Universidad Nacional de San Agustín

Facultad de Producción y Servicios

Escuela Profesional de Ciencia de la Computación

Curso :

Algoritmos Paralelos

Tema :

Multiplicación de Matrices usando CUDA - Tiled Matrix Multiplication Kernel

Docente :

Mgter. Alvaro Henry Mamani Aliaga

Alumno :

Vidal Antonio Soncco Merma

AREQUIPA - PERÚ

2017

1 Multiplicación de Matrices usando CUDA

La multiplicación de la matrices, entre una matriz M $i \times j$ (i filas por j columnas) y una matriz N $j \times k$ produce una matriz P $i \times k$. Esta función es la base de muchos solucionadores de álgebra lineal tales como la descomposición LU. 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.

Cuando se realiza una multiplicación matricial, cada elemento de la matriz de salida P es un producto interno de una fila de M y una columna de N. Continuaremos usando la convención donde $P_{Row,Col}$ es el elemento en la posición Row^{th} en la dirección vertical y la posición Col^{th} en la dirección horizontal. El producto interior, también denominado producto punto, de dos vectores es la suma de productos de los elementos vectoriales individuales, es decir, $P_{Row,Col} = \sum M_{Row,k} \times N_{k,Col}, \text{for} k = 0, 1, \ldots \text{Width} - 1$.

Código del kernel de multiplicación de matrices usando un hilo para calcular un elemento P.

```
//Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float *d_M , float *d_N , float *d_P , int Width) {
    //2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    //Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0.0;

    for(int k = 0; k < Width ; ++k) {
        float Mdelement = d_M[ty*Width + k];
        float Ndelement = d_N[k*Width + tx];
        Pvalue += (Mdelement*Ndelement);
    }

    d_P[ty*Width + tx] = Pvalue;
}
```

2 Kernel de Multiplicación de Matrices mediante Tiling

Presentamos un kernel de multiplicación de matrices por mosaicos(tiled) que utiliza memoria compartida para reducir el tráfico a la memoria global. El kernel implementa las fases ilustradas en la Figuras de abajo, se declaran Mds y Nds como variables de memoria compartida. Recuerdar 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.

Se guardan los valores de $threadIdx$ y $blockIdx$ en variables automáticas y, por tanto, en registros para un acceso rápido. Recuerde que las variables escalares automáticas se colocan en los registros. Su alcance está en cada hilo individual; Es decir, una versión privada de tx, ty, bx y by es creada por el sistema en tiempo de ejecución para cada subproceso y residirá en los registros que son accesibles por el subproceso. Se inicializan con los valores de $threadIdx$ y $blockIdx$ y se utilizan muchas veces durante la duración del subproceso. Una vez que el hilo termina, los valores de estas variables dejan de existir.

Luego de determinan los índices de fila y columna del elemento P que ha de producir el hilo. El código asume que cada hilo es responsable de calcular un elemento P. La posición horizontal

(x), o el índice de columna del elemento P que se producirá por un hilo, se puede calcular como $bx \times TILE\_WIDTH + tx$ porque cada bloque cubre elementos $TILE\_WIDTH$ en la dimensión horizontal. Un hilo en el bloque bx tendría bx bloques de hilos, o ($bx \times TILE\_WIDTH$) hilos, antes de él; Cubren $bx \times TILE\_WIDTH$ elementos de P. Otros hilos del tx dentro del mismo bloque cubrirían otros elementos del tx. Por lo tanto, el hilo con bx y tx debe ser responsable de calcular el elemento P cuyo índice x es $bx \times TILE\_WIDTH + tx$.

