UNIVERSIDAD NACIONAL DE SAN AGUSTIN

Algoritmos Paralelos– 14 de julio de 2017

Informe CUDA multiplication tiled with thread granularity

Ciencia de la Computación

Integrantes:
Felix Oliver Sumari Huayta

14 de julio de 2017

Índice

1. Multiplicación with Thread Granularity

En esta seccion vamos a ver la version de la multiplicación de matrices con tile pero con reduccion de redundancia, para ello se utiliza la granularidad de hilos, esto es ventajoso cuando algunos datos se vuelven redundante como en el cosa de una multiplicación de matrices , la primera fila se usa varias veces para multiplicar con cada columna, si nosotros redusimos los hilos usados , obtendremos mejor rendimiento en el paralelismo.

A continuación el codigo del granulado de threads, redundancia de 4 , por thread ejecutaremos una fila de la matriz **A** y 4 columnas de la matriz **B** por thread de ese modo reduciremos la redundancia de threads.

Declaramos la funcion $mult\_matrix\_tile_4$ como una funcion kernel que se ejecutara en memoria global con sus respectivos parametros.

```
__global__ void mult_matrix_tile_4(int a[], int b[], int c[],int fila
   ,int columna){
```

Declaramos los tile que ejecutaremos por thread, a_ds representara la fila de la matriz A y $b\_ds\_0, b\_ds\_1, b\_ds\_2, b\_ds\_3$ seran las columnas de la matriz b , todas estas declaraciones en memoria compartida.

```
                __shared__ int a_ds[tile_width][tile_width];
        __shared__ int b_ds_0[tile_width][tile_width];
        __shared__ int b_ds_1[tile_width][tile_width];
        __shared__ int b_ds_2[tile_width][tile_width];
        __shared__ int b_ds_3[tile_width][tile_width];
```

Asignamos los indices a cada variable para que mas adelante sean llamados y se pueda efectuar el producto punto. creamos una varaible row y 4 variables col.

```
        const uint bx = blockIdx.x;
        const uint by=blockIdx.y;

        const uint tx = threadIdx.x;
        const uint ty=threadIdx.y;

        const uint row = by * tile_width + ty;

        const uint col_0 = (4*bx+0 )* tile_width + tx;
        const uint col_1 = (4*bx+1 )* tile_width + tx;
        const uint col_2 = (4*bx+2 )* tile_width + tx;
        const uint col_3 = (4*bx+3 )* tile_width + tx;
```

En esta parte hacemos una sentencia for para copiar los valores de las matrizes a los tile declarado en memoria compartida.

```

```

```
        float Pvalue_0 = 0.0f, Pvalue_1 = 0.0f, Pvalue_2 = 0.0f,
           Pvalue_3 = 0.0f;
        for(uint ph=0;ph<fila/tile_width;++ph)
        {
                a_ds[ty][tx] = a[row*fila+ ph*tile_width + tx];

                b_ds_0[ty][tx] = b[(ph*tile_width+ty)*fila+col_0];
                b_ds_1[ty][tx] = b[(ph*tile_width+ty)*fila+col_1];
                b_ds_2[ty][tx] = b[(ph*tile_width+ty)*fila+col_2];
                b_ds_3[ty][tx] = b[(ph*tile_width+ty)*fila+col_3];

                __syncthreads();
```

En esta sección vemos otra sentencia for, para calculara el producto punto de los valores declarados en memoria compartida por tile y se guardara en 4 valores.

```
                for(uint k=0;k<tile_width;++k)
                {
                        Pvalue_0 += a_ds[ty][k]*b_ds_0[k][tx];
                        Pvalue_1 += a_ds[ty][k]*b_ds_1[k][tx];
                        Pvalue_2 += a_ds[ty][k]*b_ds_2[k][tx];
                        Pvalue_3 += a_ds[ty][k]*b_ds_3[k][tx];
                }
                __syncthreads();
        }
```

Aqui solo asignamos los resultados obtenidos en la matriz C

```
        c[row*fila+col_0] = Pvalue_0;
        c[row*fila+col_1] = Pvalue_1;
        c[row*fila+col_2] = Pvalue_2;
        c[row*fila+col_3] = Pvalue_3;
}
```

2. Ejecución

	32 serial	32 tile	16 serial	16 tile	16 granularity
1024 x 1024	52.402176	45.370689	60.844833	43.435585	44.406654
2048 x 2048	281.318604	191.982590	230.882599	191.665436	177.246109
8192 x 8192				4019.258789	3253.973145

Cuadro 1: Ejecución.

Los resultados obtenidos se encuentran en el $Cuadro1$, la granularidad ha sido probado con $tile\ width = 16$, le hemos asignado este valor porque por thread ejecutaremos 4 operaciones

de esta manera aprovechamos mejor la granularidad a comparacion de 2 operaciones por thread.

Este es el codigo donde llamamos a la función kernel:

```
1
2   int main(int argc, char * argv[]){
3
4         int *a, *b, *c;
5         int *d_a, *d_b, *d_c;
6
7
8         int size = m*n*sizeof(int);
9
10        //separar espacion de memoria para copias en device
11
12        float gpu_elapsed_time_ms, cpu_elapsed_time_ms;
13
14        // some events to count the execution time
15        cudaEvent_t start, stop;
16        cudaEventCreate(&start);
17        cudaEventCreate(&stop);
18
19        // start to count execution time of GPU version
20        cudaEventRecord(start, 0);
21
22        cudaMalloc((void **)&d_a, size);
23        cudaMalloc((void **)&d_b, size);
24        cudaMalloc((void **)&d_c, size);
25        //separa espacio en host
26
27        a = (int *)malloc(size);
28        llenar_random_matrix(a,m,n);
29        b = (int *)malloc(size);
30        llenar_random_matrix(b,m,n);
31        c = (int *)malloc(size);
32        // imprimir matrices a , b
33
34        print_matrix(a, m, n);
35        printf("----------------\n");
36        print_matrix(b, m, n);
37        printf("----------------\n");
38        // copias entrada a device
39
40        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
41        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
42
43        // lanzar funcion kernel en gpu con N blocks
44
45
```

```
46          unsigned int grid_rows = (m)/tile_width;
47          unsigned int grid_cols = (n)/tile_width;
48
49          printf("g_row: %d\n",grid_rows);
50          printf("g_col: %d\n",grid_cols);
51
52          dim3 dimGrid(grid_cols, grid_rows);
53          dim3 dimBlock(tile_width,tile_width);
54          dimGrid.x/=4; // las columnas lo dividimos entre 4 , porque
               por fila seran 4 operaciones de columna
55          mult_matrix_tile_4<<<dimGrid,dimBlock>>>(d_a,d_b,d_c,m,n);
56
57
58          // copia resultado al host
59          cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
60
61          cudaEventRecord(stop, 0);
62          cudaEventSynchronize(stop);
63
64          // compute time elapse on GPU computing
65          cudaEventElapsedTime(&gpu_elapsed_time_ms, start, stop);
66          printf("Time elapsed on matrix multiplication of GPU: %f : \n
               ",  gpu_elapsed_time_ms);
67
68          print_matrix(c,m,n);
69          // limpiar memoria
70          free(a); free(b); free(c);
71          cudaFree(d_a); cudaFree(d_b);cudaFree(d_c);
72          return 0;
73  }
```