Dgesv Profiling and Optimization

Master HPC – Block 1, Task 3

Miguel García Folgado

1. Metodología y características de los entornos

Se ha evaluado el rendimiento de la implementación propia de `dgesv()` desarrollada en la Task 1 utilizando ICC 2021 e ICX 2024. Estas versiones están disponibles en FT3:

Versión Intel	**Módulo**
icc 2021.3.0	intel/2021.3.0
icx 2024.2.1	intel/2024.2.1

Las pruebas se realizaron sobre matrices cuadradas de tamaño 1024x1024, 2048x2048 y 4096x4096, ejecutando el binario 5 veces y tomando la mediana del tiempo medido dentro del `main` (en milisegundos).

2. Resultados

2.1. icc 2021.3.0

Módulos que se han utilizado para este caso:

- `module load cesga/2020`
- `module load intel/2021.3.0`
- `module load imkl`

La librería utilizada para la función de referencia es:

- `#include <mkl_lapacke.h>`

Matrix size	**O2**	**O3**	**fast**	**ref**
1024×1024	778 ms	781 ms	596 ms	38 ms
2048×2048	11641 ms	11711 ms	9054 ms	276 ms
4096×4096	97436 ms	97763 ms	80283 ms	1583 ms

2.2. icx 2024.2.1

Módulos que se han utilizado para este caso:

- `module load cesga/2025`
- `module load intel/2024.2.1`

- `module load imkl`

La librería utilizada para la función de referencia es:

- `#include <mkl_lapacke.h>`

Matrix size	O2	O3	Ofast	Ofast+ipo	ref
1024×1024	1023 ms	1041 ms	1041 ms	1042 ms	43 ms
2048×2048	14029 ms	14154 ms	14136 ms	14324 ms	273 ms
4096×4096	118573 ms	118451 ms	119013 ms	120023 ms	1413 ms

Para $N = 4096$, la solución obtenida mediante Gauss–Jordan deja de coincidir con LAPACK debido a la inestabilidad numérica del método. Gauss–Jordan requiere normalizar completamente cada fila y realizar eliminación en ambas direcciones, lo que aumenta la acumulación de errores de redondeo. La implementación utilizada por LAPACK no utiliza Gauss–Jordan, lo cual la hace más eficiente y estable. Tras revisar la implementación, no se han encontrado errores lógicos, por lo que se atribuye la discrepancia al método utilizado y no al código.

3. Profiling usando VTune

Para el estudio con VTune se ha empleado la ejecución del algoritmo sobre sistemas de tamaño 2048×2048.

Métrica	ICC -O2	ICC -O3	ICC -fast	MKL dgesv
IPC	1.405	1.515	0.557	1.949
DP GFLOPS	2.632	2.847	3.354	53.405
Frecuencia media CPU	2.8 GHz	2.8 GHz	2.795 GHz	2.744 GHz
Utilización CPU (lógica)	1.5 %	1.5 %	1.5 %	4.8 %
Utilización CPU (física)	1.5 %	1.5 %	1.5 %	4.7 %
Retiring	26.7 %	28.8 %	13.7 %	46.4 %
Front-End Bound	1.4 %	1.6 %	2.5 %	2.1 %
Bad Speculation	0.3 %	0.3 %	0.0 %	0.9 %
Back-End Bound (total)	71.5 %	69.2 %	83.8 %	50.5 %
Memory Bound	45.9 %	42.5 %	60.1 %	20.1 %
L1 Bound	0.5 %	0.5 %	0.1 %	7.0 %
L2 Bound	2.7 %	3.1 %	0.1 %	0.7 %
L3 Bound	17.3 %	17.1 %	11.1 %	0.8 %
DRAM Bound	19.5 %	15.6 %	44.7 %	12.7 %
Vectorización: Packed FP (%)	100 %	100 %	99.2 %	99.2 %
Vector width efectivo	128-bit	128-bit	256-bit	256-bit
ISA vectorial detectada	AVX (128)	AVX (128)	AVX2 (256)	AVX2 (256)
DP FLOPs (% de uOps)	35.8 %	35.7 %	44.8 %	68.9 %
Non-FP uOps	64.2 %	64.3 %	55.2 %	31.1 %
FP arith / mem read ratio	0.989	0.990	0.975	1.706
FP arith / mem write ratio	1.979	1.980	1.949	21.345

