UNIVERSIDAD NACIONAL DE SAN AGUSTIN

Algoritmos Paralelos– 3 de julio de 2017

Informe CUDA multiplication tiled

Ciencia de la Computación

Integrantes:
Felix Oliver Sumari Huayta

3 de julio de 2017

Índice

1. Multiplicacion Normal **3**

2. Multiplicacion con tiled **3**

3. Ejecucion **4**

1. Multiplicacion Normal

La multiplicación de matrices es un componente importante de la Basic Linear Álgebra Subprogramas (BLAS). Esta función es la base de muchos solucionadores de álgebra lineal tales como la descomposición de LU. Como veremos, la multiplicación matricial presenta oportunidades para la reducción de accesos a la memoria global que pueden ser capturados con técnicas relativamente simples. La velocidad de ejecución de las funciones de multiplicación de matriz puede variar en órdenes de magnitud, dependiendo del nivel de reducción de los accesos de memoria global. Por lo tanto, la multiplicación matricial proporciona un excelente ejemplo inicial para tales técnicas.

Time elapsed on matrix multiplication of GPU: 228.406723 milisegundos

```
1
2     __global__ void mult_matrix(int a[],int b[], int c[],int fila,int
          columna)
3   {
4           int row = blockIdx.y*blockDim.y+threadIdx.y;
5           int col = blockIdx.x*blockDim.x+threadIdx.x;
6
7           if((row<fila)&&(col<columna))
8           {
9                   int Pvalue = 0;
10                  for(int k=0;k<fila;++k)
11                  {
12                          Pvalue += a[row*fila+k]*b[k*columna+col];
13                  }
14                  c[row*fila+col] = Pvalue;
15          }
16  }
```

2. Multiplicacion con tiled

Ahora estamos listos para presentar un núcleo de multiplicación de matriz de mosaico que utiliza memoria compartida para reducir el tráfico a la memoria global. Mds y Nds como variables de memoria compartida. Recuerde que el ámbito de las variables de memoria compartida es un bloque. De este modo, se creará un par de Mds y Nds para cada bloque, y todos los hilos de un bloque podrán acceder a los mismos Mds y Nds. Esto es importante ya que todos los subprocesos de un bloque deben tener acceso a los elementos M y N cargados en Mds y Nds por sus pares para que puedan usar estos valores para satisfacer sus necesidades de entrada.

Time elapsed on matrix multiplication of GPU: 183.590912 milisegundos

```
1
2    __global__ void mult_matrix_tile(int a[], int b[], int c[],int fila
         ,int columna){
```

```
        __shared__ int a_ds[tile_width][tile_width];
        __shared__ int b_ds[tile_width][tile_width];

        int bx = blockIdx.x; int by=blockIdx.y;
        int tx = threadIdx.x; int ty=threadIdx.y;

        int row = by * tile_width + ty;
        int col = bx * tile_width + tx;

        float Pvalue = 0;
        for(int ph=0;ph<fila/tile_width;++ph)
        {
                a_ds[ty][tx] = a[row*fila+ ph*tile_width + tx];
                b_ds[ty][tx] = b[(ph*tile_width+ty)*fila+col];
                __syncthreads();

                for(int k=0;k<tile_width;++k)
                {
                        Pvalue += a_ds[ty][k]*b_ds[k][tx];
                }
                __syncthreads();
        }
        c[row*fila+col] = Pvalue;
}
```

3. Ejecucion

	32 serial	32 tile	16 serial	16 tile
1024 x 1024	52.402176	45.370689	60.844833	43.435585
2048 x 2048	281.318604	191.982590	230.882599	191.665436

Cuadro 1: Ejecución.

```

  int main(int argc, char * argv[]){

        int *a, *b, *c;
        int *d_a, *d_b, *d_c;


        int size = m*n*sizeof(int);

        //separar espacion de memoria para copias en device

```

```
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("m+tile -1: %d\n",m+tile_width-1);
50          printf("g_row: %d\n",grid_rows);
51          printf("g_col: %d\n",grid_cols);
52
53          dim3 dimGrid(grid_cols, grid_rows);
54          dim3 dimBlock(tile_width,tile_width);
55          mult_matrix_tile<<<dimGrid,dimBlock>>>(d_a,d_b,d_c,m,n);
56          //mult_matrix<<<dimGrid,dimBlock>>>(d_a,d_b,d_c,m,n);
57          // add_3<<<n,1>>>(d_a,d_b,d_c,m,n);
58
59          // copia resultado al host
60          cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

```
61          //imprimir c result
62
63          cudaEventRecord(stop, 0);
64          cudaEventSynchronize(stop);
65
66          // compute time elapse on GPU computing
67          cudaEventElapsedTime(&gpu_elapsed_time_ms, start, stop);
68          printf("Time elapsed on matrix multiplication of GPU: %f : \n
               ",  gpu_elapsed_time_ms);
69
70          print_matrix(c,m,n);
71          // limpiar memoria
72          free(a); free(b); free(c);
73          cudaFree(d_a); cudaFree(d_b);cudaFree(d_c);
74          return 0;
75  }
```