Programación del método Jacobi con distintos grados de optimización

Carlos Moldes Peña, Pablo Mouriño Lorenzo

Arquitectura de computadores
Grupo 03
{carlos.moldes,pablo.mourino.lorenzo}@rai.usc.es

Resumen

En este proyecto, tuvimos el objetivo de realizar la implementación del método Jacobi en punto flotante para la resolución de sistemas de ecuaciones con distintas estrategias capaces de reducir el número de ciclos por ejecución del programa y probarlo para distintos tamaños del problema. Estas estrategias consistían en tres: un programa secuencial con optimizaciones del uso de memoria caché sobre el programa base, un programa utilizando procesamiento vectorial SIMD y un programa utilizando programación paralela en memoria compartida(OpenMP). Gracias a esto, fuimos quienes de optimizar este método consiguiendo una clara mejoría en el rendimiento y ahorrando de esta manera tiempo.

Palabras clave: Optimización, SIMD, paralelismo, eficiencia, speedup, OpenMP...

I. Introducción

Se nos presenta en este trabajo el método de Jacobi, un algoritmo iterativo para resolver sistemas de ecuaciones lineales cuya solución se va modificando en cada iteración.

El objetivo se basa en implementar este método usando diferentes estrategias que busquen mejorar el rendimiento del programa lo máximo posible. Para comprobar la optimización en cada mejora, se tomarán medidas del número de ciclos e iteraciones necesarias para obtener el resultado y se proporcionarán distintos tamaños del problema para comparar distintas situaciones.

Las modificaciones a implementar a partir del código base son tres:

1. Optimizaciones basadas en el uso de memoria caché
2. Paralelismo de datos (procesamiento vectorial SIMD)
3. Paralelismo en memoria compartida (OpenMP)

Estas diferentes técnicas de optimización se explicarán con más detalle en sus respectivos apartados del informe.

Pasaremos a analizar en los siguientes apartados los programas hechos en esta práctica, analizando las modificaciones de cada uno. Posteriormente compararemos unos con otros con distintas optimizaciones, mostrando sus correspondientes resultados y terminaremos con una conclusión donde repasaremos todo el documento.

II. Programa secuencial base

El primer programa que usaremos será el algoritmo de Jacobi por defecto, sin recibir ninguna clase de optmización, y para ello nos ayudaremos del pseudocódigo de la figura 1.

Primero de todo, los argumentos que le pasaremos a la función serán: una **matriz a** que almacena los coeficientes de las incógnitas del sistema, pasándola dinámicamente como un doble puntero para las filas y columnas; un **vector b** de términos independientes del sistema; el **vector x** solución, que se irá actualizando a medida que se ejecute el algoritmo; la **tolerancia** máxima y el **número máximo de iteraciones**, representando estos dos últimos argumentos las condiciones de parada.

Cómputo:

```
Para iter (int) desde 0 hasta iter:
    norm2 = 0: norma del vector al cuadrado (float)

    Para i (int) desde 0 hasta n:
        sigma = 0.0 (float)
        Para j desde 0 hasta n:
            Si i ≠ j:
                sigma += a[i][j] * x[j]
        x_new[i] = (b[i] - sigma) / a[i][i]
        norm2 += (x_new[i] - x[i]) ^ 2

    x = x_new
    Si sqrt(norm2) < tol:
        Terminar
```

Figura 1: *Pseudocódigo algoritmo Jacobi*

Será necesario iniciar estas variables que usaremos. Esta inicialización será pseudoaleatoria, ya que inicializaremos la semilla de generación de números de forma que dependa del tamaño de la matriz; `srand(n)`. Una vez tenemos esto, reservaremos memoria para las estructuras de forma que quede alineada con la caché, por lo que usaremos `aligned_alloc()` o `_mm_malloc()`. Una vez realizado, procedemos a introducir valores a la matriz y al vector de términos independientes. Para esto, realizaremos un bucle que recorra la matriz fila por fila, inicializando cada fila columna por columna. A medida que hacemos esto, almacenamos en una variable el sumatorio de la fila para que cuando termine de inicializarla, se le añada este valor a la diagonal. De forma similar, pero con un solo bucle, inicializamos el vector de términos independientes. Es importante destacar que todos los valores se computan de la forma `(float) rand() / RAND_MAX`

Llegado el momento de inicializar la ejecución del método, también necesitaremos crear una variable donde vayamos acumulando las nuevas soluciones, para luego pasárselas al vector solución para actualizarlo.

