Práctica Final

Reducción tiempo de cómputo

ESCUELA SUPERIOR DE INFORMÁTICA

7 JUNIO

Master en Ingeniería Informática
Creado por: Sergio Cámara Sevilla
Daniel Ballesteros Almazán

UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

Contenido

Introducción

En este proyecto se aborda la mejora de rendimiendo de la etapa de estimación de momiviento en el algoritmo de codificación de video, ya que se trata de una operación de computación exigente.

Sistema

Para la ejecución del algoritmo y las pruebas de paralelismo se ha utilizado en siguiente sistema:

- SO Windows 10
- 64Bits
- Intel(R) Core (TM) i5-8600K CPU @ 3.60GHz 3.60 GHz
- RAM: 16GB DDR4
- Compilador: gcc 8.1.0 bajo MinGW-W64

Descomposición

Se trata de un problema estático el cual se puede abordar de una forma concisa debido a que el tamaño de la solución ya es conocido de antemano, los cuales son los vectores de movimiento y los datos de entrada. Los vectores de movimiento son calculados en relación con el coste, pero no se accede a la misma posición ya que cada ejecución del bucle accedería para escribir a una posición diferente. En cuanto a los datos necesarios para los cálculos, estos no son alterados tras la ejecución del algoritmo, ya que se generan los vectores de movimiento, por tanto, podemos decir que los datos de entrada pueden estar de forma compartida, debido a que su acceso solo es de lectura.

Tras analizar el problema comentado, se considera la descomposición de tipo de datos de salida, ya que no se dependería de una tarea final que realizase una composición.

Interfaz de programación

Cómo hemos comentado, se trata de un problema en el que los datos de origen son los mismos y no se ven alterados, por tanto, considerando las condiciones, se decide realizar una programación basada en memoria compartida, de esta forma se evita la replicación de información.

Esta decisión implica la utilización de una interfaz de programación de multiproceso basada en memoria compartida y haciendo uso de OpenMP.

Optimización

El problema principal de la ejecución del algoritmo reside en los 4 bucles anidados que calculan los vectores de movimiento.

En dichos bucles se actualizan 3 variables diferentes que son compartidas, las cuales son Vx, Vy y Coste. Por lo tanto, se decide implementar una sentencia *opm* para paralelizar los 4 bucles, colapsándolos y compartiendo esas variables. De esta forma únicamente se influyen en los datos de salida de los vectores de movimiento, ya que los datos de entrada son los mismos para todas las ejecuciones paralelas.

```
#pragma omp parallel for collapse(4) shared (Vx,Vy,costes)
   for (unsigned int y = 0; y < HEIGHT; y += BS)
   {
       for (unsigned int x = 0; x < WIDTH; x += BS)
       {
           /* Calcular MSE para todos los bloques en el área de búsqueda
.
          Las coordenadas en ref y act están alineadas.   */
           for (unsigned char j = 0; j < 2 * SA; j++)
           {
               for (unsigned char k = 0; k < 2 * SA; k++)
               {
                   /*Calcular MSE */
                   float coste_bloque = MSE(&act[y * WIDTH + x], &ref[(y
 + j) * (WIDTH + 2 * SA) + (x + k)]);
                   /* Podría haber una optimización. A igualdad de coste
, elegir aquel cuyo vector de movimiento tenga la menor distancia
                   respecto al origen */
                   if (coste_bloque < costes[y / BS][x / BS])
                   {
                       costes[y / BS][x / BS] = coste_bloque;
                       Vx[y / BS][x / BS] = j - SA;
                       Vy[y / BS][x / BS] = k - SA;
                   }


               }
           }
       }
```

