UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster Universitario en Computación en la Nube y de Altas Prestaciones

# Programación de GPUs con CUDA y OpenCL (PGPU)

**Tarea 3**

## Introducción

Para compilar los ejercicios se puede utilizar el siguiente comando:

```
nvcc -lineinfo -Xptxas=-v -arch=sm_60 -o ejercicio ejercicio.cu
```

## Ejercicio 1

En este ejercicio vamos a construir una infraestructura que vamos a utilizar en lo sucesivo para el resto de ejercicios. El código facilitado en el fichero `SumaVectores.cu` contiene dicho esqueleto. Los siguientes pasos van a consistir en el rellenado de dicho esqueleto.

1. La función `main` está completamente implementada. Se llama a la función `vector_sum`, que suma los dos vectores de tamaño `n` pasados como segundo y tercer argumentos, respectivamente, y devuelve el vector sumado como último argumento. Esta función servirá para comprobar que el resultado es correcto. La implementación es trivial y ya se encuentra implementada.

2. La función `cu_vector_sum` debe hacer lo propio en la GPU. En esta función se distinguen los punteros a la memoria de la CPU (precedidos por `h_`) de aquellos que apuntan a la memoria de la GPU (precedidos por `d_`). Véase que hay declarados tres punteros a GPU: `d_a`, `d_b` y `d_c`. Lo primero que hay que hacer es reservar memoria, apuntada por dichos punteros, en GPU mediante la función `cudaMalloc` en el lugar indicado.

3. Copiar los vectores de CPU `h_a` y `h_b` en GPU, es decir, en los vectores `d_a` y `d_b`, respectivamente, mediante la función `cudaMemcpy`.

4. Calcular los bloques de threads totales que se necesitan para "cubrir" los `n` elementos a sumar de los vectores sobre la variable `nblocks`. Por ejemplo, si `n=1300` y `blocksize=32`, el número total de bloques será de `nblocks=41`.

5. A continuación se crean dos variables, `dimGrid` y `dimBlock`, de tipo `dim3` para representar la malla de bloques y el tamaño de bloque de threads. En caso de que la malla de bloques (o del tamaño de bloque) sean "lineales" o 1D no es necesario crear esta variable pero, si lo hacemos así, nos servirá para el futuro.

6. En este paso se debe llamar al kernel mediante la notación

`nombre_de_kernel<<<...>>>( argumentos ).`

Obsérvese que al principio del fichero puede encontrarse la implementación inacabada de un kernel (`compute_kernel`). Se debe implementar dicho kernel utilizando los argumentos especificados. Para ello, hay que tener en cuenta que cada thread debe encargarse de sumar dos elementos de los vectores `d_a` y `d_b`, respectivamente, en un elemento del vector `d_c`. Para ello, hay que tener en cuenta que cada thread puede encargarse del elemento de índice

`indice = threadIdx.x + blockDim.x * blockIdx.x.`

Algo importante a tener en cuenta es que los threads "cubren" un espacio mayor que la memoria reservada para los vectores (`n`), por lo que se deberá evitar que ningún thread acceda a posiciones de memoria que no se han reservado previamente.

7. Ahora, de vuelta a la función `cu_vector_sum` y después de la llamada al kernel realizamos una copia del vector `d_c` en GPU a la CPU, o sea, en el vector `h_c` mediante la función `cudaMalloc`.

8. Terminamos la implementación de la función `cu_vector_sum` liberando la memoria creada en GPU mediante la función `cudaFree`. Este paso de liberar memoria es importante realizarlo dado que puede servir para detectar errores que podrían pasar desapercibidos.

## Ejercicio 2

En este ejercicio se va a realizar la suma de dos matrices. En realidad, se trata de una generalización de la suma de vectores anterior solo que a dos dimensiones, aunque se trata más de una cuestión conceptual. La parte importante y diferenciada es que ahora vamos a trabajar con bloques de threads bidimensionales y mallas de bloques también bidimensionales.

Lo primero que vamos a tratar es de la representación de los datos. Las matrices bidimensionales en C pueden declararse con dos indirecciones de manera que el acceso al elemento $i, j$ de la matriz $A$ se realizaría así: `A[i][j]`. Sin embargo, nosotros vamos a utilizar otra manera. Las matrices se almacenarán en un array unidimensional. Sea pues una matriz matemática $A \in \mathbb{R}^{m \times n}$, la declaración de la misma en C la realizamos de la siguiente manera:

```
float *A = (float*) malloc ( m*n*sizeof(float) );
```

Para nosotros, las filas de la matriz $A$ se almacenarán consecutivamente en memoria, es decir, el elemento $A_{i,j+1}$ se encuentra a continuación del elemento $A_{i,j}$. Para más sencillez, utilizaremos la siguiente notación para acceder al elemento $A_{i,j}$, por ejemplo, para asignarle un valor:

```
A( i, j ) = 4.1;
```

