Computación de propósito general de unidades de procesamiento gráfico

Laboratorio Final

Operaciones DGEMM y DTRSM

Integrantes

Nombre	CI	Correo
Renzo Gambone	5.155.486-4	renzo.gambone@fing.edu.uy
Germán Ouviña	4.823.566-1	german.ouvina@fing.edu.uy

Índice

1. Introducción

En el presente informe se detallan los resultados obtenidos durante la evaluación experimental de los ejercicios correspondientes al **Laboratorio Final**, así como su pertinente análisis. A continuación, se destacan ciertos aspectos a tener en cuenta del análisis y de los archivos entregables:

1. La implementación de los algoritmos se realizó empleando el lenguaje **C** y la API de **CUDA**.
2. La evaluación experimental se realizó por medio de la plataforma ClusterUY. Durante la evaluación experimental los recursos con los que se ejecutó el algoritmo fueron:
   - **GPU:** Tesla P100-PCIE-12GB
   - **CPU:** Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz
3. Debido a particularidades de la arquitectura en la tarjeta Tesla P-100 las métricas de eficiencia al experimentar con la misma no fueron la esperadas. Para simular el comportamiendo esperado por el curso, dichas métricas fueron capturadas desactivando la caché L1 de la tarjeta por medio compilar con las flags `-Xptxas -dlcm=cg`.
4. Los datos manipulados fueron matrices representadas por vectores de números decimales en punto flotante de doble precisión (*double* de **C**).
5. Los tiempos registrados al realizar distintas ejecuciones de un mismo script no fueron siempre constantes, sin embargo su variación osciló con un radio lo suficientemente bajo como para que las observaciones y conclusiones presentadas se preserven.
6. Las matrices de prueba utilizadas para extraer los datos a analizar fueron generadas aleatoriamente, utilizando una semilla fija y tomando valores no nulos en el intervalo $[0{,}1, rand\_max + 0{,}1]$, siendo $rand\_max$ una constante de C.

2. Desarrollo

2.1. Operación DGEMM

En esta sección se detallan los resultados obtenidos de la experimentación realizada con los algoritmos que implementan la operación DGEMM. Dicha experimentación consistió en registrar los tiempos de cada algoritmo, así como analizar distintas métricas en referencia al uso eficiente de la memoria. Para capturar estos datos, se utilizó la herramienta de profiling `nvprof`.

2.1.1. Construcción de Algoritmos

Por consigna, se trabajó con dos implementaciones de `dgemm_gpu`, una función que, dadas tres matrices A, B y C, dos escalares α y β, y tamaños y rangos sobre los mismos (para trabajar con submatrices), guarda en C el valor $\beta C + \alpha A \times B$. Dichas implementaciones fueron las siguientes:

- `dgemm_global_kernel`, versión paralela del algoritmo con bloques de 32×32, en donde hay tantos threads como celdas en C, y cada uno computa una celda de C, recorriendo bloques de filas de A y de columnas de B de la memoria global.
- `dgemm_shared_kernel`, versión paralela del algoritmo con bloques de 32×32, en donde hay tantos threads como celdas en C, y cada uno computa una celda de C, realizando un bucle en donde cada hilo del bloque carga un bloque de A y de B en memoria compartida, usa los valores de dichos bloques y procede a seguir iterando por los bloques de A por filas y de B por columnas, cargándolas siempre en memoria compartida (sincronizando los hilos antes y después de las operaciones). Se utilizan dos estructuras matriciales en la memoria compartida, las cuales cuentan con una celda por thread del bloque.