Se ha realizado un análisis comparativo entre las diferentes opciones de optimización del compilador ICC usando VTune. Los resultados muestran diferencias significativas en tiempo de ejecución, eficiencia del hardware y uso de memoria, lo cual permite identificar los cuellos de botella fundamentales del algoritmo y del código desarrollado.

3.1. Rendimiento general

Las tres configuraciones del compilador ICC (-O2, -O3 y -fast) producen mejoras moderadas sobre la versión base, pero ninguna se acerca al rendimiento de MKL, que resuelve el sistema casi dos órdenes de magnitud más rápido.

La causa principal no es el compilador, sino la naturaleza del algoritmo implementado: Gauss–Jordan es computacionalmente más caro y mal estructurado para la jerarquía de memoria. En contraste, MKL utiliza algoritmos optimizados.

3.2. Eficiencia del pipeline (Retiring, IPC)

El IPC con las tres configuraciones diferentes de ICC para el algoritmo desarrollado se sitúa entre 0.55 y 1.5, muy por debajo del potencial del procesador (2.0–3.0 en código bien vectorizado). Además, el *Retiring* solo alcanza entre 13 % y 28 %, lo que indica que la mayoría de ciclos de CPU no llegan a ejecutar operaciones útiles.

En cambio, MKL alcanza un IPC de 1.95 y un *Retiring* del 46 %. Esto refleja que su código aprovecha mucho mejor las unidades de ejecución, con mayor paralelismo interno, menos dependencias y un flujo de instrucciones más eficiente.

3.3. Cuellos de botella en Back-End y Memory Bound

El análisis con VTune muestra que la implementación propia presenta un comportamiento fuertemente limitado por la memoria. Los valores de *Back-End Bound* entre el 70 % y el 84 %, junto con un *Memory Bound* del 42 % al 60 %, indican que la mayor parte del tiempo de ejecución se pierde esperando accesos a datos. Este patrón está asociado a un recorrido desfavorable de memoria —operaciones fila a fila sin *blocking*— que provoca numerosos ciclos del tipo carga, operación, *store* sin reutilización efectiva en caché. Como consecuencia, el algoritmo ejerce una gran presión tanto sobre la jerarquía de caché como sobre la DRAM, especialmente visible en la versión compilada con `-fast`, donde el *DRAM Bound* llega al 45 %.

En contraste, la rutina `dgesv` de MKL muestra un perfil mucho más equilibrado. Su *Back-End Bound* se reduce en torno al 50 % y el *Memory Bound* cae a aproximadamente un 20 %, con un acceso a DRAM claramente menor (∼12 %). Esto refleja una utilización mucho más eficiente de la jerarquía de memoria, la reordenación de bucles para mejorar la localidad y la explotación intensiva de las cachés L2/L3. En conjunto, estas optimizaciones reducen de forma drástica el coste de los accesos a memoria y explican la diferencia de rendimiento respecto a la implementación manual.

3.4. Vectorización

El estudio de VTune muestra que las versiones compiladas con ICC alcanzan niveles muy altos de vectorización (99–100 % de operaciones en punto flotante vectoriales). Sin embargo, en las configuraciones `-O2` y `-O3` la vectorización efectiva se limita a registros de 128 bits, mientras que con `-fast` se habilita AVX2 de 256 bits, aunque a costa de introducir errores numéricos debido a la agresividad de las optimizaciones. Además, el propio método de Gauss–Jordan presenta dependencias entre datos que dificultan una vectorización profunda. En consecuencia, la vectorización acelera las instrucciones, pero no resuelve el verdadero cuello de botella del algoritmo: los accesos masivos y poco reutilizables a la matriz.

