Optimización Computacional del Método de Jacobi

Xabier Nóvoa Gómez, Adrián Quiroga Linares

Arquitectura de computadores
Grupo 04
{xabier.novoa,adrian.quiroga}@rai.usc.es

17 de abril de 2025

Resumen

Se implementa el método de Jacobi en C explorando distintas técnicas de optimización: mejoras en memoria caché, vectorización con AVX256 y paralelización con OpenMP. Se evalúa el rendimiento en el sistema FinisTerrae III mediante medidas de ciclos de CPU, analizando la ganancia en velocidad obtenida según el tamaño del problema y el tipo de optimización aplicada.

Palabras clave: optimización, multinúcleo, SIMD, fallos caché, OpenMP, paralelismo, scheduling...

I. Introducción

En esta práctica se aborda la implementación del método de Jacobi. El objetivo principal es analizar el rendimiento del algoritmo bajo diferentes estrategias de optimización, incluyendo la mejora del acceso a memoria caché, el uso de instrucciones vectoriales SIMD (Single Instruction, Multiple Data) y la paralelización mediante OpenMP. Como entorno de ejecución utilizaremos un nodo del supercomputador Finisterrae III del CESGA.

El informe se estructura en varias secciones. En primer lugar, se presenta una descripción detallada de los experimentos que se llevarán a cabo. Posteriormente, se exponen y analizan los resultados obtenidos durante las pruebas. Finalmente, se presentan las conclusiones derivadas de los experimentos realizados.

II. Descripción de los experimentos

Para comprobar el resultado de las multiples optimizaciones, primero creamos un ejecutable base con el algoritmo de jacobi, sin aplicar ninguna optimización, para poder tener una referencia (de aquí en adelante denominado v1).

Todos los experimentos se realizar mandandole un trabajo a un nodo del Finesterrae III, en este caso solicitamos un nodo de 64 cores y 1 gb de almacenamiento.

Para poder comparar los resultados entre múltiples optimizaciones se cuenta el número de ciclos que produce cada ejecución durante la ejecución del algoritmo (el bucle principal).

Para la matriz que vamos a resolver, primero se inicializa con valores aleatorios en cada uno de los elementos y después nos aseguramos que inicializamos la matriz como diagonal dominante, para asegurarnos la convergencia y que el sistema tenga una solución única. Además en los casos en los que las iteraciones superen las 20000 finalizamos la ejecución, para reducir el tiempo de cálculo. A continuación se muestra el pseudocódigo sobre el que vamos a estar trabajando:

```
Entradas:
a: Matriz de coeficientes (float[n x n])
b: Vector independientes (float[n])
x: Vector solución (float[n])
tol: Tolerancia (float)
max_iter: Máximo de iteraciones (int)

Variables auxiliares:
x_new: Vector nueva solución (float[n])

Para iter de 0 a max_iter:
```

```
  norm2 = 0
  Para i de 0 a n:
    suma = 0
    Para j de 0 a n:
      Si i != j:
        suma += a[i][j] * x[j]
    x_new[i] = (b[i] - suma) / a[i][i]
    norm2 += (x_new[i] - x[i])^2
  x = x_new
  Si sqrt(norm2) < tol:
    Terminar

Imprimir norm2
```

A. Optimización de uso de memoria caché

En este caso nuestro objetivo es reducir el tiempo de búsqueda (hit time), la tasa de fallos y la penalización de fallos. Por otro lado queremos aumentar el ancho de banda de caché.

Reducción del número de instrucciónes

Una reducción del número de intrucciones permite reducir el número de accesos a memoria, aumentar la eficiencia de los datos que están en la caché y mejorar la localidad espacial y temporal, haciendo que la caché sea más efectiva como veremos más adelante.

Se puede reducir el número de instrucciones evitando que se calcule varias veces una misma operación, guardando el resultado. Cuando se calcula la norma, en vez de tener que calcular dos veces la resta, se puede almacenar el resultado y así aprovechar la localidad temporal.

Calcular la raiz cuadrada de la norma para poder compararla con la tolerancia en cada iteración del bucle principal también es muy costoso y podemos ahorrarnos esa operación comparando el valor de la norma con el valor de la toleracia elevado al cuadrado.

El uso de un puntero para la fila i de la matriz a en lugar de acceder a los elementos de la fila con a[i][j] mejorar la localidad espacial, ya que se accede a los datos de forma secuencial.

