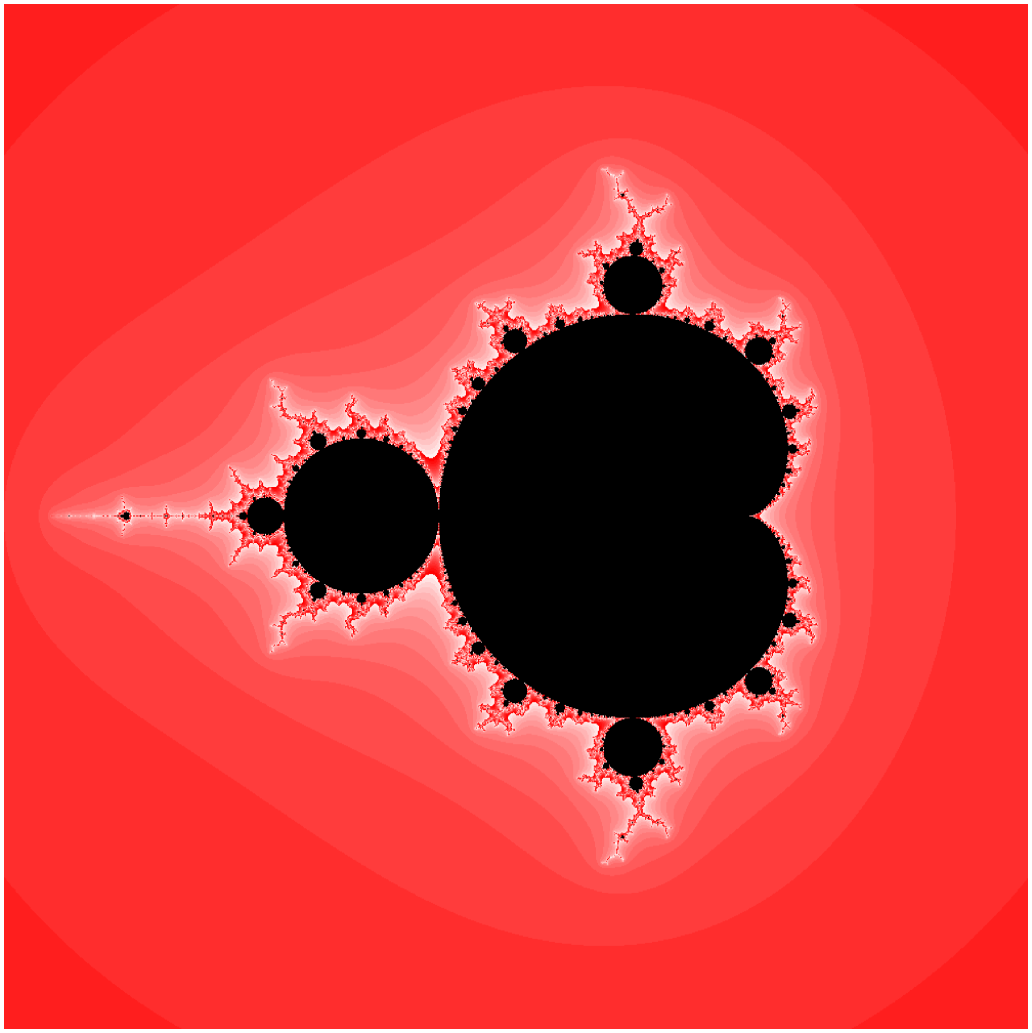
David Morilla Sorlí

Conjunto de Mandelbrot (basado en Julia)

El programa para generar una imagen fractal del conjunto de Mandelbrot basado en Julia opera mediante la iteración de ecuaciones matemáticas complejas. El conjunto de Julia se define mediante la fórmula $z_{n+1} = z_n^2 + c$, donde z_n es una variable compleja y c es una constante compleja asociada a un punto específico en el plano complejo. En este caso, c se toma como coordenada en el conjunto de Mandelbrot. La imagen se genera iterando esta fórmula para cada punto en una cuadrícula en el plano complejo.

