Informe de Prácticas
Sesión 3
Computación de Altas Prestaciones
Máster en Ingeniería Informática

Universidad de Oviedo

Autores:

Rubén Martínez Ginzo, UO282651@uniovi.es
Alejandro Rodríguez López, UO281827@uniovi.es

Abril 2025

Índice

Índice de figuras

1. Introducción

En esta sesión de prácticas de laboratorio se aborda la programación en C/CUDA. Para ello, se implementarán y analizarán dos problemas:

- Producto matricial.
- Resolución de sistemas de ecuaciones utilizando el método de *Gauss-Jordan*.

El objetivo principal de esta sesión es la implementación de ambos problemas en C/CUDA y su posterior análisis de rendimiento respecto a la implementación en CPU.

1.1. Desarrollo

Para llevar a cabo el desarrollo de esta práctica, se han seguido las indicaciones recogidas en el guion de la sesión correspondiente. Cada uno de los dos alumnos involucrados se ha centrado en la resolución de uno de los problemas, siendo el producto matricial el problema asignado al alumno Rubén Martínez Ginzo y la resolución de sistemas de ecuaciones el problema asignado al alumno Alejandro Rodríguez López.

Todo el código fuente se encuentra disponible públicamente en el siguiente repositorio de GitHub, así como en el archivo *zip* asociado a esta entrega.

1.2. *Benchmarking*

Con fines de realizar análisis de rendimiento, se han implementado *benchmarks* para simplificar la ejecución y la toma de tiempos con diferentes configuraciones y datos de entrada en ambos problemas.

Respecto al producto matricial, se ha implementado un programa en C/CUDA (*cuda_mul_benchmark.cu*) para cumplir con el propósito anterior. Este programa permite la ejecución del producto de matrices de tamaño $N \times N$ con diferentes configuraciones de *threads per block*. Además, se define un número de iteraciones para cada operación a realizar, de forma que se pueda obtener un tiempo medio de ejecución de todas ellas, minimizando así el impacto de la variabilidad en el rendimiento. De forma sencilla, modificando una simple variable (*ONLY_KERNEL_TIME*), se puede elegir si se desea medir el tiempo de ejecución del *kernel* únicamente o si se desea incluir el tiempo de reserva de memoria y el tiempo de copia de resultados. El programa devuelve un *log* en formato CSV con los resultados obtenidos.

2. Producto matricial en GPU

En esta sección se analiza el rendimiento del producto matricial en GPU. Para ello, se ha implementado un *kernel* que realiza el producto de dos matrices de tamaño $N \times N$ y se ha medido el tiempo de ejecución en función del tamaño de la matriz. Cada tamaño de matriz se ha ejecutado con todos los bloques de hilos posibles $(1, 2, 4, 8, 16, 32)$.

2.1. CPU vs GPU

Se compara a continuación el rendimiento del producto matricial en CPU y en GPU. Para ello, se grafican los resultados obtenidos con el algoritmo *Z-Order* en CPU y con el *kernel* paralelizado en GPU utilizando 1 *thread per block*. Dichos resultados se muestran en la Figura 1.

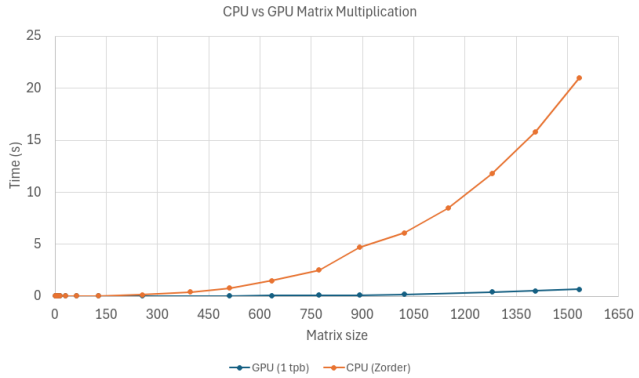


Figura 1: Rendimiento del producto matricial en CPU y GPU

Como se puede observar, el rendimientro del producto matricial en GPU es infinitamente superior al de CPU. Además, debe tenerse en cuenta que se está utilizando el algoritmo probado en CPU que mejores resultados ha dado (*Z-Order*) respecto a la configuración menos eficiente en GPU (1 *thread per block*).

2.2. *Threads per block*

En esta sección se analiza el rendimiento del producto matricial en GPU en función de la cantidad de *threads per block* utilizados. Los resultados se grafican en la Figura 2.

