Práctica 3

Implementación de algoritmos paralelos de datos en GPU usando CUDA

En esta práctica se aborda la implementación paralela del algoritmo de Floyd para el cálculo de todos los caminos más cortos en un grafo etiquetado usando el modelo de programación CUDA. Se desarrollarán igual que en prácticas anteriores dos versiones paralelas del algoritmo que difieren en el enfoque seguido para organizar las hebras.

La definición del problema a resolver (problema de los caminos más cortos) y del algoritmo a paralelizar (algoritmo de Floyd) es la misma que se dio en la práctica 1. Se aporta un código CUDA como plantilla para realizar la práctica.

```
procedure floyd secuencial
begin
I_{i,j} = A
  for k := 0 to N-1
    for i := 0 to N-1
      for j := 0 to N-1
        I^{k+1}_{i,j} = min{I^k_{i,j}, I^k_{i,k} + I^k_{k,j}}
  end;
```

Se desarrollarán dos versiones paralelas del algoritmo, tal como se hizo en la práctica 1. En cada versión, la distribución de los datos entre las hebras es de grano mucho más fino que en prácticas anteriores para aprovechar la plataforma de cómputo masivamente paralela que constituye la GPU. En este caso, igual que en la práctica 3, las hebras se comunican entre sí mediante variables compartidas pero cada hebra se encargará de calcular la actualización de un elemento de la matriz de los caminos más cortos en cada iteración del bucle de más alto nivel (iteración k-ésima). A diferencia de prácticas previas, en esta práctica, el número de vértices N puede ser cualquiera (no cumple ninguna condición de multiplicidad).

La primera versión usa una grid unidimensional de bloques de hebras unidimensionales (con forma alargada) y la segunda usa bloques de hebras cuadrados en una grid 2D.

3.1 Grid unidimensional (bloques 1D

Se diseñará un kernel que se lanzará una vez por cada iteración k. Cada hebra es responsable de un elemento de I y ejecutará el siguiente algoritmo:

```
procedure Kernel floyd paralelo 1 para actualizar I(i,j) en iteración k
```
$I_{i,j}^{k+1} = min\{I_{i,j}^{k}, I_{i,k}^{k} + I_{k,j}^{k}\}$
```
end;
```

En esta versión, las hebras CUDA se organizan como una Grid unidimensional de bloques de hebras unidimensionales. Teniendo en cuenta que cada hebra se encarga de actualizar un elemento de la matriz I en la iteración k, tendríamos que tener al menos $N \times N$ hebras. Ejemplos de tamaños de bloque usuales son: 64, 128, 256 y 512 (1024 sería viable a partir de GPUs con Compute Capability 2.0).

3.2 Uso de grids y bloques 2D

En esta versión, también se lanza un kernel por cada iteración k-ésima del algoritmo pero las hebras CUDA se organizan como una Grid bidimensional de bloques cuadrados de hebras bidimensionales. Ejemplos de tamaños de bloque usuales son: 8×8 y 16×16. (bloques 32×32 serían viables a partir de GPUs con Compute Capability 2.0).

3.3 Ejercicios propuestos

1. Implementar los algoritmos de cálculo de todos los caminos más cortos que han sido descritos previamente usando CUD C/C++. Se usará como plantilla el programa CUDA (`floyd.cu` que se encuentra en la web de descargas de la asignatura. Se debe crear una carpeta diferente para cada versión paralela (`Floyd-1` y `Floyd-2`).

2. Realizar 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:

   (a) Medidas para el algoritmo secuencial (Una sola hebra en CPU).

   (b) Medidas para el algoritmo paralelo en GPU, indicando el modelo de GPU usado.

   (c) Ganacia en velocidad de la versión GPU con respecto a la versión CPU. Las medidas deberán realizarse para diferentes tamaños de problema.

   Se presentará una tabla con el siguiente formato pero para diferentes tamaños del bloque de hebras.

Tiempo	*CPU* (secuencial)	GPU	Ganancia
$n = 400$			
$n = 800$			
$n = 1200$			
$n = 1600$			

   Obtener también una gráfica que ilustre como varía la ganancia en velocidad al aumentar el tamaño del problema