Índice

Reducción del tiempo de cómputo de la etapa de estimación de movimiento en algoritmos de codificación de vídeo

Enlace a GitHUb: https://github.com/jorgeMunozMartinez/AltasPrestacionesFinal#sin-openmp-y-con-bucles-optimizados

Para la realización de esta práctica se ha realizado varias pruebas.

El principal problema encontrado en el código han sido los bucles anidados *for*. Para intentar solventar el costo temporal de la ejecución de cada bucle se han realizado tres aproximaciones

1. **Optimización de Bucles**: En esta aproximación solo se han optimizado los bucles, no se han usado herramientas de paralelización
2. **OpenMP sin Optimización de Bucles**: En esta aproximación se ha usado únicamente la API de **OpenMP** para probar el tiempo de ejecución de los bucles sin optimizar
3. **OpenMP con Optimización de Bucles**: En esta aproximación se ha usado la API de **OpenMP**, junto con los bucles optimizados para probar el tiempo de ejecución de los bucles optimizar

Aunque se ha intentado ejecutar las pruebas con el mismo entorno. La máquina que las ha ejecutado ya estaba ejecutando otros procesos, es posible que los resultados obtenidos se vean alterados por los procesos que esté ejecutando la máquina.

Sin OpenMP y con Bucles Optimizados

En una primera aproximación para mejorar el tiempo de cómputo. Se ha reescrito los bucles anidados.

El tiempo de cómputo de un bucle *for* es **O(n)**, siendo **n** en número de iteraciones realizadas. La complejidad de un bucle anidado es de **O(n^x)** siendo **n** el número de iteraciones y **x** la cantidad de *for* anidados.

El objetivo es reducir la complejidad del algoritmo en todo los posible. En un primer momento se tenían seis bucles anidados, tras finalizar la optimización de los bucles, se tienen únicamente tres bucles anidados.

Búcle Principal sin modificar

```
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 di
         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;
          }
        }
      }
    }
  }
```

Bucle Principal modificado

```
for (i = 0; i < (HEIGHT * WIDTH)/BS; i += BS){
  x = i % WIDTH;
  y = (i / WIDTH) *BS;
  /* Calcular MSE para todos los bloques en el área de búsqueda.
  Las coordenadas en ref y act están alineadas.   */
  for (z = 0; z < 96 * SA; z++){
    /*Calcular MSE */
    j= z/ (2 * SA);
         k= z % (2 * SA);
    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   dist
    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;
    }
  }
}
```

Bucle MSE sin modificar

```
float MSE(unsigned char *bloque_actual, unsigned char *bloque_referencia){
  float error = 0;
  for (unsigned char y = 0; y < BS; y++){
    for (unsigned char x = 0; x < BS; x++){
      error += pow((bloque_actual[y * WIDTH + x] - bloque_referencia[y * (WIDTH + 2 * SA) + x]), 2);
     }
   }
 return error / (BS*BS);
}
```