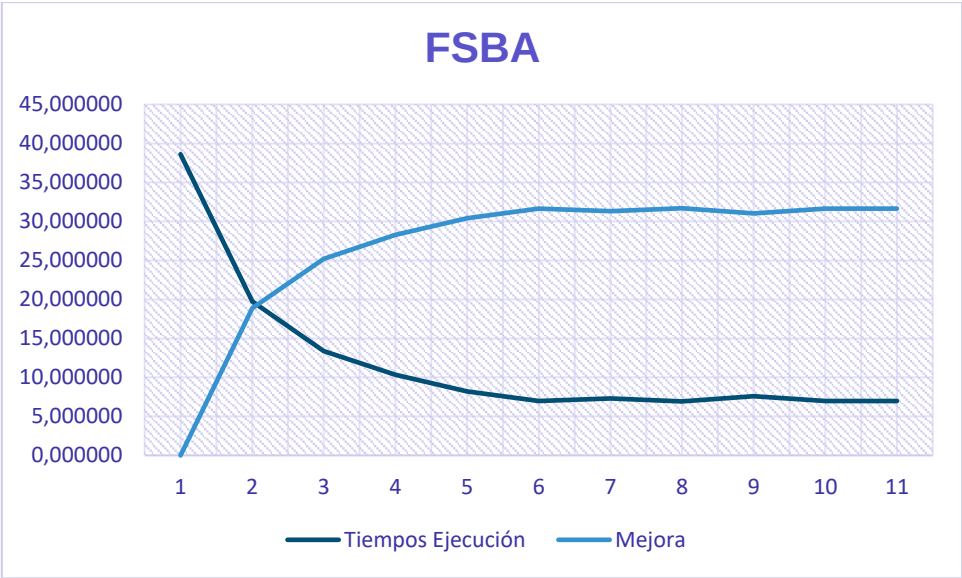
La ejecución secuencial del bucle en el sistema actual nos genera la solución en un tiempo aproximado de 39 segundos. Estos tiempos son mejorados notablemente con la

optimización de paralelismo seleccionada, la cual se ve afectada por el número de hilos disponibles en el sistema en el que nos encontramos, el cual tiene 6 hilos de ejecución.

Hilos	*Tiempos(s)*
1	38,624000
2	19,729000
3	13,400000
4	10,356000
5	8,207000
6	6,963000
7	7,308000
8	6,925000
9	7,603000
10	6,974000
11	6,980000

Cómo podemos observar en la tabla anterior, se ha realizado la ejecución del algoritmo desde 1 a 11 hilos, donde 1 es la ejecución secuencial. A medida que aumentamos el número de hilos, los tiempos disminuyen y se ven totalmente vinculados a las restricciones del sistema. En este caso, cuando se llega a 7 hilos no se consigue una mejora de rendimiento. Por lo tanto, se determina que la ejecución que mejor resultados ofrece en el equipo usado, sin ofrecer retrasos por overheads, es la de 6 hilos. En otros equipos que dispongan de mayor número de hilos de ejecución, este número de hilos óptimo para la ejecución puede variar.

En la siguiente imagen se muestra una gráfica con los tiempos por hilos y la mejora de tiempos a medida que los aumentamos.



Restricciones

Dentro del algoritmo se han detectado dos posibles mejoras de rendimiento para paralelizar que, tras su implementación, no han determinado mejoras, o incluso han repercutido de forma negativa.

En primer lugar, la inicialización de costes tras paralelizar no consigue una mejora significativa, tanto es así que se considera una mejora despreciable.

```
    /* Inicializar los costes mínimos asociados a cada bloque */
    #pragma omp parallel for collapse(2) shared (costes)
    for (unsigned int y = 0; y < HEIGHT / BS; y++)
    {
        for (unsigned int x = 0; x < WIDTH / BS; x++)
        {
            costes[y][x] = __FLT_MAX__;
        }
    }
```

Por otro lado, se intenta paralelizar el método de MSE, colapsando los bucles y utilizando un reduction del error, pero los tiempos se ven perjudicados y consiguiendo resultados errores.

```
float MSE(unsigned char *bloque_actual, unsigned char *bloque_referencia)
{
    float error = 0;
    // #pragma omp parallel for collapse(2) reduction(+:error)
    // es mas lento.
    for (unsigned char y = 0; y < BS; y++)
    {
        for (unsigned char x = 0; x < BS; x++)
        {

            error += pow((bloque_actual[y * WIDTH + x] - bloque_referenci
a[y * (WIDTH + 2 * SA) + x]), 2);
        }

    }
    return error / (BS*BS);
}
```

Acceso GitHub

https://github.com/sergiocamara/CAP-FSBMA