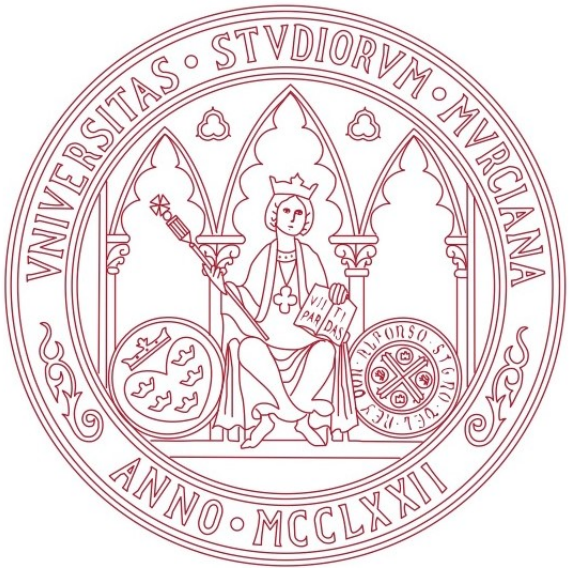


Programación de Arquitecturas Multinúcleo

PROYECTO DE PROGRAMACIÓN CON CUDA

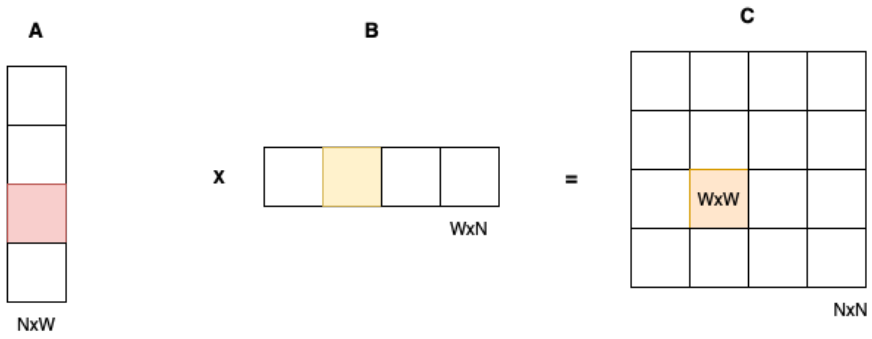
Francisco Arnaldo Boix Martínez - 52041159W

Abril 2023

Índice

1. Programa `compara_kernels`

En concreto tenemos que realizar una multiplicacion de matrices, por bloques de tamaño `WxW`.



En este primer programa dentro del proyecto de programación tenemos que realizar una multiplicación de matrices usando uno de los siguientes algoritmos proporcionados en los recursos de la asignatura:

- **simpleMultiply:** Cada thread calcula el elemento (row,col) de C recorriendo la fila row de A y la columna col de B.

Dim mat	Block Size	Time (ms)
512	4	0.16
	8	0.09
	16	0.12
	32	0.26
1024	4	0.54
	8	0.29
	16	0.40
	32	0.93
2048	4	2.50
	8	1.15
	16	1.57
	32	3.88
4096	4	10.72
	8	4.68
	16	6.44
	32	16.12

Cuadro 1: Tiempos para el algoritmo `simpleMultiply`

- **coalescedMultiply:** Cada elemento del tile de A se lee de memoria global a memoria compartida solamente una vez, en forma completamente coalesced, sin desaprovechar ancho de banda. En cada iteración: un valor de memoria compartida se distribuye a todos los threads del warp.

Dim mat	Block Size	Time (ms)
512	4	0.17
	8	0.09
	16	0.11
	32	0.20
1024	4	0.56
	8	0.28
	16	0.37
	32	0.68
2048	4	2.52
	8	1.10
	16	1.36
	32	2.83
4096	4	10.77
	8	4.48
	16	5.65
	32	11.77

Cuadro 2: Tiempos para el algoritmo `coalescedMultiply`

- **sharedABMultiply:** Para calcular cada fila del tile de C se lee el tile entero de B. Por tanto, para el tile entero de C (el trabajo que hace un bloque de threads) se lee el tile entero de B repetidamente (w veces). Véase el Cuadro 3

Es importante recalcar que en estos dos últimos algoritmos, he tenido que definir una zona de memoria compartida **dinámica**, puesto que en tiempo de **compilación** se **desconoce** el valor de `N`. En conreto el tamaño de memoria compartida que uso es: `dim_block * dim_block * sizeof(float)`.