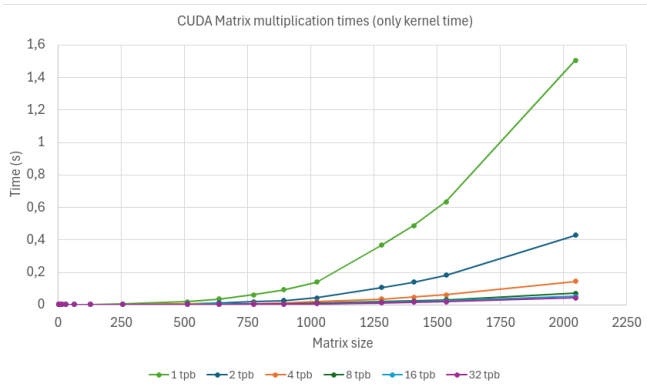


Figura 2: Rendimiento del producto matricial en GPU en función de los *threads per block*

Como es de esperar, el rendimiento mejora a medida que se incrementa el número de *threads per block*. Esta mejora de rendimiento se hace mucho más notable conforme aumenta el tamaño de la matriz, aunque en este caso, a partir de 8 *threads per block* puede considerarse despreciable. Esto se debe a que el máximo tamaño de matriz probado ($N = 2048$) es relativamente pequeño y no se aprovechan al máximo las capacidades de la GPU. En la Figura 3 se observa el rendimiento del producto matricial en GPU con matrices mucho mayores.

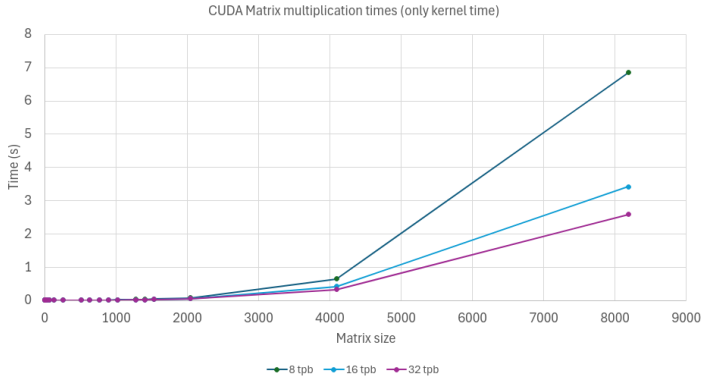


Figura 3: Rendimiento del producto matricial en GPU con matrices de tamaño $N > 2048$

Ahora sí que la diferencia de rendimiento con más de 8 *threads per block* es notable, y será aun mayor utilizando matrices de más tamaño.

2.3. Reserva de memoria y copia de resultados

Una peculiaridad de la GPU es que su memoria no es accesible desde la CPU. Por lo tanto, es necesario reservar memoria en la GPU para almacenar los resultados y luego copiarlos a la memoria de la CPU. En las gráficas anteriores sólo se ha tenido en cuenta el tiempo de ejecución del *kernel* y no el tiempo de reserva de memoria ni el tiempo de copia de resultados. En una sitación real, el tiempo de reserva de memoria y el tiempo de copia de resultados deberían ser tenidos en cuenta, ya que de poco sirve generar resultados si no son accesibles desde la CPU. En la Figura 4 se muestra la comparativa de ambas situaciones.

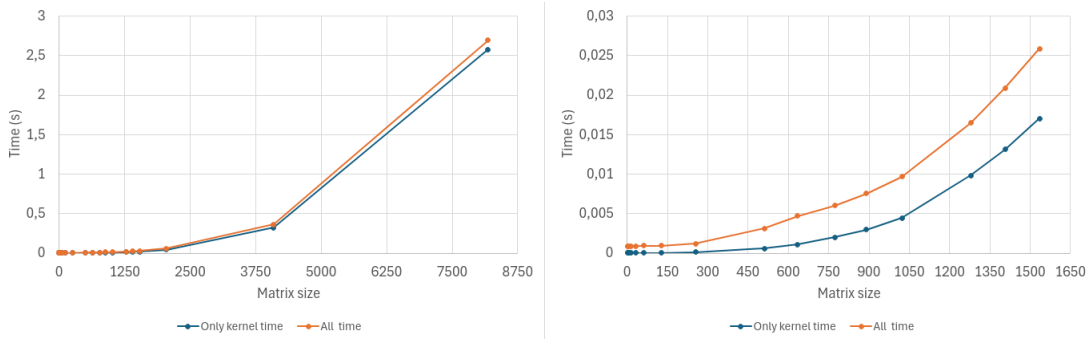


Figura 4: Sólo el tiempo de ejecución del *kernel* vs el tiempo total

Sorprendentemente, el tiempo de reserva de memoria y copia de resultados está muy optimizado ya que, con los tamaños de matriz probados, no hay apenas diferencia entre únicamente el tiempo de ejecución del *kernel* y el tiempo total. Aun así, si se probase con matrices de mucho mayor tamaño (poco funcional para esta práctica) seguramente la diferencia sería notable.