El algoritmo itera sobre cada punto, verificando si la secuencia generada por la fórmula permanece acotada. Si la secuencia diverge hacia el infinito, el punto se asigna a un color según la velocidad de divergencia. Si la secuencia permanece acotada, el punto se considera parte del conjunto y se asigna un color distinto. La combinación de estos colores en la cuadrícula produce la imagen fractal que podemos ver a continuación.



Profiling

El objetivo de esta parte de la práctica es la utilización de herramientas de *profiling* sobre un programa que genera una imagen del conjunto fractal de Mandelbrot para su posterior optimización. Se usarán perf, valgrind y VTune.

1. **Fallos de caché**

   Mediante los comandos `perf stat -d ./Mandel_J 1000 5000` y `perf report` obtenemos un informe donde se detalla que las llamadas al sistema `clear_page_erms` y `copy_user_enhanced_fast_string` son las que más fallos de caché generan con casi un 30% cada una.

   ```
   Self     Command   Shared Object          Symbol
   29.34%   Mandel_J  [kernel.kallsyms]      [k] clear_page_erms
   29.02%   Mandel_J  [kernel.kallsyms]      [k] copy_user_enhanc...
   ```

   Sin embargo, a través de valgrind y su herramienta 'cachegrind' y usando el mandato `valgrind --tool=cachegrind ./Mandel_J 1000 5000`, podemos ver como sobre el total de accesos a caché, el número de fallos es aproximadamente el 0.0% tanto para instrucciones como para datos:

   ```
   Instruction Cache (I):                Data Cache (D):
   I   refs:      78,433,887,086         D refs:     32,960,975,341
   I1  misses:             1,477         D1 misses:          50,871
   LLi misses:             1,411         LLd misses:   49,831
   I1  miss rate:          0.00%         D1 miss rate:         0.0%
   LLi miss rate:          0.00%         LLd miss rate:        0.0%
   ```

2. **Fugas de memoria**

   A través de la herramienta 'memcheck' de valgrind observamos si se producen fugas de memoria. Usando el comando `valgrind --tool=memcheck --leak-check=full --show-reachable=yes --leak-resolution=high ./Mandel_J 1000 5000`, obtenemos los siguientes resultados que muestran que no existen fugas de memoria:

   ```
   HEAP SUMMARY:
   in use at exit: 0 bytes in 0 blocks
   total heap usage: 4 allocs, 4 frees, 3,005,592 bytes allocated
   All heap blocks were freed -- no leaks are possible
   ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
   ```

