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

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 :

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 calculo 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 indices,

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

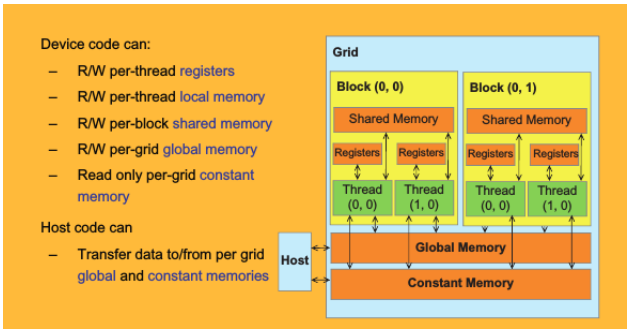


Figura 1.1: Diagrama de distribución de memoria

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.

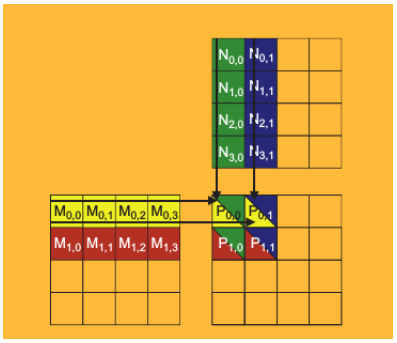


Figura 1.2: Representación de acceso en multiplicación de matrices

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
#include <fstream>
#include <iostream>
using namespace std;

#define TILE_WIDTH 16

void fillMatrix(int* a, int n)
{
   int i;
   for (i = 0; i < n*n; ++i)
        a[i] = 10;//rand()%5;
}
__global__
void matrixMulti(int *c, int *a, int *b,int n)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y ;
    int col = blockIdx.x * blockDim.x + threadIdx.x ;
    if ((row <n) && (col<n))
    {
    int suma=0;
        for(int i=0;i<n;++i)
        {
          suma+=a[row*n+i]*b[i*n+col];
        }
        c[row*n+col] = suma;
    }
}
__global__ void MatrixMulKernel(int * d_P, int * d_M, int* d_N,int
    Width)
{
  __shared__ int Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ int Nds[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x; int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
  // Identify the row and column of the d_P element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  int Pvalue = 0;
  // Loop over the d_M and d_N tiles required to compute d_P element
  for (int ph = 0; ph < Width/TILE_WIDTH; ++ph)
  {
    // Collaborative loading of d_M and d_N tiles into shared memory
    Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
    Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
    {
      Pvalue += Mds[ty][k] * Nds[k][tx];
    }
```

```
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<<
        endl; */
92
93    int blocks= (N + THREADS_PER_BLOCK -1)/THREADS_PER_BLOCK;
94    dim3 dimGrid(blocks, blocks, 1);
95    dim3 dimBlock(THREADS_PER_BLOCK,THREADS_PER_BLOCK, 1);
96    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 architectura 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 vale 8 bytes entonces nuestra memoria **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.