2.1.2. Análisis de Métricas

A continuación se adjuntan cuadros de comparación de tiempos y métricas de eficiencia de memoria para cada versión del algoritmo. Como datos de entrada, se emplearon 4 instancias distintas para los tamaños de A y B, siendo las mismas:

1. $A \in M_{512\times512}$, $B \in M_{512\times512}$
2. $A \in M_{4096\times4096}$, $B \in M_{4096\times4096}$
3. $A \in M_{4096\times512}$, $B \in M_{512\times4096}$
4. $A \in M_{512\times4096}$, $B \in M_{4096\times512}$

Las primeras dos instancias buscan evaluar casos de matrices cuadradas, variando los tamaños de las mismas. Las últimas dos buscan evaluar el algoritmo al utilizar matrices no cuadradas. De esta manera, los datos extraídos fueron los siguientes:

	Global	Shared
Tiempo Kernel (ms)	1.1108	0.3180
GLD Efficency (%)	91.68	100.00
GST Efficency (%)	100.00	100.00
Shared Efficency (%)	0.00	132.00
Achieved Occupancy	0.9074	0.9369

Cuadro 1: $A(512 \times 512)$ **y** $B(512 \times 512)$

	Global	Shared
Tiempo Kernel (ms)	863.01	125.91
GLD Efficency (%)	91.67	100.00
GST Efficency (%)	100.00	100.00
Shared Efficency (%)	0.00	132.00
Achieved Occupancy	0.9231	0.9990

Cuadro 2: $A(4096 \times 4096)$ **y** $B(4096 \times 4096)$

	Global	Shared
Tiempo Kernel (ms)	65.480	15.829
GLD Efficency (%)	91.68	100.00
GST Efficency (%)	100.00	100.00
Shared Efficency (%)	0.00	132.00
Achieved Occupancy	0.9219	0.9964

Cuadro 3: $A(4096 \times 512)$ **y** $B(512 \times 4096)$

	Global	Shared
Tiempo Kernel (ms)	9.5580	2.1925
GLD Efficency (%)	91.67	100.00
GST Efficency (%)	100.00	100.00
Shared Efficency (%)	0.00	132.00
Achieved Occupancy	0.9047	0.9318

Cuadro 4: $A(512 \times 4096)$ **y** $B(4096 \times 512)$

De estos datos, se pueden realizar las siguientes observaciones:

- La versión *shared* del algoritmo fue más rápida que la versión *global*, siendo 4 veces más rápida para todas las instancias de evaluación planteadas.
- La métrica *gld_efficency* fue 91,68 % para todas las instancias de la versión *global*, mientras que para la versión *shared* fue de 100,00 %.
- La métrica *gst_efficency* fue 100,00 % en todos los casos.
- La métrica *shared_efficency* fue 0,00 % para la versión *global* mientras que para la versión *shared* fue 132,00 %.
- La métrica *occupancy* fue mayor a 0,9 en todos los casos, siendo siempre mayor en las instancias de la versión *shared*.

Se pueden extraer múltiples conclusiones de los datos anteriormente observados:

- Comparando los tiempos del kernel en ambas versiones, se concluye que la versión *shared* es ampliamente más rápida que la versión *global*, lo cual resulta coherente debido a la utilización de memoria compartida, la cual cuenta con un tiempo de acceso notablemente inferior al de la memoria global.
- Las métricas *gld_efficency* y *gst_efficency* resultaron óptimas para la versión *shared* del algoritmo, lo cual es coherente debido a que el acceso a las matrices se hace por filas y de forma coalesced.
- La métrica *shared_efficency* tuvo un valor de 0,00 % en la versión *global* del algoritmo (lo tiene sentido, dado que el algoritmo no emplea memoria compartida para nada). Por otra parte, la versión *shared* obtiene 132,00 %. Este número si bien desconcertante se atribuye a un posible error del profiler al obtener la métrica, debido a la naturaleza del algoritmo y GPU, no obstante se presume que el acceso a la misma es óptimo, debido a que el acceso es coalesced y cada hilo de un warp lee o bien el mismo eleménto o siempre uno distinto de A y B.
- La métrica *occupancy* fue buena en general, alcanzando su pico en la versión *shared*. Se interpreta que más del 90 % en esta métrica implica que la mayoría de threads de cada warp se mantuvieron activos a la vez durante toda la ejecución del kernel, lo cual es un buena buena característica para ambos algoritmos.

2.2. Ejercicio 3 - Operación DTRSM

En esta sección se detallan los resultados obtenidos de la experimentación realizada con los algoritmos que implementan la operación DTRSM. Dicha experimentación consistió en registrar los tiempos de cada algoritmo, así como analizar distintas métricas en referencia al uso eficiente de la memoria. Para capturar estos datos, se utilizó la herramienta de profiling `nvprof`.