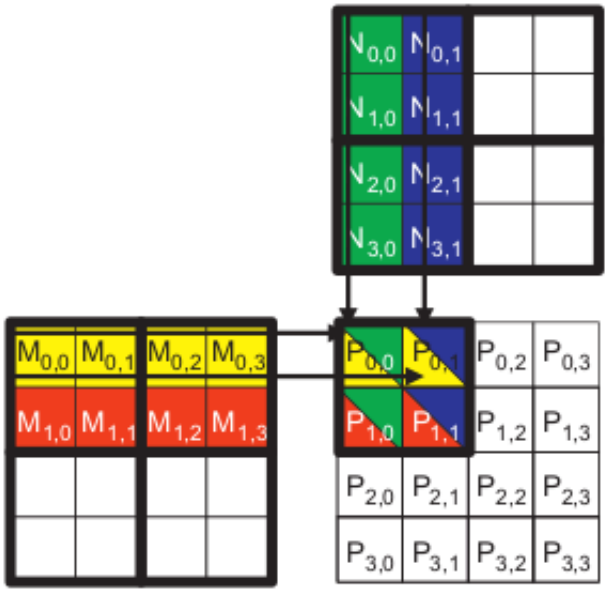

Figure 1: Tiling M y N para utilizar memoria compartida.

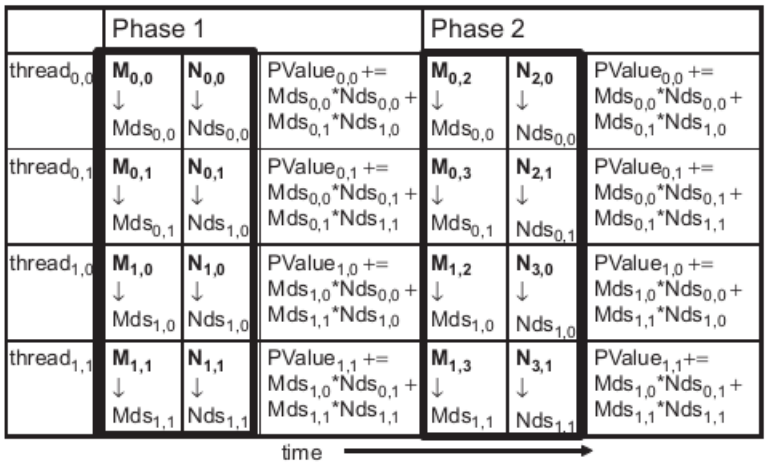

Figure 2: Fases de ejecución de una multiplicación matricial.

Código del kernel de multiplicación de matrices de mosaico (tiling) con memoria compartida.

```
__global__
void MatrixMulKernel(float *d_M , float *d_N , float *d_P , int Width) {
        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
        __shared__ float 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;

        float Pvalue = 0;
        // Loop over the d_M and d_N tiles required to compute d_P element
        // ph indicate number of phase
        for (int ph = 0; ph < ceil(Width/(float)TILE_WIDTH); ++ph) {

                // Collaborative loading of d_M and d_N tiles into shared memory
                if ((Row < Width) && ((ph*TILE_WIDTH + tx) < Width))
                        Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
                else Mds[ty][tx] = 0.0;
                if (((ph*TILE_WIDTH + ty) < Width) && (Col < Width))
                        Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
                else Nds[ty][tx] = 0.0;

                __syncthreads(); // for syncronizeing the threads

                for (int k = 0; k < TILE_WIDTH; ++k) {
                        Pvalue += Mds[ty][k] * Nds[k][tx];
                }
                __syncthreads(); // for syncronizeing the threads
        }
        if ((Row < Width) && (Col < Width))
                d_P[Row*Width + Col] = Pvalue;
}
```

3 Tiempos de ejecución

Tiempos de ejecución (en segundos) del Programas en una computadora con las siguientes características:

```
Device 0: "GeForce GT 620"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 963 MBytes (1010040832 bytes)
  ( 1) Multiprocessors, ( 48) CUDA Cores/MP:     48 CUDA Cores
  GPU Max Clock rate:                            1620 MHz (1.62 GHz)
  Memory Clock rate:                             897 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 65536 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 20
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
```

```
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Disabled
Device supports Unified Addressing (UVA):      Yes
Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
```

Hilos	Simple	Tiling
2^2	0.256128	5.307552
2^4	0.256544	5.830176
2^8	0.231744	15.408320
2^{16}	0.229088	0.000000
2^{32}	0.001280	0.001280
2^{64}	0.001440	0.001280

Table 1: Comparación de los tiempos de ejecución

References

[1] http://rubmarin.galeon.com/sisd.htm

[2] http://www.academia.edu/18404632/Taxonomías_de_Flynn

[3] https://es.slideshare.net/cramex/programacin-paralela-conceptos-y-diseo-de-sistemas-distribuido