



INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO



Unidad de Aprendizaje: Analysis and Design of Parallel Algorithms.

“Reporte Práctica 3 - CUDA.”

Gabriela Moreno González.

Profa. Sandra Luz Morales Guitrón.

3CV9.

Programa 1. Multiplicación de matrices usando memoria compartida.

Primeramente, nos pide realizar la multiplicación de matrices empleando memoria compartida pero ahora también acompletar el código para poder realizarla, así nos da un código a la mitad donde le faltan algunas funciones y especificaciones para que pueda funcionar de la mejor manera posible, por lo que el código editado y corregido es el siguiente:

```
1  #include <stdio.h>
2
3  __global__ void Multiplica_Matrices_SM(float *C, float *A, float *B, int nfil, int ncol)
4  {
5      int bx = blockIdx.x;
6      int by = blockIdx.y;
7
8      int tx = threadIdx.x;
9      int ty = threadIdx.y;
10
11     int BLOCK_SIZE = 4;
12
13     int aBegin = ncol * BLOCK_SIZE * by;
14     int aEnd = aBegin + ncol - 1;
15     int aStep = BLOCK_SIZE;
16
17     int bBegin = BLOCK_SIZE * bx;
18     int bStep = BLOCK_SIZE * ncol;
19
20     float sum_sub = 0.0f;
21
22     for(int a = aBegin, b = bBegin ; a <= aEnd ; a+= aStep, b += bStep)
23     {
24         __shared__ float As[4][4];
25         __shared__ float Bs[4][4];
26
27         As[ty][tx] = A[a + ncol * ty + tx];
28         Bs[ty][tx] = B[b + ncol * ty + tx];
29
30         __syncthreads();
31
32         for(int k = 0 ; k < BLOCK_SIZE ; k++)
33         {
34             sum_sub += As[ty][k] * Bs[k][tx];
```

```

35     }
36
37     __syncthreads();
38 }
39
40 int c = ncol * BLOCK_SIZE * by + BLOCK_SIZE * bx;
41 C[c + ncol * ty + tx] = sum_sub;
42 }
43
44 int div_up(int a, int b)
45 {
46     if (a % b) /* does a divide b leaving a remainder? */
47         return a / b + 1; /* add in additional block */
48     else
49         return a / b; /* divides cleanly */
50 }
51
52 int main(void)
53 {
54     float *A_h,*B_h,*C_h;
55     float *A_d,*B_d,*C_d;
56     int nfil = 12;
57     int ncol = 12;
58     int BLOCK_SIZE = 4;
59     int N=nfil*ncol;
60
61     size_t size=N * sizeof(float);
62
63     A_h = (float *)malloc(size);
64     B_h = (float *)malloc(size);
65     C_h = (float *)malloc(size);
66
67     for(int i=0; i<nfil; i++)
68     {

```

```

69         for(int j=0;j<ncol;j++)
70         {
71             A_h[i*ncol+j] = 1.0f;
72             B_h[i*ncol+j] = 2.0f;
73         }
74     }
75
76     cudaMalloc((void **) &A_d, size);
77     cudaMalloc((void **) &B_d, size);
78     cudaMalloc((void **) &C_d, size);
79
80     cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
81     cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);
82
83     dim3 block_size(BLOCK_SIZE,BLOCK_SIZE);
84     dim3 n_blocks(div_up(ncol,block_size.x),div_up(nfil,block_size.y));
85
86     Multiplica_Matrices_SM<<< n_blocks, block_size >>> (C_d,A_d,B_d,nfil,ncol);
87
88     cudaMemcpy(C_h,C_d,size,cudaMemcpyDeviceToHost);
89
90     printf("\n\nMatriz c:\n");
91
92     for(int i=0; i<10; i++)
93     {
94         for(int j=0; j<10; j++)
95         {
96             printf("%.2f ", C_h[i*ncol+j]);
97         }
98         printf("\n");
99     }
100
101     free(A_h);
102     free(B_h);

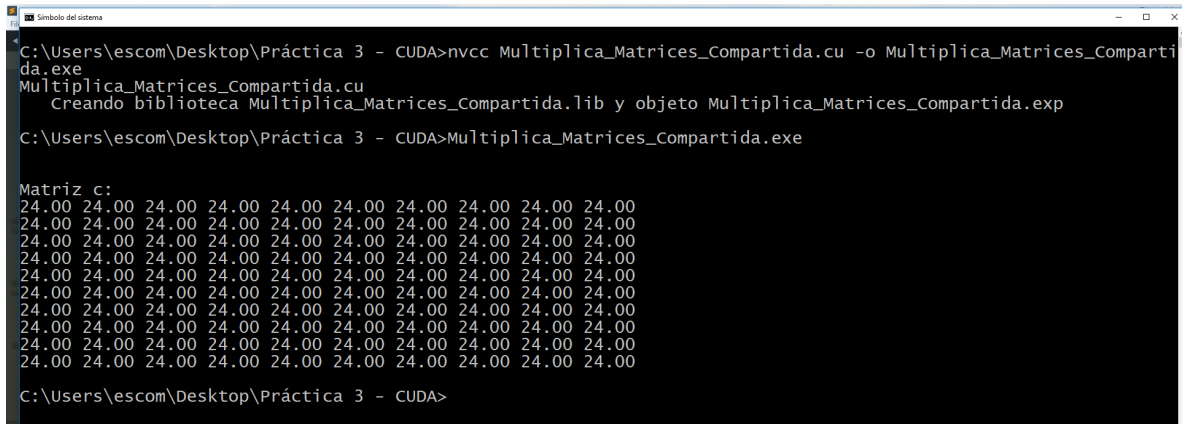
```

```

103     free(C_h);
104
105     cudaFree(A_d);
106     cudaFree(B_d);
107     cudaFree(C_d);
108
109     return(0);
110 }

