# Práctica de Programación de GPUs. Implementación de algoritmos paralelos de datos en GPU usando CUDA

En esta práctica se aborda la implementación en GPU de varios algoritmos paralelos de datos usando CUDA.

## 1 Implementación en CUDA del Algoritmo de Floyd

En la plataforma PRADO, se dispone de una implementación en CUDA del Algoritmo de Floyd para el cálculo de todos los caminos más cortos en un grafo etiquetado. El número de vértices N puede ser cualquiera. Esta implementación usa una grid unidimensional de bloques de hebras CUDA unidimensionales (con forma alargada). Se describe en la misma un kernel que se lanza una vez para cada iteración k = 0, ..., N − 1. Cada hebra es responsable de actualizar un elemento de la matriz resultado parcial A y ejecutará el siguiente algoritmo:

**Kernel para actualizar la matriz A en la iteración k**
**begin**
**A [i,j] = min(A[i,j] , A[i,k] + A [k,j] )**
**end;**

En esta versión, las hebras CUDA se organizan como una Grid unidimensional de bloques CUDA unidimensionales. Teniendo en cuenta que cada hebra se encarga de actualizar un elemento de la matriz A en la iteración k-ésima, tendríamos que tener al menos N × N hebras. Los tamaños de bloque de hebras CUDA usuales son: 64, 256 ó 1024.

**Ejercicios propuestos**

En todos los ejercicios propuestos, se exigirá que la solución obtenida con la implementación CUDA desarrollada se compare con la obtenida por el programa secuencial de referencia (que viene dado).

**1.** Modificar la implementación CUDA del algoritmo de Floyd para que las hebras CUDA se organicen como una grid bidimensional de bloques cuadrados de hebras bidimensionales. Los ejemplos de tamaños de bloque usuales son: 8 × 8, 16 × 16 y 32 × 32. Para realizar este ejercicio, se recomienda tomar como referencia, además de la implementación CUDA del algoritmos de Floyd que se os proporciona, la implementación de la suma de matrices cuadradas ya que, en este ejemplo, también se usa una grid bidimensional de bloques bidimensionales.

Realizar también medidas de tiempo de ejecución sobre los algoritmos implementados. El programa plantilla que se ofrece incluye la toma de los tiempos de ejecución del algoritmo en CPU y en GPU. Deberán realizarse las siguientes medidas para problemas de diferentes tamaños y diferentes tamaños de bloque CUDA:

(a) Medidas de tiempo de ejecución para el algoritmo secuencial (Una sola hebra en CPU): TCPU.

(b) Medidas para ambos algoritmos paralelos en GPU (habría que indicar previamente el modelo de GPU usado en los experimentos): $TGPU_{1D}$ y $TGPU_{2D}$.

(c) Ganacia en velocidad de las versiones en GPU con respecto a la versión monohebra en CPU: $SGPU_{1D}$ y $SGPU_{2D}$ , que se obtiene dividiendo el tiempo de ejecución secuencial entre el tiempo de la correspondiente versión en GPU. Las medidas deberán realizarse para diferentes tamaños de

problema (valores de N). Se presentará una tabla con el siguiente formato para cada tamaño del bloque de hebras (BSize = 64, 256, 1024).

| | **TCPU(sec)** | **$TGPU_{1D}$** | **$SGPU_{1D}$** | **$TGPU_{2D}$** | **$SGPU_{2D}$** |
|---|---|---|---|---|---|
| **N = 400** | | | | | |
| **N = 1000** | | | | | |
| **N = 1400** | | | | | |
| **N = 2000** | | | | | |

Se valorará que se obtenga también una gráfica que ilustre cómo varía la ganancia en velocidad al aumentar el tamaño del problema para cada valor de Bsize.

**2.** Extender la implementación en CUDA C desarrollada para que se calcule también la media aritmética de todos los caminos más cortos encontrados. Usar para ello un kernel CUDA de reducción aplicado al vector  que almacena los valores de la matriz resultado. Se pueden usar los kernels de reducción disponibles en las CUDA Samples:

https://github.com/nvidia/cuda-samples

o usar el kernel de reducción que se muestra en las diapositivas y en la carpeta "Ejemplo_suma_vectores_reduccion" de los códigos CUDA que se os proporciona.

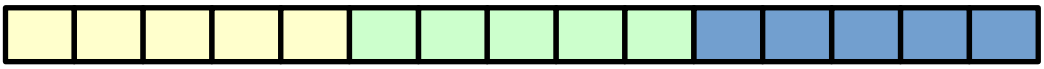
También se recomienda, al realizar la suma definitiva en el host, usar una variable destino de tipo long int para evitar desbordamientos.

## 2 Implementación CUDA de una operación vectorial

Se pretende realizar una serie de cálculos utilizando dos vectores de entrada A y B de N floats donde N es un parámetro entero positivo del problema (*A={A[0], A[1],…,A[N-1]}, B={B[0],…, B[N-1]}*) tal que N es múltiplo de un entero $M \in \{64, 128, 256\}$ ($N = NBlocks \times M$ , siendo *NBlocks* un entero positivo). Estos vectores se inicializan con los siguientes valores:

$$A[i]=1.5\frac{1 + 5i \mod 7}{1+i \mod 5}, \qquad B[i]=2\frac{2+i \mod 5}{1+i \mod 7}, \quad i=0,\ldots,N-1$$

Los vectores A y B se encuentran (solo a nivel conceptual) divididos en *NBlocks* bloques de elementos contiguos de tamaño *M*). Como ejemplo de cómo se organizan los vectores A y B, la siguiente figura muestra un vector de 15 elementos divididos en 3 grupos de 5 elementos cada uno (aquí, *M=5, NBlocks=3 y N=15*).