Evidentemente, lo anterior no es notación C. Para que funcione, es necesario que estén definidas las macros correspondientes, en este caso:

```
#define A(i,j)          A[ (j) + ((i)*(n)) ]
```

El fichero `SimpleMatrixSum.cu` contiene las macros necesarias ya definidas. Hay que prestar atención a esta definición, es decir, si quisiéramos que las matrices estuviesen almacenadas "por columnas", por ejemplo, porque queremos utilizar las bibliotecas BLAS/LAPACK, la macro tendría el siguiente aspecto:

```
#define A(i,j)          A[ (i) + ((j)*(m)) ]
```

Esta manera de trabajar es cómoda pero tiene inconvenientes ya que debe haber coherencia entre la longitud de las filas (columnas) de la matriz declarada y de la definición de su macro correspondiente, lo que suele generar errores. También es necesario declarar una macro por matriz.

A continuación realizaremos los siguientes pasos sobre el fichero anterior:

1. La función principal se encuentra implementada. Pasamos a la función `cu_matrix_sum` que contiene la construcción de la infraestructura necesaria para llamar al kernel que resolverá el problema: `compute_kernel`. Implementamos las partes indicadas: reserva de memoria, transferencia de datos, cálculo de la malla de bloques y llamada al kernel. Es importante observar que la malla de bloques de threads debe "cubrir" completamente el espacio de `m`×`n` elementos que ocupan las matrices a sumar.

2. Implementación del kernel. En este punto hay que tener en cuenta que un thread tiene dos coordenadas dentro del bloque bidimensional de threads (`threadIdx.x` y `threadIdx.y`) y, además, cada bloque de threads tiene sus coordenadas dentro de la malla bidimensional de bloques de threads (`blockIdx.x` y `blockIdx.y`). Teniendo en cuenta que las dimensiones de los bloques son `blockDim.x` y `blockDim.y` para las dos dimensiones, respectivamente, podemos deducir fácilmente que cada thread tiene acceso a un elemento de la matriz `d_A` (o a un elemento de las otras dos, ya que todas se han almacenado de la misma manera en la GPU) utilizando, por ejemplo, las siguientes coordenadas:

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
int j = threadIdx.y + blockDim.y * blockIdx.y;
```

Una vez implementado y probado el programa, hay que pasar al análisis correspondiente para ver si el alineamiento en memoria que hemos utilizado es adecuado. Este análisis se puede realizar con `computeprof` o con `nvprof`. Tomamos nota del tiempo que tarda en ejecutarse el kernel.

A continuación realizamos otra implementación del mismo kernel, esta vez ocupándonos de que los threads que tienen posiciones consecutivas en la dirección `x` accedan a posiciones consecutivas en memoria. Por ejemplo, el thread cuyo valor `threadIdx.x=17` debe acceder a la posición siguiente en memoria a la que accede el thread `threadIdx.x=16`. Una vez realizado esto se debe volver a analizar la aplicación con el profiler para ver si el tiempo de ejecución del kernel ha cambiado.

## Ejercicio 3

El siguiente ejercicio sirve de repaso. Se parece mucho al primer ejercicio. La idea aquí consiste en implementar la operación `saxpy`. Esta operación viene definida en la biblioteca BLAS y se define de la siguiente manera. Dados dos vectores $x, y \in \mathbb{R}^n$ y un escalar $a \in \mathbb{R}$ la operación `saxpy` tiene la forma

$$y = a \cdot x + y.$$

El fichero `cu_saxpy.cu` contiene el esqueleto necesario para realizar la implementación de dicha operación en GPU. Dado que se trata de dos vectores lineales (1D) no va a ser necesario crear una malla bidimensional de bloques rectangulares de threads, será suficiente con crear bloques de threads 1D y una malla 1D de bloques de threads.

## Ejercicio 4

Este ejercicio resuelve el mismo problema que el ejercicio anterior. En este caso, vamos a tratar la situación en la que la malla de threads no "cubre" todo el tamaño de los vectores. El código facilitado (`cu_saxpy1.cu`) pide tanto el tamaño de bloque de threads como el tamaño de malla o número de bloques. El número total de threads (`blockDim.x*gridDim.x`) puede no ser suficiente para que cada thread pueda tener asignado una posición de los vectores, tal como sucedía antes. La solución consiste en que cada thread se encargará de acceder a las posiciones:

```
threadIdx.x + blockId.x + blockIdx.x * blockDim.x + blockDim.x * gridDim.x
threadIdx.x + blockId.x + blockIdx.x * blockDim.x + 2*blockDim.x * gridDim.x
threadIdx.x + blockId.x + blockIdx.x * blockDim.x + 3*blockDim.x * gridDim.x
...
```

de los vectores para realizar la operación *saxpy* con esas componentes. Esto significa también que el kernel tendrá un bucle. La parte interesante de este ejercicio es averiguar la forma que tendrá dicho bucle.

## Ejercicio 5

El siguiente ejercicio también es sencillo aunque no tanto como parece. La idea aquí consiste en implementar la operación *dot product* o producto escalar. Esta operación viene definida en la biblioteca BLAS y se define de la siguiente manera. Dados dos vectores $x, y \in \mathbb{R}^n$ la operación `dot` tiene la forma

$$a = \sum_{i=0}^{n-1} x_i \cdot y_i.$$

donde $a$ es un escalar, $a \in \mathbb{R}$.

Hasta ahora el tamaño de la salida (cantidad de datos del resultado) ha sido siempre el mismo que el de la entrada. Eso ha facilitado bastante las cosas a la hora de paralelizar un algoritmo. Sin embargo, ahora nos enfrentamos al hecho de que el resultado tiene tamaño 1, es decir, es un escalar, mientras que la entrada es de tamaño $n$. Esto, que en paralelismo se le conoce con el nombre de *reducción*, supone un problema tanto de implementación como de eficiencia.

En un principio vamos a adoptar una solución "para salir del paso" utilizando las herramientas de las que disponemos. La solución pasará por implementar los dos kernels siguientes que podemos encontrar en el código facilitado (`cu_dot.cu`):

1. `compute_kernel1`: Este kernel, cuya signatura se puede ver en el fichero facilitado, recibe el tamaño de los vectores, los vectores a multiplicar y un vector de 32 elementos. El kernel será llamado de manera fija por una malla formada por un solo bloque de threads. Este bloque será 1D de 32 threads. La forma más fácil de implementar la solución de este kernel es aquella en la que cada thread se va a encargar de realizar la siguiente operación:

   $$a_t = x_t \cdot y_t + x_{32\cdot 1+t} \cdot y_{32\cdot 1+t} + x_{32\cdot 2+t} \cdot y_{32\cdot 2+t} + \ldots + x_{32\cdot i+t} \cdot y_{32\cdot i+t} + \ldots ,$$

   siendo $t$ el identificador del thread, o sea, `threadIdx.x`. Se calcularán términos mientras se cumpla $32i + t \leq n$. Cada thread guardará el resultado calculado, $a_t$, en una posición del vector `v` pasado como argumento, es decir, `v[threadIdx.x] = a;`.

2. `compute_kernel2`: El kernel anterior ha realizado un cálculo parcial que se encuentra almacenado en las entradas del vector `d_v`, un vector para el que se ha reservado memoria en la GPU. Esto se ha realizado así porque, al menos hasta ahora, no sabemos cómo

comunicar datos entre los threads de un bloque. Lo que sí sabemos es que los datos en memoria de la GPU son persistentes entre llamadas a kernels diferentes, es decir, mientras dura la ejecución del programa. El kernel que proponemos aquí se va a encargar de sumar los valores del vector `d_v` y devolver el resultado, que será el resultado de la operación producto escalar.

Este kernel será llamado por una malla de bloques de threads consistente en un solo bloque, igual que antes, pero ahora el bloque de threads estará formado por un solo thread. El segundo argumento del kernel será un vector de un solo elemento, que habrá sido creado antes de la llamada a dicho kernel como cualquier otro vector. El kernel se limitará a sumar los elementos del vector `v` y asignar el resultado a ese último vector de un solo elemento (`d_result`). De momento esto lo hacemos así dado que los kernels no pueden devolver un valor, siempre devuelven `void`.

## Ejercicio 6

Este ejercicio es el mismo que el anterior pero ahora, la idea es racionalizar el uso de la variable `d_result` ya que, tratándose de una variable simple, no tiene sentido tratarla como un vector y reservar espacio en memoria dinámicamente. Lo primero que haremos es copiar el archivo `cu_dot.cu` en el archivo `cu_dot1.cu` para conservar ambos en caso de que se pidan.

1. La modificación consiste en declarar una variable en espacio global de la siguiente manera:

   ```
   __device__ float d_result;
   ```

   lo que implica que la variable anterior `d_result` debe desaparecer allí donde se había utilizado.

2. Esta nueva variable la vamos a inicializar a 0 (aunque no es necesario en este caso, pero así vemos cómo se hace). Para ello, utilizaremos la función `cudaMemcpyToSymbol`.

3. Seguidamente actualizamos la utilización de la variable dentro del kernel correspondiente. Ahora no se recibirá dicha variable como argumento ya que puede accederse como variable global que es.

4. Una vez calculado el valor de la variable `d_result` hay que devolver su valor a la CPU mediante la utilización de la función `cudaMemcpyFromSymbol`.

## Ejercicio 7

Siguiendo con el ejemplo anterior, ahora vamos a copiar el fichero `cu_dot1.cu` en el `cu_dot2.cu`, y vamos a realizar las siguientes modificaciones:

1. De la misma manera que hemos creado un variable simple en la memoria del dispositivo (`d_result`) podemos también declarar un vector, en este caso, el vector `d_v` con 32 elementos. Esto se puede hacer porque se trata de un vector de tamaño constante, conocido en tiempo de compilación. Esta vez no es necesario que inicialicemos el vector `d_v` a ningún valor en particular. Modificamos consecuentemente los kernels que acceden a dicho vector, haciendo desaparecer el vector anterior. Observaremos que, ahora, el segundo kernel no tiene argumentos.

2. La segunda modificación consiste en unir los dos kernels en uno solo. Siempre va a ser más eficiente tener el número mínimo de kernels si esto es posible ya que la llamada a un kernel desde el host tiene coste. La idea es copiar el código del `compute_kernel2` en el `compute_kernel1`. Hay que tener en cuenta que el nuevo código que hemos introducido en el primer kernel solo ha de hacerlo uno de los threads, pongamos que es el `threadIdx.x=0`.

(No estaría de más mirar el cronograma de los ejercicios anteriores para ver el coste temporal.)

## Ejercicio 8

Este ejercicio es trivial en comparación con los anteriores pero servirá para comenzar a trabajar con la memoria compartida. Se realizará en el fichero `cu_dot3.cu` que generaremos copiando de `cu_dot2.cu`. Se trata de "mover" la declaración del vector `d_v` como vector en la memoria del dispositivo a la memoria compartida del kernel, es decir, realizar la siguiente declaración