2.2.1. Construcción de Algoritmos

Por consigna, se trabajó con cuatro implementaciones de `dtrsm_gpu`, una función que, dadas dos matrices A y B, un escalar α, y tamaños y rangos sobre los mismos (para trabajar con submatrices), guarda en B los resultados X de resolver la ecuación matricial $A \times X = \alpha B$. Las implementaciones fueron las siguientes:

- `dtrsm_32_shared_kernel`, versión paralela del algoritmo con bloques de 32×32, en donde A es de 32×32, y hay tantos threads como valores en $B(32 \times n)$, con n múltiplo de 32. Para que un hilo de un warp comunique el resultado de su variable al resto de hilos de ese mismo warp (a modo que el resto calcule iterativamente sus valores), el valor es cargado en memoria compartida. Tras ser cargado, los hilos del warp se sincronizan mediante un `__syncwarp()` y el resto de hilos del warp lee el valor que acaba de ser cargado en memoria compartida.

- `dtrsm_32_shuffle_kernel`, versión paralela del algoritmo con bloques de 32×32, en donde A es de 32×32, y hay tantos threads como valores en $B(32 \times n)$, con n múltiplo de 32. Para que un hilo de un warp comunique el resultado de su variable al resto de hilos de ese mismo warp (a modo que el resto calcule iterativamente sus valores), el valor es calculado en un registro. Tras ser calculado, los hilos del warp se sincronizan mediante un `__syncwarp()` y mediante la operación `__sfl_sync`, el hilo del warp que acaba de calcular su valor realiza un broadcast del mismo al resto de los hilos del warp.

- `dtrsm_32k_kernel`, versión del algoritmo con bloques de 32×32, en donde A es de $32k \times 32k$, y hay tantos threads como valores en $B(32k \times n)$, con n múltiplo de 32. La matriz A es divida en bloques de 32×32, así como B en bloques de $32 \times n$. Debido a la dependencia de operaciones entre subloques de A y B, este algoritmo **itera secuencialmente** a través de los bloques de A, de izquierda a derecha y de arriba hacia abajo, invocando al kernel `dtrsm_32_shared_kernel` al estar en un subloque triangular o al kernel `dgemm_shared_kernel` al estar en un subloque no triangular, realizando las respectivas operaciones para cada caso.

- `dtrsm_recursive`, versión del algoritmo con bloques de 32×32, en donde A es de $32k \times 32k$, y hay tantos threads como valores en $B(32k \times n)$, con n múltiplo de 32. La matriz A es divida en bloques de 32×32, así como B en bloques de $32 \times n$. Debido a la dependencia de operaciones entre subloques de A y B, este algoritmo **procesa recursivamente *(in order)*** los bloques de A, invocando en el paso base al kernel `dtrsm_32k_kernel` con $k = 2$. Mientras que en el paso recursivo divide la matriz A en cuatro submatrices $(A_{11}, 0, A_{21}, A_{22})$ y a B en (B_1, B_2) acorde. Realizando pasos recursivos para $[A_{11}|B_1]$ y $[A_{22}|B_2]$, y entre ambas recursiones invocando al kernel `dgemm_shared_kernel` realizando la operación $B_2 = B_2 - A_{21} \times B_1$.

A su vez también se trabajó con la implementación `cublasDtrsm`, proporcionada por *CuBlas* y empleada a modo de marco de referencia para evaluar la correctitud y el desempeño de los algoritmos implementados.

Se destaca que, si bien las implementaciones propuestas de los algoritmos empleando los kernels `dtrsm_32k_kernel` y `dtrsm_recursive` son correctas (resuelven el problema $DTRSM$) y dan el exacto mismo resultado entre sí, durante la comparación con el resultado de CuBlas se encontraron pequeñas fluctuaciones en los resultados, para matrices A a partir de 64×64. Estas fluctuaciones fueron del orden de 10^6 ($-22138{,}037681$ vs $-22138{,}037680$). Si bien son errores despreciables, estos mismos se arrastran a lo largo de las columnas de B; comprometiendo cada vez más la precisión de las columnas inferiores. Se cree que dichas fluctuaciones son debido a la magnitud de los números y el pasaje entre C (código del laboratorio) y *Fortran* (CuBlas). Se decidió ignorar dichas inconsistencias a efectos de determinar la correctitud de los algoritmos implementados.