Una vez hecho esto, iniciamos el contador de ciclos para medir el algoritmo de Jacobi. Al finalizar de esto, paramos el contador, no contabilizando la impresión de valores.

III. Optimizaciones basadas en uso de caché

En esta nueva implementación, el objetivo es optimizar el código base, obteniendo el mismo resultado final y reduciendo el tiempo de ejecución mediante el uso de mejoras de la memoria caché.

En nuestro caso, nuestras implementaciones fueron las siguientes:

1. **Reducción de instrucciones**
   En primer lugar, realizaremos algunos cambios que nos reduzcan el número de instrucciones. Concretamente, haremos 2 optimizaciones en este apartado: cuando sumamos el cuadrado de las diferencias, decidimos almacenar la diferencia en una variable aparte para que de esta forma sólo se calcule una vez, consiguiendo una reducción de 1M de ciclos; por otro lado, al actualizar el vector solución, lo hicimos con intercambio de punteros creando un puntero auxiliar, lo cual nos reducirá 2M de ciclos. Estas mejoras, si bien no son tan eficientes como las que próximamente comentaremos, son un buen recurso en el código para conseguir una mayor optimización en cada iteración y reducir algunos ciclos en sus respectivas operaciones. Podemos ver el código de estas mejoras que acabamos de mencionar en 2 y 3, respectivamente.

2. **Desenrollamiento de lazos**
   Para esto, desenrollamos por un lado el bucle de las filas de la matriz para que fuese de dos en dos, teniendo que declarar dos `sigma` donde se van acumulando las contribuciones de las demás incógnitas, y actualizando la solución de la fila `i` y de la `i+1` en cada iteración, consiguiendo de esta forma mucha más eficiencia.

   Por otro lado, también nos encargamos de desenrrollar el bucle de las columnas de 4 en 4, permitiendo procesar 8 elementos en total en una iteración. Para lograr esto, decidimos comprobar en todo momento que no sumásemos la diagonal ya que, al procesar dos filas a la vez, esta varía. Esto nos consiguió una reducción de aproximadamente 30M de ciclos. Otra alternativa sería dividir el bucle en 2 fases: antes y después de la diagonal. Sin embargo, como desenrollamos el bucle

de las filas también, la diagonal cambia respecto de una fila a otra, complicando la lógica y produciendo más ciclos. Un ejemplo del desenrollo de bucles que acabamos de comentar se puede observar en el código 1. Esta figura nos muestra las dos variables `sigma`, cada una para una fila y, a su vez, los bucles que se encargan de procesar las columnas restantes, en caso de que `n` no sea múltiplo de 4, ya que si esto ocurre, el bucle principal encargado de operar sobre los elementos no cubrirá todos los elementos.

3. **Realización de operaciones por bloques** Antes de llegar a la conclusión de que desenrollar el bucle por filas nos reducía ciclos, probamos también los bloques con un *blocking size* de 64. Es importante esta afirmación previa ya que el código tiene el bucle de las columnas desenrollado y dividido en 2 para evitar comprobar la diagonal, pero como ya mencionamos antes, al desenrollar el bucle por filas se nos presentaba el problema de tener 2 diagonales diferentes, siendo mejor desenrollar el bucle en menos elementos por fila (obteniendo el mismo resultado al tener dos filas) y comprobar con if las diagonales.

```
float sigma1 = 0.0, sigma2 = 0.0;
for (; j < n; j++) {
if (i != j) {
    sigma1 += a[i][j] * x[j];
}
if (i + 1 < n && i + 1 != j) {
    sigma2 += a[i + 1][j] * x[j];
}
}
```

Listing 1: *Desenrollo de bucles*

```
float diff1 = x_new[i] - x[i];
norm2 += diff1 * diff1;
if (i + 1 < n) {
float diff2 = x_new[i + 1] - x[i + 1];
norm2 += diff2 * diff2;
}
```

Listing 2: *Almacenamiento de diferencia en variable*

```
float* temp = x;
x = x_new;
x_new = temp;
```

Listing 3: *Intercambio de punteros*

IV. Paralelismo a nivel de datos (SIMD)

El siguiente nivel de optimización que recibirá el método de Jacobi será el paralelismo. En este apartado se trata el paralelismo a nivel de datos con procesamiento vectorial SIMD. El paralelismo a nivel de datos SIMD (Single Instruction, Multiple Data) es un modelo de paralelismo en el que una única instrucción se aplica simultáneamente a múltiples datos. Este enfoque permite que múltiples operaciones sean ejecutadas en paralelo en diferentes elementos de datos, lo que mejora la eficiencia conseguida. Utilizando SIMD, aprovechamos los registros que tiene el procesador los cuales pueden manejar múltiples elementos de datos en paralelo, realizando la misma operación en cada uno de estos elementos en un solo ciclo de reloj.