División de lazos

Para evitar que `suma` se calcule utilizando la diagonal, en cada iteración del tercer bucle se está realizando una operación de comparación. Podemos eliminarla dividiendo en dos el bucle, uno para la parte superior de la matriz y otro para la parte inferior.

Desenrrollamiento de lazos

En este caso por cada iteración en cada bucle vamos a calcular el valor de `suma` de varias iteraciones a la vez, lo que permitirá reducir el número de iteraciones, por lo tanto el número de comprobaciones del bucle. Además permite un mayor aprovechamiento de la caché ya que se mantiene la información en la caché sin que se produzca cambio de contexto.

En nuestra implementación se ha optado por desenrollar los bucles de 10 en 10 elementos. Esta decisión se basa en que los tamaños de las matrices proporcionadas en los ejemplos son múltiplos de 10, lo cual permite aprovechar al máximo el desenrollamiento sin necesidad de gestionar casos residuales o iteraciones sobrantes. De este modo, se evita la ejecución de bucles adicionales que gestionan las iteraciones no cubiertas por el bloque principal, los cuales, dependiendo del tamaño específico del problema, podrían tener un comportamiento menos eficiente y afectar negativamente al rendimiento global.

Realización de operaciones por bloques

En lugar de acceder a filas o columnas enteras, las subdividimos en bloques y reusamos datos antes de que el bloque sea reemplazado de la caché. Requiere más acceso de memoria pero mejora la localidad. Es importante tener en cuenta que elegir el tamaño de bloque adecuado para que los datos utilizado en un momento dado quepan en la memoria caché.

$$blockSize = \frac{lineSize}{sizeDataTipe}$$

Método de Jacobi con reducción de instrucciones, división de lazos y desenrollamiento de lazos

Finalmente después de aplicar las anteriores optimizaciones quedaría un código como el siguiente:

```
Para i = 0 hasta n-1:
  a_fila = a[i]
  suma = 0.0f
```

```
// Parte 1: debajo de la diagonal
Para j = 0 hasta i-10 paso 10:
    suma += a_fila[j] * x[j]
    ...
    suma += a_fila[j+9] * x[j+9]

Para j = i hasta i-1:
    suma += a_fila[j] * x[j]

  // Parte 2: encima de la diagonal
Para j = i+1 hasta n-10 paso 10:
    suma += a_fila[j] * x[j]
    ...
    suma += a_fila[j+9] * x[j+9]

Para j = n-10 hasta n:
    suma += a_fila[j] * x[j]

xi_nueva = (b[i] - suma)/a_fila[i]
diferencia = xi_nueva - x[i]
xNuevo[i] = xi_nueva
norma += diferencia * diferencia
```

B. Procesamiento vectorial SIMD

Las extensiones SIMD (Single Instruction, Multiple Data) permiten realizar la misma operación sobre varios datos de forma paralela, aumentando significativamente el rendimiento. En este proyecto utilizamos las extensiones AVX256, que permiten operar con registros de 256 bits, equivalentes a 8 valores 'float' de 32 bits cada uno.

Para evitar problemas al acceder a memoria alineada y optimizar el cálculo, usamos una variable auxiliar llamada `diagonal`, que almacena previamente los valores de la diagonal de la matriz. De este modo, establecemos la diagonal de la matriz original a cero, eliminando la necesidad de comprobar la condición `i != j` durante el procesamiento vectorial. Además, usamos una variable auxiliar llamada `temp` donde cargamos el resultado de la multiplicación vectorial y nos permite incrementar el valor de `suma`.

```
Para i = 0 hasta n-1:
    a_fila = a[i]
    suma = 0
    Para j = 0 hasta n-8 paso 8:
        vecA = cargar_8_floats(A[i][j])
        vecX = cargar_8_floats(x[j])
        vecMul = vecA * vecX
        vecStore(temp, vecMul)
        suma += temp[0] + ... + temp[7]

    Para j hasta n-1 paso 1:
        suma += a_fila[j] * x[j]
```

C. Programación paralela en memoria compartida (OpenMP)

La programación paralela en memoria compartida permite distribuir el trabajo entre múltiples hilos de ejecución que comparten el mismo espacio de memoria.