Bucle MSE modificado

```
float MSE(unsigned char * bloque_actual, unsigned char * bloque_referencia) {
  float error = 0;
  int i = 0;
  unsigned char x, y, z;
  for (y = 0; y < BS ; y++) {
    error += pow((bloque_actual[y * WIDTH + 0] - bloque_referencia[y * (WIDTH + 2 * SA) + 0]), 2);
    error += pow((bloque_actual[y * WIDTH + 1] - bloque_referencia[y * (WIDTH + 2 * SA) + 1]), 2);
    error += pow((bloque_actual[y * WIDTH + 2] - bloque_referencia[y * (WIDTH + 2 * SA) + 2]), 2);
    error += pow((bloque_actual[y * WIDTH + 3] - bloque_referencia[y * (WIDTH + 2 * SA) + 3]), 2);
    error += pow((bloque_actual[y * WIDTH + 4] - bloque_referencia[y * (WIDTH + 2 * SA) + 4]), 2);
    error += pow((bloque_actual[y * WIDTH + 5] - bloque_referencia[y * (WIDTH + 2 * SA) + 5]), 2);
    error += pow((bloque_actual[y * WIDTH + 6] - bloque_referencia[y * (WIDTH + 2 * SA) + 6]), 2);
    error += pow((bloque_actual[y * WIDTH + 7] - bloque_referencia[y * (WIDTH + 2 * SA) + 7]), 2);
    error += pow((bloque_actual[y * WIDTH + 8] - bloque_referencia[y * (WIDTH + 2 * SA) + 8]), 2);
    error += pow((bloque_actual[y * WIDTH + 9] - bloque_referencia[y * (WIDTH + 2 * SA) + 9]), 2);
    error += pow((bloque_actual[y * WIDTH + 10] - bloque_referencia[y * (WIDTH + 2 * SA) + 10]), 2);
    error += pow((bloque_actual[y * WIDTH + 11] - bloque_referencia[y * (WIDTH + 2 * SA) + 11]), 2);
    error += pow((bloque_actual[y * WIDTH + 12] - bloque_referencia[y * (WIDTH + 2 * SA) + 12]), 2);
    error += pow((bloque_actual[y * WIDTH + 13] - bloque_referencia[y * (WIDTH + 2 * SA) + 13]), 2);
    error += pow((bloque_actual[y * WIDTH + 14] - bloque_referencia[y * (WIDTH + 2 * SA) + 14]), 2);
    error += pow((bloque_actual[y * WIDTH + 15] - bloque_referencia[y * (WIDTH + 2 * SA) + 15]), 2);
  }
```

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

Desenrrollado de bucles

Se ha decidido aplicar la técnica de Desenrollado de bucles *loop unrolling*, usado para mejorar la velocidad de ejecución del programa. El único problema de esta técnica es que empeora la legibilidad del código, aunque en nuestro caso nuestro objetivo principal es la mejora del tiempo de ejecución, para entregar un código más "limpio" hemos decidido aplicarlo solo en el bucle más pequeño a modo de ejemplo, pero de igual forma esta técnica se podría aplicar a los demás bucles.

Conclusiones

Tiempo de ejecución sin optimizar los bucles

```
altasPrestaciones@altasPrestaciones:~/AltasPrestaciones/Final$ ./fsbma

> Tiempo Total: 80.092631
```

Tiempo de ejecución con los bucles optimizados

```
altasPrestaciones@altasPrestaciones:~/AltasPrestaciones/Final$ ./fsbma

> Tiempo Total:79.245109
```

Como se puede apreciar. La diferencia de los tiempos de ejecución, solo modificando los bucles no es significativa. **No hay casi diferencia en tiempo de ejecución entre los bucles optimizado y los sin optimizar**.

No se aprecia la optimización de los bucles anidados ya que creemos que, al ser un código pequeño, la complejidad del algoritmo no influye mucho en el tiempo final.

Con OpenMP y sin Bucles Optimizados

En esta segunda aproximación se quiere probar el tiempo de ejecución del algoritmo haciendo uso únicamente de la API de **OpenMP**.

Parámetros optimizados

Bucle principal

```
#pragma omp parallel for private(x,y,j,k,z, coste_bloque) schedule(dynamic) num_threads(hilos)
```

- **num_threads(hilos)**: Asigna el número de hilos introducidos por la entrada de texto para la ejecución del a sección paralela
- **Pragma omp parallel for**: Indica que el bucle se ejecutará con los hilos indicados anteriormente
- **Scheduler**: Aquí definimos la forma de planificar las iteraciones de los bloques, en este caso no se han observado diferencias significativas, por lo que se ha decidido mantener el modo dinámico
- **Variables privadas**:
  - **X**: Corresponde con el valor de la anchura de la imagen, su rango es de 0-1280
  - **Y**: Corresponde con el valor de la altura de la imagen, su rango es de 0-720
  - **J**: Usada en el bucle interior, cuyo rango es 96 *16
  - **K,Z**: Usadas para calcular el ME, su rango va de 0-32
  - **Coste_bloque**: Variable que almacena el resultado de la función MSE

Bucle de la función MSE

```
#pragma omp reducer(+:error) parallel  for  private(x,y)  schedule(dynamic) num_threads(hilos)
```

