



UNIVERSIDAD NACIONAL DE SAN AGUSTIN

CIENCIA DE LA COMPUTACIÓN

---

# Informe de la multiplicación de matrices usando *tile* modificado

---

***Alumnas :***

*Judith Escalante Calcina*

***Profesor:***

*Mg. Alvaro Henry Mamani*

*Aliaga*

## Índice

<b>1. Multiplicación en cpu</b>	<b>2</b>
<b>2. Multiplicación en gpu</b>	<b>2</b>
2.1. Multiplicación sin memoria compartida . . . . .	2
2.2. Multiplicación con memoria compartida (tile) . . . . .	3
2.3. Multiplicación con memoria compartida (tile) modificada . . . . .	6
<b>3. Resultados</b>	<b>8</b>
<b>4. Conclusión</b>	<b>8</b>

## 1. Multiplicación en cpu

La multiplicación de matrices es un proceso altamente utilizado y conocido , este funciona perfectamente con matrices pequeñas pero no con matrices de grandes dimensiones, en el siguiente código podemos ver la función principal de una multiplicación de matrices en general :

```
void cpu_matrix_mult(int *h_a, int *h_b, int *h_result ,
int m, int n, int k) {
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            int tmp = 0.0;
            for (int h = 0; h < n; ++h)
            {
                tmp += h_a[i * n + h] * h_b[h * k + j];
            }
            h_result[i * k + j] = tmp;
        }
    }
}
```

## 2. Multiplicación en gpu

### 2.1. Multiplicación sin memoria compartida

La multiplicación realiza en la gpu de una computadora es mucho más eficiente que que la hecha en la cpu , por que utiliza varios thread para cada elemento de la matriz , que qa su vez es dividida en una cierta cantidad de bloques. Aún asi este no es uno de las ejecuciones más eficientes. Aqui podemos ver el código principal:

```
--global-- void matrix_mult(int *a,int *b, int *c, int m,
int n, int k)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    if( col < k && row < m)
```

```
{
    for(int i = 0; i < n; i++)
    {
        sum += a[row * n + i] * b[i * k + col];
    }
    c[row * k + col] = sum;
}
```

## 2.2. Multiplicación con memoria compartida (tile)

La multiplicación utilizando memoria compartida y *tileds* es una versión optimizada del producto de dos matrices,  $A \times B$ , en la que cada bloque de hilos computa una submatriz o *tiled* de la matrix resultado  $C$ . Esto permite reducir el cuello de botella del ancho de banda de la memoria, puesto que varios elementos del bloque acceden a la misma fila de  $A$  y columna de  $B$ .

