Paralelización con CUDA

Joan Rosell, Sergi Molina

1 Introducción

En esta práctica se nos pide continuar paralelizando el perceptrón multicapa empleando aceleradoras. Para ello se realizará una paralelización con CUDA, siglas de *Compute Unified Device Architecture*, la cual hace referencia a la plataforma creada por NVIDIA. Algunos de los retos que se plantean en esta entrega son la gestión eficiente del ancho de banda entre Host y GPU, el agrupamiento de threads para maximizar el paralelismo SIMT y el manejo de variables de control tanto en el Host como en la GPU.

1.1 Herramientas

Para la realización de la práctica, ha sido necesario el uso de varias herramientas tales como, *Valgrind*, con la que se pueden observar problemas de memoria y rendimiento del programa, *perf*, con la que se puede realizar un análisis completo de rendimiento de los programas sobre la plataforma Linux, *CMake*, la cual permite automatizar la generación de código y *SLURM*, acrónimo de Simple Linux Utility for Resource Management, con la que se ha lanzado el trabajo a los nodos del clúster. Las versiones utilizadas de cada uno de los programas ha sido la siguiente:

Herramienta	Versión
Valgrind	3.11.0
perf	3.10.0
CMake	3.13.4
SLURM	19.05.4

Como sistema de control de versiones se ha utilizado Git, sobre la plataforma GitHub.

1.2 Plataforma de cómputo

Para la realización de esta práctica se ha trabajado sobre el clúster Aolin, donde se encuentran varios nodos con GPU. Los nodos en cuestión son *AOLIN24 (RTX3080), AOLIN23 (RTX3080), AOCLSD (RTX2070 y GTX750Ti) y AOMASTER (GTX1080Ti y GTX750Ti)*. De entre estos nodos se ha utilizado AOLIN24 para los experimentos que se van a presentar ya que tiene la GPU más potente al momento de escribir este trabajo.

2 Paralelización con CUDA

2.1 Análisis inicial de rendimiento

Cómo ya conocíamos donde estaban los cuellos de botella de la versión secuencial no vamos a detenernos volviendo a comentar lo mismo que en las prácticas anteriores. Si es interesante comentar que ahora si se puede trabajar el problema con una granularidad muy fina, debido a que todo el cómputo se realizará en el mismo nodo, en la GPU.

2.2 Propuesta de paralelización con CUDA

Con el objetivo de paralelizar el programa de forma eficiente con CUDA vamos a reescribir los bucles más internos como kernels que serán ejecutados en la GPU. Como desde uno de estos bucles se calcula el error por cada batch (Batch Error) debemos tener cuidado con cómo se gestiona la sincronización de este valor entre el Host y la GPU. Estos detalles serán comentados en el apartado de **Comunicación**. Además, algunos bucles pueden reescribirse ligeramente para aprovechar el paralelismo masivo de la GPU y el funcionamiento de los accesos fusionados a memoria (*coalesced load/store*).

2.2.1 Particionamiento

Tal y como hemos comentado previamente, hemos dividido el programa secuencial en pequeños kernels. Estos kernels representan los bucles más internos donde residían los *hotspots* de la versión base. Cabe destacar que alguno de los bucles ha sido sujeto a optimizaciones de tipo *loop fission* para evitar problemas de divergencia y simplificar algunos cálculos.

2.2.2 Comunicación

En esta práctica la comunicación entre Host y Device era clave para alcanzar unos buenos resultados de rendimiento. Una de las primeras versiones funcionales tenía el problema de que sincronizaba el *Batch Error* tras cada patrón procesado. Esto era tremendamente pernicioso para el rendimiento, llegando a consumir hasta un 60% del tiempo de ejecución.

Para evitar estos problemas todos los kernels reutilizan la memoria global del Device durante el cómputo de los batches y tras cada batch se actualizan los pesos y los errores haciendo las copias del Host al Device pertinentes. De esta forma se maximiza la localidad de los datos ya que se mantienen en la GPU para todos los cálculos necesarios.

2.2.3 Agrupación y Asignación

En lo que respecta a la agrupación, hemos intentado en todo momento agrupar las iteraciones de los bucles externos dentro de un mismo bloque para que los bucles más

internos sean procesados íntegramente por los threads de un solo bloque y no exista la necesidad de hacer comunicaciónes y sincronizaciones entre bloques, cosa bastante compleja al no existir soporte nativo mediante CUDA. De la misma forma, hemos asignado cada iteración a cada thread del bloque, cosa sencilla ya que el bucle de mayor envergadura ejecutaba 1024 iteraciones, la mitad que el máximo soportado por la RTX3080 de Aolin24.