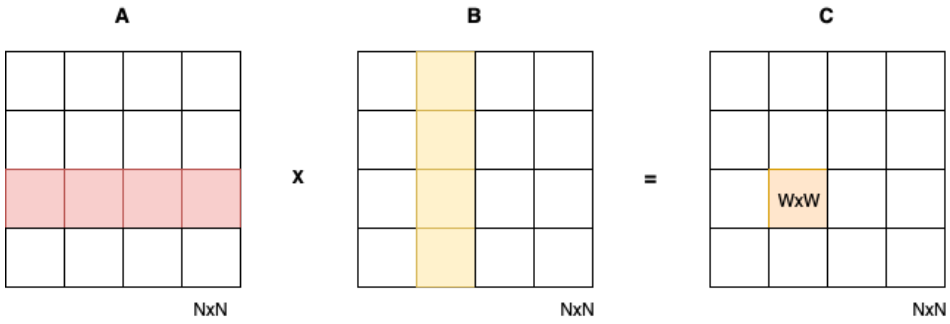
En concreto, para el algoritmo `sharedMultiply`, como hace uso de dos matrices de tamaño `WxW` dentro de su memoria compartida, hay que reservar el doble de espacio. Sabiendo que no conocemos el valor de `N`, tampoco podemos conocer la **dirección de memoria** a la que **B** debe de **apuntar** (al menos en tiempo de compilación), por tanto, hago un **cálculo** sencillo al principio del código del kernel.

Dim mat	Block Size	Time (ms)
512	4	0.19
	8	0.10
	16	0.10
	32	0.19
1024	4	0.59
	8	0.24
	16	0.33
	32	0.65
2048	4	2.52
	8	0.91
	16	1.28
	32	2.47
4096	4	10.31
	8	3.60
	16	5.09
	32	9.85

Cuadro 3: Tiempos para el algoritmo `sharedMultiply`

2. Programa `mulmatcua_1G`

En esta ocasión vamos a hacer el mismo cálculo pero ahora teniendo en cuenta matrices cuadradas de `NxN`. En concreto haré uso del algoritmo `sharedMultiply` descrito en la sección 1.



Para poder realizar este nuevo ejercicio propuesto tenemos que tener en cuenta, que tenemos que recorrer los bloques dentro de la matriz, por ello añado un nuevo bucle `for`. Si nos fijamos en las iteraciones que hacen los dos bucles anidados, nos damos cuenta que suman un total de: `N / tile_dim * tile_dim = N`

```
__global__ void sharedABMultiply(float *a, float *b, float *c, int N,
    const int tile_dim)
{
    extern __shared__ float aTile[];

    float sum = 0.0f;
```

```
6
7       int row, col;
8
9       for (int tileIdx = 0; tileIdx < N / tile_dim; tileIdx++)
10      {
11          float *bTile = aTile + (tile_dim * tile_dim);
12          row = blockIdx.y * blockDim.y + threadIdx.y;
13          col = tileIdx * blockDim.x + threadIdx.x;
14
15
16          aTile[threadIdx.y * tile_dim + threadIdx.x] = a[row * N + tile_dim
        * tileIdx + threadIdx.x];
17          bTile[threadIdx.y * tile_dim + threadIdx.x] = b[threadIdx.y * N +
       tile_dim * tileIdx * N + col];
18
19          __syncthreads();
20
21          for (int i = 0; i < tile_dim; i++)
22          {
23              sum += aTile[threadIdx.y * tile_dim + i] * bTile[i * tile_dim
       + threadIdx.x];
24          }
25      }
26
27      c[row * N + col] = sum;
28  }
```

Pasemos ahora a ver los tiempos obtenidos al ejecutar el programa:

Dim mat	Block Size	Time (ms)
512	4	15.27
	8	3.65
	16	2.42
	32	2.38
1024	4	120.89
	8	28.89
	16	19.07
	32	18.08
2048	4	950.37
	8	220.79
	16	155.37
	32	143.67
4096	4	8018.87
	8	1860.30
	16	1228.49
	32	1162.46

Cuadro 4: Tiempos para el programa `mulmatcua_1G`

3. Programa `mulmat_1G`

En esta ocasión tenemos que realizar una multiplicación muy similar a la del ejercicio de la Sección 2, pero sin asumir que las matrices son cuadradas, es decir, tenemos que hacerlo de forma genérica.

De forma previsora el ejercicio anterior, lo realicé usando algún parámetro **innecesario**, teniendo en cuenta que posteriormente tendría que **generalizar** el algoritmo para cualquier tamaño de matriz. De esta forma, la realización de este ejercicio fue prácticamente inmediata apoyándome en el ejercicio anterior.

```
__global__ void sharedABMultiply(float *a, float *b, float *c, const int M
    , const int N, const int K, const int tile_dim)
{
    extern __shared__ float aTile[];
    float *bTile = aTile + (tile_dim * tile_dim);

    float sum = 0.0f;

    int row, col;

    for (int tileIdx = 0; tileIdx < K / tile_dim; tileIdx++)
    {
        row = blockIdx.y * blockDim.y + threadIdx.y;
        col = blockIdx.x * blockDim.x + threadIdx.x;


        aTile[threadIdx.y * tile_dim + threadIdx.x] = a[row * K + tileIdx
    * tile_dim + threadIdx.x];
        bTile[threadIdx.y * tile_dim + threadIdx.x] = b[threadIdx.y * N +
    tileIdx * tile_dim * N + col];

        __syncthreads();

        for (int i = 0; i < tile_dim; i++)
        {
            sum += aTile[threadIdx.y * tile_dim + i] * bTile[i * tile_dim
    + threadIdx.x];
        }

    }

    c[row * N + col] = sum;
}
```

