Universidad Nacional de San Agustin

Escuela Profesional de
Ciencia de la Computación

Algoritmos Paralelos

CUDA : Optimización de la Multiplicación de Matrices

Alumna :
Rosa Yuliana Gabriela
Paccotacya Yanque
Profesor:
Mg.Alvaro Henry Mamani
Aliaga

Índice

1. Multiplicación de matrices con tiles optimizado

1.1. Multiplicación de matrices con tiles optimizado

En la multiplicación de matrices con tiles simple, al momento de generar los productos se cargaban en memoria compartida, los mismos bloques, generandose asi redundancia. Para evitar esto se crea un tile adicional en memoria compartida, de manera que calculemos tambien el valor del producto del bloque correlativo como se observa en la Figura 1.

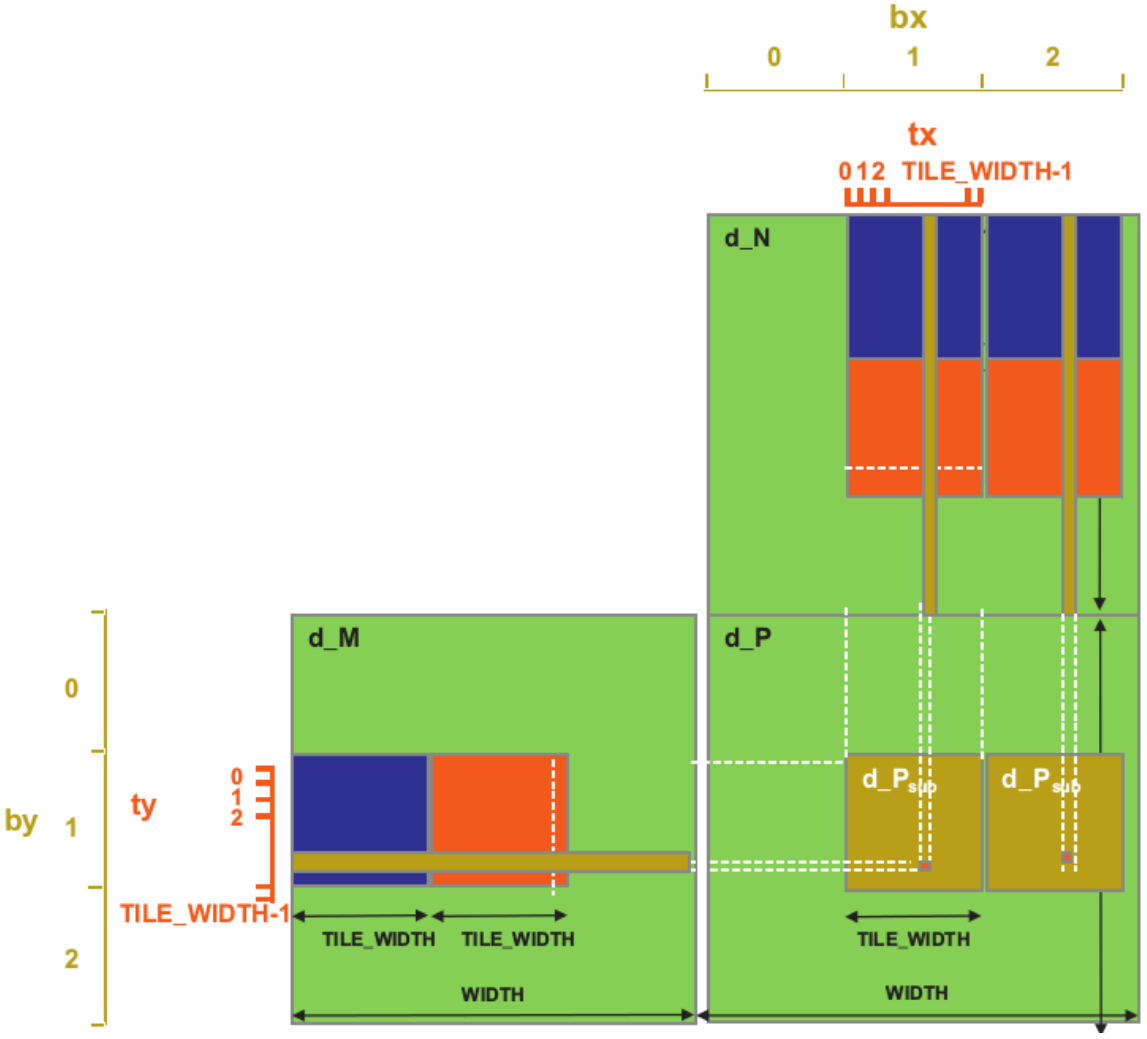

Figura 1: Multiplicación de matrices con tiles incrementando thread granularity

Para esto se modifica el código de manera que podamos aprovechar los bloques ya cargados en memoria, haciendo la multiplicación para dos elementos de la matriz resultante, por lo que tendriamos un grid de n/2 x n bloques(tener en cuenta al llamr al kernel).