```
__shared__ float d_v[32];
```

dentro del kernel. Eso es todo para este ejercicio aunque hay que tener en cuenta que, cuando se escribe en la memoria `__shared__` suele ser necesario sincronizar los threads mediante una llamada a la función `__syncthreads()`, ya que un *warp* en ejecución puede acceder a dicha memoria "antes" de que esta haya sido escrita por otro *warp*. Por lo tanto, aunque aquí no es necesario dado que solo trabajamos con un *warp*, sería recomendable introducir dicha sincronización después de que los 32 threads escriban en el vector `d_v` y justo antes de que el thread 0 realice la suma de los 32 elementos de `d_v`.

## Ejercicio 9

En este ejercicio seguimos trabajando con el mismo problema, el producto escalar de dos vectores, pero ahora va a ser algo más complicado ya que queremos utilizar una malla con más bloques de threads, no solo uno.

Ahora vamos a trabajar con un kernel al que llamaremos así:

```
compute_kernel<<< nblocks, BLOCKSIZE >>>( n, d_x, d_y, d_v );
```

donde `nblocks` se corresponde con la cantidad de bloques necesarios para "cubrir" el tamaño de los vectores, `n`, siendo `BLOCKSIZE` igual a 32. Este último valor es una constante definida en el programa y es usual hacerlo así puesto que la memoria *shared* que se declara ha de ser constante (no puede ser dinámica) y su tamaño suele ser una función lineal del tamaño de bloque de threads, como es el caso. Como antes, `d_x` y `d_y` serán los vectores a multiplicar, mientras que `d_v` será un vector auxiliar, creado igual que los anteriores, en memoria del dispositivo y de tamaño `nblocks`. La esencia de este vector consiste en lo siguiente: cada bloque de threads calculará un resultado parcial del producto escalar, que es un escalar, y lo guardará en la posición correspondiente de este vector. Hay que tener en cuenta que los bloques de threads no pueden comunicarse entre sí, salvo a través de la memoria del dispositivo, por esa razón necesitamos dicho vector.

Para este ejercicio contamos con un código esqueleto que podemos utilizar para rellenarlo (`cu_dot4.cu`). Sean $x$ e $y$ los vectores a multiplicar, $s$ un vector de 32 elementos declarado en memoria *shared* y $v$ un vector de tamaño `nblocks` declarado en memoria del dispositivo, para la implementación del kernel se sugiere el siguiente código:

**Algoritmo 1** Productor escalar.

1: $threadIdx \leftarrow \ldots$ ▷ Índice del thread dentro del bloque de threads.
2: $blockIdx \leftarrow \ldots$ ▷ Índice del bloque dentro de la malla de bloques.
3: $i \leftarrow \ldots$ ▷ Índice del vector $x$ e $y$ que va a multiplicar este thread.
4: **if** $i < n$ **then**
5: $s_{threadIdx} \leftarrow x_i \times y_i$
6: `__syncthreads()`
7: **if** $threadIdx = 0$ **then**
8: Sumar los 32 elementos de $s$ en la variable $a$.
9: $v_{blockIdx} \leftarrow a$
10: **if** $blockIdx = 0$ **then**
11: Sumar los `nblocks` elementos de $v$ en la variable $a$.
12: `d_result` $\leftarrow a$
13: **end if**
14: **end if**
15: **end if**

Puede observarse en el código anterior que todos los threads realizan la operación de la línea 5. Después, solo los threads con índice 0 realizan la operación de las líneas 7–14, que consiste en sumar los valores de la memoria *shared* (línea 8) y guardar el resultado en la memoria del dispositivo (línea 9). Finalmente, solo el thread 0 del bloque 0 (líneas 10–13) suma los `nblocks` elementos del vector $v$ y lo guarda en la variable global `d_result`.

Una vez implementado el código hay que probarlo varias veces y con tamaños de vector diferentes y preferiblemente grandes. Observaremos que para un mismo tamaño de vector el error que devuelve puede diferir de una ejecución a otra. También veremos valores excesivamente grandes para el valor del error. Eso indica que el programa no funciona bien. Es importante realizar una reflexión acerca de por qué esto es así antes de avanzar. De momento, lo dejamos ahí.

## Ejercicio 10

En este ejercicio vamos a implementar un kernel que realice una transposición de matrices en la GPU. Dada la matriz $A \in R^{m \times n}$ el kernel obtiene la matriz $B \in R^{n \times m}$ tal que

$$B = A^T \ ,$$

esto es, una transposición "out of place".

El código proporcionado (`MatrixTransposition_v1.cu`) lee los valores $m$ y $n$. A continuación, transpone la matriz en la CPU y almacena la transpuesta en otra. La rutina `cu_transpose` hace todo lo necesario para llamar al kernel. Antes de llamar al kernel, se forma una malla bidimensional de bloques cuadrados de threads (Código 1).