```

Y, ejecutando el programa obtenemos lo siguiente:



```

C:\Users\escom\Desktop\Práctica 3 - CUDA>nvcc Multiplica_Matrices_Compartida.cu -o Multiplica_Matrices_Compartida.exe
Multiplica_Matrices_Compartida.cu
Creando biblioteca Multiplica_Matrices_Compartida.lib y objeto Multiplica_Matrices_Compartida.exp
C:\Users\escom\Desktop\Práctica 3 - CUDA>Multiplica_Matrices_Compartida.exe

Matriz c:
24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00
24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00
24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00
24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00
24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00
24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00
24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00
24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00
24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00
24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00 24.00
C:\Users\escom\Desktop\Práctica 3 - CUDA>

```

Lo cual nos indica que efectivamente el código se ejecutó de forma correcta con los valores elegidos.

Programa 2. Producto escalar empleando CUDA.

El siguiente programa realiza el producto escalar visto con MPI pero ahora empleando CUDA, el código de dicho programa es el siguiente:

```
1  #include <stdio.h>
2
3  __global__ void Suma_vectores(float *c, float *a, float *b, int N)
4  {
5      int idx = blockIdx.x * blockDim.x + threadIdx.x;
6
7      if(idx < N)
8      {
9          c[idx] = a[idx] + b[idx];
10     }
11 }
12
13 int main(void)
14 {
15     float *a_h, *b_h, *c_h;
16     float *a_d, *b_d, *c_d;
17     const int N = 24;
18
19     size_t size = N * sizeof(float);
20
21     a_h = (float *)malloc(size);
22     b_h = (float *)malloc(size);
23     c_h = (float *)malloc(size);
24
25     for(int i=0; i<N; i++)
26     {
27         a_h[i] = (float)i;
28         b_h[i] = (float)(i + 1);
29     }
30
31     printf("\nArreglo a:\n");
32
33     for(int i=0; i<N; i++)
34     {
```

```

35     printf("%f ", a_h[i]);
36 }
37
38 printf("\nArreglo b:\n");
39
40 for(int i=0; i<N; i++)
41 {
42     printf("%f ", b_h[i]);
43 }
44
45 cudaMalloc((void **) &a_d,size);
46 cudaMalloc((void **) &b_d,size);
47 cudaMalloc((void **) &c_d,size);
48
49 cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
50 cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
51
52 int block_size = 8;
53 int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
54
55 Suma_vectores <<< n_blocks, block_size >>> (c_d, a_d, b_d, N);
56
57 cudaMemcpy(c_h, c_d, size, cudaMemcpyDeviceToHost);
58
59 printf("\n\nArreglo c:\n");
60
61 for(int i=0; i<N; i++)
62 {
63     printf("%f ", c_h[i]);
64 }

```

```

65
66 printf("\n");
67 system("pause");
68
69 free(a_h);
70 free(b_h);
71 free(c_h);
72
73 cudaFree(a_d);
74 cudaFree(b_d);
75 cudaFree(c_d);
76
77 return(0);
78 }

```

Así, corriendo el programa nos arroja lo siguiente:

```

Arreglo a:
0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
9.000000 10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000
17.000000 18.000000 19.000000 20.000000 21.000000 22.000000 23.000000
Arreglo b:
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000 17.000000
18.000000 19.000000 20.000000 21.000000 22.000000 23.000000 24.000000
Arreglo c:
1.000000 3.000000 5.000000 7.000000 9.000000 11.000000 13.000000 15.000000 17.00
0000 19.000000 21.000000 23.000000 25.000000 27.000000 29.000000 31.000000 33.00
0000 35.000000 37.000000 39.000000 41.000000 43.000000 45.000000 47.000000
Presione una tecla para continuar . . .
C:\Users\alumno\Desktop\Práctica 2 - CUDA>

```

Programa 3. MergeSort.