En nuestro caso nos permite paralelizar el cálculo iterativo de un método numérico , dividiendo el bucle principal entre hilos. Para obtener un buen rendimiento es esencial encontrar un equilibrio adecuado entre el número de hilos (`numHilos`) y la política de reparto de trabajo (`schedule`). También utilizamos una cláusula `reduction` para acumular correctamente la variable `norma` sin condiciones de carrera.

```
paralelo (i = 0 hasta n) con reducción
        en norma y numHilos:
    a_fila = a[i]
    suma = 0.0f
    /*
    *   mismo pseudo-código que en la
    *   última version del apartado B
    */
```

III. Resultados

A continuación, se presentan los resultados obtenidos tras la ejecución de los distintos experimentos.

Para obtener estos resultados, utilizamos el supercomputador **Finisterrae III**, ejecutando las diferentes versiones del código mediante un script de Bash automatizado. Por cada combinación de versión y tamaño del problema, realizamos **15 ejecuciones independientes**, utilizando el comando `sbatch script.sh` para enviar los trabajos a la cola de ejecución. De las 15 mediciones obtenidas por cada configuración, calculamos la **mediana** como medida representativa del tiempo de ejecución, minimizando así el efecto de posibles fluctuaciones o valores atípicos.

En la versión **v2**, se aplicó una combinación de optimizaciones que ofrecieron los mejores resultados: reducción del número de instrucciones, división de bucles y desenrollamiento manual de los mismos. Esta combinación permitió mejorar la eficiencia computacional sin comprometer la precisión del algoritmo.

En cuanto a la versión **v4**, que incorpora paralelismo mediante OpenMP, para las comparaciones con el resto de versiones (v1, v2 y v3), utilizamos la configuración con **64 hilos**, ya que fue la que obtuvo el mejor rendimiento entre todas las variantes evaluadas.

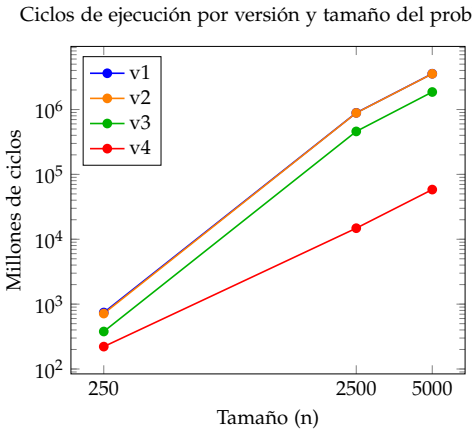

Figura 1: *Comparación de millones de ciclos de ejecución entre versiones para distintos tamaños de entrada*

La figura 1 muestra la evolución del número de millones de ciclos de CPU requeridos por cada versión del programa en función del tamaño del problema. Gracias a esta figura podemos observar una clara mejora progresiva en el rendimiento desde la versión **v1** hasta la **v4**. Esta evolución va a ser desglosada en los siguientes apartados de este capítulo.

A. Programa optimizando el uso de la caché

Al aplicar las optimizaciones descritas en el apartado A se observa una ligera mejora de rendimiento con un una reducción del número de ciclos. Esto se debe a varias razones: Al reducir el número de instrucciones, aumenta la localidad espacial y temporal, los datos que están en la caché se pueden reutilizar más veces sin tener gastar tiempo en recuperarlos de la memoria. La eliminación de la comprobación de la diagonal en el bucle que calcula el valor de `suma`, reduce también el número de ciclos. Al utilizar dos bucles, la predicción de salto es mucho más fiable, ya que solo falla al final de cada uno de los bucles. El desenrrollamiento de lazos, también aumenta el rendimiento, ya que reduce el número de iteraciones y en cada iteración se aprovecha más la localidad espacial sin que se produzcan cambios de contexto.

Inicialmente se planteó el uso de bloques para mejorar aún más el rendimiento en términos de localidad. Sin embargo, durante las pruebas experimentales, observamos que esta técnica no proporcionaba mejoras reales. De hecho, los resultados eran incluso más desfavorables, probablemente debido al incremento en la complejidad del código (más bucles y cálculos adicionales), lo que introducía un mayor overhead y afectaba negativamente al rendimiento. Por esta razón, decidimos prescindir de esta técnica en la versión final optimizada.

Como se observa en la Figura 2, el speed-up conseguido respecto a la versión original (v1) es ligero. Sin embargo, a medida que aumenta el tamaño de la matriz, el beneficio relativo disminuye. Esto se debe a que el coste de acceso a memoria principal se vuelve más relevante y la eficiencia de las optimizaciones orientadas a caché pierde peso frente al volumen total de datos procesados.

