Sesión 4 – Producto de Matrices

Introducción

En esta sesión vamos a trabajar con un algoritmo muy conocido, el producto de matrices. Un posible algoritmo secuencial podría ser el siguiente:

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    for (k=0; k<P; k++)
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

Kernel 00 (MM00.cu)

Este test os lo damos ya hecho, sólo tenéis que compilarlo y ejecutarlo.

Para compilar el programa sólo hay que hacer:

```
make kernel00.exe
```

Para ejecutarlo hay que enviarlo al sistema de colas, tal y cómo hicimos en la sesión anterior.

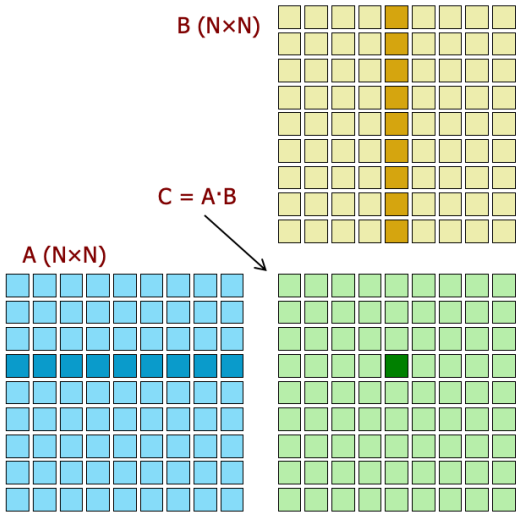
Al ejecutar el *kernel* podemos pasarle 2 parámetros. El primero es el tamaño del problema (este *kernel* sólo funciona si las matrices son cuadradas y múltiplo del número de *Threads* en cada dimensión). El segundo es un flag ('Y' o 'N') para indicar si queremos comprobar el resultado. Para comprobar que el resultado es correcto, os recomendamos que no utilicéis tamaños muy grandes, el tiempo de ejecución del test es muy elevado. Una ejecución podría ser la siguiente:

```
./kernel00.exe 640 Y
```

Ejecuta el *kernel* con matrices cuadradas de 640x640 elementos y comprueba el resultado. El resultado de esta ejecución podría ser el siguiente (todos los resultados que aparecen en documento corresponden a la ejecución con una K40c):

```
KERNEL 00
Dimensiones: 640x640
nThreads: 32x32 (1024)
nBlocks: 20x20 (400)
NO usa Pinned Memory [1]
Tiempo Global: 8.789408 milseg
Tiempo Kernel: 4.148928 milseg
Rendimiento Global: 59.65 GFLOPS
Rendimiento Kernel: 126.37 GFLOPS
TEST PASS
```

El programa calcula el rendimiento del kernel incluyendo el movimiento de datos entre CPU y GPU, y el rendimiento del kernel exclusivamente.



Este kernel es el que vimos en clase. En cada thread se calcula un elemento de la matriz resultado (**`C[i][j]`**). Cada thread ha de leer una fila de la matriz A (**`A[i][-]`**) y una columna de B (**`B[-][j]`**). El código del kernel se muestra a continuación:

```
 __global__ void Kernel00 (int N, float *A,
float *B, float *C) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  float tmp = 0.0;
  for (int k=0; k<N; k++)
    tmp += A[row*N+k] * B[k*N+col];
  C[row*N+col] = tmp;
```

Se puede comprobar que este kernel funciona perfectamente cuando el tamaño del problema es múltiplo del número de threads (32, en nuestro ejemplo) en cada dimensión. Pero no funciona en caso contrario. Si ejecutamos el siguiente código:

[1] El código que os proporcionamos permite probar la memoria PINNED. Simplemente hay que cambiar el flag correspondiente en el Makefile.

```
./kernel00.exe 641 Y
```

El resultado que obtenemos es el siguiente:

```
KERNEL 00
Dimensiones: 641x641
nThreads: 32x32 (1024)
nBlocks: 20x20 (400)
NO usa Pinned Memory
Tiempo Global: 9.858720 milseg
Tiempo Kernel: 5.117536 milseg
Rendimiento Global: 53.43 GFLOPS
Rendimiento Kernel: 102.93 GFLOPS
0:640: 158.175903 - 0.000000 = 158.175903
TEST FAIL
```

Se puede comprobar fácilmente que el número de *Thread Blocks* no es correcto, el elemento (0, 640) de la matriz resultado, no se está calculando.

Kernel 01 (MM01.cu)

El *kernel* 00 sólo funciona cuando las dimensiones del problema son múltiplo del número de *threads* en cada dimensión. Modificad el *kernel* para que funcione con matrices de cualquier dimensión y con matrices no cuadradas.