En nuestro experimento, como debíamos operar con las extensiones AVX256, decidimos sustituír las operaciones realizadas en el bucle desenrollado por las operaciones SIMD. De esta forma, nos permitirá mantener las optimizaciones basadas en el uso de caché, y además, mantener el bucle desenrollado pero con menos instrucciones. De esta forma, aumentaremos la eficiencia de nuestro programa, dado que realizaremos varias operaciones en un ciclo de reloj. Sin embargo, debido a que el método de Jacobi debe excluír el valor de la diagonal, se nos presenta un problema para manejar esto. Nuestra solución consistirá en una leve sobrecarga al código en la que con un bucle, almacenaremos la diagonal de la matriz en un vector y, a su vez, igualamos la diagonal de la matriz a cero. Esto, a pesar de ser una sobrecarga y tener que contabilizarla respecto a la versión anterior, nos permite también eliminar todas las cláusulas `if` que se encargaban de comprobar este requisito.

El bucle resultante será el visible en el código 4.

```
for (j = 0; j <= n - 8; j += 8) {
    __m256 va1 = _mm256_load_ps(&a[i][j
        ]);
    __m256 vx = _mm256_load_ps(&x[j]);
```

```
4
5       //Cargamos 8 elementos de la fila i
            +1 en caso de que exista
6       __m256 va2 = (i + 1 < n) ?
            _mm256_load_ps(&a[i + 1][j]) :
            _mm256_setzero_ps();
7
8       __m256 mul1 = _mm256_mul_ps(va1, vx
            );
9       __m256 mul2 = _mm256_mul_ps(va2, vx
            );
10
11      _mm256_store_ps(temp1, mul1);
12      _mm256_store_ps(temp2, mul2);
13
14      for (int k = 0; k < 8; k++) {
15          sigma1 += temp1[k];
16          if (i + 1 < n) sigma2 += temp2[
                k];
17      }
18  }
```

Listing 4: *Bucle con operaciones SIMD*

V. Paralelismo en memoria compartida

La última optimización a realizar sobre el código base se basa en el uso de hilos con **OpenMP**. OpenMP nos facilita la programación con paralelismo en programas que emplean memoria compartida, este funciona creando múltiples hilos de ejecución que comparten el mismo espacio de memoria, permitiendo dividir tareas o bucles entre ellos para acelerar el procesamiento.

En esta versión se nos solicita que experimentemos con diferentes parámetros para, una vez observados los diferentes comportamientos del programa, lleguemos a la mejor combinación posible. Los diferentes experimentos solicitados son:

1. **Variación del número de hilos**
   El número de hilos es una característica que va, en cierta parte, de la mano con el tamaño de la matriz. Sin embargo, entre los tres tamaños de matriz con los que experimentamos, hay tendencia a que el mejor número de hilos es 64, aprovechando los 64 cores de los que dispone un nodo del `Cesga`. Es destacable mencionar que con un `n` pequeño, funcionan mejor 32 hilos que 64.
   Para establecer el número de hilos a usar en el programa se debe llamar a la función `omp_set_num_threads(num_threads);`

2. **Mecanismos de scheduling**
   Estas cláusulas nos permiten repartir el trabajo entre los hilos que se ejecutan en la zona deseada. Las cláusulas `dynamic` y `guided` asignan un número de iteraciones a los hilos y, cuando las terminan, solicitan más. El número de iteraciones es variable, pero la diferencia es que con `guided`, el número de iteraciones dadas a un hilo se va reduciendo. Como es de esperar `static` otorga un número igual de iteraciones a todos los hilos, además de asignarse anticipadamente. En nuestro caso, **esta última cláusula es la que mejor nos funciona**. Esto dependerá de las matrices y su tamaño, que las nuestras serán cuadradas, y del número de hilos trabajando, permitiendo o no un reparto equivalente entre los hilos. Además, las asignaciones dinámicas del trabajo suponen más carga en la gestión de las mismas.

