Tarea #3
Herramientas HPC

Iñaki Amatria Barral
Alonso Rodríguez Iglesias

Noviembre 2021 – Universidade da Coruña

1 Introducción y objetivos

En este documento se expone el proceso de análisis y optimización del código de la Tarea #2 con el objetivo de conseguir un mejor tiempo de ejecución mediante la paralelización con OpenMP y siguiendo las recomendaciones de las herramientas ofrecidas por Intel oneAPI.

2 Problemas de memoria

En lo relativo a nuestra implementación, esta es muy sencilla y todos los `malloc` o `calloc` realizados tienen sus correspondientes `free` al final de la función. Sin embargo, el código original tiene memoria reservada, que al final de la ejecución, es *still reachable*. En concreto, el programa termina sin liberar la memoria de `a`, `b`, `aref`, `bref`, `ipiv` e `ipiv2`. Esto no supone un gran problema, puesto que los recursos se liberan al finalizar el proceso, pero es una buena práctica el correcto manejo de la memoria.

Una vez solucionados estos detalles, se puede comprobar que no hay problemas de memoria mediante la ejecución de Valgrind (ver Figura 1).

3 Paralelización y vectorización

Para la paralelización del código se ha seguido un enfoque incremental. De esta forma, se describe ahora el proceso seguido para paralelizar el código.

Como nuestro código no es, todavía, complejo, previo a la implementación se ha realizado un análisis en papel sobre los patrones de acceso a memoria de los diferentes bucles del código, así como un cálculo de la complejidad que estos representan. De esta forma, hemos podido entender mucho mejor como funciona nuestra implementación en términos de caché y, así, podemos entender mejor como paralelizar el código; evitando bloqueos innecesarios, por ejemplo.

