Algunas ideas para implementar actividades adicionales en CUDA

1. En `AddVectorsInto` (kernel para la suma de vectores) y `MatrixMulGPU` (kernel para el producto de matrices), investigar el tamaño del problema a partir del cual la GPU mejora los tiempos de la CPU, teniendo siempre en cuenta en la versión CUDA el tiempo que se tarda en transferir los datos en ambas direcciones. A partir de ahí, tratar de cuantificar el factor de aceleración en GPU en función del tamaño del problema.

2. En `AddVectorsInto`, alinear los 3 vectores a, b y c consecutivamente en memoria de forma que se puedan transferir en una sola sentencia de comunicación, amortizando así el coste del start-up y aprovechando mejor el ancho de banda de las comunicaciones con CPU.

3. En `AddVectorsInto`, migrar el tipo de datos de `float` a `double` y evaluar el impacto que tiene en el tiempo final el doble hecho de que ahora la computación es más lenta y las transferencias necesarias duplican el tamaño en bytes. Hacer lo mismo en el sentido inverso, es decir, simplificando la computación al migrar de `float` a `int` (NOTA: En CUDA, un entero y un float ocupan lo mismo, 4 bytes).

4. En `matrixMulGPU`, realizar el mismo cambio del tipo de dato: (1) de `float` a `double`, y (2) de `float` a `int`, e investigar si los tiempos mejoran o empeoran más o menos respecto al caso de `AddVectorsInto`. Tened en cuenta para el análisis que este kernel tiene complejidad computacional $O(N^3)$, mientras que el anterior sólo tiene O(N).

5. En `AddVectorsInto`, comparar el rendimiento de las dos versiones del kernel que hemos discutido en clase. Un lazo para cada hilo como solución más elegante y genérica que incluye los casos en los que haya más datos que hilos, y una segunda versión en la que siempre declaremos un hilo para cada dato del tamaño del problema como versión más eficiente y escalable.

6. En `AddVectorsInto`, adicionalmente sobre las versiones anteriores, implementar la estrategia del "padding" que elimina la necesidad del "if (i<N)" para los hilos porque siempre completa los datos de entrada a la potencia de dos por exceso más cercana al valor N, rellenando con valores nulos el vector.

7. En `AddVectorsInto`, comparar el rendimiento de la versión con memoria unificada respecto a la que declara explícitamente el vector en CPU y GPU `(malloc/cudamalloc)` y realiza las transferencias manualmente con `cudamemcpy`.

8. En `SAXPY` (ejercicio propuesto por el DLI al final del módulo 2), utilizar distintas estrategias de prebúsqueda en escenarios más diversos que los que plantea el DLI, y evaluar su rendimiento con el Visual Profiler. Es decir, de igual forma que en `AddVectorsInto` se plantea inicializar el kernel en CPU o GPU, anticipar resultados, etc, el decorado se puede también alterar en `SAXPY`, y seguidamente, argumentar qué prebúsquedas se podrían utilizar en cada caso.

9. Aprovechar las llamadas que pueden hacerse al hardware para conocer la arquitectura de mi GPU (ejercicio 4 del segundo módulo) para elegir la mejor configuración de bloques/malla e hilos/bloque al desplegar el paralelismo en el lanzamiento del kernel, argumentando cuál sería la mejor opción, y acompañando el análisis de resultados aportados por el Visual Profiler en el tercer módulo.

10. Plantear mejoras sobre cualquiera de los kernels de ciencia base que están resueltos en el Teaching Kit (disponible en el Campus Virtual, al final de los recursos del capítulo de CUDA). La mayoría de ellos, por ejemplo, no sacan partido de la prebúsqueda de datos.

Actividades avanzadas (no entran en el temario del curso):

1. En `matrixMulGPU`, utilizar la memoria compartida para optimizar la ejecución, aprovechando la carga de las matrices de entrada en mosaicos *(tiles)* suficientemente pequeños que quepan de forma alterna en la memoria compartida disponible en cada multiprocesador.
2. Completar los ejercicios que hay propuestos tras la certificación en el módulo 3, acerca del uso de streams y comunicaciones asíncronas.