Como podemos observar, los únicos cambios que han habido han sido que tenemos que tener en cuenta los nuevos tamaños de las matrices a la hora de acceder a ellas. Del mismo modo, a la hora de reservar memoria para ellas fuera del kernel debemos tener cuidado con los nuevos tamaños, con respecto a la versión de matrices cuadradas de tamaño `NxN`. Veamos un ejemplo en el siguiente fragmento de código.

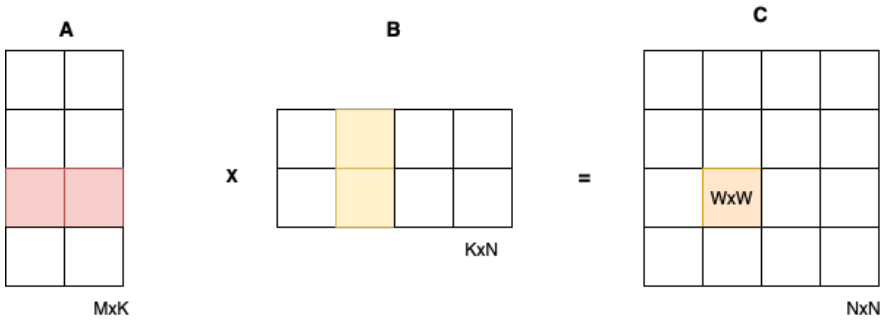
```
    dim3 grid(t, s);
    dim3 block(w, w);
```

```
    ...

    size_A = m * k;
    size_B = k * n;
    size_C = m * n;

    nBytes_A = size_A * sizeof(float);
    nBytes_B = size_B * sizeof(float);
    nBytes_C = size_C * sizeof(float);
```

Nótese que aquí tampoco hemos tenido que modificar el tamaño de la **memoria compartida** para la ejecución del kernel puesto que la memoria compartida para el algoritmo `sharedMultiply` usa dos matrices del tamaño de un bloque, es decir, `WxW`.



4. Programa `mulmat_1C1G`

Partiendo de la versión de la Sección 3, debemos dividir el la carga de trabajo entre la **GPU** y la **CPU**. En concreto, la CPU deberá encargarse de las `F` últimas filas de la matriz resultante. En concreto en la Figura 1, `F = 1`.

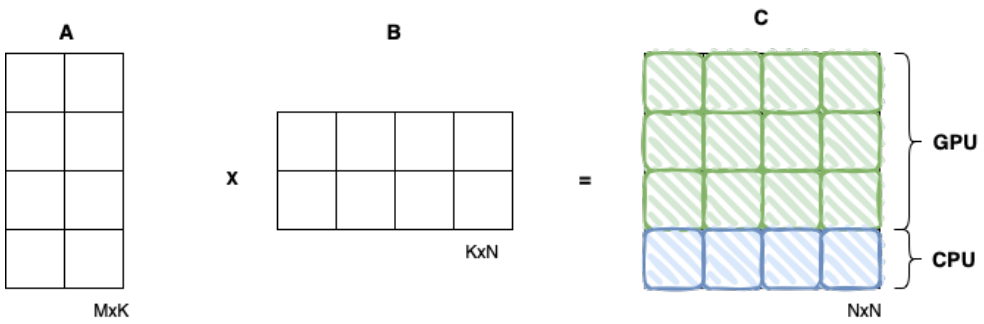
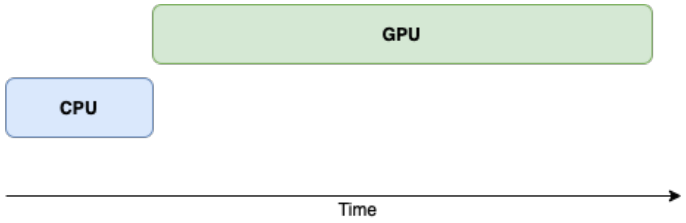


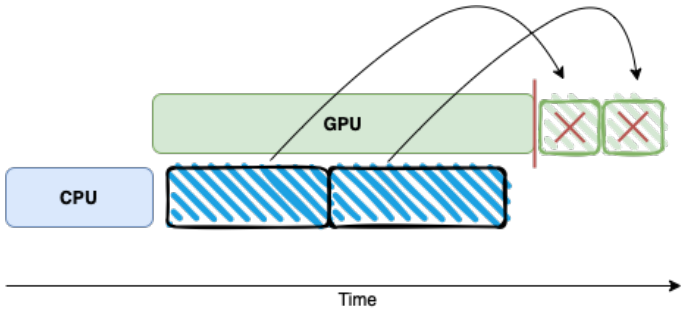
Figura 1: Visualización del último ejercicio a realizar

Nuestro objetivo es encontrar un *sweet spot* o *zona favorable* **específica** para la máquina en la que estemos ejecutando los programas, en la que la CPU y la GPU se estén aprovechando

al máximo. Si nos damos cuenta, en la versión de la Sección 3, nos encontramos con algo similar a la siguiente Figura.



