

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN

ESCUELA PROFESIONAL DE CIENCIA DE LA
COMPUTACIÓN



ALGORITMO PARALELOS

Práctica de 6to Laboratorio

Alumno:

Eddy René Cáceres
Huacarpuma

CUI:

20101934

28 de junio de 2017

Índice general

1. Introducción	2
1.1. MATRIX MULTIPLICATION	2
1.2. Experimentos	6
1.3. Conclusiones	10

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/6to%20Laboratorio](https://github.com/eddyrene/Laboratorio/tree/master/6to%20Laboratorio)

1.1. MATRIX MULTIPLICATION

La multiplicación de matrices es una operación útil para varias aplicaciones como IA, computación gráfica, etc. Una de las aceleraciones posibles es utilizar el device para el cálculo independiente de cada resultado, sin embargo es importante tener en cuenta ciertos aspectos al momento de paralelizar con Cuda.

Uno de estos aspectos es el acceso a memoria, como se ha visto en clases existe una forma intuitiva de acceder a las ubicaciones de las matrices dados los índices,

```
1  __global__
2  void matrixMulti(int *c, int *a, int *b, int n)
3  {
4      int row = blockIdx.y * blockDim.y + threadIdx.y ;
5      int col = blockIdx.x * blockDim.x + threadIdx.x ;
6      if ((row < n) && (col < n))
7      {
8          int suma=0;
9          for (int i=0; i<n; ++i)
10             {
11                 suma+=a[row*n+i]*b[i*n+col];
12             }
```

```

13     }
14     c[row*n+col] = suma;
15 }
16 }

```

Sin embargo a pesar de generarse una mejora en la ejecución, aun se puede mejorar si se hace uso de la memoria compartida por el bloque y no solo se accede continuamente a memoria principal del device.

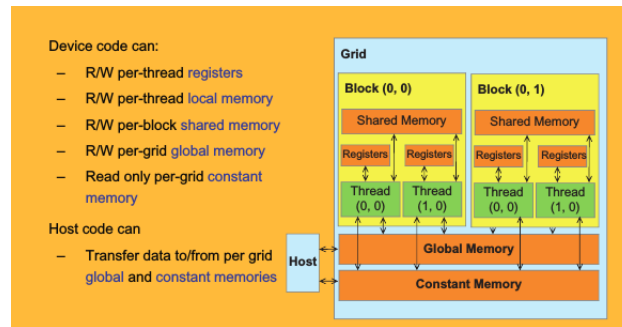


Figura 1.1: Diagrama de distribución de memoria

Como se aprecia en la imagen anterior el acceso a la memoria global es la más cara.

En el ejemplo de multiplicación de matrices sucede una acceso repetitivo a una misma fila de la Matriz A , para multiplicar con una diferente fila de B, en este caso , para hallar un valor de la matriz C.

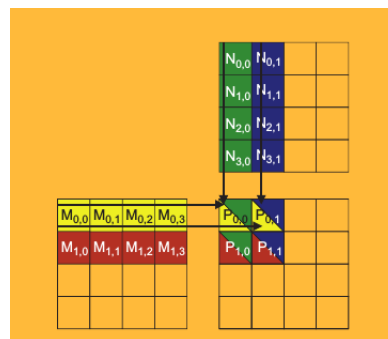


Figura 1.2: Representación de acceso en multiplicación de matrices

Modificamos el código para aprovechar esta propiedad de memoria compartida, obteniendo el código:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <fstream>
4 #include <iostream>
5 using namespace std;
6
7 #define TILE_WIDTH 16
8
9 void fillMatrix(int* a, int n)
10 {
11     int i;
12     for (i = 0; i < n*n; ++i)
13         a[i] = 10; //rand() %5;
14 }
15 __global__
16 void matrixMulti(int *c, int *a, int *b, int n)
17 {
18     int row = blockIdx.y * blockDim.y + threadIdx.y ;
19     int col = blockIdx.x * blockDim.x + threadIdx.x ;
20     if ((row < n) && (col < n))
21     {
22         int suma=0;
23         for(int i=0; i<n; ++i)
24         {
25             suma+=a[row*n+i]*b[i*n+col];
26         }
27         c[row*n+col] = suma;
28     }
29 }
30 __global__ void MatrixMulKernel(int * d_P, int * d_M, int* d_N, int
    Width)
31 {
32     __shared__ int Mds[TILE_WIDTH][TILE_WIDTH];
33     __shared__ int Nds[TILE_WIDTH][TILE_WIDTH];
34     int bx = blockIdx.x; int by = blockIdx.y;
35     int tx = threadIdx.x; int ty = threadIdx.y;
36     // Identify the row and column of the d_P element to work on
37     int Row = by * TILE_WIDTH + ty;
38     int Col = bx * TILE_WIDTH + tx;
39     int Pvalue = 0;
40     // Loop over the d_M and d_N tiles required to compute d_P element
41     for (int ph = 0; ph < Width/TILE_WIDTH; ++ph)
42     {
43         // Collaborative loading of d_M and d_N tiles into shared memory
44         Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
45         Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
46         __syncthreads();
47         for (int k = 0; k < TILE_WIDTH; ++k)
48         {

```

