# PRÁCTICA 3:

Versión paralelizada con CUDA del programa “Simulación de bombardeo de partículas”.

Autores: Grupo 3
Blanco de la Cruz, Luis
González Ruiz, Rubén

## 1) Desarrollo de la práctica

En un primer lugar, lo que hicimos fue identificar las operaciones susceptibles de paralelizar: la actualización, la relajación y la comparación de máximos. Por la manera de trabajar de CUDA, teníamos que reducir la transferencia de datos ente el Host y el Device ya que es la operación más costosa.

Hemos ido trabajando en distintas versiones en las cuales íbamos añadiendo más operaciones de paralelización, comenzando por una que tuviera únicamente la función "actualiza" paralelizada, hasta la última donde tenemos paralelizadas las tres últimas operaciones.

Para las versiones siempre hemos utilizado un tamaño de bloque de 512 o 256 hilos y el tamaño de grid se calcula de esta manera:

```
if (layer_size % tamBlock == 0) {
      tamGrid = layer_size/tamBlock;
} else {
      tamGrid = (layer_size/tamBlock)+1;
}
```

Dentro de los kernel, la fórmula que utilizamos para calcular el id global de cada hilo es esta:

```
int id = (threadIdx.x + blockIdx.x * blockDim.x) + (threadIdx.y + blockIdx.y * blockDim.y) * blockDim.x * gridDim.x;
```

**Versión 1:** Comenzamos paralelizando la función "actualiza" transformándola en un kernel. Reservamos espacio [3] en el Device para dos vectores nuevos (layerDevice y layer_copyDevice) y añadimos dos memcpy para pasar los valores de dichos vectores al vector layer del host. En esta primera versión, realizamos el paso de datos desde Host a Device y viceversa porque hacemos a relajación de capas de manera secuencial después [2].

Inicialmente no obtenemos los resultados correctos porque estábamos haciendo las transferencias iniciales fuera del bucle de tormentas. Al mover memcpy dentro del bucle obtuvimos la primera versión funcional, que tarda 1m 35 seg.

**Versión 2:** Una vez que ya tenemos la primera versión funcional, nos pusimos a trabajar con la siguiente parte a reducir: la relajación. Antes de esto, estuvimos probando distintos tamaños de bloque para ver cual tardaba menos. Los que mejor resultado nos dieron fueron 256-512 hilos por bloque.

Creamos un kernel de relajación, en el cual hacíamos la copia de layerDevice a layer_copyDevice y la relajación. Conseguimos una versión funcional que bajaba el tiempo a 1m 2 seg pero que tenía condiciones de carrera.

Estas venían dadas porque entre la copia de datos de layerDevice a layer_copyDevice y la relajación (la media entre el valor de una posición del vector y sus vecinos) poníamos una sincronización de hilos para evitar condiciones de carrera. En parte, la idea era buena, no dejar realizar la relajación hasta no haber copiado todos los valores, pero el problema es que la sincronización de hilos se hace a nivel de bloque. Todos los hilos que pertenezcan al mismo bloque se actualizarán bien, pero puede ser que el último hilo de un bloque y el primero del siguiente no realicen sus operaciones cuando deben, produciendo errores en el resultado.

**Versión 3:** Para esta versión teníamos que eliminar las condiciones de carrera y paralelizar la comparación de máximos. Para lo primero, creamos un kernel llamado "copia" que lo único que hace es copiar layerDevice en layer_copyDevice, ya que todos los hilos se sincronizan para cada lanzamiento de kernel. Con esto, eliminamos las condiciones de carrera. Nos dimos cuenta también, que los vectores layer y layercopy del Host no los vamos a utilizar. Por lo que elimanos de la parte secuencial la inicialización a ceros de ambos vectores y lo realizamos directamente en el device [4]. Al hacerlo, eliminamos una transferencia también, que es una de las operaciones más costosas para cuda. Para esto último, hicimos un kernel llamado "inicializa" que rellena el vector de ceros antes del bucle de tormentas.

Con estos cambios, bajamos del minuto en el leaderboard, pero no era suficiente para pasar la Ref1. Nos quedaba aún paralelizar la búsqueda de máximos.

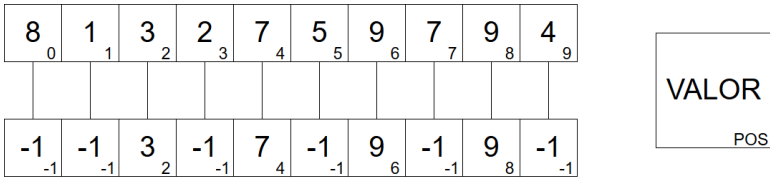
**Versión 4:** Procedemos a la paralelización durante la búsqueda de máximos. Primeramente, estábamos obcecados en intentar paralelizar el código dado directamente, sin resultado alguno.