La API de CUDA no nos ofrece muchas oportunidades de asignar los kernels más allá de jugar con la indexación de los bloques para que accedan a distintas matrices, pero en esta práctica esto no es tan relevante.

2.3 Implementación del algoritmo paralelo

Para implementar esta propuesta reescribimos el código de cada bucle cómo un kernel individual. De esta forma pudimos aislar las entradas y salidas de cada kernel para desarrollar de forma iterativa el algoritmo. Utilizamos en todo momento los datos calculados en el Host como Ground Truth para asegurarnos que no introducíamos errores de precisión por el cálculo en la GPU, cosa que nos facilitó en gran medida la detección de bugs y problemas con la gestión de la indexación de las diferentes estructuras de datos.

De esta forma, la primera implementación completamente funcional con CUDA la vamos a llamar *V1*. A esta primera versión se le añaden varias mejoras para mejorar el rendimiento de los kernels más lentos. De entre estas mejoras podemos destacar el desenrollado de las reducciones cuando se alcanza un *stride* de 32 con el objetivo de eliminar la divergencia dentro del primer *warp*. Este desenrollado se implementa con una primitiva de CUDA llamada `__shfl_down`, que va aplicando una reducción de tipo suma de forma descendiente dentro del *warp*.

Otras mejoras de menor importancia son adelantar el cómputo de ciertos arrays antes de entrar en regiones condicionales, evitando en cierta medida los efectos de la divergencia y la transformación de ciertas multiplicaciones que emplean la variable *tset* ya que al final lo que se está haciendo es decidir si realizar un cómputo o no. Estas dos mejoras se complementan ya que se puede lograr que todos los threads calculen la mayor parte del cómputo sin divergencia al cambiar la multiplicación del *tset* por un operador ternario y precalcular todos los operandos de la operación que se va a realizar condicionalmente. Hay un ejemplo de esta técnica en el kernel `k_compute_delta_ih`.

A esta segunda versión la hemos llamado *V2*. Con esta versión se logra mejorar ligeramente el tiempo de ejecución, por lo que hemos decidido incorporarla en los resultados de rendimiento.

2.4 Resultados de rendimiento

Pasemos a evaluar los resultados obtenidos tras la implementación del algoritmo con CUDA. Empecemos con el tiempo de ejecución:

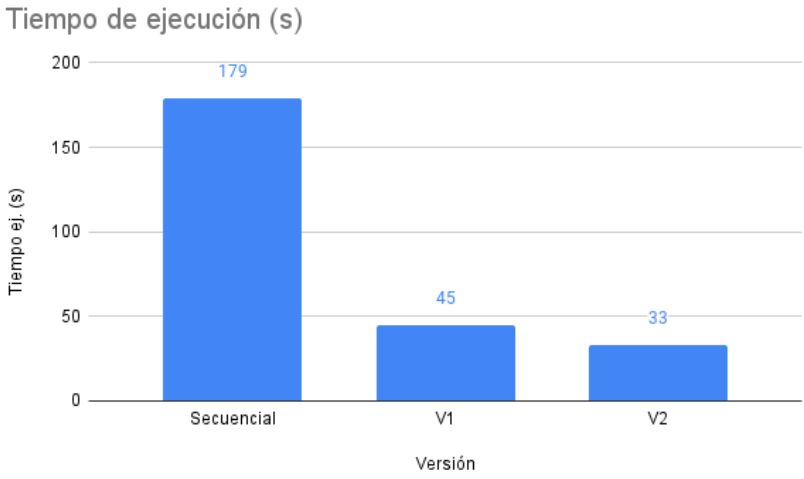

FIGURE 1. *Tiempo de ejecución*

Tal y como se puede apreciar en el gráfico, las dos versiones propuestas reducen en gran medida el tiempo de ejecución respecto la versión secuencial. Se presentan a continucación los Speedups obtenidos:

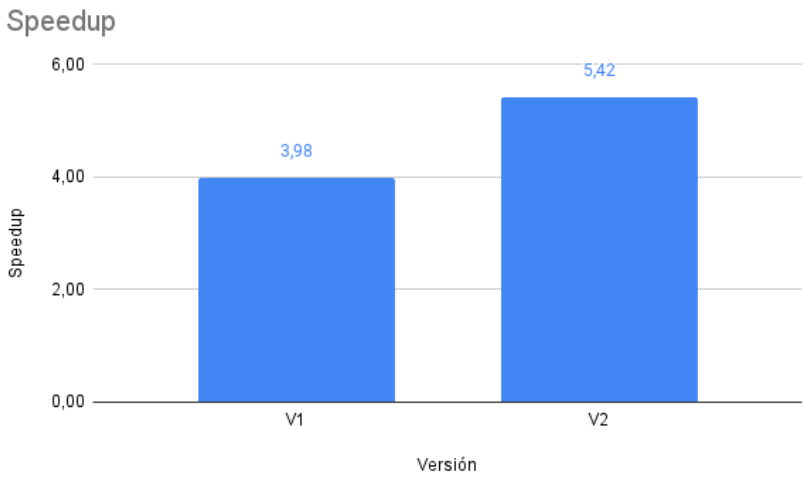

FIGURE 2. *Speedups obtenidos*

A partir de los Speedups obtenidos podemos ver que la versión V2 obtiene un

speedup total de 5,42 y un Speedup relativo respecto a la versión V1 de $\frac{45}{33} = 1.36$. Podemos concluir que las mejoras introducidas en la segunda versión son significativas, ya que se mejora el tiempo de ejecución en casi un 40%.

Al haber logrado mantener el número de iteraciones constante el throughput también se ha incrementado en casi un 40% tal y como se puede ver en esta comparativa:

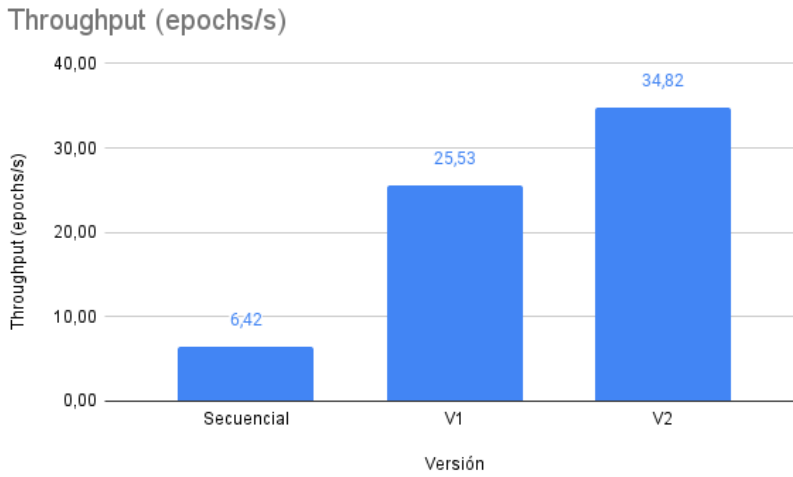

FIGURE 3. *Comparativa de Throughput*

Tal y como hemos comentado justo ahora, si nos fijamos en la diferencia de throughput entre la versión V1 y V2 obtenemos que $\frac{34.82}{25.53} = 1.36$, el mismo factor de aceleración que al comparar los tiempos de ejecución.

3 Instrucciones de compilación y ejecución

Este proyecto emplea CMake con el objetivo de hacer el código más portable y simplificar el proceso de compilación. Se recomienda utilizar CMake 3.13 cómo versión mínima para poder compilar el proyecto.

La compilación del proyecto sigue un patrón llamado *out-of-source build*, en el que se almacenan los archivos resultantes de la compilación en un directorio específico para no mezclar estos archivos con el código fuente. Además, este directorio se excluye del VCS (Git en nuestro caso) para evitar problemas.

3.1 Ejecución/Compilación en Aolin (mediante SLURM)

Para ejecutar el programa en el clúster Aolin se debe utilizar el script llamado *submit.sh*.
A continuación, se presenta un ejemplo de uso:

```
$ ./submit.sh myFile
```

4 Apéndice

Ver.	Tiempo ej. (s)	Epochs	Throughput (epochs/s)	Speedup
Secuencial	179	1149	6,42	-
V1	45	1149	25,53	3,98
V2	33	1149	34,82	5,42

FIGURE 4. *Datos de rendimiento*