2.2.2. Análisis de Métricas

En la presente sección se realiza una comparativa entre distintas versiones del algoritmo en cuestión. Para facilitar el análisis, se presenta la información en 3 subsecciones distintas.

2.2.2.1 Caso 1 - Shared VS Shuffle ($B \in M_{32\times n}$)

A continuación se adjunta una tabla comparando tiempos y métricas de eficiencia de memoria para cada versión del algoritmo con $B \in M_{32\times n}$. Como datos de entrada, se emplearon 2 instancias distintas para los tamaños de A y B, siendo las mismas:

1. $A \in M_{32\times 32}$, $B \in M_{32\times 512}$
2. $A \in M_{32\times 32}$, $B \in M_{32\times 4096}$

De esta manera, los datos extraídos fueron los siguientes:

	Shared	Shuffle
Tiempo Kernel (us)	28.448	28.480
GLD Efficency (%)	25.00	25.00
GST Efficency (%)	25.00	25.00
Shared Efficency (%)	9.99	6.10
Achieved Occupancy	0.4411	0.4476

Cuadro 5: $A(32 \times 32)$ **y** $B(32 \times 512)$

	Shared	Shuffle
Tiempo Kernel (us)	74.495	74.815
GLD Efficency (%)	25.00	25.00
GST Efficency (%)	25.00	25.00
Shared Efficency (%)	9.99	6.10
Achieved Occupancy	0.7983	0.7922

Cuadro 6: $A(32 \times 32)$ **y** $B(32 \times 4096)$

Se pueden realizar las siguientes observaciones:

- En comparación, ambas versiones del algoritmo cuentan con métricas muy similares.
- Las métricas *gld_efficency* y *gst_efficency* resultaron iguales para ambas versiones e instancias, con un valor de 25 %.
- La métrica *shared_efficency* resultó en valores particularmente bajos (9,99 % y 6,10 % respectivamente), menores para la versión *shuffle*.
- La métrica *occupancy* fue notablemente más alta para la instancia de mayor tamaño en B, pero aún así, estuvo más lejos de alcanzar valor óptimo en comparación a los algoritmos de DGEMM.

Se pueden extraer múltiples conclusiones de los datos anteriormente observados.

- Debido a la similitud de las métricas tanto en tiempos como en eficiencia, es difícil determinar que versión tuvo mejor performance. Tomando los tiempos de la instancia con B de mayor tamaño, y teniendo en cuenta que el uso de la memoria compartida fue apenas más eficiente para la versión *shared*, se determinó que esta versión es la mejor en comparación.
- Respecto a las métricas de eficiencia en el acceso a memoria, resulta coherente que las mismas resulten bajas, ya que se recorre a la matriz B por columnas de forma no coalesced, impactando directamente en estas métricas.
- Puede decirse algo similar respecto a los bajos resultados obtenidos para *shared_efficency*, aunque en este caso el tener acceso no coalesced resulta en conflictos de bancos, disminuyendo el valor resultante para esta métrica.
- Respecto a la *occupancy*, debido a la estructura triangular de A y a la naturaleza del algoritmo, tiene sentido no haber alcanzado un buen ratio de threads activos.

2.2.2.2 Caso 2 - Secuencial VS Recursivo ($B \in M_{32k \times n}$)

A continuación se adjunta una tabla comparando tiempos y métricas de eficiencia de memoria para cada versión del algoritmo con $B \in M_{32k \times n}$. Como datos de entrada, se emplearon 2 instancias distintas para los tamaños de A y B, siendo las mismas:

1. $A \in M_{4096 \times 4096}$, $B \in M_{4096 \times 512}$
2. $A \in M_{4096 \times 4096}$, $B \in M_{4096 \times 4096}$