Decidimos cambiar la estrategia y dividir el proceso en dos partes diferenciadas:

- Búsqueda de todos los candidatos a máximo (máximos locales).
- Reducción de los máximos locales hasta hallar el mayor de ellos.

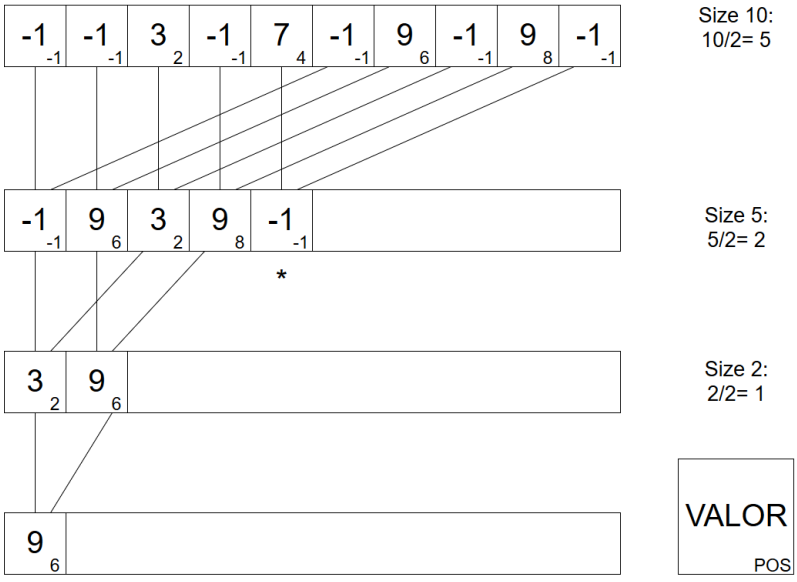
Para ello necesitaremos a mayores otros dos vectores del mismo tamaño que layerDevice. En uno almacenaremos los máximos, y en otro las posiciones de los mismos ya que son los dos datos que nos interesan.

En la primera parte, buscaremos los máximos locales (aquellas posiciones del vector que sean mayores que sus vecinos sin ser los extremos del vector) y escribiremos su valor en el vector de máximos en la misma posición. En el vector de posiciones, en la misma posición que en máximos, escribiremos la posición en la que se encuentra el máximo. En el resto de posiciones de ambos vectores, pondremos -1 como bandera. No nos interesan esos valores ni sus posiciones.




En la segunda parte, trabajaremos sobre el vector de máximos locales para hallar el máximo absoluto utilizando una operación de recursión. Iremos comparando los valores dos a dos y quedándonos con el valor más alto entre ellos. Existen varias opciones de comparar los valores, nos quedamos con la que compara un elemento con aquel situado n posiciones más adelante (siendo n la mitad del tamaño del vector). Escogemos esta opción porque es más rápida, por ejemplo, que la compara un elemento con su siguiente porque traemos directamente una línea de caché con valores consecutivos en vez de varias líneas para traer valores salteados, es decir tenemos en cuenta la coalescencia.

Para el proceso de reducción debemos tener en cuenta no solo el máximo, sino también la posición. Aunque es difícil ya que los valores float tienen bastantes decimales, en caso de que hubiera que comparar dos valores exactamente iguales nos quedaríamos con aquel que tuviera menor posición (aquel que en secuencial hubiera aparecido antes). En tamaños de vector impares el primer hilo es el encargado de hacer una comparación a mayores (*).




Con todos los cambios anteriores y un tamaño final de bloque de 256, conseguimos pasar la Ref1 y tener un tiempo de 43.383s n el leaderborad.

## 2) Bibliografía

[1] NVIDIA, <<Documentacion CUDA>> [Online]. Disponible en: http://horacio9573.no-ip.org/cuda/index.html .[Accedido 20-may.-18]

[2] NVIDIA, <<cudaMemcpy>> [Online]. Disponible en: http://horacio9573.no-ip.org/cuda/group__CUDART__MEMORY_g48efa06b81cc031b2aa6fdc2e9930741.html. [Accedido 20-may.-18]

[3] NVIDIA, <<cudaMalloc>> [Online]. Disponible en: http://horacio9573.no-ip.org/cuda/group__CUDART__MEMORY_gc63ffd93e344b939d6399199d8b12fef.html#gc63ffd93e344b939d6399199d8b12fef. [Accedido 20-may.-18]

[4] Foro Colaborativo, <<Uso de datos tras enviarlos>> [Online]. Disponible en: http://campusvirtual2017.uva.es/mod/forum/discuss.php?d=26604. [Accedido 20-may.-18]