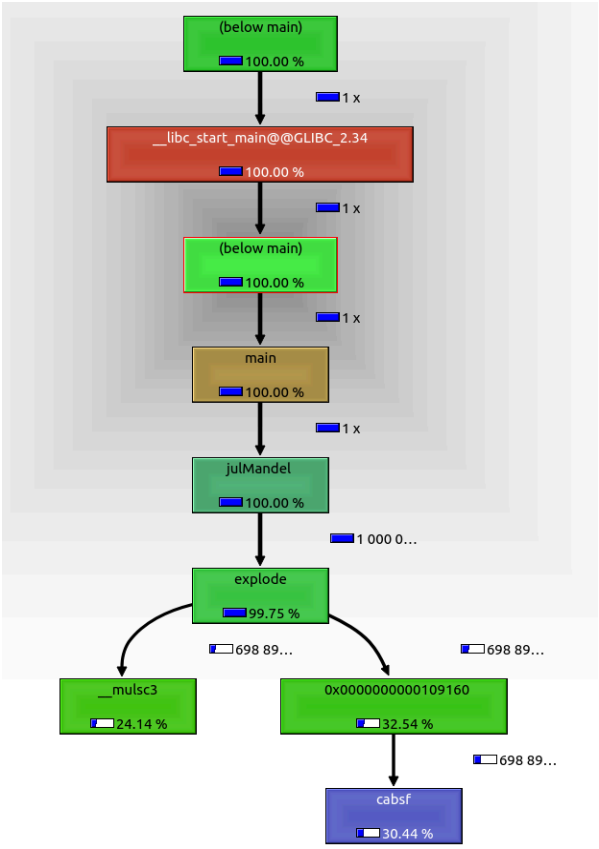
3. Hotspots

Mediante el uso de los comandos `perf record -g ./Mandel_J 1000 5000` y `perf report` obtenemos un informe acerca del tiempo empleado en cada función de nuestro código. Se muestran aquellas funciones con más de un 5% de ciclos totales:

Self	Command	Shared Object	Symbol
39.66%	Mandel_J	Mandel_J	[.] explode
32.10%	Mandel_J	libm-2.31.so	[.] __hypotf_f...
19.21%	Mandel_J	Mandel_J	[.] __mulsc3
5.04%	Mandel_J	libm-2.31.so	[.] __hypotf

De estas funciones, hay dos que son funciones de librería y una que también lo es (`__mulsc3`) pero que se monta dinámicamente, por lo que aparece como parte de nuestro programa. Por ahora parece que es la función `explode` en donde podría ser interesante explorar su optimización.

Usaremos ahora la herramienta 'callgrind' de valgrind para ver los posibles cuellos de botella. Con los comandos `valgrind --tool=callgrind--dump-instr=yes ./Mandel_J 1000 5000` y `kcachegrind callgrind.out.[pid_nr]` obtenemos un grafo de tiempos:



Podemos observar como la función `explode` (sin contar las llamadas a `__mulsc3` y `0x0`…) usa aproximadamente el 43.07% de los ciclos totales del programa.
Por último usaremos VTune para ver en detalle los hotspots del programa:

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
__libm_hypot_e7	mandel	14.050s	57.1%
explode	mandel	6.911s	28.1%
cabsf	mandel	3.360s	13.7%
__libm_hypot	mandel	0.248s	1.0%
mapPoint	mandel	0.012s	0.0%
[Others]	N/A*	0.029s	0.1%

Vemos como el mayor tiempo de CPU lo consume una función de librería y que `explode` consume un 28.1% del tiempo de trabajo de la CPU. Si observamos el call-stack activando la opción de incluir bucles vemos lo siguiente:

Function / Call Stack	CPU Time	Module	Function (Full)	Source File	Start Address
cabsf	17.658s	mandel	cabsf		0x401b90
[Loop at line 37 in explode]	6.911s	mandel	[Loop at line 37 in explode]	Mandel_J.c	0x401411
mapPoint	0.012s	mandel	mapPoint	Mandel_J.c	0x4015b0
fcolor	0.012s	mandel	fcolor	Mandel_J.c	0x401260
tga_write	0.009s	mandel	tga_write	Mandel_J.c	0x401a70
[Loop at line 78 in julMandel]	0.008s	mandel	[Loop at line 78 in julMan…	Mandel_J.c	0x4016f3

Del total de tiempo que consume `explode`, en el 100% es responsable el bucle de la línea 37. Si nos adentramos un poco más en los recursos que nos ofrece VTune, podemos ver exactamente el código que ocupa más a la CPU dentro de dicho bucle.

Source Line	Source	CPU Time: Total	CPU Time: Self
37	`while ((k<=n) && (modul<=4)){`	8.8%	2.156s
38	`z0 = z1;`	0.4%	0.096s
39	`if (modul<=4){`	0.3%	0.068s
40	`z1 = (z0*z0)+c;`	2.1%	0.528s
41	`modul = cabsf(z1);`	15.5%	3.813s
42	`k++;`	1.0%	0.250s
43	`valor = k;`		
44	`}`		
45	`else`		
46	`valor = k;`		
47	`}`		

Evidentemente, como ya sabíamos, la llamada a la función de librería `cabsf` es responsable de gran parte del tiempo de CPU; sin embargo, podemos ver como el bucle while es candidato a ser optimizado.