```
1 // Calculate blocks grid size
2 int blocks_row = ( m + BLOCKSIZE - 1 ) / BLOCKSIZE;
3 int blocks_col = ( n + BLOCKSIZE - 1 ) / BLOCKSIZE;
4 // Execute the kernel
5 dim3 dimGrid( blocks_col, blocks_row );
6 dim3 dimBlock( BLOCKSIZE, BLOCKSIZE );
7 compute_kernel<<< dimGrid, dimBlock >>>( m, n, d_A, d_B );
```

Código 1: Llamada al kernel de transposición de matriz.

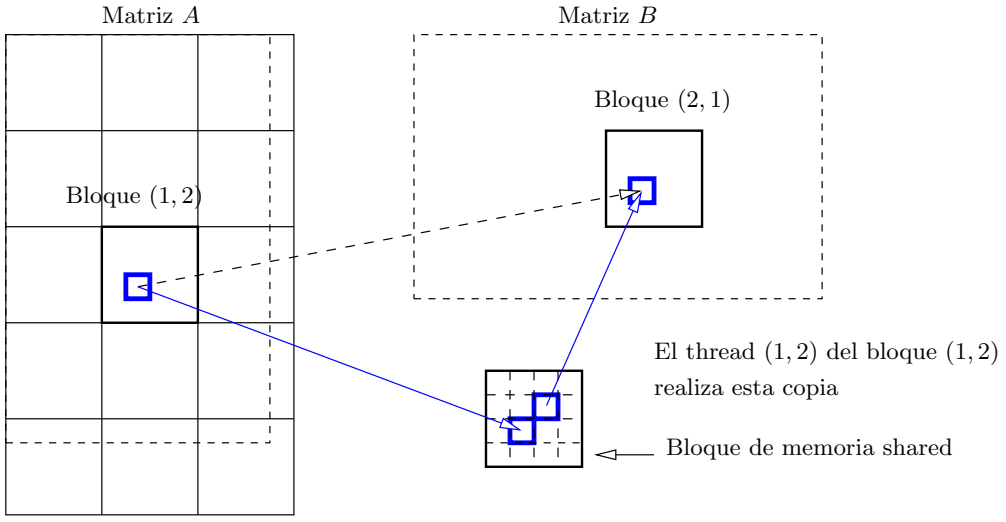


Figura 1: Ejemplo de transposición de una matriz $A$ de $17 \times 11$ a una matriz $B$ de $11 \times 17$ en una GPU utilizando memoria compartida. El rectángulo formado por líneas punteadas representa la matriz. Un thread (Thread$(1,2)$) de un bloque (Block$(1,2)$) copia un elemento de la matriz $A$ ($A(11,6)$) en un elemento de la matriz $B$ ($B(6,11)$) a través de la memoria compartida.

Para implementar el kernel sigue los pasos siguientes:

1. Obtener el índice `i` global de una fila de la matriz `d_A`.
2. Obtener el índice `j` global de una columna de la matriz `d_A`.
3. Copiar el elemento `d_A(i,j)` en `d_B(j,i)` para formar la matriz transpuesta. Hay que tener cuidado de no acceder a elementos más allá de los límites de la matriz.

El código que se ha implementado no accede correctamente a las posiciones de memoria de ambas matrices (puede que acceda correctamente a unas direcciones pero no a las de otra). Tomad nota del tiempo de transposición de una matriz cuadrada de tamaño 8192 para tener una referencia[1].

Ahora, vamos a utilizar una versión diferente que permita acceder a elementos consecutivos de ambas matrices. Para ello, tenemos que utilizar memoria *shared*.

La nueva versión se implementará en el fichero `MatrixTransposition_v2.cu`. Sigue los siguientes pasos:

1. Obtener las coordenadas $(x, y)$ del thread en ambas dimensiones dentro del bloque.
2. Guardar los índices del bloque en dos variables, por ejemplo, `BLOQUE_X` y `BLOQUE_Y`.

Observar la Figura 1 para entender los siguientes pasos. En esta versión, utilizaremos un valor constante para el tamaño de bloque `BLOCKSIZE`. Esto es porque vamos a utilizar ese valor para declarar memoria compartida y su tamaño de asignación debe ser conocido en tiempo de compilación. Como se ve, para simplificar se está utilizando un bloque cuadrado. A la izquierda de la figura podemos ver la matriz $A$. Supongamos que un bloque de threads está procesando el bloque de matriz $(1,2)$. Como las matrices se almacenan por filas, utilizaremos la primera dimensión del bloque de threads (`blockIdx.x`) para acceder a los elementos de fila de $A$. Con el

[1] Por ejemplo, `nvprof ./MatrixTransposition_v1 16384 16384`.