Hay que modificar bastantes cosas:

- El main, para leer de la línea de comandos las 3 dimensiones del problema:

  $$C(N \times M) \leftarrow A(N \times P) \cdot B(P \times M)$$

  Queremos que la invocación del kernel sea algo parecido a ésto:

```
./kernel01.exe 639 641 1023 Y
```

  Siendo N = 639, M = 641 y P = 1023.

- La creación e inicialización de las matrices y las transferencias entre CPU y GPU.

- El kernel, para que tenga en cuenta que el tamaño del problema no es múltiplo del número de *Threads*. Los *Thread Blocks* de los bordes han de hacer menos cálculos y hay que evitar que escriban en memoria.
- La invocación del kernel, para que lance el número correcto de *Thread Blocks*.
- El cálculo de los GFLOPS.

Si analizamos el kernel que hemos utilizado en este programa y el anterior tenemos que:

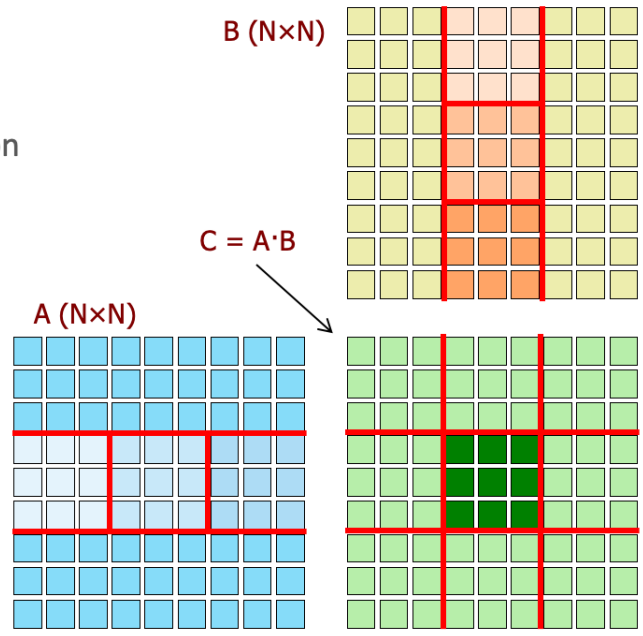
- Cada thread calcula un elemento de la matriz C.
  - Cada fila de A es leída N veces desde la memoria global.
  - Cada columna de B es leída N veces desde la memoria global.
- Se desperdicia mucho ancho de banda.
- Mal equilibrio entre cálculo y ancho de banda. En definitiva, es un código bastante ineficiente.

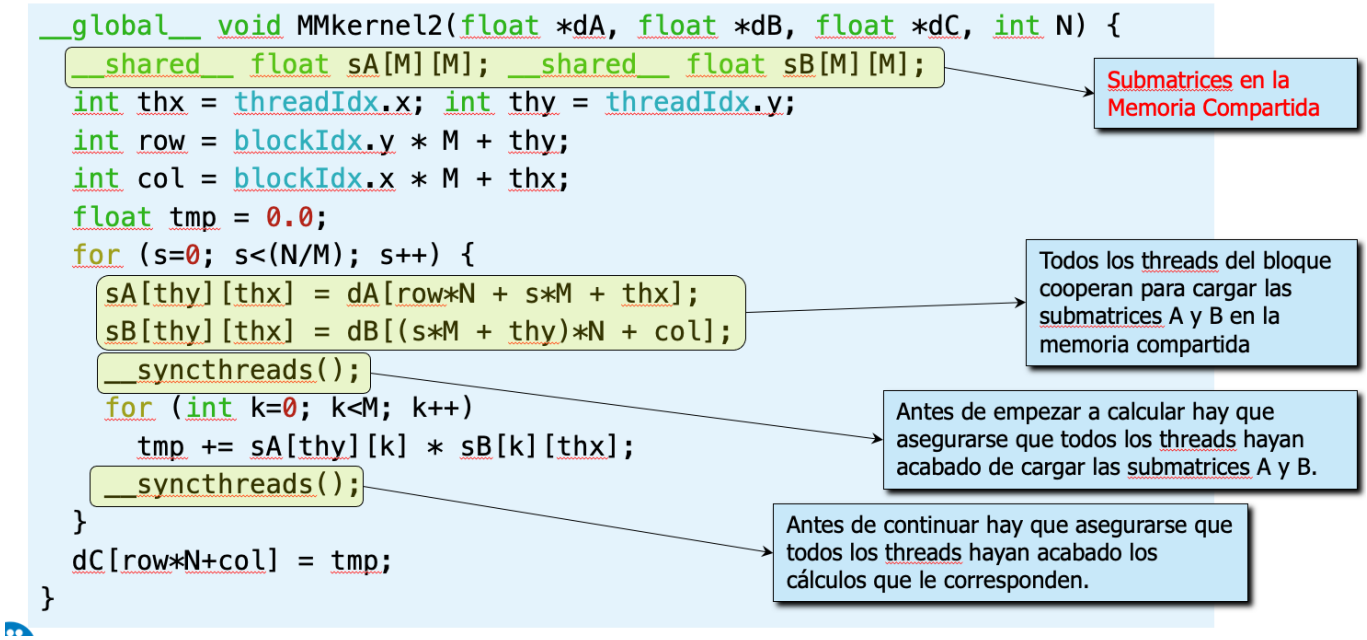
Kernel 10 (MM10.cu)

En este *kernel* vais a implementar el producto de matrices a bloques que vimos en las clases de teoría. Os incluimos las dos transparencias dónde se explicaba el *kernel*.