Optimización del código secuencial (función `explode`)

Primero debemos obtener los tiempos promedios de ejecución del programa tras cinco ejecuciones (dos en triqui1 y una en triqui2, triqui3 y triqui4).

Programa	**Opciones**	**Optimizaciones**	**t-min**	**t-medio**	**t-max**
Mandel_J	size: 1000X1000 iter: 5000	Ninguna	24.237 s	24.457 s	24.649 s

Tras analizar la función `explode` podemos ver como hay alguna ineficiencia dentro del bucle while. Debido a que la condición del while ya impide que modul sea mayor que cuatro, y nada modifica a modul antes de entrar en el if, esta sentencia condicional puede ser eliminada, quedando el código de la siguiente manera:

```
int explode(float _Complex z0, float _Complex c, int n) {
	int k = 1;
	int valor = 1;
	float _Complex z1 = (z0 * z0) + c;
	float modul = cabsf(z1);
	while (k <= n && modul <= 4) {
		z0 = z1;
		z1 = (z0 * z0) + c;
		modul = cabsf(z1);
		k++;
		valor = k;
	}
	return valor;
}
```

Tras este cambio en la función obtenemos los siguientes tiempos de ejecución:

Programa	**Opciones**	**Optimizaciones**	**t-min**	**t-medio**	**t-max**
Mandel_J	size: 1000X1000 iter: 5000	Eliminación de redundancias	24.336 s	24.492 s	24.718 s

Como vemos, se obtienen tiempos muy similares a la ejecución anterior; de hecho, el speedup medio es ligeramente inferior a 1 (0.998). Esto nos puede indicar que realmente la optimización anterior no ha sido tal y debemos buscar otros lugares del código para optimizar.

Optimización del código secuencial (función `julMandel`)

Dado que la única función que llama a `explode` es `julMandel`, y visto que `explode` está lo más optimizado posible, nos planteamos la casuística de que sea en `julMandel` dónde debamos hacer los trabajos de optimización.

Cambiaremos el malloc de rgb por un calloc, inicializando a cero (negro) todos los valores de rgb. De esta manera simplificamos la sentencia condicional de dentro de los bucles. Además, vamos a juntar ambos bucles en uno solo porque esto nos ayudará posteriormente para la paralelización y vectorización. Debido a que vamos a querer que cada iteración sea independiente la una de la otra, vamos a reescribir la manera de asignar el valor a k, y lo haremos multiplicando el índice lineal por tres al comienzo de cada iteración. El código queda de la siguiente manera:

```
unsigned char *julMandel(int width,int height,int n){
	float x_min = -2.15;
	float x_max = 1.15;
	float y_min = -1.65;
	float y_max = 1.65;
	int x,y,i,k,linearIndex,iterations;
	float _Complex z0,z1;
	float _Complex c;
	color asp;
	unsigned char *rgb;
	rgb = calloc ((width*height*3) ,sizeof(unsigned char));
	iterations=width*height;
	for(linearIndex=0; linearIndex<iterations;linearIndex++){
		k = linearIndex*3;
		x=linearIndex%width;
		y=linearIndex/width;
		c = mapPoint(width,height,x,y,x_min,x_max,y_min,y_max);
		z0=0+0*I;
		i = explode(z0,c,n);
		if (i<n) {
			asp = fcolor(i,n);
			rgb[k]   = asp.bl;
			rgb[k+1] = asp.gr;
			rgb[k+2] = asp.re;
		}
    }
return rgb;
}
```

Programa	Opciones	Optimizaciones	t-min	t-medio	t-max
Mandel_J	size: 1000X1000 iter: 5000	Anterior + Unificación en un solo bucle en julMandel.	24.191 s	24.379 s	24.753 s

Speedup: 1.003
Como vemos, el speedup es prácticamente nulo, sin embargo, los cambios que hemos realizado nos ayudarán a realizar la paralelización y vectorización posterior.