En comparación, MKL también presenta una vectorización muy elevada (99.2 %), pero con una proporción significativamente mayor de operaciones en doble precisión dentro del total de micro-operaciones (69 %, frente al 35–45 % en la función desarrollada). Además, MKL explota de forma consistente vectores de 256 bits gracias a su diseño interno optimizado para AVX2. No obstante, la ventaja real no proviene solo de usar instrucciones vectoriales más anchas, sino de aplicarlas sobre bloques de datos con alta localidad y reutilización. En conclusión, la vectorización

existe en todos los casos en el algoritmo desarrollado, pero no es capaz de compensar el elevado coste del acceso a la memoria.

4. Análisis de la memoria usando Cachegrind

Para el estudio con Valgrind y Cachegrind se ha empleado la ejecución del algoritmo sobre sistemas de tamaño 1024×1024.

Métrica	**ICC -O2**	**ICC -O3**	**ICC -fast**	**MKL**
I refs	6,440,298,140	6,436,142,567	2,106,631,415	1,117,894,720
I1 misses	2,091	2,099	2,104	7,093
LLi misses	1,917	1,918	1,921	2,914
D refs	3,298,385,997	3,291,591,587	3,316,820,259	399,453,351
D1 misses	270,033,326	270,033,357	270,045,461	34,169,577
LLd misses	537,299	537,299	537,300	1,135,504
D1 miss rate	8.2 %	8.2 %	8.1 %	8.6 %
LLd miss rate	0.0 %	0.0 %	0.0 %	0.3 %
LL refs	270,035,417	270,035,456	270,047,565	34,176,670
LL misses	539,216	539,217	539,221	1,138,418
LL miss rate	0.0 %	0.0 %	0.0 %	0.1 %

Para complementar el análisis dinámico realizado con VTune, se ha ejecutado el código bajo Cachegrind con matrices de 1024×1024. A diferencia de VTune, que mide tiempo real, ancho de banda, utilización del procesador y comportamiento del *pipeline*, Cachegrind proporciona métricas precisas sobre accesos a memoria simulada, instrucciones retiradas y tasas de fallo en las distintas cachés. Este análisis resulta especialmente útil para comprender por qué la versión manual del algoritmo presenta un rendimiento tan inferior al de la rutina `dgesv` de MKL, y qué patrones de acceso a memoria son los responsables de los cuellos de botella observados.

4.1. Estructura general de los accesos a memoria

Los resultados muestran que las tres compilaciones de ICC del algoritmo desarrollado (`-O2`, `-O3` y `-fast`) presentan cifras prácticamente idénticas en términos de instrucciones ejecutadas e intensidad de accesos a memoria. Esto indica que, pese a las optimizaciones del compilador, la estructura algorítmica impone un patrón de acceso muy rígido.

Las versiones compiladas con ICC realizan aproximadamente:

- $\sim 6{,}4 \times 10^9$ instrucciones
- $\sim 3{,}3 \times 10^9$ accesos a datos (*D refs*)
- $\sim 270 \times 10^6$ fallos en caché L1 de datos

Estas magnitudes son extraordinariamente elevadas para un sistema de 1024×1024, y reflejan un patrón de acceso característico de Gauss–Jordan:

- barrido completo por filas
- falta de *blocking*
- reutilización mínima de datos en caché
- dependencia directa entre filas y columnas que obliga a recargar elementos repetidas veces

El compilador puede alterar la granularidad o el orden de algunas instrucciones, pero no puede transformar el kernel en un algoritmo de tipo BLAS-3, por lo que el coste final es prácticamente idéntico entre `-O2`, `-O3` y `-fast`.

Este comportamiento coincide con los resultados obtenidos en VTune: las compilaciones del algoritmo desarrollado están fuertemente limitadas por memoria (*Memory Bound* entre 42 % y 60 %), con un *Back-End Bound* muy elevado y bajo *Retiring*, lo que indica que la CPU pasa la mayor parte del tiempo esperando cargas desde memoria.