3. Equations

3.1. Problema a resolver

El problema consiste en la resolución de un sistema de ecuaciones lineales (SEL) de un tamaño cualquiera mediante el método de Gauss-Jordan. Para ello, se recibe como dato de entrada el tamaño del SEL (i.e. su número de incógnitas). De esta forma, un problema $N = 3$ sería de la forma representada en la ecuación 1.

$$\begin{array}{ccccccc} a_{0,0} \cdot x_0 & + & a_{0,1} \cdot x_1 & + & a_{0,2} \cdot x_2 & = & y_0 \\ a_{1,0} \cdot x_0 & + & a_{1,1} \cdot x_1 & + & a_{1,2} \cdot x_2 & = & y_1 \\ a_{2,0} \cdot x_0 & + & a_{2,1} \cdot x_1 & + & a_{2,2} \cdot x_2 & = & y_2 \end{array} \tag{1}$$

Sistema de ecuaciones lineales (SEL)

Se utiliza como dato de entrada una matriz de tamaño $N \times N + 1$. Esta estructura contiene los N coeficientes del SEL en sus primeras N columnas y el vector de términos independientes en la columna $N + 1$. (Véase 2)

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,N} & y_0 \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,N} & y_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{N,0} & a_{N,1} & a_{N,2} & \cdots & a_{N,N} & y_N \end{bmatrix} \tag{2}$$

Matriz de coeficientes extendida de un SEL

A esta matriz se le aplicarán una serie de transformaciones según el método de Gauss Jordan para obtener una matriz identidad, en la que su diagonal principal está compuesta de unos y el resto de elementos son cero (salvo los términos independientes de la columna $N+1$). Estos términos independientes serán el resultado del SEL. Esta matriz sería de la forma:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & x_0 \\ 0 & 1 & 0 & \cdots & 0 & x_1 \\ 0 & 0 & 1 & \cdots & 0 & x_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & x_N \end{bmatrix} \tag{3}$$

Matriz identidad de un sistema de ecuaciones lineales

3.2. CPU

Las observacinoes empíricas del algoritmo en CPU siguen fielmente las predicciones teóricas para este algoritmo. Se presenta una gráfica que muestra el comportamiento potencial del tiempo de ejecución respecto al tamaño del problema. Nótese que ambos ejes de la gráfica son logarítmicos. Véase la Figura 5.

3.3. GPU

La aceleración de un algoritmo paralelo se decide en el instante en que se planifican las secciones paralelas, sus tareas, comunicaciones, sincronización y dependencias. En el caso de Gauss Jordan, paralelizar el cómputo de cada

columna es inviable, ya que cada columna depende de la columna anterior. En su lugar, se debe dividir el cómputo de cada columna en dos partes paralelas: en la primera se normaliza la fila para obtener el 1 en la diagonal principal (`normalize_row`), y en la segunda se obtienen ceros en la columna actual (`eliminate_column`). Este acercamiento reporta una tiempo notablemente mejor al de la CPU. Véase 5