Vectorización

A través de la directiva `#pragma omp simd` podemos vectorizar el bucle for de la función `julMandel`. El código del bucle for de la función `julMandel` queda así:

```
    #pragma omp simd
    for(linearIndex=0; linearIndex<iterations;linearIndex++){
        k = linearIndex*3;
        x=linearIndex%width;
        y=linearIndex/width;
        c = mapPoint(width,height,x,y,x_min,x_max,y_min,y_max);
        z0=0+0*I;
        i = explode(z0,c,n);
        if (i<n) {
            asp = fcolor(i,n);
            rgb[k]   = asp.bl;
            rgb[k+1] = asp.gr;
            rgb[k+2] = asp.re;
        }
}
```

Programa	Opciones	Optimizaciones	t-min	t-medio	t-max
Mandel_J	size: 1000X1000 iter: 5000	Anterior + Vectorización	13.165 s	13.224 s	13.267 s

Speedup: 1.85

Paralelización

A continuación, utilizaremos OpenMP para paralelizar el bucle de la función `julMandel` y enviar cada iteración a los diferentes cores creando múltiples threads para aumentar el speedup del programa. Debemos usar la directiva `#pragma parallel for` para paralelizar. Es importante que se hagan copias privadas de las variables k, linearIndex, x, y, z0, c, i y asp, puesto que estas variables son específicas de cada iteración y no deben verse modificadas por el resto de hilos en ejecución.

```
#pragma omp parallel for simd schedule(*) private(k,linearIndex, x, y,
z0, c, i, asp)
for(linearIndex=0; linearIndex<iterations;linearIndex++){
      k = linearIndex*3;
      x=linearIndex%width;
      y=linearIndex/width;
      c = mapPoint(width,height,x,y,x_min,x_max,y_min,y_max);
      z0=0+0*I;
      i = explode(z0,c,n);
      if (i<n) {
            asp = fcolor(i,n);
            rgb[k]   = asp.bl;
            rgb[k+1] = asp.gr;
            rgb[k+2] = asp.re;
      }
}
```

*Ahora vamos a ver como cambia el speedup según el tipo de schedule que utilicemos.

Programa	**Opciones**	**Optimizaciones**	**t-min**	**t-medio**	**t-max**
Mandel_J	size: 1000X1000 iter: 5000	Anterior + Paralelización (static)	1.428 s	1.455 s	1.491 s

Speedup: 16.81

Programa	**Opciones**	**Optimizaciones**	**t-min**	**t-medio**	**t-max**
Mandel_J	size: 1000X1000 iter: 5000	Anterior + Paralelización (dynamic)	0.554 s	0.560 s	0.573 s

Speedup: 43,67

Programa	Opciones	Optimizaciones	t-min	t-medio	t-max
Mandel_J	size: 1000X1000 iter: 5000	Anterior + Paralelización (guided)	0.579 s	0.621 s	0.690 s

Speedup: 39,38

La paralelización con "dynamic" muestra un mejor rendimiento en términos de speedup en comparación con "static" y "guided". Especialmente en el caso de "dynamic", el speedup alcanza un valor impresionante de 43.67. Algunas de las razones por las cuales `schedule(dynamic)` puede funcionar bien en este caso son que, al haber condiciones, no todos los hilos realizan la misma carga de tareas, por lo que la asignación dinámica puede ayudar a distribuir mejor el trabajo entre los hilos. Algunos hilos pueden manejar más iteraciones que otros, lo que evita la inactividad de algunos hilos mientras otros aún tienen trabajo. Además, en comparación con `schedule(static)`, que divide las iteraciones en bloques fijos antes de la ejecución, `schedule(dynamic)` reduce la sobrecarga de planificación al asignar iteraciones en tiempo de ejecución en lotes más pequeños.