4.2. Eficiencia de caché y comparación con MKL

El contraste con MKL es extremadamente revelador. La rutina `dgesv` reduce los accesos a memoria a cifras muy inferiores:

- I refs: $6{,}4 \times 10^9 \rightarrow 1{,}1 \times 10^9$
- D refs: $3{,}3 \times 10^9 \rightarrow 0{,}39 \times 10^9$
- D1 misses: $270 \times 10^6 \rightarrow 34 \times 10^6$

Es decir, MKL realiza:

- 6 veces menos instrucciones
- 8 veces menos accesos a datos
- 8 veces menos fallos en L1
- 5 veces menos fallos globales

Este resultado es completamente coherente con los hallazgos de VTune, en particular:

- reduce el *Memory Bound* del 45–60 % a 20 %
- reduce drásticamente la presión sobre L3 y DRAM
- mantiene un IPC muy superior ($\approx 1{,}95$ frente a 0,55–1,5)
- presenta un *Retiring* del 46 %, frente al 13–28 % del código manual

En Cachegrind se observa que, aunque MKL pueda tener tasas de fallo algo mayores en proporción (0.1 % frente a $< 0{,}01$ %), el volumen total de accesos es tan bajo que el coste global es muy inferior.

Del mismo modo, MKL consigue reducir en más de un orden de magnitud la cantidad de datos que llegan a los niveles superiores de memoria (L3 y DRAM), lo que coincide perfectamente con la reducción del *DRAM Bound* observada en VTune (del 12 % frente al 15–45 % del algoritmo manual).

4.3. Relación con la vectorización y las uOps

Aunque VTune muestra que tanto ICC como MKL alcanzan niveles muy altos de vectorización (99 % de operaciones de coma flotante empaquetadas), Cachegrind confirma que el ancho vectorial no es el factor dominante: el número de accesos a memoria es abrumadoramente más determinante que la eficiencia en operaciones vectoriales.

Las optimizaciones de MKL reducen la carga total de trabajo a nivel de memoria, lo que permite que las unidades vectoriales funcionen con mayor continuidad, generando un IPC mayor y un volumen de *uOps* mucho más orientado a cálculo (hasta un 69 % de *FP uOps*, frente al 35–45 % en el algoritmo implementado).

5. Memory Leaks

El análisis del uso de memoria en el código revela la presencia de fugas asociadas a las estructuras asignadas dinámicamente. En concreto, las matrices `a`, `b`, `aref`, `bref` y el vector de pivotes `ipiv` se reservan mediante `malloc()` pero no se liberan explícitamente antes de finalizar la ejecución. Aunque el sistema operativo recupera toda la memoria al terminar el proceso, este patrón constituye una fuga desde el punto de vista del análisis del *heap*, ya que los bloques permanecen sin `free()` dentro del flujo del programa. En aplicaciones iterativas o en ejecuciones de larga duración, este comportamiento podría ocasionar un crecimiento progresivo del consumo de memoria.

La herramienta Valgrind Memcheck no pudo ejecutarse directamente en el entorno del clúster. No obstante, un análisis estático del código permite identificar inequívocamente estas fugas, que serían clasificadas como *definitely lost* por Memcheck en un entorno compatible. La solución es sencilla: añadir las correspondientes llamadas a `free()` al final de la ejecución para garantizar una gestión adecuada del *heap* y evitar pérdidas de memoria en ejecuciones repetidas o integradas en flujos más complejos.

6. Vectorización

El análisis detallado con Intel Advisor confirma que los bucles principales de la rutina `my_dgesv` están completamente autovectorizados por el compilador ICC. El informe muestra que el bucle dominante en la eliminación de Gauss–Jordan (línea 53) aparece en sus versiones *body*, *peeled* y *remainder*, y que todas ellas han sido vectorizadas utilizando instrucciones AVX2 de 256 bits. Además, el compilador aplica transformaciones adicionales como *loop unrolling*, *fusion* y *peeling* para maximizar la alineación y aprovechamiento de las unidades SIMD. Advisor estima un *speedup* teórico inferior a 3.5× para este bucle, lo cual es coherente con las limitaciones algorítmicas del método.