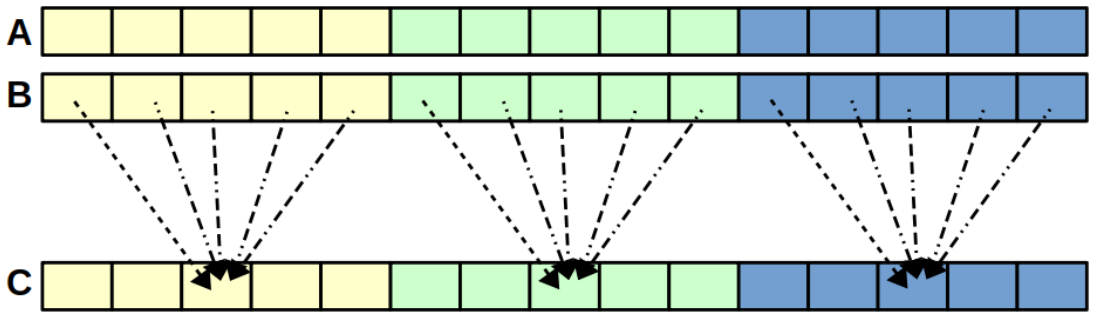
Con estos datos de entrada se calculan los siguientes resultados:

1. Un vector resultado **C** con N celdas tal que el valor C[i] se calcula de la siguiente forma:

$$C[i]=\sum_{j\in M_i}\left|\frac{\left(i\cdot B[j]-A[j]^2\right)}{(i+2)\cdot max\left(A[j],B[j]\right)}\right|$$

donde $M_i$ es el conjunto de M índices (M es el tamaño de bloque) del bloque de elementos de A y B al que pertenece el índice i. Cada índice i en el conjunto {0,…,N-1} se encuentra en el bloque j-ésimo (j ∈ {0, ..., *NBlocks*-1}) donde j=i *div* M (*div* denota la división entera).

La siguiente figura ilustra las dependencias de datos en este cálculo:



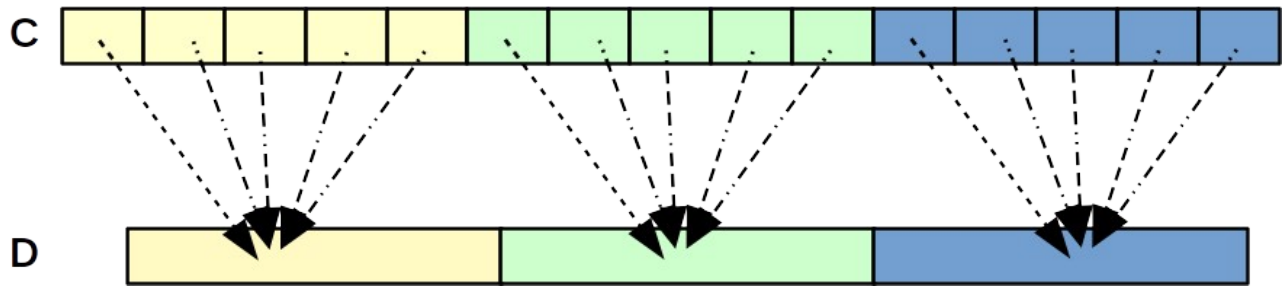

2. El máximo de todos los valores almacenados en C, independientemente de la división en bloques.

$$mx=max_{i=0,\ldots,N-1}\{C_i\}$$

3. La media aritmética de los valores de C pertenecientes a cada bloque, almacenando el resultado en un vector **D** con *NBlocks* posiciones (tantas como bloques se hayan definido):

$$D[k] = \frac{\sum_{i \in B_j} C[j]}{M}, \qquad k = 0, \ldots, NBlocks - 1$$

donde Bj es el conjunto de M índices del j-ésimo bloque.




Para que sirva de referencia, en los códigos CUDA que se proporcionan como muestra, se dispone de un programa C++ (transformacion.cc) que realiza todos estos cálculos secuencialmente en CPU.

**Ejercicios propuestos**

**1.** Se deberán realizar dos implementaciones CUDA C++: una que realiza el cálculo del vector C (primera fase de cálculo) utilizando variables en memoria compartida y otra que no utilice este tipo de almacenamiento para esta fase de cálculo. Para el resto de fases se recomienda usar un vector en memoria compartida ya que es necesario para realizar reducciones a nivel de bloque de forma eficiente en CUDA.

**2.** Se deberán comparar los resultados obtenidos, tanto en tiempo de ejecución como en la precisión de los valores obtenidos con respecto al código CPU (disponible en el archivo transformacion.cc) solo para el cálculo del vector C. Para ello, se utilizarán valores muy altos de N (N > 2000000), probando los tres tamaños de bloque indicados al inicio de este ejercicio y usando las versiones con y sin memoria compartida para el cálculo del vector C. En los tiempos tomados para el cálculo del vector C en GPU, solo se medirá el tiempo tomado en la ejecución del kernel de cálculo de C, por lo que habrá que invocar la función “cudaDeviceSynchronize()” justo después de invocar el kernel correspondiente y antes de tomar el instante de tiempo final.