MPI (Paso de mensajes)

Para implementar paso de mensajes al programa, primero debemos dividir la tarea entre los diferentes procesos. Dado que se está generando una imagen del conjunto de Mandelbrot de tamaño `width * height`, podríamos dividir la imagen en tiras de filas y asignar a cada proceso la tarea de calcular las iteraciones para esas filas específicas. Los cambios realizados en el código son los siguientes:

1. **`main`**
   Declaramos las variables nproc y myrank y global_rgb e inicializamos MPI. Este código lo ejecutan todos los procesos:

```
int i,j;
int width, height, nproc, myrank;
complex c;
unsigned char *rgb, *global_rgb = NULL;
struct timeval tv_start, tv_end;
float tiempo_trans;
MPI_Init (&argc,&argv);
MPI_Comm_size (MPI_COMM_WORLD , &nproc);
MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
```

A continuación, debemos gestionar que tareas realizará el maestro y cuáles los esclavos. En este caso, hemos decidido que todas las impresiones de mensajes y toma de tiempos las realice el maestro:

```
if(argC != 3) {
	if (myrank == 0) {
		printf("Uso : %s\n", "<dim de la ventana, n_iteraciones>");
	}
	MPI_Finalize();
	exit(1);
}

//resto de código intermedio ...

if (width > DIM) {
	if (myrank == 0) {
		printf("El tamaño de la ventana debe ser menor que
		1024\n");
	}
	MPI_Finalize();
	exit(1);
}

//resto de código intermedio ...

if (myrank == 0) {
	 printf("Mandelbrot: %d, %d, %d\n", width, height, atoi(argV[2]));
	 gettimeofday(&tv_start, NULL);
}
```

Debemos determinar el número de filas de la imagen de las que se van a ocupar cada uno de los procesos pesados. Esto lo hacemos definiendo las siguientes variables y teniendo en cuenta que en caso de que el número de procesos no sea divisor del número de filas, el último proceso será el encargado de las filas finales restantes. Este código lo ejecutan todos los procesos:

```
int rows_per_process = height / nproc;
int start_row = myrank * rows_per_process;
int end_row = (myrank + 1) * rows_per_process;
if (myrank == nproc - 1) {
	 end_row = height;
}
```

Lo siguiente que debemos hacer es llamar a la función `julMandel`, a la que, por razones que se explicarán a continuación, hemos añadido dos parámetros de entrada: start_row y end_row. Tras la llamada a la función, tenemos que recoger los resultados locales de cada proceso y agruparlos en la variable global_rgb declarada anteriormente.

```
rgb=julMandel(width, height, atoi(argV[2]), start_row, end_row);
if (myrank == 0) {
      global_rgb = (unsigned char *)malloc(width * height * 3
      * sizeof(unsigned char));
}
MPI_Gather(rgb, 3 * width * (end_row - start_row),
MPI_UNSIGNED_CHAR, global_rgb,
3 * width * (end_row - start_row), MPI_UNSIGNED_CHAR, 0,
MPI_COMM_WORLD);
```

Por último, dejamos que el maestro calcule el tiempo transcurrido, llame a tga_write para que se realice la impresión de la imagen y libere la variable dinámica global_rgb. Finalmente, todos los procesos deben liberar la variable dinámica rgb y hacer la llamada a `MPI_Finalize`:

```
if (myrank==0){
      gettimeofday(&tv_end, NULL);
      tiempo_trans=(tv_end.tv_sec - tv_start.tv_sec) * 1000000 +
      (tv_end.tv_usec - tv_start.tv_usec); //en us
      printf("Tiempo Mandel_J = %f segundos\n",
      tiempo_trans/1000000);
}

// resto de código intermedio ...

if (myrank == 0) {
    tga_write ( width, height, global_rgb, "mandelbrot.tga" );
    free(global_rgb);
}
free(rgb);
MPI_Finalize();
```

