Computación Paralela y Distribuida
Departamento de Ciencia de la Computación
Universidad Católica San Pablo
21 de Junio de 2023

Multiplicación de matrices con CUDA

Rafael Alonso David Peñalva
rafael.david@ucsp.edu.pe

1. Introducción

Se utiliza `CUDA C/C++` para implementar dos kernels de multiplicación de matrices:

- Multiplicación de matrices (convencional)
- Multiplicación de matrices + Memoria compartida

Empezaremos detallando el input y el ouput de los kernels. Más adelante se explicará cada kernel por separado. Finalmente se verá la configuración de lanzamiento.
El código fuente se puede encontrar en: CUDA-Matrix-Multiplication.

2. I/O

2.1. Input

El input para ambos kernels es una matriz generada aleatoriamente, se utiliza la función `generateRandomMatrix`:

```
void generateRandomMatrix(vector<float> &matrix, int N)
{
  random_device rd;
  mt19937 gen(rd());
  uniform_real_distribution<float> dis(0.0, 9.0);

  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      matrix[i * N + j] = round(dis(gen));
}
```

Listing 1: Implementación de generateRandomMatrix

Esta función genera números de coma flotante del 0 al 9 y se redondean. Esto se hizo para verificar las matrices resultantes con más facilidad, sin embargo, puede ser alterado para emitir flotantes no redondeados y en cualquier rango.

2.2. Output

El output de los kernels es una matriz producto de la multiplicación, se puede imprimir añadiendo el argumento `-DEBUG` al ejecutar.

```
matrixMulKernel 4 -DEBUG
```

Listing 2: Imprimir las matrices al ejecutar

3. Kernels

3.1. Multiplicación de matrices (convencional)

Este kernel realiza una multiplicación de matrices en paralelo.

```
__global__
void MatrixMulKernel(float *M, float *N, float *P, int Width)
{
  // Se calculan las coordenadas globales de cada hilo en la matriz P
  int Row = blockIdx.y * blockDim.y + threadIdx.y;
  int Col = blockIdx.x * blockDim.x + threadIdx.x;
  // Se verifica que el hilo este dentro de los limites v lidos de la
   matriz P
  if ((Row < Width) && (Col < Width))
  {
    // Pvalue es una variable temporal para almacenar el resultado
   parcial de la multiplicacion realizada por cada hilo
    float Pvalue = 0;
    // Itera a traves del width de las matrices para realizar la
   multiplicacion
    for (int k = 0; k < Width; ++k)
    {
      // Se realiza la multiplicacion de matrices y se acumulan los
   resultados
      Pvalue += M[Row * Width + k] * N[k * Width + Col];
    }
    // Se actualiza la matriz de salida P con el resultado parcial
   calculado por el hijo
    P[Row * Width + Col] = Pvalue;
  }
}
```

Listing 3: Implementación de MatrixMulKernel

3.2. Multiplicación de matrices + Memoria compartida

Este kernel realiza una multiplicación de matrices en paralelo utilizando la memoria compartida de cada bloque.

```
__global__
void MatrixMulKernelShared(float *M, float *N, float *P, int width)
{
  // Se definen dos matrices en memoria compartida del tama o del
   bloque (TILE_WIDTH)
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  // Obtenemos los  ndices  de bloque y de hilo
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Se calculan las coordenadas globales de la matriz P
   correspondientes al hilo actual
  int Row = by * TILE_WIDTH + ty;
```

```
16   int Col = bx * TILE_WIDTH + tx;
17
18   // Pvalue es una variable temporal para almacenar el resultado
      parcial de la multiplicacion realizada por cada hilo
19
20   float Pvalue = 0;
21
22   // Se inicia un bucle para dividir la matriz en bloques mas peque os
23   for (int ph = 0; ph < width / TILE_WIDTH; ph++)
24   {
25     // Se cargan los bloques de las matriz M y N en memoria compartida
      Mds Nds
26     Mds[ty][tx] = M[Row * width + ph * TILE_WIDTH + tx];
27     Nds[ty][tx] = N[(ph * TILE_WIDTH + ty) * width + Col];
28
29     // Se sincronizan los hilos del bloque antes de continuar, caso
      contrario no habrian datos para multiplicar
30     __syncthreads();
31
32     // Bucle para realizar la multiplicacion de matrices en bloques
      cargados en memoria compartida
33     for (int k = 0; k < TILE_WIDTH; ++k)
34     {
35       // Se realiza la multiplicaci n de matrices y se acumulan los
      resultados
36       Pvalue += Mds[ty][k] * Nds[k][tx];
37     }
38     // Se sincronizan los hilos del bloque para asegurar que todos los
      hilos hayan terminado de multiplicar antes de iterar nuevamente
39     __syncthreads();
40   }
41   // Se actualiza la matriz de salida P con el resultado parcial
      calculado por el hijo
42   P[Row * width + Col] = Pvalue;
43 }
```

