

## Práctica 3

# 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.


### 1.1 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 (no cumple ninguna condición de divisibilidad). 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  $k = 0, \dots, N - 1$ . Cada hebra es responsable de actualizar un elemento de la matriz resultado parcial  $A^{(k+1)}$  y ejecutará el siguiente algoritmo:

```
Kernel Actualizacion A(i,j) en iteración k
   $A_{i,j}^{k+1} = \min\{A_{i,j}^k, A_{i,k}^k + A_{k,j}^k\}$ 
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  $A$  en la iteración  $k$ , tendríamos que tener al menos  $N \times N$  hebras. Los tamaños de bloque de hebras CUDA usuales son: 64, 128, 256, 512 ó 1024.

#### 1.1.1 Ejercicios propuestos

-  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:  $8 \times 8$ ,  $16 \times 16$  y  $32 \times 32$ . 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 tamaño y diferentes tamaño de bloque CUDA:

- Medidas de tiempo de ejecución para el algoritmo secuencial (Una sola hebra en CPU):  $T_{CPU}$ .
- Medidas para ambos algoritmos paralelos en GPU (habría que indicar previamente el modelo de GPU usado):  $T_{GPU_{1D}}$  y  $T_{GPU_{2D}}$ .
- Ganancia en velocidad de las versiones en GPU con respecto a la versión monohebra en CPU:  $SGPU_{1D}$  y  $SGPU_{2D}$ . Las medidas deberán realizarse para diferentes tamaños de problema.



Se presentará una tabla con el siguiente formato para cada tamaño del bloque de hebras ( $BSize = 64, 128, 256, 512, 1024$ ).

	$T_{CPU}$ (sec)	$T_{GPU_{1D}}$	$SGPU_{1D}$	$T_{GPU_{2D}}$	$SGPU_{2D}$
$n = 400$					
$n = 1000$					
$n = 1400$					
$n = 2000$					

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  $BSize$ .

- Extender la implementación en CUDA C desarrollada para que se calcule también la longitud del camino de mayor longitud dentro de 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:

<http://docs.nvidia.com/cuda/cuda-samples/index.html>

## 1.2 Implementación CUDA de una operación vectorial

Se pretende realizar una serie de cálculos utilizando dos vectores de entrada  $A$  y  $B$ . Estos vectores podrán ser de longitud variable  $N$  ( $A[i], B[i] \quad i = 0, \dots, N - 1$  y contendrán números en coma flotante. Los vectores  $A$  y  $B$  se encuentran (solo a nivel conceptual) divididos en bloques de elementos contiguos de igual tamaño. El tamaño de bloque puede ser de 64, 128 ó 256 elementos ( $N = k \times M$ , donde  $M \in \{64, 128, 256\}$  y  $k \in \mathbb{N}$ ). Como ejemplo de cómo se organizan los vectores  $A$  y  $B$ , la figura 1.1 muestra un vector de 15 elementos divididos en 3 grupos de 5 elementos cada uno.

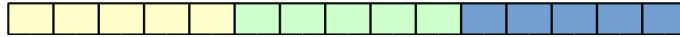


Figura 1.1: Organización de los vectores de entrada para 3 bloques de 5 elementos

Con estos datos de entrada se calcula:

- Un vector resultado  $C$  con  $N$  celdas tal que el valor  $C[i]$ ,  $i = 0, \dots, N - 1$ , se calcula de la siguiente forma:

$$C[i] = \sum_{j \in M_i} (A[j] \cdot i + B[j]), \quad \text{Si } \text{ceil}(A[j] \cdot i) \text{ es par.}$$

$$C[i] = \sum_{j \in M_i} (A[j] \cdot i - B[j]), \quad \text{Si } \text{ceil}(A[j] \cdot i) \text{ es impar.}$$

donde  $M_i$  es el conjunto de  $M$  índices ( $M$  es el tamaño de bloque) al que pertenezca la posición  $i$ .

2. La suma de los valores de  $C$  pertenecientes a cada bloque, almacenando el resultado en un vector  $D$  con tantas posiciones como bloques se hayan definido:

$$D[j] = \sum_{i \in B_j} C[i], \quad j = 1, \dots, k$$

donde  $B_j$  es el conjunto de  $M$  índices del  $j$ -ésimo bloque y  $k$  es el número de bloques considerado.

3. El valor máximo  $mx$  de todos los valores almacenados en  $C$ , independientemente de la división en bloques.

$$mx = \max\{C[i], \quad i = 0 \dots N - 1\}.$$

En la plataforma PRADO se dispone de un programa C++ (`transformacion.cc`) que realiza estos cálculos secuencialmente en CPU.

### 1.2.1 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 memoria compartida.
2. Se deberán comparar los resultados obtenidos, tanto en tiempo de ejecución como en los valores obtenidos con respecto al código disponible para CPU. Para ello, se utilizarán valores altos para  $N$  ( $N > 20000$ ), probando los tres tamaños de bloque indicados al inicio de este ejercicio y las versiones con y sin memoria compartida para el cálculo del vector  $C$ .