Por último se destaca que como todas estas implementaciones invocan múltiples kernels, las métricas de eficiencia se capturaron tomando el promedio. De esta manera, los datos extraídos fueron los siguientes:

	Secuencial	Recursiva
Tiempo Total (ms)	158.440	65.388
Tiempo Kernel (ms)	115.181	14.064

Cuadro 7: Tiempos $A(4096 \times 4096)$ **y** $B(4096 \times 512)$

	Secuencial	Recursiva
Tiempo Total (ms)	289.456	218.223
Tiempo Kernel (ms)	143.632	79.1870

Cuadro 8: Tiempos $A(4096 \times 4096)$ **y** $B(4096 \times 4096)$

	Secuencial	Recursiva
GLD Efficency (%)	100.00	100.00
GST Efficency (%)	100.00	100.00
Shared Efficency (%)	132.00	132.00
Achieved Occupancy	0.4898	0.5534

Cuadro 9: Métricas Kernel DGEMM $A(4096 \times 4096)$ **y** $B(4096 \times 512)$

	Secuencial	Recursiva
GLD Efficency (%)	100.00	100.00
GST Efficency (%)	100.00	100.00
Shared Efficency (%)	132.00	132.00
Achieved Occupancy	0.8694	0.9087

Cuadro 10: Métricas Kernel DGEMM $A(4096 \times 4096)$ **y** $B(4096 \times 4096)$

	Secuencial	Recursiva
GLD Efficency (%)	25.00	25.00
GST Efficency (%)	25.00	25.00
Shared Efficency (%)	9.99	9.99
Achieved Occupancy	0.4464	0.4477

Cuadro 11: Métricas Kernel DTRSM $A(4096 \times 4096)$ **y** $B(4096 \times 512)$

	Secuencial	Recursiva
GLD Efficency (%)	25.00	25.00
GST Efficency (%)	25.00	25.00
Shared Efficency (%)	9.99	9.99
Achieved Occupancy	0.8294	0.8295

Cuadro 12: Métricas Kernel DTRSM $A(4096 \times 4096)$ **y** $B(4096 \times 4096)$

Se pueden realizar las siguientes observaciones:

- En comparación, ambas versiones del algoritmo cuentan con métricas muy similares, observándose una mejora notable en relación a los tiempos de la versión recursiva y una pequeña mejora con respecto a la *occupancy* de esa misma versión.
- Las métricas *gld_efficency*, *gst_efficency* y *shared_efficency* arrojaron los mismos resultados que en sus evaluaciones previas.
- La métrica *occupancy* fue notablemente más alta para la instancia de mayor tamaño en B, tanto para el kernel DGEMM como para el kernel DTRSM.

Se pueden extraer múltiples conclusiones de los datos anteriormente observados.

- Si bien las métricas obtenidas para ambos kernels fueron exactamente las mismas que en sus anteriores evaluaciones, los tiempos de la versión recursiva fueron ampliamente menores, lo cual permite concluir que esta versión resultó ser mejor. Esto tiene sentido, debido a que la principal diferencia entre la implementación secuencial y la recursiva es que en lugar de realizar múltiples operaciones DGEMM (con bloques de 32×32) se realiza un DGEMM con el cuarto de matriz A_{21}, luego otras dos con dieciseisavos de matriz de A y así sucesivamente. Por ende la cantidad de kernels invocados es exponencialmente menor, lo cual reduce el overhead entre invocaciones de kernel. Además, se consigue una mejor performance de `dgemm_shared_kernel`, que durante la experimentación incrementó su occupancy al escalar de 512 a 4096.
- Respecto a las métricas de eficiencia en el acceso a memoria, resulta coherente que los resultados no hayan variado respecto a las experimentaciones anteriores, ya que estas versiones del algoritmo particionan el problema en menores unidades, delegando el trabajo en GPU a los kernels previamente analizados.
- Respecto a la *occupancy*, se vuelve a notar que al crecer B, mejora el valor de esta métrica notablemente. Este fenómeno se comportó igual para el kernel que resuelve DTRSM, el cual en la experimentación previa también obtuvo peor occupancy cuando B era de 32×512 respecto a la instancia en donde B era de 32×4096. Se presume que la disminución de *occupancy* en el kernel DGEMM para B de tamaño 4096×512 es debido a que en cada paso base se realizan dos operaciones DGEMM con $A(32 \times 32)$ y $B(32 \times 512)$. No se experimentó en DGEMM con dicha configuración, por lo cual resulta probable que en la misma, múltiples recursos de procesamiento de la GPU queden ociosos.