Tras esta primera aproximación puramente teórica, se observan los *hotspots* del código usando Intel VTune, que coinciden con los previstos. Esto es, los bucles `for` que realizan la descomposición LU, sustitución hacia delante y sustitución hacia atrás.

```
==50130== Command: ./target/gnu/debug.out 24
==50130==
Time taken by LAPACK: 0.10s
Time taken by my implementation: 0.00s
Result is ok!
==50130==
==50130== HEAP SUMMARY:
==50130==     in use at exit: 0 bytes in 0 blocks
==50130==   total heap usage: 50 allocs, 50 frees, 98,736 bytes allocated
==50130==
==50130== All heap blocks were freed -- no leaks are possible
==50130==
==50130== For lists of detected and suppressed errors, rerun with: -s
==50130== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 1: Salida de Valgrind una vez solucionados los errores de memoria

Después, se ha implementado una versión paralela del mismo mediante directivas de OpenMP, haciendo especial énfasis en dichos *hotspots*, así como en el último bucle de transposición, donde la paralelización es trivial. Además, gracias al análisis en papel del código, se ha podido evitar sincronizaciones entre hilos innecesarias usando la directiva `nowait`.

Por último, y siguiendo los consejos de Intel Advisor y los reportes de optimización del Intel C Compiler, se han indicado los bucles sin dependencias (en este caso, todos) mediante el uso de `#pragma ivdep`. Gracias a esta confirmación explícita por parte del programador, ICC (que no GCC) es capaz de vectorizar el código haciendo uso de las instrucciones AVX2, arrojando un *Gain Estimate* de entre 2.15x y 3.64x (ver Figura 4).

4 Métricas del código

La paralelización obtiene buenos rendimientos y la eficiencia de la paralelización es muy buena (ver Figura 5). Sin embargo, exceptuando tamaños pequeños de matrices (menores a $n = 1024$), la implementación de LAPACK es mucho mejor, entre otros, porque nuestra implementación está *memory-bound*, tal y como indica Intel VTune (ver Figura 6). La causa de esto está clara: la intensidad aritmética de nuestro algoritmo es muy baja. Es decir, tenemos un código que realiza operaciones sencillas (y muy rápidas) sobre datos que usa sólo una vez y, esto, acaba por saturar el bus y limitando la escalabilidad del código —i.e. mal rendimiento—.