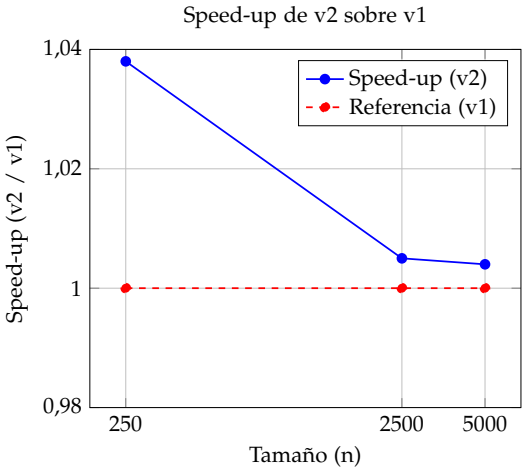

Figura 2: *Comparación de velocidad: v2 sobre v1 sin optimizar*

B. Programa optimizado y paralelizado utilizando procesamiento vectorial (SIMD)

En esta versión se ha incorporado el procesamiento vectorial mediante instrucciones SIMD. En particular, se ha aplicado esta técnica al cálculo de la variable `suma`. Este enfoque reduce de forma significativa el número de ciclos de CPU requeridos, al realizar 8 multiplicaciones en paralelo, como se puede observar en la Figura 3. En dicha gráfica se muestra el *speed-up* logrado por la versión v3 respecto a la v2, evidenciando una mejora de rendimiento cercana al doble en todos los tamaños de problema analizados.

Además, el rendimiento de esta versión se mantiene estable a medida que crece el tamaño de la matriz, lo cual indica que la vectorización sigue siendo efectiva incluso con cargas de trabajo mayores. Esto contrasta con las optimizaciones basadas exclusivamente en el uso de caché, cuya eficiencia relativa decrece en matrices de gran tamaño.

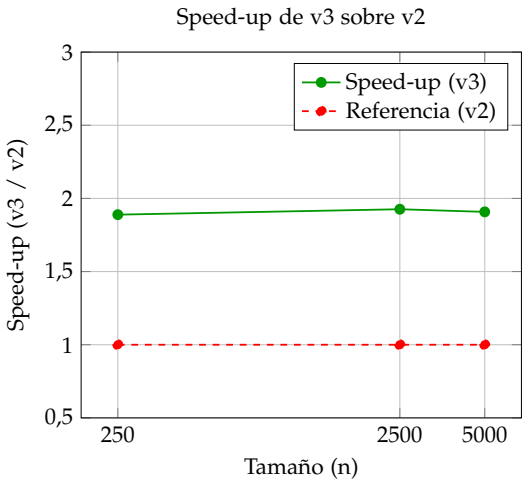

Figura 3: *Comparación de velocidad: v3 sobre v2*

C. Programa optimizado paralelizado utilizando programación paralela en memoria compartida (OpenMP)

La paralelización con memoria compartida, permite acelerar mucho los calculos, ya que se divide el trabajo entre varios nucleos de procesamiento que trabajan de manera concurrente. Además permite aprovechar más otras optimizaciones anteriores como el desenrrollamiento de lazos. Como los hilos cuando acaban cada iteración tienen que cargar los datos de la memoria, es más eficiente si pueden aprovechar la localidad espacial y cargar varios a la vez. Así se reduce el número de accesos a memoria principal y por tanto el número de ciclos.

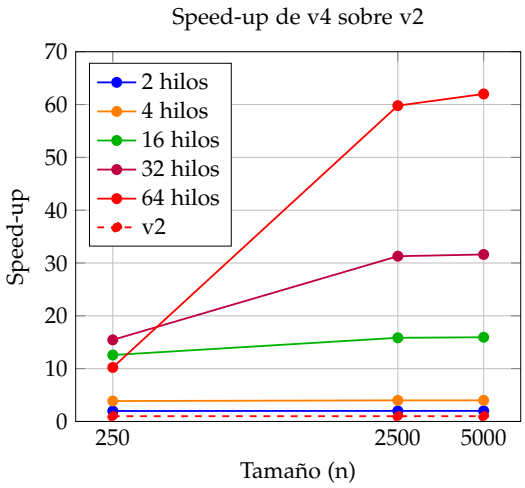

Figura 4: *Speed-up de v4 sobre v2 con diferentes números de hilos*

En la figura 4 se puede observar el aumento de rendimiento según aumenta el número de hilos, con pruebas ejecutadas con más de 64 hilos el rendimiento baja drasticamente. Es importante recalcar que el procesador en el que realizamos las pruebas tiene 64 cores, por lo que con 64 hilos estamos aprovechando la capacidad del procesador completamente. Una vez que superamos ese número de hilos, los cambios de contexto que se tienen que llevar a cabo entre cada hilo provocan una bajada del redimiento