2.2.2.3 Caso 3 - Recursivo VS CuBlas ($B \in M_{32k \times n}$)

A continuación se adjunta una tabla comparando tiempos y métricas de eficiencia de memoria para la versión recursiva del algoritmo con $B \in M_{32k \times n}$. En esta ocasión, se compara con los resultados obtenidos de la versión recursiva implementada en *CuBlas*. Como datos de entrada, se emplearon 2 instancias distintas para los tamaños de A y B, siendo las mismas:

1. $A \in M_{4096 \times 4096}$, $B \in M_{4096 \times 512}$
2. $A \in M_{4096 \times 4096}$, $B \in M_{4096 \times 4096}$

Por último se destaca que como todas estas implementaciones invocan múltiples kernels, las métricas de eficiencia se capturaron tomando el promedio. De esta manera, los datos extraídos para el caso $B \in M_{4096 \times 512}$ fueron los siguientes:

	Recursiva	CuBlas
Tiempo Total (ms)	65.388	46.1318
Tiempo Kernel (ms)	14.064	7.18246

Cuadro 13: Tiempos $A(4096 \times 4096)$ **y** $B(4096 \times 512)$

	Recursiva
GLD Efficency (%)	100.00
GST Efficency (%)	100.00
Shared Efficency (%)	132.00
Achieved Occupancy	0.5534

Cuadro 14: Métricas Kernel DGEMM $A(4096 \times 4096)$ **y** $B(4096 \times 512)$

	Recursiva
GLD Efficency (%)	25.00
GST Efficency (%)	25.00
Shared Efficency (%)	9.99
Achieved Occupancy	0.4477

Cuadro 15: Métricas Kernel DTRSM $A(4096 \times 4096)$ **y** $B(4096 \times 512)$

	CuBlas
GLD Efficency (%)	99.98
GST Efficency (%)	99.91
Shared Efficency (%)	160.65
Achieved Occupancy	0.0842

Cuadro 16: Métricas Kernel Maxwell DGEMM 64x64 $A(4096 \times 4096)$ **y** $B(4096 \times 512)$

	CuBlas
GLD Efficency (%)	100.00
GST Efficency (%)	100.00
Shared Efficency (%)	83.24
Achieved Occupancy	0.1467

Cuadro 17: Métricas Kernel TRSM L Mul32 $A(4096 \times 4096)$ **y** $B(4096 \times 512)$

	CuBlas
GLD Efficency (%)	50.00
GST Efficency (%)	50.00
Shared Efficency (%)	128.57
Achieved Occupancy	0.0932

Cuadro 18: Métricas Kernel Magma LDS128 DGEMM $A(4096 \times 4096)$ **y** $B(4096 \times 512)$

Mientras que los datos extraídos para el caso $B \in M_{4096\times4096}$ fueron:

	Recursiva	CuBlas
Tiempo Total (ms)	218.223	117.789
Tiempo Kernel (ms)	79.1870	32.4057

Cuadro 19: Tiempos $A(4096 \times 4096)$ **y** $B(4096 \times 4096)$

	Recursiva
GLD Efficency (%)	100.00
GST Efficency (%)	100.00
Shared Efficency (%)	132.00
Achieved Occupancy	0.9087

Cuadro 20: Métricas Kernel DGEMM $A(4096 \times 4096)$ **y** $B(4096 \times 4096)$

	Recursiva
GLD Efficency (%)	25.00
GST Efficency (%)	25.00
Shared Efficency (%)	9.99
Achieved Occupancy	0.8295

Cuadro 21: Métricas Kernel DTRSM $A(4096 \times 4096)$ **y** $B(4096 \times 4096)$

	CuBlas
GLD Efficency (%)	100.00
GST Efficency (%)	100.00
Shared Efficency (%)	159.52
Achieved Occupancy	0.0856