```

49     Pvalue += Mds[ty][k] * Nds[k][tx];
50 }
51 __syncthreads();
52 }
53 d_P[Row*Width + Col] = Pvalue;
54 }
55
56 void printMatrix(string s, int *a, int tam){
57     cout<<s;
58     for(int i=0;i<tam;i++)
59     {
60         for(int j=0;j<tam;j++)
61         {
62             cout<<a[i*tam+j]<<" ";
63         }
64         cout<<endl;
65     }
66 }
67
68 int main(int argc, char *argv[])
69 {
70     srand (time(NULL));
71     int N= strtol(argv[1], NULL, 10);
72     //cout<<N<<endl; return 1;
73     int THREADS.PER.BLOCK =16;
74     int *a, *b, *c; // host copies of a, b, c
75     int *d_a, *d_b, *d_c; //device copies of a,b,c
76     //int size = N*N*sizeof(int);
77     int size=N*N*sizeof(int);
78     cudaMalloc((void **)&d_a, size);
79     cudaMalloc((void **)&d_b, size);
80     cudaMalloc((void **)&d_c, size);
81
82     a = (int *)malloc(size);
83     fillMatrix(a, N);
84     b = (int *)malloc(size);
85     fillMatrix(b, N);
86     c = (int *)malloc(size);
87     cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
88     cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
89     /*int blocks= (N*N + THREADS.PER.BLOCK -1)/THREADS.PER.BLOCK;
90     int blocks2= (N + THREADS.PER.BLOCK -1)/THREADS.PER.BLOCK;
91     cout<<"blocks : \n"<<blocks2<<"\n threds: \n "<<THREADS.PER.BLOCK<<
92         endl; */
93
94     int blocks= (N + THREADS.PER.BLOCK -1)/THREADS.PER.BLOCK;
95     dim3 dimGrid(blocks, blocks, 1);
96     dim3 dimBlock(THREADS.PER.BLOCK,THREADS.PER.BLOCK, 1);
97     cout<<"blocks : \n"<<blocks<<"\n threds: \n "<<THREADS.PER.BLOCK<<

```

```

    endl;
97  cudaEvent_t start, stop;
98  float elapsedTime;
99  cudaEventCreate(&start);
100 cudaEventRecord(start, 0);
101  //matrixAdition<<<blocks,THREADS_PER_BLOCK>>>(d_c, d_a, d_b,N);
102  //matrixAditionRow<<<blocks2,THREADS_PER_BLOCK>>>(d_c, d_a, d_b,N);
103  //matrixMulti<<<blocks2,THREADS_PER_BLOCK>>>(d_c, d_a, d_b,N);
104  //matrixAditionCol<<<blocks2,THREADS_PER_BLOCK>>>(d_c, d_a, d_b,N);
    ;
105  //matrixMulti<<<dimGrid,dimBlock>>>(d_c, d_a, d_b, N);
106  MatrixMulKernel<<<dimGrid,dimBlock>>>(d_c, d_a, d_b, N);
107  cudaEventCreate(&stop);
108  cudaEventRecord(stop, 0);
109  cudaEventSynchronize(stop);
110  cudaEventElapsedTime(&elapsedTime, start, stop);
111  printf("Elapsed time : %f ms\n", elapsedTime);
112  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
113
114  //printMatrix("Printing Matrix A \n",a,N);
115  //printMatrix("Printing Matrix B \n",b,N);
116  //printMatrix("Printing Matrix C \n",c,N);
117  free(a); free(b); free(c);
118  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
119  return 0;
120 }

```

1.2. Experimentos

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

Donde en cantidad total para memoria compartida por bloque es: 49152 bytes = 49 kbytes.

Una de las cosas a tener en cuenta es el tamaño de threads permitidos por bloque en nuestro caso: **1024 thread por Bloque.**

Ya que usamos Grid de **2 dimensiones** como maximo podemos considerar en cada dimensión, teniendo total de 32x32 threads.

Cada bloque tiene 256 threads que realizan 256 acceso a memoria y lo que deseamos es reducir esos accesos, entonces debemos crear memoria del mismo tamaño de bloques, entonces creamos memoria compartida del mismo tamaño 32*32 en este caso usamos 2 matrices que serian igual a una matriz de 512 datos, ahora estos datos son

de tipo *double* que ocupa 8 bytes entonces nuestra memoria seria $512 * 8 = 4096$ bytes.

Estamos usando la capacidad máxima de threads permitidos y nuestra separación de memoria compartida es permitida por ser menor al máximo.