1.2. Código

```
__global__ void MatrixMulTiledMod(int * d_P, int * d_M, int* d_N,int Width)
{
  __shared__ int Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ int Nds[TILE_WIDTH][TILE_WIDTH];
  __shared__ int Nds2[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*2 + tx;
  int Pvalue =0 , Pvalue2=0;
  Mds[ty][tx]=0;
  Nds[ty][tx]=0;
  Nds2[ty][tx]=0;
  __syncthreads();

  // Loop over the d_M and d_N tiles required to compute d_P element
  if((Row < Width) && (Col < Width)){
    for (int ph = 0; ph <Width/TILE_WIDTH; ph++)
    {
      // Collaborative loading of d_M and d_N tiles into shared memory
      //printf("%i - %i -%i \n",ph, Row, Col );
      if ((Row< Width) && (ph*TILE_WIDTH+tx)< Width)
        Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
      if ((ph*TILE_WIDTH+ty)<Width && Col<Width)
        Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
      //printf("%i %i\n",(ph*TILE_WIDTH+ty),Col+TILE_WIDTH);
      if (((ph*TILE_WIDTH + ty)*Width + Col+TILE_WIDTH)<(Width*Width))
      {
        Nds2[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col+TILE_WIDTH];
      }
      __syncthreads();
      for (int k = 0; k < TILE_WIDTH; k++)
      {
        Pvalue += Mds[ty][k] * Nds[k][tx];
        Pvalue2 += Mds[ty][k] * Nds2[k][tx];
      }
      __syncthreads();
    }

    d_P[Row*Width + Col] = Pvalue;
    d_P[Row*Width + Col +TILE_WIDTH] = Pvalue2;
  }
}
```

Código 1: Multiplicación de Matrices

1.3. Ejemplo

```
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ nvprof ./mod3 8
==22872== NVPROF is profiling process 22872, command: ./mod3 8
N: 8    Bloques : 3      Hebras/Bloque: 3
Tiempo  : 0.054464 ms
------A------------
4 5 4 2 5 2 5 4
5 3 2 1 5 2 3 4
1 4 5 1 1 1 3 4
3 2 5 5 2 5 1 2
5 1 4 4 5 3 4 4
2 1 1 4 2 4 2 5
2 4 2 2 4 5 3 3
3 2 5 1 3 2 3 2
------B------------
3 1 3 4 5 1 2 2
3 5 5 5 3 3 1 4
3 2 2 2 3 4 1 5
2 5 1 5 4 3 3 3
3 2 1 4 3 4 5 5
3 1 1 5 1 1 5 3
5 2 1 2 2 2 2 1
3 2 2 1 4 4 3 3
------C------------
64 59 54 89 72 63 44 59
53 39 37 74 61 47 41 50
38 38 37 48 40 41 14 26
61 57 41 90 67 57 33 39
62 31 26 88 74 59 48 51
38 40 36 63 42 33 27 30
34 22 18 54 42 36 34 46
26 24 23 39 34 24 26 32
==22872== Profiling application: ./mod3 8
==22872== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 46.50%  4.2560us         1  4.2560us  4.2560us  4.2560us  MatrixMulTiledMod(int
*, int*, int*, int)
 27.62%  2.5280us         2  1.2640us  1.2160us  1.3120us  [CUDA memcpy HtoD]
 25.87%  2.3680us         1  2.3680us  2.3680us  2.3680us  [CUDA memcpy DtoH]
```

Figura 2: Multiplicación de matricesde longitud 8 con ancho de tile 3

2. Características del Dispositivo

Se trabajó en una máquina con tarjeta gráfica NVIDIA GT740-M y Cuda 8.0. Las características se detallan a continuación:

```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 740M"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 2004 MBytes (2101542912 bytes)
  ( 2) Multiprocessors, (192) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1032 MHz (1.03 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  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: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  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): (2147483647, 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
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime Version = 8.0, NumDevs = 1, Dev
ice0 = GeForce GT 740M
```

3. Análisis de Factores limitantes del dispositivo

3.1. Registros

El Kernel de la multiplicación de matrices con tiles requiere almacenar 11 variables de tipo int(4 bytes) en registro, necesitandose así como mínimo 11 registros.