Cuadro 22: Métricas Kernel Maxwell DGEMM 64x64 $A(4096 \times 4096)$ **y** $B(4096 \times 4096)$

	CuBlas
GLD Efficency (%)	100.00
GST Efficency (%)	100.00
Shared Efficency (%)	83.24
Achieved Occupancy	0.7244

Cuadro 23: Métricas Kernel TRSM L Mul32 $A(4096 \times 4096)$ **y** $B(4096 \times 4096)$

Se pueden realizar las siguientes observaciones:

- Los tiempos de la versión de CuBlas son considerablemente menores a la implementación recursiva.
- Dependiendo del tamaño de B, el algoritmo de *CuBlas* invoca distintos kernels para para procesar subpartes de la misma.
- *CuBlas* siempre alcanzó una mayor eficiencia en cuanto a la manipulación de memoria global y a la eficiencia alcanzada con la memoria compartida.
- Los kernels de *CuBlas* alcanzaron una muy baja tasa de *occupancy* en comparación al resto de los algoritmos.

Se pueden extraer múltiples conclusiones de los datos anteriormente observados:

- En relación a los tiempos, si bien la implementación recursiva tuvo tiempos mayores a *CuBlas*, esto no resulta sorprendente, ya que *CuBlas* es considerado como una cota superior difícil de alcanzar. Esto es debido a que es la implementación más eficiente de los algoritmos hasta ahora, exprimiendo al máximo las particularidades de cada arquitectura. Inclusive las arquitecturas de GPU se diseñan teniendo en cuenta maximizar el impacto en operaciones como DGEMM.

- Los kernels de *CuBlas* siempre superaron o igualaron las métricas de escritura/lectura en memoria global. Esta diferencia es particularmente notable comparando con el kernel DTRSM implementado. Se presume que una implementación de kernel DTRSM que haga un mejor uso de la memoria (leyendo y recorriendo de forma espejada por filas) alcanzará mejores tiempos. Durante la implementación de `dtrsm_32_shared_kernel`, este cambio fue implementado por accidente, y los resultados capturados no sólo alcanzaban 100 % de *GLD y GST efficiencies* sinó también un 132 % de *shared efficiency*. Adicionalmente sus tiempos de kernel fueron drásticamente menores a la implementación final que recorre por columnas (41,442us vs 74,815us) para $A(32 \times 32)$ y $B(32 \times 4096)$. Se decidió abandonar esta implementación, a causa de que imposibilitaba la implementación de `dtrsm_32_shuffle_kernel` recorriendo de la misma forma que la implementación shared (por filas) y no era la pedida por la letra (que especificaba la recorida de B en columnas dentro de cada warp). Hubiera sido de valor contar con el desempeño del resto de los algoritmos usando esta versión, desafortunadamente por cuestiones de tiempos no fue posible capturar y analizar dichos resultados en nuestra experimentación.

- Es sorprendente que los kernels de *CuBlas* alcancen tan poca *occupancy*, especialmente si se tiene en cuenta que el tiempo de kernel del algoritmo en cuestión es menor a la implementación recursiva. Una posible interpretación de esto es que la implementación de los kernels de *CuBlas* tienen un alto desempeño para la cantidad de núcleos de procesamiento usados (necesitando muy pocos recursos). Por otra parte, puede que nuestra implementación de los kernels del laboratorio tengan una alta *occupancy* debido a estar haciendo un mal uso de los recursos de computo (Por ejemplo en el bucle de DTRSM los kernels una vez calculan su incógnita, quedan ociosos, meramente evaluando en *false* las condiciones de ambos *if* y participando en un `__syncwarp()` mientras el resto de incógnitas en la columna son calculadas). Adicionalmente, el haber incluido `__syncwarp()` en el loop de DTRSM y de dos `__syncthreads()` en el loop de DGEMM puede ser una de las razones a las que atribuir que los tiempos alcanzados fueran peores (suponiendo que *CuBlas* implemente dichos kernels con un menor grado de sincronización entre hilos y por ende alcanzando una mayor paralelización).