3. **Regiones críticas, reducción y operaciones atomáticas** En esta versión del programa, una carrera crítica ocurre cuando varios hilos acceden y modifican una misma variable de forma concurrente, provocando resultados inconsistentes. Para evitarlo, se utilizan las secciones de carreras críticas. Por otro lado existen las reducciones, en las cuales cada hilo tiene su resultado privado sobre una variable. Las reducciones permiten combinar los resultados parciales de varios hilos en una sola variable compartida de forma segura. Por último, las operaciones atómicas son instrucciones que garantizan que una operación sobre una variable compartida se ejecute de forma indivisible, evitando condiciones de carrera pero sin bloquear toda una sección de código. Con esto explicado, el mejor resultado obtenido fue usando la cláusula **reduction**, ya que esta permite el trabajo simultáneo de los hilos en un bloque de código y, muy importante, sobre una copia privada de la variable especificada, ya que

`critical` bloquea un bloque de código y `atomic` tiene que gestionar el acceso a la variable **compartida**.

Con estos apartado explicados, la mejor combinación obtenida es un único pragma, como vemos en el código 5.

```
#pragma omp parallel for schedule(
    static) shared(a,b,x,x_new,n)
    private(i,j,diff1,diff2,sigma1,
    sigma2) reduction(+:norm2)
for(i=0; i<n; i+=2){
    sigma1=0, sigma2=0;
    //Aprovechamos las mejoras de la
        versi n v2, como el desenrollo
         de bucles
    for (j = 0; j <= n - 4; j += 4) {
        {...}
    }
    //Calculamos, en caso de ser
        necesario, las iteraciones
        restantes
    int k = 0;
    for (k = j; k < n; k++) {
       {...}
    }
    //Calculamos los nuevos valores de
        x[i] y x[i+1]
    {...}
    //Calculamos las diferencias para
        la norma
    {...}
}
```

Listing 5: *Ubicación pragma OpenMP*

VI. Resultados

En este apartado vamos a exponer comparaciones de los resultados que hemos obtenido de cada uno de los códigos de la práctica para comprobar la ganancia en velocidad (también llamada speedup). Todas estas medidas las conseguimos ejecutando los diferentes programas en el `Finisterrae III` del CESGA y usando el comando `sbatch` para ejecutar un script que creamos especialmente para iniciar los códigos. Concretamente realizamos las siguientes tablas:

A. Sec. base vs Sec. optimizada

En esta primera comparación queremos ver cuanto mejora la versión secuencial optimizada con respecto a la secuencial base sin ninguna mejora en el compilador (-O0). Para los tamaños indicados en el eje

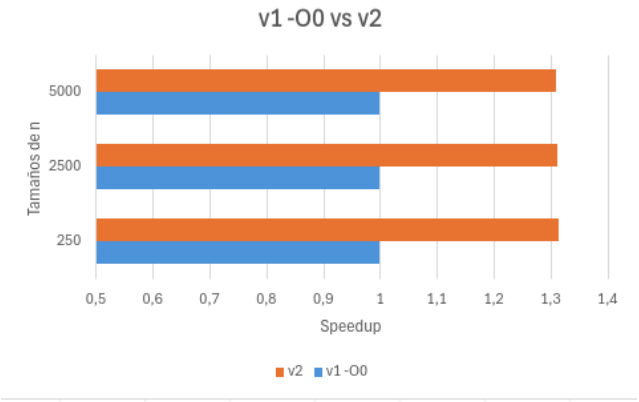

Figura 2: *v1 -O0 con v2*

vertical, como podemos observar, siempre supone una mejoría de aproximadamente 1,3 en todos los tamaños, es decir, un 30 % de optimización, lo que supone una buena reducción del tiempo de ejecución, mas comparado con otras gráficas, sigue habiendo bastante margen de mejora.

B. Sec. Optimizada vs SIMD vs OpenMP

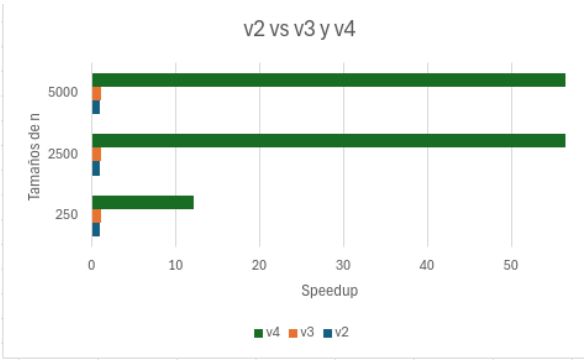

Figura 3: *Secuencial optimizada vs SIMD vs OpenMP*

Posteriormente, comprobamos el caso de la versión secuencial optimizada con respecto a SIMD y a OpenMP, y como podemos ver, mientras que la versión usando SIMD mejora apenas un 10 %, la versión utilizando OpenMP se dispara y mejora ampliamente el programa, especialmente para los tamaños de 2500 y 5000, dando a entender que esta implementación es la óptima a la hora de realizar este algoritmo.