3.2. Memoria Compartida

En memoria compartida se necesitará la cantidad de bytes que usarán Mds, Nds y Nds2 (el tiplo int es de 4 bytes) $tile * tile * 4 * 3 = 12 * tile^2$ bytes.

tile=16 $16^2 * 12 = 3072bytes = 2Kb$

tile=32 $32^2 * 12 = 12288bytes = 12Kb$

tile=64 $64^2 * 12 = 49152bytes = 48Kb$

El dispositivo cuenta con 49152 bytes de memoria compartida por bloque, así la memoria compartida no limitará las operaciones.

3.3. CUDA Occupancy

Las limitaciones del hardware son principalmente:

- 2048 hebras por multiprocesador
- 1024 hebras por bloque
- 32 hebras en el warp

Lo recomendable es que el tamaño del bloque sea múltiplo del warp, para asi tener todas las hebras activas.Por lo que usando un blocksize de 32 o 16 maximizaría el número de hebras en el SM, como se observa en las figuras 3 y 4.

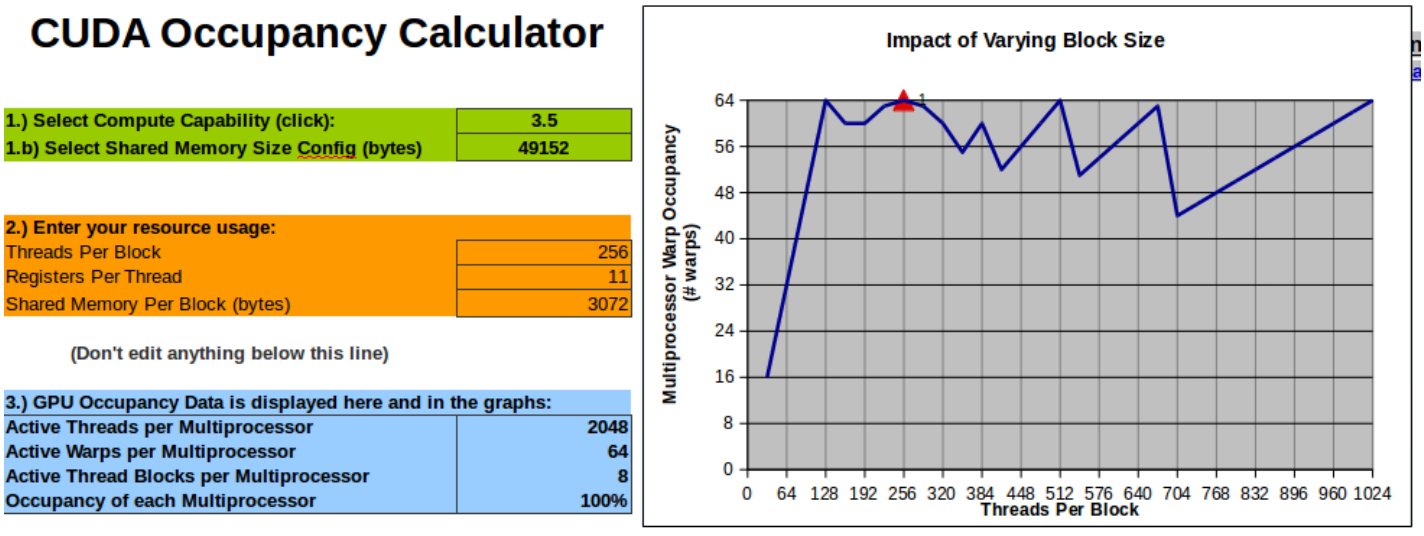

Figura 3: Occupancy para tile de ancho 16

BlockSize	Nro Blocks	Nro Hebras en SM
32x32	2	2048
16x16	8	2048

Se realizarán experimentos con estas dos dimensiones para evaluar su desempeño.