5 Resultados

Habiendo visto los resultados de ejecutar el código con Intel VTune, ya estamos en condiciones de interpretar las Figuras 2 y 3, que muestran el *speedup* de las funciones paralelas `my_dgesv` con respecto a la implementación secuencial de `my_dgesv` usando 1,

2, 4, 8 y 16 *cores*[1].

Los resultados son sencillos de explicar. En general, como se pudo ver ya en la tarea anterior, compilar con O3 sobre O2 no ofrece prácticamente ningún tipo de mejora en tiempo de ejecución. Las aceleraciones son buenas, pero se puede comprobar como la presión sobre la memoria, en gran parte, evita que se pueda conseguir un speedup(p) = p. Además, por este mismo motivo, la vectorización no es extremadamente beneficiosa, con un código más intenso aritméticamente se podrían ver diferencias más notables entre una versión y la otra.

6 Conclusiones

La conclusión que se puede sacar es que las herramientas de Intel oneAPI son muy potentes. Gracias a conocer la arquitectura sobre la que se ejecutan las aplicaciones, se puede obtener muchos detalles sobre qué limita nuestro código y dónde lo podemos mejorar. Sin embargo, por potentes que sean estas herramientas y por muy bien que nos aconsejen, nosotros, como ingenieros, seguimos teniendo muchísima responsabilidad a la hora de obtener el mejor rendimiento de nuestro código. ¿Hemos programado un algoritmo bueno y eficiente? ¿Tiene una intensidad aritmética suficiente como para poder vectorizarlo y paralelizarlo y no estar limitados por la memoria? En nuestro caso hemos podido ver que, claramente, no. La intensidad aritmética de nuestro algoritmo es baja y acaba por limitar el beneficio de las paralelizaciones y vectorizaciones sobre el código. Por tanto, un siguiente paso para conseguir una mejora continuada sería revisar el algoritmo entero para mejorar esta métrica.