El 3er programa realiza un ordenamiento de números dividiendo los conjuntos a la mitad y así hasta que queda un solo elemento y los compara, devuelve el conjunto pero ordenado. Así nuestro programa sería el siguiente:

```
1 #include <algorithm>
2 #include <vector>
3 #include "mpi.h"
4 #include <iostream>
5 using namespace std;
6
7 int main(int argc, char *argv[])
8 {
9     int rank, size, tama;
10    vector<int> Global;//Vector a ordenar
11    vector<int> *Local;//parte del vector
12
13    MPI_Init(&argc, &argv);//iniciamos el entorno MPI
14    MPI_Comm_rank(MPI_COMM_WORLD,&rank);//obtenemos el identificador del proceso
15    MPI_Comm_size(MPI_COMM_WORLD,&size);//obtenemos el numero de procesos
16
17    if( size % 2 != 0 ){//El numero de procesos deberia de ser par para aplicar este algoritmo
18        cout << "El numero de procesos debe ser par" << endl;
19        MPI_Abort(MPI_COMM_WORLD,1);//abandonamos la ejecucion.
20    }
21
22    if(argc < 2){
23        if(rank == 0)
24            cout<< "No recibí parametro con el tamaño de vector, por defecto sera 1000"<<endl;
25        tama = 1000;
26    }else{
27        tama = atoi(argv[1]);
28    }
29
30    if(rank == 0){//el proceso 0 genera un vector desordenado.
31        for(int i = 0; i < tama;++i){
32            Global.push_back(rand()%1000);
33        }
34    }
```

```
35    Local = new vector<int>(tama/size);// reservamos espacio para el vector local a cada proceso.
36
37    //Repartimos el vector entre todos los procesos.
38
39    MPI_Scatter(&Global[0],tama/size,MPI_INT,&((*Local)[0]),tama/size,MPI_INT,0,MPI_COMM_WORLD);
40
41    //Cada proceso ordena su parte.
42    sort(Local->begin(),Local->end());
43
44    vector<int> *ordenado;
45    MPI_Status status;
46    int paso = 1;
47
48    //Ahora comienza el proceso de mezcla.
49    while(paso<size)
50    {
51        // Cada pareja de procesos
52        if(rank%(2*paso)==0) // El izquierdo recibe el vector y mezcla
53        {
54            if(rank+paso<size)// los procesos sin pareja esperan.
55            {
56                vector<int> localVecino(Local->size());
57                ordenado = new vector<int>(Local->size()*2);
58
59                MPI_Recv(&localVecino[0],localVecino.size(),MPI_INT,rank+paso,0,MPI_COMM_WORLD,&status);
60                merge(
61                    Local->begin(),Local->end(),localVecino.begin(),localVecino.end(),ordenado->begin() );
62
63                delete Local;
64                Local = ordenado;
65                ordenado = NULL;
66            }
67        }
68    }
```

```

69     else // El derecho envia su vector ordenado y termina
70     {
71         int vecino = rank-paso;
72         MPI_Send(&(*Local)[0], Local->size(), MPI_INT, vecino, 0, MPI_COMM_WORLD);
73         break; // Sale del bucle
74     }
75     paso = paso*2; // el paso se duplica ya que el numero de procesos se reduce a la mitad.
76 }
77
78 if(rank == 0){
79     cout<<endl<<" ";
80     for(unsigned int i = 0; i<Local->size(); ++i){
81         cout<< (*Local)[i]<<" ";
82     }
83     cout<<"]"<<endl;
84 }
85
86 MPI_Finalize();
87 return 0;
88 }

```

Y corriendo el programa podemos observar la siguiente salida:

```

MacBook-Pro-de-Gabriela:Desktop gabriela$ mpirun -np 8 mergesort 100
[1, 13, 24, 42, 63, 73, 91, 94, 97, 99, 105, 115, 123, 124, 149,
153, 157, 165, 169, 188, 194, 195, 196, 228, 228, 249, 267, 272, 278,
298, 298, 303, 327, 335, 336, 357, 393, 404, 408, 425, 440, 451,
485, 490, 492, 501, 503, 505, 512, 517, 530, 536, 544, 549, 560,
566, 579, 612, 629, 633, 635, 658, 666, 668, 669, 672, 708, 709, 709,
722, 729, 745, 752, 801, 807, 810, 814, 816, 821, 826, 840, 853,
865, 878, 882, 903, 923, 930, 933, 933, 944, 967, 977, 979, 981,
987, ]
MacBook-Pro-de-Gabriela:Desktop gabriela$

```

Conclusiones:

Usar CUDA de diversas maneras es muy similar a sistemas operativos, donde puedes meter todo en un solo hilo o bien empezar a crear procesos para poder tener tareas en paralelo y tener un mayor rendimiento de tu equipo de cómputo, lo mismo sucede con CUDA, podemos compartir la memoria (algo similar que sucedía con MPI), pero de todas maneras sería muy similar, además de que también podemos elegir a que parte de los procesadores les vamos a mandar qué tarea.