Multiplicación de matrices en GPU

Gabriel Aparicio Tony

July 3, 2017

1 Introducción

Como algoritmo, la multiplicación de matrices resulta una operación costosa para realizar en la CPU, sobre todo teniendo en cuenta que forma parte de algoritmos más complejos. Supone muchos accesos a memoria (la mayoría duplicados). Además es un algoritmo muy uniforme en el sentido que siempre realiza el mismo cómputo sobre distintos datos. Estos factores hacen que se ajuste al modelo SIMD de la GPU. En el presente trabajo se realizan pruebas con los algoritmos estudiados en clase. Para determinar de forma precisa la diferencia entre ejecutar la versión serial del algoritmo de multiplicación de matrices y su contraparte paralela en GPU. Además de esto, se muestra los efectos de identificar y acceder de forma eficiente a los distintos niveles en la jerarquía de memoria. Específicamente se muestra la diferencia de desempeño entre utilizar un algoritmo que accede de forma indiscriminada a la memoria global y otro que a través del uso de la memoria compartida que tiene cada bloque optimiza la manera en que se utilizan los datos y por ende el tiempo de ejecución.

2 Código fuente

2.1 Kernel - Multiplicación normal

```
__global__
void mult_matrix(int* a, int* b, int* c,int n)
{
        int col = blockDim.x*blockIdx.x+ threadIdx.x;
        int row = blockDim.y*blockIdx.y+ threadIdx.y;


        if ( col<n && row<n )
        {
                int i;
                c[row*n+col] = 0;

                for(i=0;i<n;i++)
                {
                        c[row*n + col] += a[ row*n + i ]*b[ i*n +
                            col ];

                }

        }
}
```

2.2 Kernel - Multiplicación usando memoria compartida

```
__global__
void mult_matrix_shared(int* a, int* b, int* c,int n)
{
        __shared__ float sub_a[THREAD_PER_BLOCK][THREAD_PER_BLOCK];
        __shared__ float sub_b[THREAD_PER_BLOCK][THREAD_PER_BLOCK];

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

        int Row = by * THREAD_PER_BLOCK + ty;
        int Col = bx * THREAD_PER_BLOCK + tx;

        int Pvalue = 0;

        for (int ph = 0; ph < n/THREAD_PER_BLOCK; ++ph) {

                sub_a[ty][tx] = a[Row*n + ph*THREAD_PER_BLOCK + tx
                   ];
                sub_b[ty][tx] = b[(ph*THREAD_PER_BLOCK + ty)*n +
                   Col];
                __syncthreads();

                for (int k = 0; k < THREAD_PER_BLOCK; ++k) {
                        Pvalue += sub_a[ty][k] * sub_b[k][tx];
                }
                __syncthreads();
        }
        c[Row*n + Col] = Pvalue;
}
```

2.3 Código para medir el tiempo de ejecución en GPU

```
int main()
{
        int *a,*b,*c;
        int *d_a,*d_b,*d_c;

        int mat_elem = 800;
        int my_size = mat_elem*mat_elem*sizeof(int);

        cudaEvent_t my_start,my_stop;
        cudaEventCreate(&my_start);
        cudaEventCreate(&my_stop);

        a = (int*) malloc(my_size);
        b = (int*) malloc(my_size);
        c = (int*) malloc(my_size);

        fill_mat(a,mat_elem);
        fill_mat(b,mat_elem);

        cudaMalloc((void**)&d_a,my_size);
        cudaMalloc((void**)&d_b,my_size);
        cudaMalloc((void**)&d_c,my_size);
```

```
        cudaMemcpy(d_a,a,my_size,cudaMemcpyHostToDevice);
        cudaMemcpy(d_b,b,my_size,cudaMemcpyHostToDevice);

        dim3 my_block(THREAD_PER_BLOCK,THREAD_PER_BLOCK);
        dim3 my_grid((mat_elem + THREAD_PER_BLOCK-1)/my_block.x,(
            mat_elem + THREAD_PER_BLOCK-1)/my_block.y);

    cudaEventRecord(my_start,0);
    mult_matrix<<<my_grid,my_block>>>(d_a, d_b, d_c,mat_elem);
    //mult_matrix_shared<<<my_grid,my_block>>>(d_a, d_b,d_c,
        mat_elem);
    cudaEventRecord(my_stop,0);
    cudaEventSynchronize(my_stop);

    float elapsed_time;
    cudaEventElapsedTime(&elapsed_time,my_start,my_stop);
        cudaMemcpy(c,d_c,my_size,cudaMemcpyDeviceToHost);
        printf("Execution time: %f\n",elapsed_time);
        return 0;
}
```