Mecanismos de schedulling en OpenMP

OpenMP permite definir varios tipos de "scheduling" para los bucles que se ejecutan de manera paralela. Estas directivas nos permiten establecer como queremos que las iteraciones de los bucles se repartan entre los hilos. El resumen de especificaciones de OpenMP proporciona una visión clara de las directivas disponibles [3]. Como las operaciones de nuestro bucle paralelizado son uniformes, una directiva en la que se reparta la carga de cada hilo, de forma equitativa, es ideal. En este caso la directiva `static` consigue los mejores resulta-

dos.

Uso de la clausula de reduction

Para el cálculo de la `norma`, la suma de cada hilo a la vez, en la misma variable, podría causar carreras críticas, por lo que para evitar esto, se utiliza la cláusula `reduction(+=norma)`, provocando que cada hilo acumule su propio valor de `norma`. Finalmente cuando finaliza la ejecución paralela, se suman los valores de las normas de todos los hilos. Con esto evitamos tener que utilizar el código o utiliza clausulas atomic innecesariamente.

D. Optimización manual frente a optimización a nivel de compilador

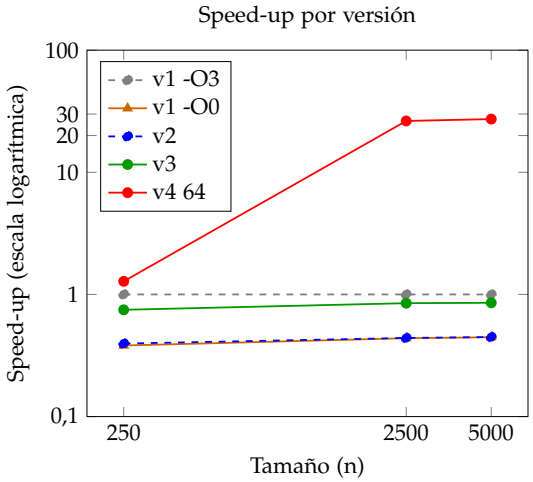

Figura 5: *Comparación de Speed-up por versión y tamaño*

En la figura 5 se muestra el speed up de las diferentes optimizaciones que se han aplicado manualmente frente a las optimizaciones que aplica el compilador automáticamente con la opción -O3. Esta opción permite que el compilador, de manera automática, implemente algunas optimizaciones, como el desenrollamiento de lazos, una mejor predicción de salto y vectorización. Como la opción -03 aplica las mejoras que hemos aplicado en v2 y optimiza los bucles con extensiones vectoriales, es entendible que produca un mejor rendimiento con respecto a v2.

IV. Conclusiones

Con el objetivo de mejorar el rendimiento del algoritmo de Jacobi computacionalmente, se partió de un código base al que se le fueron aplicando diferentes optimizaciones de forma tanto manual como automática por el compilador. A partir de estas optimizaciones se crearon 4 ejecutables.

Analizando el resultado de los 3 ejecutables, en primer lugar en las optimizaciones de uso de caché, el desenrrollamiento de lazos consigue una reducción significativa del número de iteraciones, sin embargo al realizar las operaciones por bloques no se aprecia una mejora. Partiendo del primer ejecutable, se creó otro ejecutable aplicando las extensiones AVX256, consiguiendo una mejora de rendimiento gracias a la posibilidad de poder operar con 8 floats a la vez. En el último ejecutable se explora la paralelización del algoritmo, mediante la cual conseguimos un aumento de rendimiento sustancial, hasta alcanzar el máximo con 64 hilos. En investigaciones futuras sería de especial interés analizar el resultado de utilizar las extensiones AVX512, que permiten operar utilizando registros de 512 bits.

Referencias

[1] Intel Corporation. *Intel Intrinsics Guide.* Disponible en: `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#`, [en línea], última visita: 17 de abril de 2025.

[2] CESGA. *FT3 User Guide.* Disponible en: https://cesga-docs.gitlab.io/ft3-user-guide/index.html, [en línea], última visita: 17 de abril de 2025.

[3] OpenMP Architecture Review Board. *OpenMP 3.0 Summary Specification (C/C++).* Disponible en: https://www.openmp.org/wp-content/uploads/OpenMP3.0-SummarySpec.pdf, [en línea], última visita: 17 de abril de 2025.