A su vez, esta localidad de acceso será aprovechada para utilizar la memoria compartida, lo que también nos obligará a realizar algunas sincronizaciones entre hilos dentro de un bloque. La memoria compartida dentro de cada multiprocesador se utilizará para almacenar cada submatriz antes de los cálculos, acelerando el acceso a memoria global. A continuación podemos ver el código principal:

```
--global-- void matrix_mult_tiled(float*A, float*B, float*C,
int ARows, int ACols, int BRows, int BCols, int CRows,
int CCols)
{
    float CValue = 0;
    int Row = blockIdx.y*TILE_DIM + threadIdx.y;
    int Col = blockIdx.x*TILE_DIM + threadIdx.x;

    __shared__ float As[TILE_DIM][TILE_DIM];
    __shared__ float Bs[TILE_DIM][TILE_DIM];

    for (int k = 0; k < (TILE_DIM + ACols - 1)/TILE_DIM; k++)
    {
        if (k*TILE_DIM + threadIdx.x < ACols && Row < ARows)
            As[threadIdx.y][threadIdx.x] =
```

```
        A[Row*ACols + k*TILE_DIM + threadIdx.x];
    else
        As[threadIdx.y][threadIdx.x] = 0.0;

    if (k*TILE_DIM + threadIdx.y < BRows && Col < BCols)
        Bs[threadIdx.y][threadIdx.x] =
            B[(k*TILE_DIM + threadIdx.y)*BCols + Col];
    else
        Bs[threadIdx.y][threadIdx.x] = 0.0;

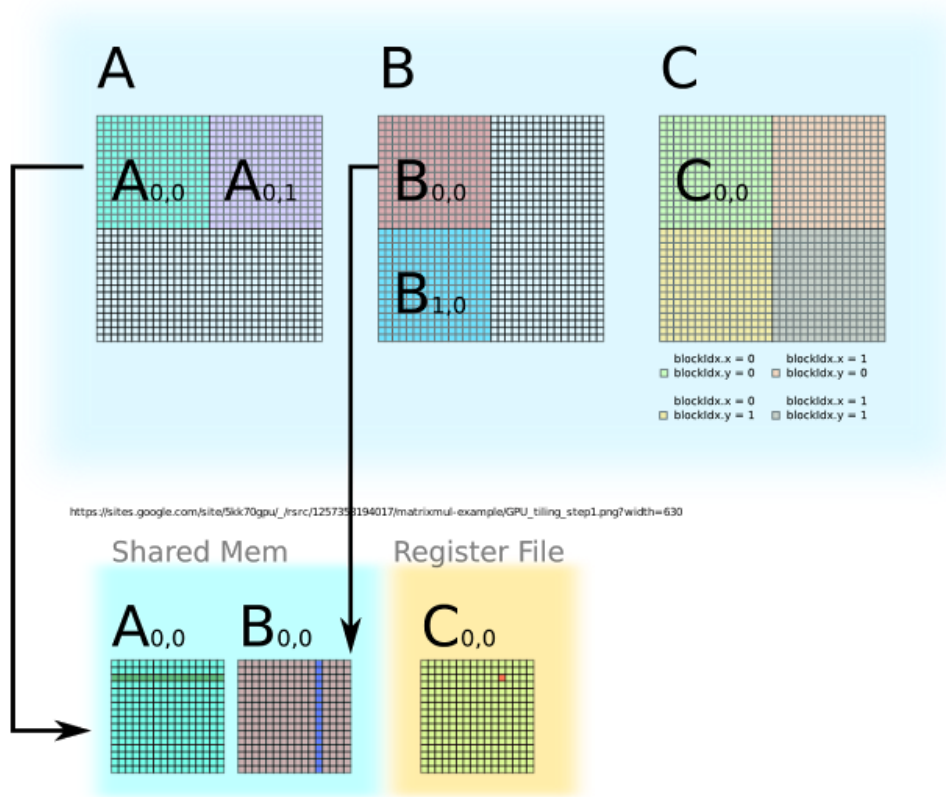
    __syncthreads();

    for (int n = 0; n < TILE_DIM; ++n)
        CValue += As[threadIdx.y][n] * Bs[n][threadIdx.x];

    __syncthreads();
}

if (Row < CRows && Col < CCols)
    C[((blockIdx.y * blockDim.y + threadIdx.y)*CCols)+
      (blockIdx.x*blockDim.x)+threadIdx.x]=CValue;
}
```

Figura 1: Multiplicación usando memoria compartida



Para que la multiplicación con memoria compartida se debe tener la arquitectura del device :

Figura 2: Arquitectura del GPU

```
cs01@slave3:/usr/local/cuda-8.0/extras/demo_suite$ ./deviceQuery
./deviceQuery Starting...

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

Detected 1 CUDA Capable device(s)

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, 2048)
  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
  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, Device0 = GeForce GT 620
Result = PASS
```

Es necesario saber el tamaño de la memoria compartida, para no exceder en el tamaño de los tiles; en este caso se está usando un device GeForce GT 620 para hacer las pruebas, el cual tiene 1024 threads por bloque y 49152 de memoria compartida.