Listing 4: Implementación de MatrixMulKernelShared

3.2.1. Lanzamiento

El lanzamiento debe configurado de esta manera:

```
1 int blockDim = TILE_WIDTH;
2 int gridDim = ceil(width / blockDim);
3
4 dim3 dimGrid(gridDim, gridDim, 1);
5 dim3 dimBlock(blockDim, blockDim, 1);
6
7 MatrixMulKernelShared<<<dimGrid, dimBlock>>>(d_M, d_N, d_P, width);
```

Listing 5: Configuración de lanzamiento

4. Comparación de rendimiento

Se realiza la comparación pasándole a ambas implementaciones n, operando matrices de n. En la gráfica 1 podemos observar en el eje X el tamaño de las matrices, y en el eje Y, los tiempos

en milisegundos.

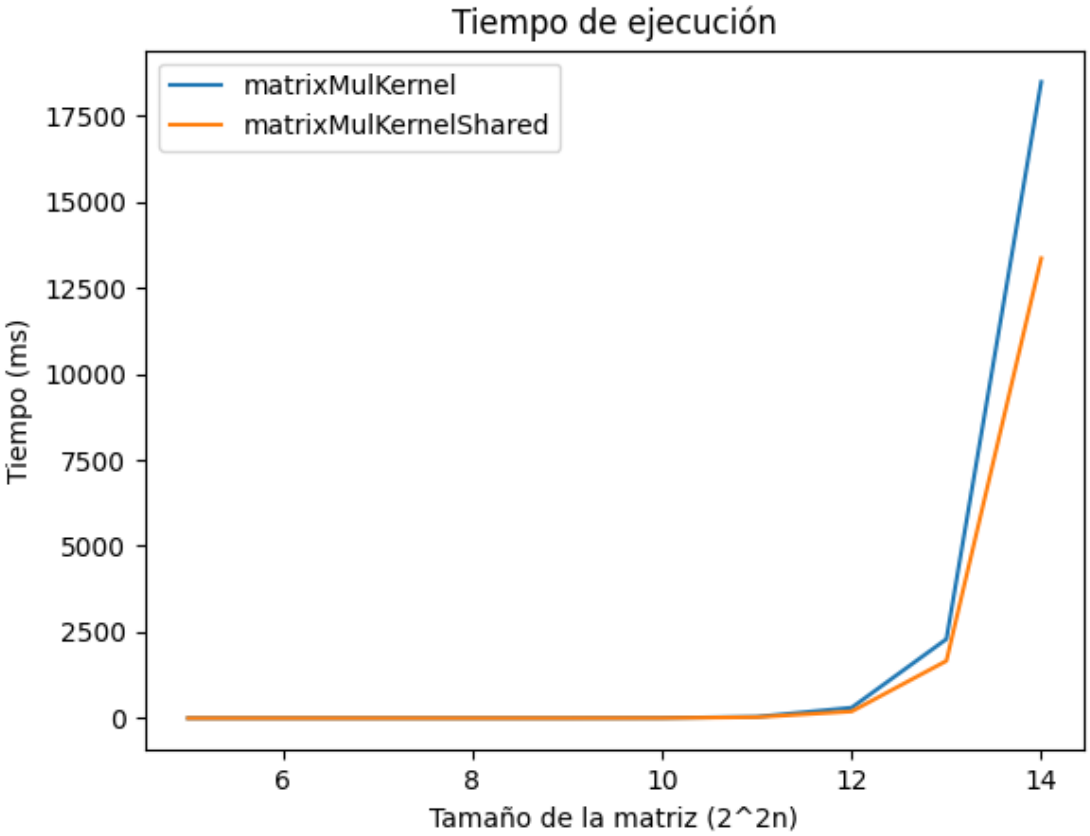

Figura 1: Comparación de tiempos