2. **julMandel**

   Las modificaciones que deben hacerse en `julMandel` son bastante sencilla. Primero, debemos añadir dos nuevos parámetros enteros de entrada a la función: start_row y end_row. Esto se debe a que el índice del pixel a calcular debe ahora ser diferente para cada proceso llamante de la función, ya que van a estar repartiéndose la tarea. A

continuación, es preciso recalcular el número de iteraciones del bucle for, puesto que solo debe iterar sobre las filas asignadas al proceso llamante. De la misma manera, el espacio en memoria que debemos reservar para la variable local rgb es menor, así que debemos actualizarlo, quedando el código de esta función de la siguiente manera:

```
unsigned char *julMandel(int width, int height, int n, int
start_row, int end_row){
      float x_min = -2.15;
      float x_max = 1.15;
      float y_min = -1.65;
      float y_max = 1.65;
      int x,y,i,linearIndex, iterations;
      float _Complex z0,z1;
      float _Complex c;
      color asp;
      int k;
      unsigned char *rgb;

      rgb = calloc ((width * (end_row - start_row) * 3),
      sizeof(unsigned char));
      iterations = width* (end_row - start_row);

      #pragma omp parallel for simd schedule(dynamic)
      private(k,linearIndex, x, y, z0, c, i, asp)
      for(linearIndex=0; linearIndex<iterations;linearIndex++){
            k = linearIndex*3;
            x=linearIndex%width;
            y=linearIndex/width + start_row; // offset
            c = mapPoint(width, height, x, y, x_min, x_max, y_min,
            y_max);
            z0=0+0*I;
            i = explode(z0,c,n);
            if (i<n) {
                  asp = fcolor(i,n);
                  rgb[k]   = asp.bl;
                  rgb[k+1] = asp.gr;
                  rgb[k+2] = asp.re;
            }
      }
      return rgb;
}
```

Tras estas modificaciones, obtenemos los siguientes tiempos utilizando dos procesos pesados, que es el número de procesos que he encontrado que mejores resultados nos arroja.

Programa	**Opciones**	**Optimizaciones**	**t-min**	**t-medio**	**t-max**
Mandel_J	size: 1000X1000 iter: 5000	Anterior + MPI (2 procesos)	0,723 s	0,727 s	0,735 s

Speedup: 33,64

CUDA

Para la versión de CUDA utilizaremos como base la versión pre-[vectorización, paralelización y MPI]. Lo primero que debemos hacer es incluir la librería `cuComplex.h`. Esta librería es la que debe usarse para tratar con números complejos en los devices. Además, debemos definir el tamaño de bloque que vamos a usar, que deberá ser múltiplo del tamaño del wrap (en este caso, 32). Tras haber hecho múltiples pruebas, llego a la conclusión de que no hay gran diferencia entre números de thread por bloque comprendidos entre 64 y 512. Sin embargo, he podido observar como de media se obtienen ligeramente mejores resultados con 128, por lo que es el tamaño que utilizaremos.

```
#include <cuComplex.h>
#define THREADS_PER_BLOCK 128
```

Posteriormente, vamos a separar la función `julMandel` en dos, `julMandel` y `julMandelKernel`. Esto lo hacemos para preparar en `julMandel` los datos que a continuación van a ser llevados al device para su cómputo.

1. **`julMandel`**

```
unsigned char *julMandel(int width, int height, int n){
      unsigned char *rgb;
      rgb = (unsigned char *)calloc((width * height * 3),
      sizeof(unsigned char));
      int size = width * height * 3 * sizeof(unsigned char);
      unsigned char *d_rgb;
      cudaMalloc((void **)&d_rgb, size);
      int numBlocks = (width * height + THREADS_PER_BLOCK - 1) /
      THREADS_PER_BLOCK;
      julMandelKernel<<<numBlocks, THREADS_PER_BLOCK>>>(width,
      height, n, d_rgb);
      cudaMemcpy(rgb, d_rgb, size, cudaMemcpyDeviceToHost);
      cudaFree(d_rgb);
      return rgb; }
```