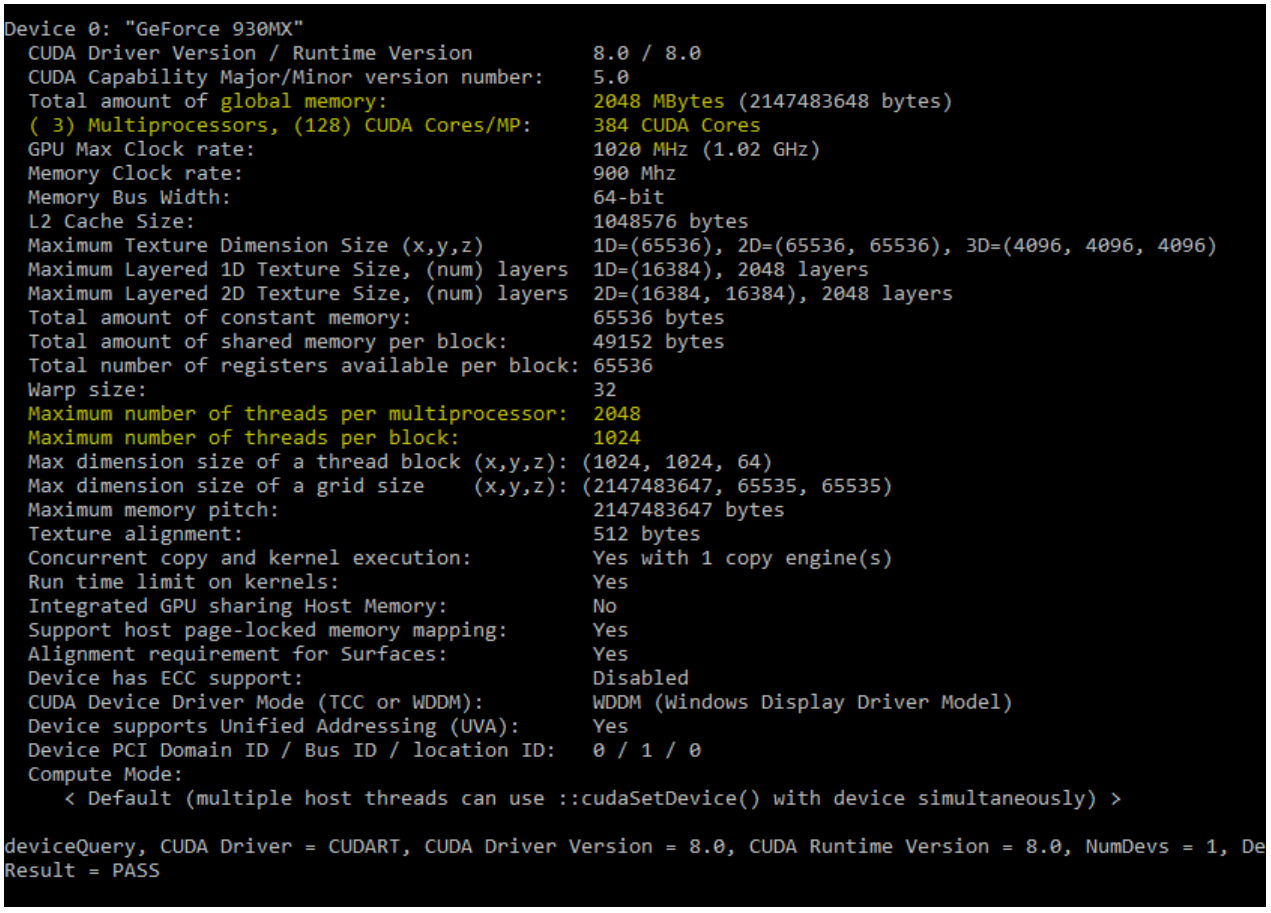


Figura 1.3: Carácteristicas del Dispositivo de Video

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

Siendo los resultados obtenidos:

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

Figura 1.4: Cuadro Comparativo usando BlockThreads 16x16

En la tabla anterior podemos ver la mejora de tiempo al usar la máxima capacidad de treads 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 :
625
 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 $
```

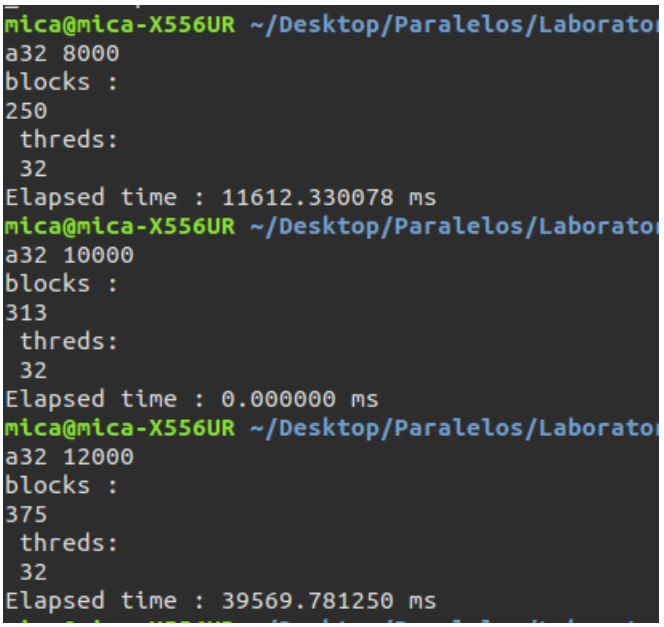
- **Ejecución Tiled Matriz Multiplicación de orden 10K y 12K**

```
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**



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 $\mathbf{N/t}$, donde N es la dimensión de la matriz y el $\mathbf{t}$ la dimensión del bloque.

Para el ejemplo de N=8000,e l numero 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 aasignacióna la manera estática y del tamaño mismo del BlockSize.