fin de lograr accesos a memoria *coalescentes*, el mismo bloque de threads almacenará el bloque $(2, 1)$ de la matriz $B$. Obsérvese que una entrada de bloque de $A$ ($A(i, j)$) y la misma entrada en $B$ ($B(i, j)$), ambas deben ser gestionadas por el mismo thread dentro del bloque de threads para poder acceder a ambos elementos de la matriz por filas (como si se tratara de una mera copia sin transposición). Para transponer los elementos de ese bloque, utilizaremos una copia intermedia en memoria compartida.

Continúa con los pasos siguientes. Ten en cuenta que el índice de columna debe recorrer entradas consecutivas de la matriz y que la diferencia entre la matriz $A$ y $B$ cae en el índice de bloque de threads.

3. Obtener el índice global a una fila de $A$ (llamarlo `r_A`).
4. Obtener el índice global de una columna de $A$ (llamarlo `c_A`).
5. Obtener el índice global de una fila de $B$ (llamarlo `r_B`).
6. Obtener el índice global de una columna de $B$ (llamarlo `c_B`).
7. Declarar un bloque de memoria compartida de dimensión `BLOCKSIZE`×`BLOCKSIZE`.
8. Dentro de una cláusula `if` que impide acceder a los elementos de $A$ más allá de los límites $m \times n$, escribir la copia del elemento correspondiente de $A$ a $B$ a través de memoria compartida. Recuerda sincronizar los threads una vez que los datos se han guardado en la memoria compartida (`__synchthreads()`).

Utiliza el profiler de CUDA (p.e. `nvprof`) de nuevo para analizar el tiempo de ejecución y comparar con la versión anterior. Ten en cuenta que, por simplicidad, vamos a utilizar siempre un tamaño de matriz que sea múltiplo de `BLOCKSIZE`.

Para terminar, haced la prueba de declarar memoria *sin conflicto de acceso a bancos de memoria*.

## Ejercicio 11

Vamos ahora a retomar el Ejercicio 9. Deberíamos haber llegado a la conclusión de que el problema está en el acceso a los datos en la memoria del dispositivo, que este acceso se está realizando de manera concurrente y, por tanto, hay kernels que acceden a datos que todavía no han sido escritos debidamente en esta memoria por parte de otros kernels. Por lo tanto, hay que "esperar" a que todos los kernels escriban sus datos en memoria antes de proceder a leer dichos datos con objeto de realizar la suma de los mismos.

La solución es sencilla: dividir el kernel implementado en el Algoritmo 1 en dos kernels diferentes, uno para calcular los resultados parciales y almacenarlos en la memoria del dispositivo (lineas 5–9) (lo llamaremos `compute_kernel1`) y otro para realizar la suma de dichos números y calcular la suma final (lineas 10–13) (lo llamaremos `compute_kernel2`).

La signatura y estructura de bloques de threads del primer kernel debería permanecer intacta mientras que, para el segundo kernel, es suficiente con que el trabajo lo haga un thread de un bloque, es decir, el kernel sería llamado así:

```
compute_kernel2<<< 1, 1 >>>( nblocks, d_v );
```

Ciertamente, es discutible la implementación de un kernel que va a ser ejecutado por un solo thread de los miles que tiene la GPU. Puede ser más productivo enviar los datos a la CPU para terminar la operación allí. Esto depende también de las operaciones que vayan a hacerse después. Por el momento nos interesa hacerlo así y seguir avanzando.

Para no dejar ningún código erróneo en nuestra base de datos de códigos CUDA realizaremos la modificación sobre el mismo código `cu_dot4.cu`.

## Ejercicio 12

El siguiente ejercicio, que trata otra vez del producto escalar, lo vamos a realizar copiando el código `cu_dot4.cu` sobre `cu_dot5.cu`. En lugar de utilizar la memoria *shared* vamos a utilizar la capacidad que tienen los threads de acceder a registros de otros threads. Esta capacidad se denomina *Warp-Level primitives* y está formada por un API sencilla compuesta por una serie de rutinas que pueden consultarse aquí y en la *CUDA Programming Guide*.

La rutina que vamos a utilizar aquí es `__shfl_down_sync`, de hecho, el código será exactamente el que forman la tres lineas siguientes:

```
1 #define FULL_MASK 0xffffffff
2 for (int offset = 16; offset > 0; offset /= 2)
3     val += __shfl_down_sync( FULL_MASK, val, offset );
```

El código anterior sirve para sumar valores que cada thread tiene almacenado en un registro sin necesidad de utilizar memoria *shared*. Al terminar la ejecución del código anterior el `threadIdx.x==0` tendrá la suma de todos los valores que tenía cada thread en su variable `val`.

La idea es reimplementar el kernel `compute_kernel1` adaptándolo a la utilización del código anterior. En este caso, ya no necesitamos declarar nada en memoria *shared*, solo una variable (p.e. `val`) que almacene el producto de dos entradas de los vectores `d_x` y `d_y`, como antes (linea 5 del Algoritmo 1). Justo después se introduce el código anterior para que, seguidamente, solo el thread 0 guarde el resultado en el lugar correspondiente del vector `d_v`, tal como se hacía en la versión anterior (linea 9 del Algoritmo 1).

## Ejercicio 13

Siguiendo con el ejercicio de multiplicación escalar vamos a introducir en la solución la utilización de operaciones atómicas. Realizaremos dos versiones de esta solución, partiendo del código finalizado en el Ejercicio 11 (`cu_dot4.cu`) para escribir el código `cu_dot6.cu`.

### Primera solución

En esta primera solución vamos a modificar únicamente el primer kernel (que habíamos llamado `compute_kernel1`). Declararemos una variable en memoria compartida, p. e. `sh_a`, y utilizaremos la función `atomicAdd` para actualizarla, sumando el producto de dos entradas de los vectores a multiplicar. En realidad se trata simplemente de sustituir el vector declarado en memoria compartida por una única variable que será actualizada por todos los threads concurrentemente. Con esto podemos observar una mejora en la simplicidad del código con respecto a las versiones anteriores contando, además, con la ventaja de ser este tipo de operaciones muy rápidas al estar implementadas por hardware.

### Segunda solución

Como alternativa a la solución anterior, ahora se trata de sustituir los dos kernels anteriores (comentarlos, no borrarlos) por un kernel que realiza lo mismo que en el caso anterior pero aplicando la operación atómica directamente sobre la variable `d_result`, aprovechando que las operaciones atómicas pueden aplicarse tanto a variables en memoria compartida como en la memoria del dispositivo.

## Ejercicio 14

Volvemos otra vez al ejercicio de multiplicación escalar para probar cómo utilizar *paralelismo dinámico* con este ejemplo. Partimos de la solución al Ejercicio 11, que hemos almacenado en el fichero `cu_dot4.cu`, y copiamos dicho código en el fichero `cu_dot7.cu`.

La idea es evitar el tener que utilizar dos llamadas a kernels, con lo que esto significa, es decir, costes de sincronización. Ahora realizaremos una llamada a un solo kernel, que será así:

```
compute_kernel<<< 1, 1 >>>( n, d_x, d_y, nblocks, d_v );
```

Observamos que este kernel solo utiliza un bloque con un solo thread. También observamos que le hemos pasado un argumento más: la variable `nblocks`. Este kernel se encargará, mediante *paralelismo dinámico*, de llamar a otro kernel, llamémosle por ejemplo `child_kernel`, aunque será una copia exacta del kernel `compute_kernel1` de `cu_dot4.cu`.

La implementación del kernel "padre" (`compute_kernel`) tiene la llamada al kernel "hijo" y, después de una sincronización necesaria, se encargaría de realizar la suma de las componentes del vector `d_v`, es decir, de realizar lo mismo que `compute_kernel2`. En pocas palabras, hemos sustituido la llamada a dos kernels secuenciales desde el host por una sola llamada a un kernel desde el host que, a su vez, realiza una llamada a otro kernel de la misma manera que se lleva a cabo desde el host, pero ahorrando así llamadas desde el host y trasladando más código al dispositivo.

Para compilar esto es necesario hacerlo así:

```
nvcc -rdc=true -arch=sm_75 -o cu_dot7 cu_dot7.cu
```

## Ejercicio 15

Seguimos con el producto escalar de dos vectores para introducir la utilización de la herramienta *Cooperative Groups* de CUDA. En este ejercicio partiremos del código `cu_dot8.cu` donde podemos observar que ya está implementado el kernel principal (`compute_kernel`) que aparece a continuación:

```
1  __global__ void compute_kernel( const unsigned int n, float *d_x, float
       *d_y ) {
2
3      float mi_suma = thread_dot(n,d_x,d_y);
4
5      extern __shared__ float temp[];
6      auto g = this_thread_block();
7      float block_dot = thread_block_dot(g, temp, mi_suma);
8
9      if (g.thread_rank() == 0) atomicAdd( &d_result, block_dot );
10
11 }
```

En este kernel, cada thread calcula de manera independiente (función `thread_dot`) una suma parcial de productos

$$\texttt{mi\_suma} = \sum_{\texttt{i}} \texttt{x}_\texttt{i} \cdot \texttt{y}_\texttt{i}.$$

para determinados índices $i$, que son los indicados en el código en comentarios. Es decir, el conjunto de todos los threads "no cubre" la totalidad del tamaño de los vectores. La función `thread_dot` devuelve esta suma.

En la linea 6 de `compute_kernel` se obtiene el *manejador* o *handle* `g` para el grupo de threads que forman el bloque de threads. Dicho *manejador* será pasado a la función `thread_block_dot` que se encarga de realizar una reducción de la suma obtenida por cada uno de los threads en el paso anterior.

La implementación de la función `thread_block_dot`, que está por hacer, puede realizarse de diferentes maneras. Sabemos, por ejemplo, que si el tamaño de bloque fuese de 32 se podrían utilizar operaciones sobre registros a nivel de *warp*, tal como vimos en el Ejercicio 12. Sin embargo, vamos a realizar una reducción más genérica haciendo uso de los grupos cooperativos que, bien implementada, puede forzar a que el compilador optimice el código introduciendo por sí mismo este tipo de operaciones a nivel de *warp*. Por eso, de momento, se hará uso de la memoria *shared*, que se ha declarado fuera y se ha pasado como argumento a esta función, y servirá para realizar los intercambios necesarios entre threads. El algoritmo a implementar aquí es de tipo *recursive doubling* y es el que se especifica en el Algoritmo 2. Hay que hacer notar que se necesita una sincronización a nivel de grupo en los lugares indicados (lineas 5 y 9).

**Algoritmo 2** Reducción a una suma mediante la técnica *recursive doubling*.

```
1:  lane ← ...                          ▷ Índice del thread dentro del bloque de threads.
2:  i ← group_size                      ▷ Tamaño del bloque de threads.
3:  while i > 0 do
4:      temp_lane ← val
5:      Sincronización
6:      if lane < i then
7:          val ← val + temp_lane+i
8:      end if
9:      Sincronización
10:     i ← i/2
11: end while
12: return val                          ▷ Solo el thread 0 devuelve el valor válido de la suma final.
```

1. La longitud de los vectores puede ser mayor que el número total de threads en la malla que, al ser lineal (1D), es de tamaño `blockDim.x*gridDim.x`. El código facilitado no lo calcula así, sino de la manera "tradicional", es decir, "cubriendo" todo el vector:

   ```
   int nblocks = ( n + blockSize - 1 ) / blockSize ;
   ```

   sin embargo, podríamos reducir el número de bloques de la siguiente manera:

   ```
   int nblocks = ( n/2 + blockSize - 1 ) / blockSize ;
   ```

   o dividiendo `n` por otro valor.

2. Otro aspecto que hemos introducido en este ejercicio es el de la utilización de un tamaño variable de memoria *shared*. Este tamaño se calcula en bytes

   ```
   int sharedBytes = blockSize * sizeof(float);
   ```

   y es pasado como tercer argumento en la llamada al kernel:

   ```
   compute_kernel<<< nblocks, blockSize, sharedBytes >>>(...
   ```

   De esta manera podemos elegir en tiempo de ejecución el tamaño de los bloques de threads.