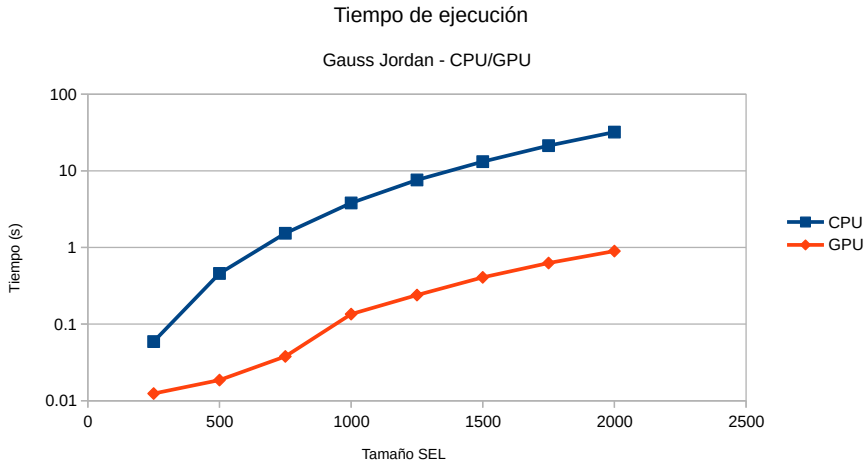
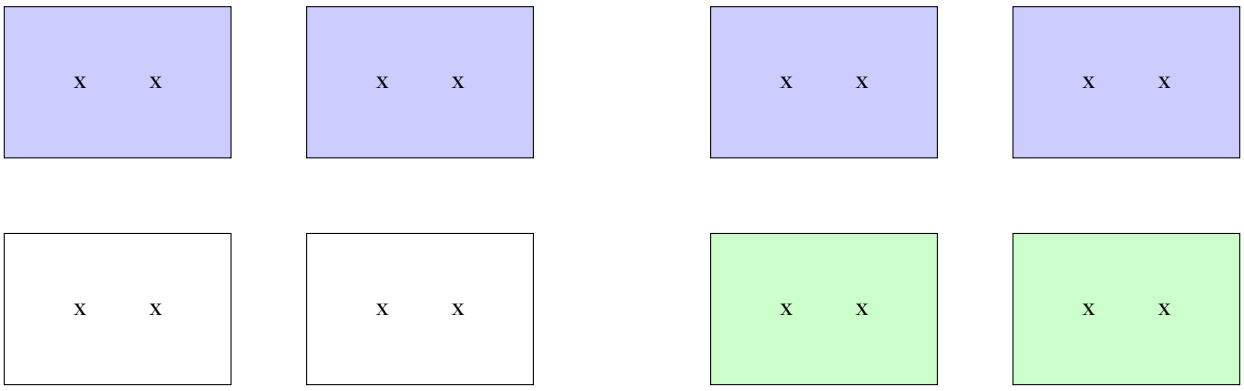


Figura 5: Tiempos de ejecución de para varios tamaños.

Debido al gran tamaño de las matrices probadas, en las que se exceden los 1024 elementos por fila y columna, se han utilizado varios bloques de hilos, calculados dinámicamente en tiempo de ejecución a partir del tamaño de la matriz. En todos los casos, el tamaño de la matriz se reparte entre bloques del máximo tamaño posible. De los dos kernels utilizados, el primero (`normalize_row`) utiliza una estructura unidimensional de hilos y bloques para iterar por todas las columnas de una fila de la matriz. El segundo (`eliminate_column`) utiliza una estructura bidimensional de hilos y bloques para iterar por todas las columnas y filas de la matriz, anulando la columna actual y aplicando el mismo factor al resto de columnas de la fila. Véase la Figura 6.



(a) Kernel `normalize_row` (b) Kernel `eliminate_column`

Figura 6: Kernels de Gauss Jordan en GPU

Siguiendo este sistema, el trabajo realizado por los hilos para cada columna se representa con el diagrama 7. Se pueden ver dos puntos de sincronización en los que el paralelismo se rompe (nodos azules), el kernel `normalize_row` que se ejecuta sobre el número de filas (nodos verdes) y el kernel `eliminate_column` que se ejecuta sobre el número total de celdas (nodos rojos).