2.4 Multiplicación serial

```
void mult_matrix(int m1[][m_size],int m2[][m_size],int r[][m_size],
    int n)
{
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            r[i][j]=0;
            for(int k=0;k<n;k++)
            {
                r[i][j]+=m1[i][k]*m2[k][j];
            }
        }
    }
}
```

2.5 Código para medir el tiempo de ejecución en CPU

```
int main()
{
        int m1[m_size][m_size];
    fill_matrix(m1,m_size,2);
    int m2[m_size][m_size];
    fill_matrix(m2,m_size,2);
    int r[m_size][m_size];
    fill_matrix(r,m_size,0);

    auto _start = chrono::system_clock::now();
    mult_matrix(m1,m2,r,m_size);
    auto _end = chrono::system_clock::now();
```

```
    auto elapsed = chrono::duration_cast<chrono::milliseconds>(_end
        - _start);
    cout<<"Execution␣time:␣"<<elapsed.count()<<endl;
        return 0;
}
```

Algoritmo	400	600	800
Serial	400	832	2826
GPU	12.75	37.99	99.69

Table 1: Comparación del tiempo de ejecución serial vs GPU. Tiempo en milisegundos

N	$TILE = 20$	$TILE = 32$	$TILE = 70$
400	3.52	0.002432	0.002432
600	11.4	0.002432	0.002432
800	26.64	0.002432	0.002432

Table 2: Tiempo de ejecución en milisegundos de la multiplicación en GPU utilizando memoria compartida

3 Experimentos

En esta sección se muestra los experimentos hechos con matrices cuadradas NxN para 3 valores distintos de N, comparando el desempeño de los algoritmos de multiplicación en serial y en paralelo. El tiempo se muestra en milisegundos.

En la tabla 1 se muestra la diferencia básica y evidente entre la ejecución del algoritmo serial y la ejecución en GPU, con la salvedad de que esta versión no optimiza los accesos a memoria y realiza todos sus cálculos utilizando únicamente la memoria global.

En la tabla 2 se realizan experimentos con la versión de la multiplicación de matrices que utiliza la memoria compartida, se consideran 3 tamaños distintos de TILE. Es así que se puede observar la ventaja contundente de utilizar esta versión y a su vez identificar que la determinación del tamaño de TILE correcto afecta considerablemente el desempeño del algoritmo; nótese que la variación del tiempo de ejecución al utilizar un tamaño de TILE de 50 y 70 en lugar de 20 es notoria.

4 Conclusiones

En el presente trabajo se ha mostrado que para obtener mejores tiempos de ejecución de un algoritmo no es suficiente implementar su versión paralela sin mayor consideración. Es importante conocer el sistema y jerarquía de la memoria para determinar el uso correcto de la misma de acuerdo a la circunstancia, problema o aplicación en cuestión. Se utilizó un algoritmo clásico y ampliamente aplicado en distintos dominios, la multiplicación de matrices, para demostrar los efectos evidentes de hacer un correcto uso de los mecanismos que nos brinda la arquitectura GPU, específicamente la memoria compartida que posee cada bloque. Además también se mostró que para obtener el máximo beneficio del uso memoria compartida el tamaño de TILE que se considera es de crítica importancia.