- **num_threads(hilos)**: Asigna el número de hilos introducidos por la entrada de texto para la ejecución del a sección paralela
- **Pragma omp parallel for**: Indica que el bucle se ejecutará con los hilos indicados anteriormente
- **Scheduler**: Aquí definimos la forma de planificar las iteraciones de los bloques, en este caso no se han observado diferencias significativas, por lo que se ha decidido mantener el modo dinámico
- **Reducer (+:error)**: Cláusula específica de openmp para indicar la reducción de un bucle.
- **Variables privadas**:
  - **X,Y**: Usado para calcular el error, sus rangos son de 0-16

Resultados obtenidos

Para comprobar la optimización de código haciendo uso de OpenMP, se van a realizar varias ejecuciones modificando el número de los hilos

Los resultados obtenidos en la columna de **Tiempo Optimizado**, es la media aritmética del resultado obtenido tras 10 ejecuciones

Scheduler Dinámico

	Tiempo sin optimizar	Tiempo optimizado
2 Hilos	80.092631	41.286586
4 Hilos	80.092631	20.998403
8 Hilos	80.092631	13.037493
16 Hilos	80.092631	12.401389
32 Hilos	80.092631	12.913650
64 Hilos	80.092631	12.206220
128 Hilos	80.092631	12.148841
256 Hilos	80.092631	12.270545
512 Hilos	80.092631	12.182388

Scheduler Estático

	Tiempo sin optimizar	Tiempo optimizado
2 Hilos	80.092631	41.812975
4 Hilos	80.092631	21.543593
8 Hilos	80.092631	12.527022
16 Hilos	80.092631	12.766653
32 Hilos	80.092631	12.678141
64 Hilos	80.092631	12.601649

	Tiempo sin optimizar	Tiempo optimizado
128 Hilos	80.092631	12.143620
256 Hilos	80.092631	12.530999
512 Hilos	80.092631	12.622883

Conclusiones

- **Scheduler Estático VS Scheduler Dinámico**: Comparando en tiempo de ejecución de la paralelización con los dos Scheduler
- **Scheduler Estático**: Mejora con un número elevado de hilos, pero empeora con un número bajo de hilos
- **Scheduler Dinámico**: Mejora con un número bajo ed hilos, pero empeora con un número alto de hilos

A partir de 8 hilos es muy costoso reducir el tiempo de ejecución. El coste de realizar la paralelización con un número mayor de 8 hilos creemos que o es rentable.

No se aprecia la optimización de los bucles anidados ya que creemos que, al ser un código pequeño, la complejidad del algoritmo no influye mucho en l tiempo final

Con OpenMP y Bucles Optimizados

En esta tercera aproximación se quiere probar el tiempo de ejecución del algoritmo haciendo uso de la API de **OpenMP y la Optimización de Bucles** realizada.

Parámetros optimizados

Bucle principal

```
#pragma omp parallel for private(x,y,j,k,z, coste_bloque) schedule(dynamic) num_threads(hilos)
```

- **num_threads(hilos)**: Asigna el número de hilos introducidos por la entrada de texto para la ejecución del a sección paralela
- **Pragma omp parallel for**: Indica que el bucle se ejecutará con los hilos indicados anteriormente
- **Scheduler**: Aquí definimos la forma de planificar las iteraciones de los bloques, en este caso no se han observado diferencias significativas, por lo que se ha decidido mantener el modo dinámico
- **Variables privadas**:
  - **X**: Corresponde con el valor de la anchura de la imagen, su rango es de 0-1280
  - **Y**: Corresponde con el valor de la altura de la imagen, su rango es de 0-720
  - **J**: Usada en el bucle interior, cuyo rango es 96 *16
  - **K,Z**: Usadas para calcular el ME, su rango va de 0-32
  - **Coste_bloque**: Variable que almacena el resultado de la función MSE

Bucle de la función MSE

```
#pragma omp reducer(+:error) parallel  for  private(x,y)  schedule(dynamic) num_threads(hilos)
```

- **num_threads(hilos)**: Asigna el número de hilos introducidos por la entrada de texto para la ejecución del a sección paralela
- **Pragma omp parallel for**: Indica que el bucle se ejecutará con los hilos indicados anteriormente
- **Scheduler**: Aquí definimos la forma de planificar las iteraciones de los bloques, en este caso no se han observado diferencias significativas, por lo que se ha decidido mantener el modo dinámico
- **Reducer (+:error)**: Cláusula específica de openmp para indicar la reducción de un bucle.