Como sabemos, si comparamos el número de cálculos que podemos hacer en una **GPU** frente a una **CPU** aplicados a este problema, es inmensamente superior, por tanto, por poco trabajo que asignemos a la CPU, la barra azul crecerá de mucho mas rápido que la verde. Nuestro objetivo es encontrar el punto óptimo en el cual, la CPU alivie un poco el trabajo de la GPU (teniendo en cuenta el tiempo que la CPU está ociosa mientras espera a que la CPU termine de hacer los cálculos y enviarlos copiarlos desde su memoria a la de la CPU), pero sin llegar a sobrepasar el tiempo de la CPU, es decir, queremos lograr algo similar a lo que se muestra en la siguiente Figura.



Para ello, haremos uso de `OpenMP` para intentar optimizar al máximo los cálculos que se van haciendo en la CPU, intentando que los rectángulos azules sean lo menos **alargados** posible.

El kernel queda idéntico al de la versión anterior, a excepción de que tenemos que indicarle hasta que fila tiene que realizar los cálculos. Para ello, es necesario añadir un par de comprobaciones dentro del kernel.

```
__global__ void sharedABMultiply(float *a, float *b, float *c, const int M
    , const int N, const int K, const int tile_dim, const int F)
{
    extern __shared__ float aTile[];
    float *bTile = aTile + (tile_dim * tile_dim);

    float sum = 0.0f;

    int row, col;

    for (int tileIdx = 0; tileIdx < K / tile_dim; tileIdx++)
    {
        row = blockIdx.y * blockDim.y + threadIdx.y;
        col = blockIdx.x * blockDim.x + threadIdx.x;
```

```
14
15          if (row >= (M - F))
16              break;
17
18          ...
19      }
20
21      if (row < (M - F))
22          c[row * N + col] = sum;
23  }
```

Para la parte de cómputo de la CPU he usado el mismo algoritmo de multiplicación de matrices en **secuencial** estándar que he estado usando durante la realización de todos los ejercicios para comprobar la **corrección** de los mismos. Las únicas diferencia es que le he añadido un pragma `OpenMP`, y le he indicado desde donde tiene que empezar a hacer los cálculos de la matriz final, es decir, las últimas `F` filas.

```
1  float *multiply_row(float *A, float *B, float *C, int m, int n, int w, int
       row)
2  {
3
4      int i, j, k;
5
6  #pragma omp parallel for private(i, j, k)
7      for (i = row; i < m; i++)
8      {
9          for (j = 0; j < n; j++)
10         {
11             C[i * n + j] = 0.0f;
12             for (k = 0; k < w; k++)
13                 C[i * n + j] += A[i * w + k] * B[k * n + j];
14
15         }
16     }
17
18     return C;
19 }
```

Pasemos ahora a ver los tiempos obtenidos:

Rows CPU	Time (ms)
1	1210.59
4	1207.96
8	1205.38
12	1203.30
16	1199.92
20	1198.99
24	1195.07
28	1302.94
32	1411.90

Cuadro 5: Tiempos para el programa `mulmat_1C1G` y un tamaño de matrices de `2048`

Como vemos en el Cuadro 5, observamos el efecto que he comentado anteriormente. En concreto, para esta **máquina** y para un tamaño de problema de `2048`, el *número óptimo* de filas a computar por la **CPU** es de `24`.

Nótese la diferencia de capacidad de cómputo que existe entre la CPU y la GPU. Tardan más o menos lo mismo en calcularse `2024` filas en la GPU que `24` filas en la CPU usando un algoritmo de multiplicación de matrices paralelizado con `OpenMP`.

5. Testing

Para cada uno de los programas previamente descritos, se puede definir con una etiqueta del preprocesador la etiqueta `TEST` (no es necesario darle ningún valor) y se puede hacer la comprobación de que el algoritmo es correcto, comparandolo con una versión de multiplicación de matrices secuencial. Es importante tener esto en cuenta a la hora de medir el rendimiento.

```
1 #define TEST
```

Del mismo modo, hay algunas etiquetas comentadas con nombres descriptivos, que de igual manera se pueden definir para obtener información adicional.