C = A · B (tamaño N × N)

- Cada block thread calcula un bloque de datos de **M×M** elementos de la matriz C.
- El algoritmo itera **N/M veces**. En cada iteración
  - Traemos un bloque **M×M** de A a la memoria compartida. La matriz A se lee **N/M veces**.
  - Traemos un bloque **M×M** de B a la memoria compartida. La matriz B se lee **N/M veces**.
- El valor de M es importante:
  - **Las submatrices han de caber en la memoria compartida.**
  - Necesitamos **M×M** threads por bloque.
- Utilizamos menos ancho de banda. El equilibrio entre cálculo y ancho de banda mejorará.





Implementad esta nueva versión del *kernel*. Utilizad como punto de partida el fichero **MM01.cu**. Tenéis que implementar el *kernel*, pensando en que ha de aceptar matrices NO cuadradas:

$$C(N \times M) \leftarrow A(N \times P) \cdot B(P \times M)$$

En principio, en esta versión sólo hay que preocuparse de que funcione correctamente con dimensiones de matriz múltiplo del número de *threads*. Por ejemplo, así:

```
./kernel10.exe 640 512 1024 Y
```

Kernel 11 (MM11.cu)

Modificad el *kernel* para que funcione con matrices de cualquier dimensión. Si utilizáis como punto de partida el código de **MM10.cu**, sólo hay que modificar el *kernel*. Queremos que el nuevo *kernel* funcione con algo parecido a esto:

```
./kernel11.exe 639 641 1023 Y
```

¡Atención con esta versión! El código no es trivial. Es un algoritmo bastante complejo. Procurad que no se pierda mucho rendimiento con respecto a la versión del kernel 10.

Pinned Memory

Se puede modificar el *Makefile* de la siguiente forma:

```
PROG_FLAGS = -DPINNED=1
```

Al cambiar este flag el programa utilizará *Pinned Memory* (obtenida con **`cudaMallocHost`**) en vez de memoria paginada (obtenida con **`malloc`**).

Volver a compilar y ejecutar los programas anteriores y observad como cambia el rendimiento de nuestra aplicación.

Otras cosas a probar

SI tenéis tiempo, se pueden probar algunas cosas. Cosas interesantes que podéis intentar son las siguientes:

- Probad diferentes dimensiones de Block Thread: 16×16, 8×8, 16×32, 8×16, … Algunas de ellas (las no cuadradas) requieren que modifiquéis ligeramente el código. Para las cuadradas, es suficiente con modificar el *Makefile* de la siguiente forma:

```
PROG_FLAGS = -DSIZE=16
```

- Modificad **`MM01.cu`** y **`MM11.cu`** para que se ejecute en un bucle diferentes invocaciones del Kernel01 y Kernel11. El objetivo es comprobar cómo cambia el rendimiento al aumentar el tamaño, y la pérdida de rendimiento cuando el tamaño de las matrices no es múltiplo del número de Threads. El código para ver la influencia del tamaño podría ser algo así:

```
for (N=128; N<2048; N=N+32)
  Kernel01<<<dimGrid, dimBlock>>>(N, N, N, d_A, d_B, d_C);
```

  El código para ver cómo influye que el tamaño del problema no sea múltiplo del número de *threads* podría ser algo así:

```
for (N=1024; N<1256; N=N+2)
  Kernel01<<<dimGrid, dimBlock>>>(N, N, N, d_A, d_B, d_C);
```

Obviamente, habrá que modificar el código adecuadamente para obtener los tiempos de ejecución. En este caso, no será necesario comprobar el resultado. Ya sabemos que los *kernels* funcionan bien.

Un desafio

Si miramos el rendimiento de pico de la NVIDIA RTX 3080 vemos que es de 29,77 TFLOPs. Con el algoritmo que utiliza la memoria compartida, podemos llegar (más o menos) a los 2,4 TFLOPs. Estamos muy lejos. ¿Se os ocurre alguna estrategia, que no sea usar Tensor Cores, para mejorar estos rendimientos?

Consideraciones Finales

Igual que hicimos en la sesión anterior, para facilitaros las cosas, y sólo para evitar que os quedéis encallados, hemos incluido un directorio en el que están todos los *kernels* solucionados.