- **Variables privadas**:
  - **X,Y**: Usado para calcular el error, sus rangos son de 0-16

Resultados obtenidos

Para comprobar la optimización de código haciendo uso de OpenMP, se van a realizar varias ejecuciones modificando el número de los hilos

Los resultados obtenidos en la columna de **Tiempo Optimizado**, es la media aritmética del resultado obtenido tras 10 ejecuciones

Scheduler Dinámico

	Tiempo sin optimizar	Tiempo optimizado
2 Hilos	80.092631	40.457573
4 Hilos	80.092631	20.006617
8 Hilos	80.092631	12.962127
16 Hilos	80.092631	12.998002
32 Hilos	80.092631	12.946937
64 Hilos	80.092631	12.465711
128 Hilos	80.092631	12.279916
256 Hilos	80.092631	12.041000
512 Hilos	80.092631	11.860091

Scheduler Estático

	Tiempo sin optimizar	Tiempo optimizado
2 Hilos	80.092631	42.355106
4 Hilos	80.092631	20.713961
8 Hilos	80.092631	13.325617
16 Hilos	80.092631	12.817600
32 Hilos	80.092631	12.532386
64 Hilos	80.092631	12.728742
128 Hilos	80.092631	12.201151
256 Hilos	80.092631	11.997898
512 Hilos	80.092631	11.897121

Conclusiones

- **Scheduler Estático VS Scheduler Dinámico**: Comparando en tiempo de ejecución de la paralelización con los dos Scheduler

- **Scheduler Estático**: Mejora con un número elevado de hilos, pero empeora con un número bajo de hilos
- **Scheduler Dinámico**: Mejora con un número bajo ed hilos, pero empeora con un número alto de hilos

A partir de 8 hilos es muy costoso reducir el tiempo de ejecución. El coste de realizar la paralelización con un número mayor de 8 hilos creemos que no es rentable.

No se aprecia la optimización de los bucles anidados ya que creemos que, al ser un código pequeño, la complejidad del algoritmo no influye mucho en el tiempo final

Comandos Usados

Para realizar la ejecución de la práctica se necesitan ejecutar tres comandos

1. Compilación del código

```
gcc fsbma.c -o fsbma -lm -fopenmp
```

2. Ejecución del código

```
./fsbma  <número de Hilos>
```

3. La comparativa del resultado obtenido con el resultado inicial sin optimizar

```
./fsbma <número de Hilos> | diff resultado.txt -
```

Conclusiones Finales

Tras la realización de este trabajo podemos concluir que la paralelización por sí sola es muy beneficiosa en cuestión de tiempo. Pero para realizarla de forma adecuada es necesario realizar un análisis del código para buscar dependencias de variables entre hilos y condiciones de carrera.

También hemos podido comprobar que, aunque openmp realiza una paralelización de bucles muy eficiente. Hemos conseguido mejorar en los tiempos de ejecución al realizar distintas modificaciones en los bucles. Dicho esto, podemos afirmar que con una mayor reducción manual de los bucles se podría mejorar aún más los resultados de esta práctica, dada la limitación de alcance de esta práctica no se ha podido realizar una mayor reducción.

Problemas encontrados

En la realización de la práctica se han encontrado ciertos problemas:

1. La medición con la función **clock_gettime()**, no ofrecía un tiempo correcto al ejecutar el programa con varios hilos, el problema era debido a el parámetro *CLOCK_PROCESS_CPUTIME_ID*, si se sustituye por *CLOCK_MONOTONIC* el tiempo es correcto, para evitar problemas se ha sustituido por **omp_get_wtime()** ya que es una medición específica del tiempo para su uso en programas paralelos.

```
clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&inicio);
```

2. Error con **collapse** en Scheduler: Se ha intentado aplicar **collapse** en el bucle principal, pero al aplicarlo se alteraba el resultado de la salida, por lo que se ha descartado