En este caso el tamaño del tile puede ser de 16 y 32 ya que no sobrepasa el tamaño de threads por bloque, ni tampoco el tamaño de la memoria compartida y también porque el tamaño de un warp es de 32 threads. Donde cada MP tiene 8 SPs (Streaming Processor = CUDA Core)

### 2.3. Multiplicación con memoria compartida (tile) modificada

```
--global-- void matrix_mult_tiled(float*A, float*B, float*C,
```

```

int ARows, int ACols, int BRows, int BCols, int CRows,
int CCols)

{

    float CValue = 0 , CValue2=0;
    int Row = blockIdx.y*TILE_DIM + threadIdx.y;
    int Col = blockIdx.x*TILE_DIM + threadIdx.x;

    __shared__ float As[TILE_DIM][TILE_DIM];
    __shared__ float Bs[TILE_DIM][TILE_DIM];
    __shared__ float Bs_sub[TILE_DIM][TILE_DIM];

    for (int k = 0; k < (TILE_DIM + ACols - 1)/TILE_DIM; k++)
    {
        if (k*TILE_DIM + threadIdx.x < ACols && Row < ARows)
            As[threadIdx.y][threadIdx.x] =
            A[Row*ACols + k*TILE_DIM + threadIdx.x];

        else As[threadIdx.y][threadIdx.x] = 0.0;

        if (k*TILE_DIM + threadIdx.y < BRows && Col < BCols)
        {
            Bs[threadIdx.y][threadIdx.x] =
            B[(k*TILE_DIM + threadIdx.y)*BCols + Col];

            Bs_sub[threadIdx.y][threadIdx.x] =
            B[(k*TILE_DIM + threadIdx.y)*BCols + (Col+TILE_DIM)];
        }
        else
        {
            Bs[threadIdx.y][threadIdx.x] = 0.0;
            Bs_sub[threadIdx.y][threadIdx.x] = 0.0;
        }

        __syncthreads();

        for (int n = 0; n < TILE_DIM; ++n)
    }
}

```



```

    {
        CValue += As[threadIdx.y][n] * Bs[n][threadIdx.x];
        CValue2 += As[threadIdx.y][n] * Bs_sub[n][threadIdx.x]
    }

    __syncthreads();

}

if (Row < CRows && Col < CCols)
C[((blockIdx.y * blockDim.y + threadIdx.y)*CCols)+
 (blockIdx.x*blockDim.x)+threadIdx.x]= CValue;

C((((blockIdx.y * blockDim.y + threadIdx.y)*CCols)+
 (blockIdx.x*blockDim.x)+threadIdx.x)+ TILE_DIM)= CValue2;
}

```

### 3. Resultados

Cuadro 1: Tabla comparativa entre los tiempos de ejecución de la multiplicación de matrices.

Tamaño de la matriz	Matriz sin tile	Matriz con tile	Matriz modificada
1000 x 1000	10.57	5.61	3.79
2000 x 2000	127.32	97.73	55.95
4000 x 4000	954.35	564.81	123.78
8000 x 8000	2988.04	975.96	765.54
16000 x 16000	45378.71	7492.734	7309.797

### 4. Conclusión

Como podemos ver en todos los casos el algoritmo ejecutado en GPU es mucho más rápido que en CPU pero aún así la misma GPU tiene algunas mejoras , como por ejemplo utilizar memoria compartida.

Para ello, se debe saber como explotar los recursos disponibles en la GPU, en el que miles de hilos pueden colaborar con un objetivo común, las optimizaciones sobre el flujo de instrucciones y el paralelismo a nivel de instrucción.

Por último podemos mejorar aun más nuestro algoritmo si generamos dos resultados a la vez como podemos ver en el algoritmo de matriz modificada , en esta podemos ver , que en vez de cargar dos veces la misma fila de la primera matriz , cargaremos dos matrices tiled de la segunda matriz , así haremos la multiplicación con dos datos de la segunda matriz a la vez , y generaremos dos resultados que ubicaremos de forma correcta donde corresponda en la matriz resultante.