C. Sec. base vs Sec. optimizada vs SIMD vs OpenMP

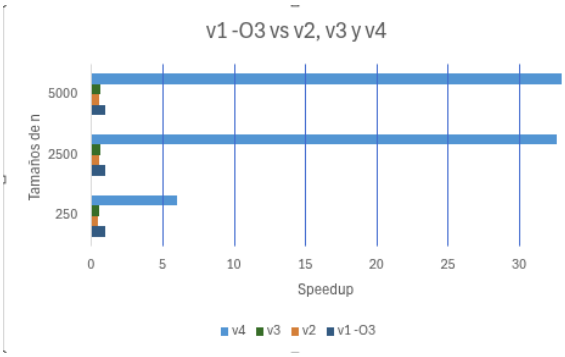

Figura 4: *Secuencial base vs Secuencial optimizada vs SIMD vs OpenMP*

En esta gráfica se aprecia el speedup conseguido en las versiones secuancial optimizada, SIMD y OpenMP, todas respecto a la versión base compilada con -O3. Como podemos apreciar, la dominancia de OpenMP sigue siendo muy clara al dividir el trabajo entre diferentes hilos pero, sin embargo, la versión secuencial optmizada y la versión SIMD tienen peor rendimiento que las mejores hechas por el compilador.

D. speedup de N

Como es obvio, ejecutar el programa con `OpenMP` con un número diferente de hilos dará una eficacia diferente. Es por eso que se ejecutaron diferentes cantidades de hilos para los tres tamaños de las matrices.

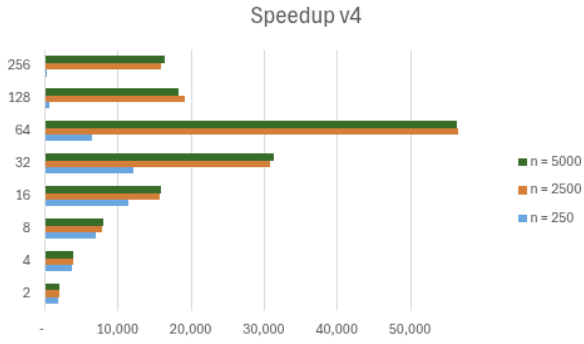

Figura 5: *Speedup en función de los hilos*

Como podemos ver y se mencionó anteriormente, hay una tendencia de que el mejor número de hilos está entre 32 y 64. Sin embargo, el caso de 32 solamente es en un tamaño de `n` pequeño, mientras que 64 es más general. Esto se debe a que con 64 hilos ocupamos los 64 cores que nos permite un nodo del `Finisterrae III`. Si tenemos menos, el trabajo de los hilos será mayor; y si tenemos más de 64, se perderá eficiencia al gestionar los cambios de contexto.

VII. Conclusiones

En este proyecto hemos tratado el problema de implementar el algoritmo de Jacobi y su constante mejoría de eficiencia con las técnicas realizadas. Para ello, nos encargamos de hacer comparaciones entre ellos para observar las optimizaciones a la hora de medir el rendimiento en forma de ciclos. Una vez realizado esto, llegamos a la conclusión de que se debe intentar realizar un código lo más eficiente posible, además de legible o no usar técnicas poco convencionales como SIMD, ya que estas optimizaciones son fácilmente recreables, y de manera más eficiente, por el compilador. Lo que sí es de gran interés, siempre que se pueda, debemos intentar paralelizar a nivel de memoria el código, ya que se aprecia una gran mejoría.

Referencias

[1] Loop Optimizations Where Blocks are Required www.intel.com/content/www/us/en/developer/articles/technical/loop-optimizations-where-blocks-are-required.html [online] ult. visita: 2025/04/12

[2] Intel® Intrinsics Guide https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.htmltechs=AVX$_A LL$ [online] ult. visita: 2025/04/15

[3] OpenMP 3.0 SummarySpec pdf https://www.openmp.org/wp-content/uploads/OpenMP3.0-SummarySpec.pdf [online] ult. visita: 2025/04/16

+ Diapositivas de la asignatura