```
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 1.3: Características del Dispositivo de Video

Una vez ejecutados las dos funciones obtenemos la tabla que resume los resultados
Siendo los resultados obtenidos:

Tabla de comparación - Multiplicación 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 x 16
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

Figura 1.4: Cuadro Comparativo usando BlockThreads 16x16

En la tabla anterior podemos ver la mejora de tiempo al usar la máxima capacidad de threads por grid que se ejecuta.

Las capturas de pantalla son

- Ejecución 8000 x 8000

```
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $ ./gl 8000
blocks :
500
threds:
16
Elapsed time : 42584.480469 ms

mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $ ./sa 8000
blocks :
500
threds:
16
Elapsed time : 13756.043945 ms
```

- Ejecución normal Matriz Multiplicación de orden 10K y 12K

```
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $ nvcc -arch=sm_30 Multiplicacion.cu -o sa
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $ ./sa 10000
blocks :
525
threds:
16
Elapsed time : 66605.929688 ms
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $ ./sa 12000
blocks :
750
threds:
16
Elapsed time : 113883.984375 ms
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $
```

- Ejecución Tiled Matriz Multiplicación de orden 10K y 12K

```
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $ ./sa 10000
blocks :
625
threds:
16
Elapsed time : 25683.476562 ms
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $ ./sa 12000
blocks :
750
threds:
16
Elapsed time : 44429.492188 ms
```

- Ejecución con Bloques de threads de 32 x 32

```
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $ ./normthrea32 8000
blocks :
250
threds:
32
Elapsed time : 31243.914062 ms
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $ ./normthrea32 10000
blocks :
313
threds:
32
Elapsed time : 68641.921875 ms
mica@mica-X556UR ~/Desktop/Paralelos/Laboratorio/6to Laboratorio $ ./normthrea32 12000
blocks :
375
threds:
32
Elapsed time : 112952.718750 ms
```

- Ejecución con Bloques de thread de 32 x 32 y 32 y bloques de memoria de 32 x 32

```
mica@mica-X556UR ~/Desktop/Paralelos/Laborato
a32 8000
blocks :
250
threds:
32
Elapsed time : 11612.330078 ms
mica@mica-X556UR ~/Desktop/Paralelos/Laborato
a32 10000
blocks :
313
threds:
32
Elapsed time : 0.000000 ms
mica@mica-X556UR ~/Desktop/Paralelos/Laborato
a32 12000
blocks :
375
threds:
32
Elapsed time : 39569.781250 ms
```

1.3. Conclusiones

Del trabajo realizado se observó que existe una memoria de más rápido acceso , la memoria compartida, con la cual haciendo el respectivo análisis de tamaño y desempeño que se requiera realizar se puede reducir el tiempo de procesamiento. En el caso visto de multiplicación de matrices N/t , donde N es la dimensión de la matriz y el t la dimensión del bloque.

Para el ejemplo de $N=8000$, el número de accesos por cada posición $M[i][j]$ es 8000 por cada uno. pero aplicando los bloques de memoria compartida se reducen a 250 accesos a memoria Global.

Aunque no se utilizó en este trabajo existe también manera de asignar memoria compartida dinámicamente, ya que cada thread estaba encargado del calculo de una posición no era necesario y se limita la asignación a la manera estática y del tamaño mismo del BlockSize.