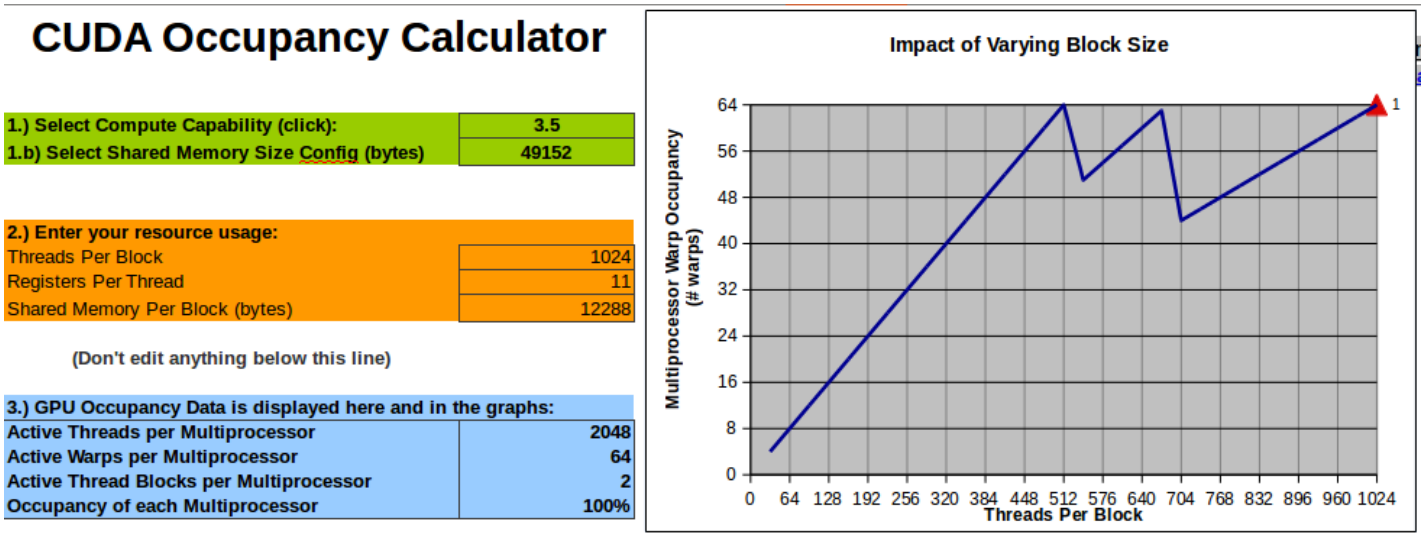

Figura 4: Occupancy para tile de ancho 32

4. Experimentos

4.1. Tabla de resultados (ms)

Matriz	256x256		1024x1024		4096x4096		8192x8192	
Block/Tile Size	16x16	32x32	16x16	32x32	16x16	32x32	16x16	32x32
Mult. Simple	2.28	0.59	150.6	145.04	8745.83	8370.59	77907.42	69143.46
Mult. Tiled	0.73	2.21	51.23	43.94	2764.95	2383.15	22154.74	19305.83
Mult. Tiled Opt.	0.605	0.631	38.94	48.93	2272.07	2789.64	1808.81	22842.91

Al optimizar la multiplicación de matrices con tiles, se observa una mejora en los tiempos usando un tile de 16x16, sin embargo al usar un tile de 32x32, el tiempo incrementa ya que al cargar datos más grandes a memoria compartida repetidamente el bandwith decrementa, aumentando la latencia. Para disminuir la latencia, es necesario disminuir el número de hebras y aumentar el trabajo por cada una de ellas.

4.2. Capturas de Pantalla

```
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./modi16 256
N: 256  Bloques : 16     Hebras/Bloque: 16
Tiempo  : 0.605408 ms
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./modi32 256
N: 256  Bloques : 8      Hebras/Bloque: 32
Tiempo  : 0.630720 ms
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./modi16 1024
N: 1024 Bloques : 64     Hebras/Bloque: 16
Tiempo  : 38.949535 ms
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./modi32 1024
N: 1024 Bloques : 32     Hebras/Bloque: 32
Tiempo  : 48.936226 ms
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./modi16 4096
N: 4096 Bloques : 256    Hebras/Bloque: 16
Tiempo  : 2272.076904 ms
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./modi32 4096
N: 4096 Bloques : 128    Hebras/Bloque: 32
Tiempo  : 2789.647705 ms
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./modi16 8192
N: 8192 Bloques : 512    Hebras/Bloque: 16
Tiempo  : 18088.148438 ms
rose@Satellite-S55-A:~/CS_AlgoritmosParalelos/cuda$ ./modi32 8192
N: 8192 Bloques : 256    Hebras/Bloque: 32
Tiempo  : 22842.908203 ms
```

5. Conclusiones

- La multiplicación con tiles y memoria compartida es más rápida que la multiplicacion simple, disminuyendo la lectura de datos en un radio de 1 a 16 o 32(según sea el tamaño del Tile). Mientras que al optimizar se disminuye la redundancia de datos y el acceso a memoria en un radio de 1 a 4 comparada con la Multiplicación con Tiles .
- Para el dispositivo GeForce 740M, el mejor tamaño de tile para una multiplicación de matrices optimizada es de 16x16 ya que maximiza al 100 % la ocupancia de cada multriprocesador.
- Al modificar la multiplicación con tiles se obtiene mejor eficiencia y mejor tiempo.