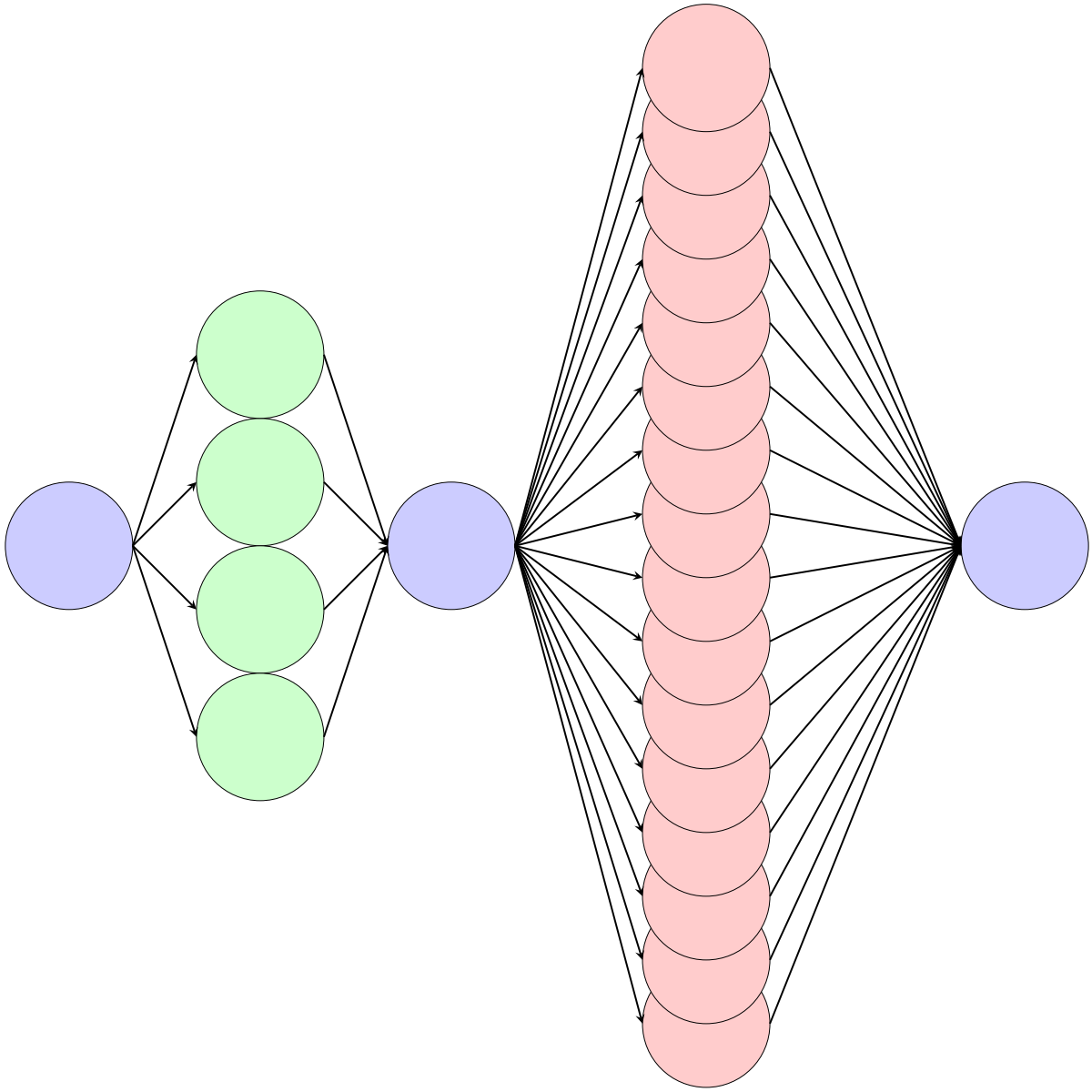


Figura 7: Cálculo de Gauss Jordan en GPU

4. Conclusiones

El desarrollo de esta sesión de prácticas de laboratorio ha permitido analizar y comprender la enorme diferencia de rendimiento entre las implementaciones en CPU y GPU para tareas de cómputo intensivo, como lo son el producto matricial y la resolución de sistemas de ecuaciones lineales mediante el método de Gauss Jordan. Debe recalcarse que este rendimiento es muy notable en problemas que permiten una paralelización eficiente, lo que aprovecha al máximo la arquitectura de la GPU.

4.1. Producto matricial

En el caso del producto matricial, la GPU ofrece una mejora de rendimiento muy significativa, incluso frente a la major implementación del problema en CPU (*Z-Order*). Este hecho demuestra que, incluso con configuraciones poco óptimas (como es el uso de un sólo *thread per block*), la GPU supera ampliamente a la CPU, confirmando que este tipo de cómputo intensivo se beneficia enormemente de la paralelización masiva.
Asimismo, se ha comprobado que la sobrecarga asociada a la reserva de memoria y la transferencia de datos entre CPUC y GPU es despreciable para los tamaños de matriz utilizados. Esto demuestra que el uso de la GPU es muy ventajoso. Otro aspecto a destacar es la escalabilidad del enfoque. Mientras que la CPU sigue una tendencia exponencial del tiempo de ejecución, la GPU mantiene una relación mucho más contenida y predecible a medida que aumenta el tamaño de los datos. Esto reafirma su utilidad no sólo para fines académicos, sino también para escenarios reales en ciencia de datos, simulación numérica y aprendizaje automático, donde las matrices de gran tamaño son la norma.

4.2. Gauss Jordan

En el caso del método de Gauss Jordan, la GPU ha demostrado una mejora de rendimiento notable siempre y cuando se paralelicen los procesos internos del cálculo de cada columna en lugar de repartir las columnas entre hilos dependientes. Este ejemplo es demostración de la viabilidad de paralelizar procesos internos cuando no es posible paralelizar el problema completo. Esta decisión podría ser descartada por miedo a introducir una mayor deuda técnica en el código a cambio de poco rendimiento, cuando en realidad el problema podría ser resuelto con mucha mayor eficiencia a cambio de poca deuda técnica.