[1]Los tiempos de ejecución y speedups, así como los logs de ejecución, se pueden encontrar en el la carpeta del proyecto, bajo la ruta `doc/resources`.

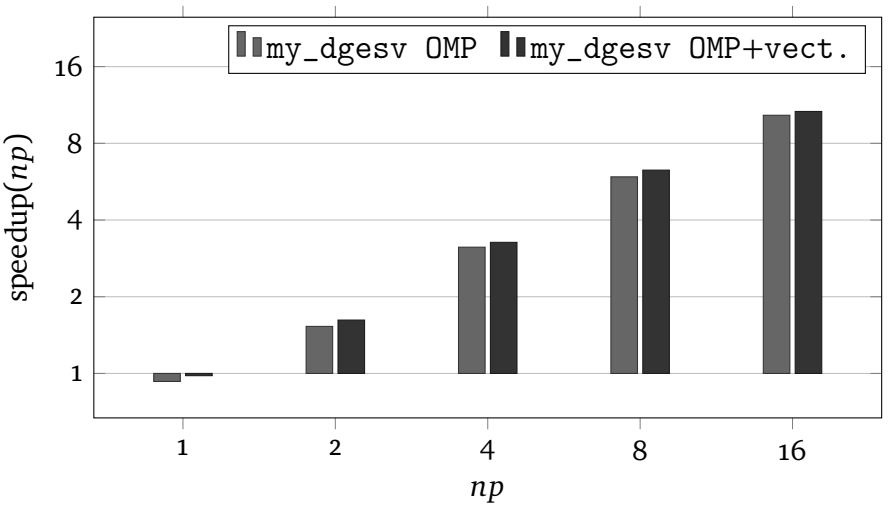

Figura 2: *Speedup* de `my_dgesv` en el supercomputador FinisTerrae 2 con optimización O2 (ICC)

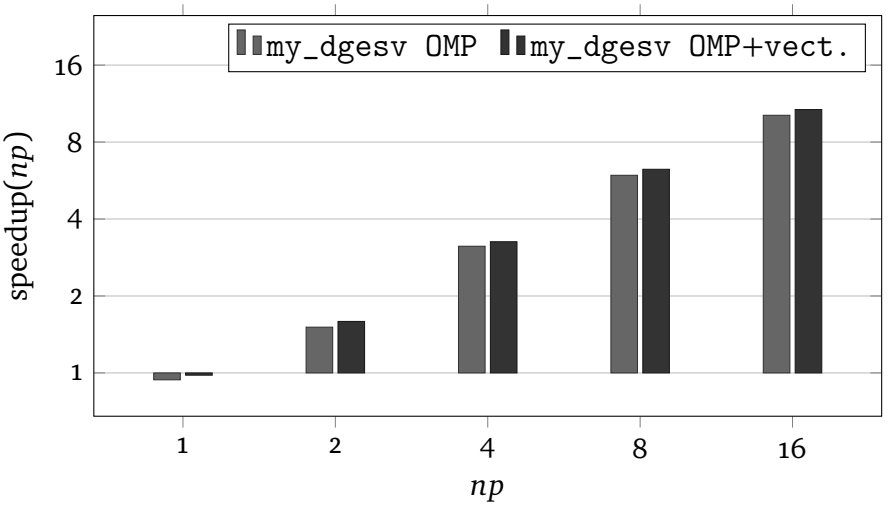

Figura 3: *Speedup* de `my_dgesv` en el supercomputador FinisTerrae 2 con optimización O3 (ICC)

Apéndice de figuras

Function Call Sites and Loops	CPU Time		Type	Why No Vectorization?	Vectorized Loops		
	Total Time	Self Time			Vector ISA	Efficiency	Gain Estimate
[loop in my_dgesvompparallel@54 at dgesv.c:75]	1,316s	1,316s	Vectorized (Body; ...		AVX; AVX2	100%	2,15x
[loop in my_dgesvompparallel@54 at dgesv.c:62]	1,000s	1,000s	Vectorized (Body; ...		AVX; AVX2	100%	2,15x
[loop in my_dgesvompparallel@54 at dgesv.c:88]	0,212s	0,212s	Vectorized (Body; ...		AVX; AVX2	90%	3,58x
[loop in my_dgesvompparallel@54 at dgesv.c:101]	0,106s	0,106s	Vectorized (Body; ...		AVX; AVX2	91%	3,64x
[loop in my_dgesvompparallel@54 at dgesv.c:98]	0,170s	0,064s	Scalar	inner loop was already vectorized			
[loop in my_dgesvompparallel@54 at dgesv.c:72]	1,344s	0,028s (	Threaded (OpenMP)	inner loop was already vectorized			
[loop in my_dgesvompparallel@54 at dgesv.c:59]	1,012s	0,012s (	Threaded (OpenMP)	inner loop was already vectorized			
[loop in generate_matrix at dgesv.c:21]	0,010s (	0,010s (	Scalar	function call cannot be vectorized			
[loop in generate_matrix at dgesv.c:21]	0,010s (	0,010s (	Scalar	function call cannot be vectorized			
[loop in generate_matrix at dgesv.c:21]	0,010s (	0,010s (	Scalar	function call cannot be vectorized			
[loop in my_dgesvompparallel@54 at dgesv.c:85]	0,220s	0,008s (	Scalar	inner loop was already vectorized			

Figura 4: Datos de rendimiento de los bucles vectorizados relevantes

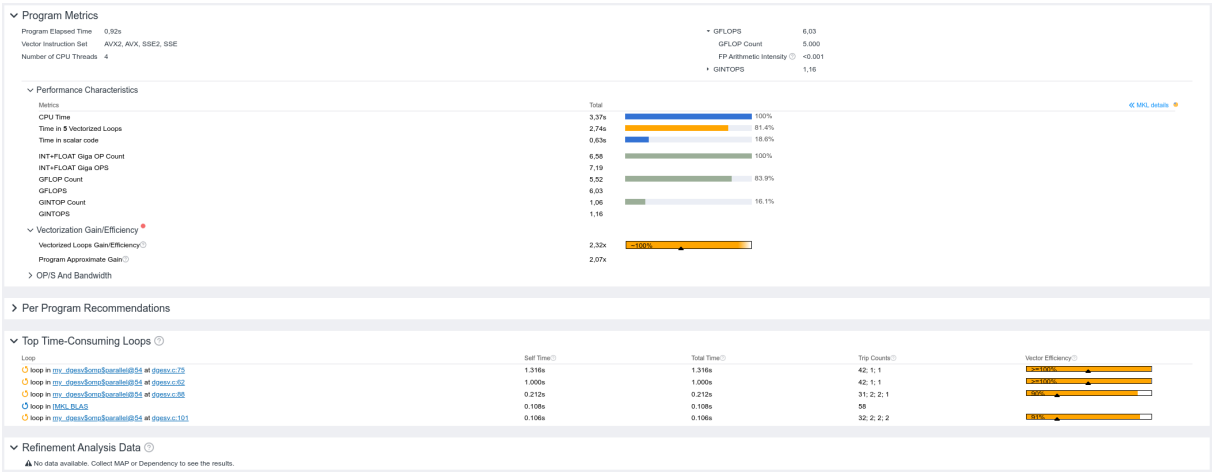

Figura 5: Métricas del programa completo, compilado con ICC

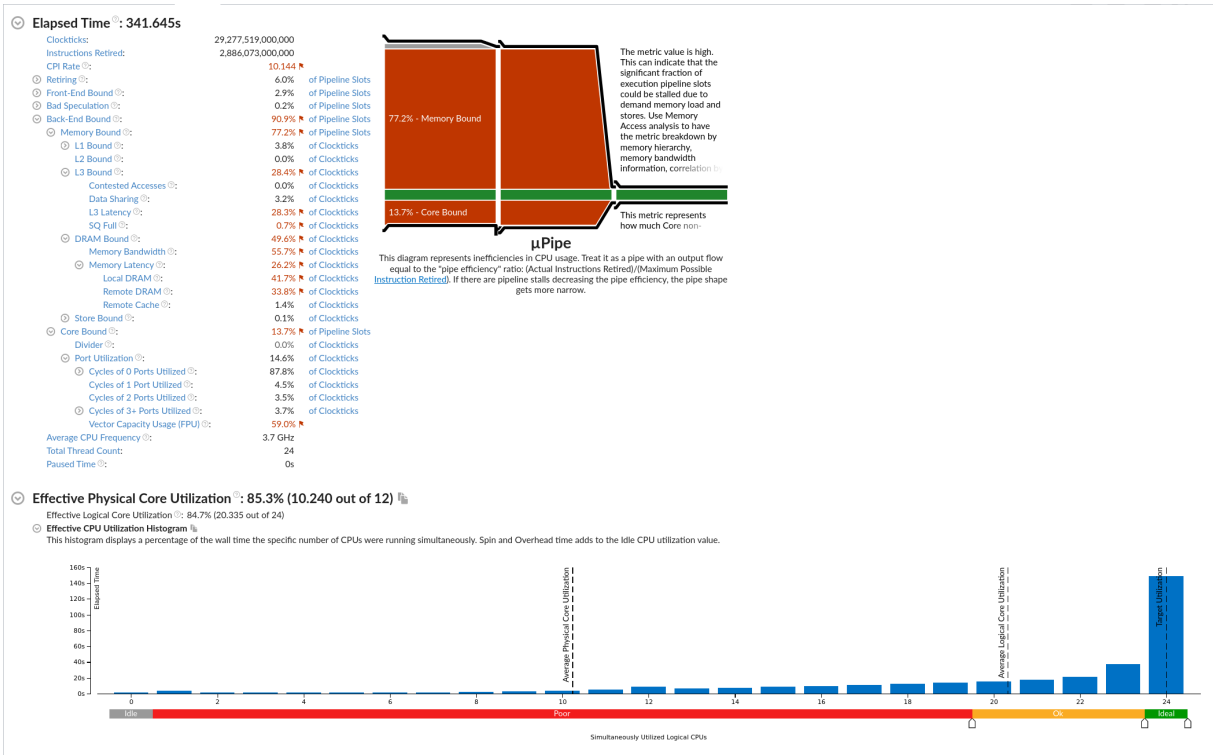

Figura 6: Datos de utilización de la arquitectura para el programa completo