 23520 23965 24410 24855
25350 25895 26440 26985 27530 28075 28620 29165 29710 30255
29850 30495 31140 31785 32430 33075 33720 34365 35010 35655
34350 35095 35840 36585 37330 38075 38820 39565 40310 41055
38850 39695 40540 41385 42230 43075 43920 44765 45610 46455
43350 44295 45240 46185 47130 48075 49020 49965 50910 51855
```

- Se muestra la captura de pantalla de la ejecución de la multiplicacion de matrices aumentando la granularidad.

```
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/7toLaboratorio $ ./p3 8000
Elapsed time : 8890.048828 ms
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/7toLaboratorio $ ./p3 10000
Elapsed time : 17361.007812 ms
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/7toLaboratorio $ ./p3 12000
Elapsed time : 30236.447266 ms
```

1.3. Conclusiones

Del trabajo realizado se observa que a pesar que utilizar memoria compartida puede mejorar el calculo del kernel, tenemos que tener en cuenta que mientras más procesos que ejecute un thread mejor será el rendimiento del mismo.

Aumentar el calculo a dos posiciones por thread en el ejemplo de la multiplicación de matrices reduce el tiempo en promedio 23 % según los experimentos realizados.