El informe también identifica bucles no vectorizables, como el de generación de la matriz mediante `rand()`, que aparece marcado como *Scalar – function call cannot be vectorized*, o bucles externos cuya vectorización no es necesaria porque el bucle interior ya está vectorizado. En conjunto, estos resultados confirman que el compilador no encuentra impedimentos técnicos relevantes para vectorizar las partes críticas del algoritmo. Sin embargo, el rendimiento global continúa estando limitado por el comportamiento de memoria: aunque las operaciones aritméticas se vectoricen eficientemente, el acceso repetido y poco reutilizable a la matriz provoca un carácter fuertemente *Memory-Bound*, de modo que la vectorización no se traduce en mejoras significativas de rendimiento.

7. Conclusión

El estudio conjunto con VTune, Cachegrind e Intel Advisor demuestra que el principal cuello de botella de la implementación del método de Gauss–Jordan no se encuentra en la capacidad de cálculo ni en la ausencia de vectorización, sino en la estructura del propio algoritmo y su patrón de acceso a memoria. La rutina realiza barridos completos por filas sin ningún tipo de *blocking*, lo que provoca un volumen masivo de accesos a datos con muy poca reutilización en caché. Este comportamiento se traduce en valores extremadamente altos de *Memory Bound* y *Back-End Bound* en VTune, así como en miles de millones de referencias a memoria y centenares de millones de fallos en caché según Cachegrind.

La vectorización tampoco supone un factor limitante. Intel Advisor confirma que los bucles principales del algoritmo se autovectorizan por completo empleando AVX2, con estimaciones de *speedup* teórico de hasta 3.5× en las secciones críticas. Sin embargo, este potencial no se manifiesta en el rendimiento real debido a que el código está dominado por los accesos a memoria. Esto

coincide con los resultados de VTune, donde se observa un IPC bajo, un porcentaje reducido de *Retiring* y un claro dominio de esperas asociadas a la jerarquía de memoria.

En comparación, la rutina `dgesv` de Intel MKL ofrece un rendimiento entre 50 y 60 veces superior. Este comportamiento no se debe al compilador, sino al algoritmo: MKL utiliza un diseño orientado a maximizar la localidad espacial y temporal, reducir el tráfico a memoria y explotar de manera óptima las cachés. El resultado es un volumen de accesos a memoria entre 6 y 8 veces menor, una reducción drástica de fallos en caché y un IPC cercano a 2, muy superior al obtenido con la implementación manual.

Además, el análisis del uso del *heap* muestra que la versión desarrollada presenta fugas de memoria, ya que las matrices `a`, `b`, `aref`, `bref` y el vector `ipiv` se asignan dinámicamente pero no se liberan explícitamente. Aunque estas fugas no afectan al tiempo de ejecución del experimento, sí representan un problema de calidad del software y serían detectadas como *definitely lost* por Valgrind Memcheck en un entorno compatible.

En conjunto, los resultados indican que:

- El código está correctamente vectorizado, pero la vectorización no compensa el elevado coste de los accesos a memoria.
- El compilador no es el factor decisivo: las diferencias entre `-O2`, `-O3` y `-fast` son marginales frente al coste intrínseco del algoritmo.
- El verdadero cuello de botella es el acceso a memoria, que limita el *throughput* de cálculo independientemente de la vectorización.
- Las optimizaciones efectivas requieren cambiar la estructura algorítmica, no los *flags* del compilador.

Por tanto, la mejora de rendimiento solo puede obtenerse adoptando algoritmos con mayor intensidad aritmética y mejor comportamiento en la jerarquía de memoria, como los empleados en bibliotecas optimizadas de tipo BLAS/LAPACK.