Tenemos que crear un puntero a char para el device y lo hacemos llamando a `cudaMalloc`. Calculamos el número de bloques que van a ser necesarios. Esto lo hacemos obteniendo el número total de píxeles, añadiendo el tamaño del bloque, restando uno y dividiendo todo por el tamaño del bloque. Esto nos asegura que en caso de que el número de píxeles no sea múltiplo del tamaño del bloque, vamos a tener un bloque extra para encargarse de los píxeles últimos. Tras esto, hacemos la llamada a la función kernel y copiamos los datos de d_rgb a rgb. Por último, liberamos la memoria de d_rgb.

2. **`julMandelKernel`**
   Esta función es __global__ porque es llamada desde el host, pero se ejecuta en el device. Es la encargada de calcular los resultados de cada pixel.

```
__global__ void julMandelKernel(int width, int height, int n,
unsigned char *rgb){
      int linearIndex = threadIdx.x + blockIdx.x * blockDim.x;
      if(linearIndex<width*height){
            float x_min = -2.15;
            float x_max = 1.15;
            float y_min = -1.65;
            float y_max = 1.65;
            int x = linearIndex % width;
            int y = linearIndex / width;
            int k = linearIndex * 3;
            cuFloatComplex c = mapPoint(width, height, x, y,
            x_min, x_max, y_min, y_max);
            cuFloatComplex z0 = make_cuFloatComplex(0,0);
            int i = explode(z0, c, n);

            if (i < n){
                  color asp = fcolor(i, n);
                  rgb[k] = asp.bl;
                  rgb[k + 1] = asp.gr;
                  rgb[k + 2] = asp.re;
            }
      }
}
```

Como podemos ver tenemos un parámetro de entrada extra con respecto a `julMandel`. Esto se debe a que necesitamos pasar el buffer donde se almacenarán los píxeles. linearIndex nos calcula el índice del pixel que vamos a calcular representando su posición como si se encontrase en un array de una dimensión. A partir de este índice, y solo si

resulta menor que el total de píxeles, se procede a calcular cuáles serían sus coordenadas x e y en la imagen. Una vez hecho esto, se procede de la misma manera que en la versión secuencial, exceptuando la existencia del bucle for. Este bucle ha de quitarse, ya que no es necesario, puesto que cada thread calcula un solo pixel. También podemos ver como en CUDA la sintaxis de un número complejo es ligeramente distinta, habiendo de llamar a la función de creación `make_cuFloatComplex(real,imag)`.

3. **`fcolor, explode, mapPoint`**
   Estas funciones se mantienen exactamente igual en términos de actuación; sin embargo, deben hacerse ciertos cambios que detallo a continuación.
   a) Hay que añadir la palabra reservada `__device__` al comienzo de la función:

```
__device__ color fcolor(int iter,int num_its)
__device__ int explode( cuFloatComplex z0, cuFloatComplex c,
int n)
__device__ cuFloatComplex mapPoint(int width,int height,int
x,int y,float x_min, float x_max, float y_min, float y_max)
```

   Esto es así porque son funciones que van a ejecutarse en el device y son llamadas desde una función que también se ejecuta en el device.
   b) Deben hacerse cambios en la sintaxis de las declaraciones y las operaciones de números complejos. Se tienen que usar los equivalentes de la librería `cuComplex.h` (cN: complex number).

   i) **Declaraciones e inicializaciones:**
```
cuFloatComplex cN;
cN = make_cuFloatComplex(real,imag);
```
   ii) **Multiplicación:**
```
cuFloatComplex cN1 = cuCmulf(cN2, cN3);
```
   iii) **Suma:**
```
cuFloatComplex cN1 = cuCaddf(cN2, cN3);
```
   iv) **Módulo o magnitud:**
```
float n = cuCabsf(cN1, cN2);
```

Programa	**Opciones**	**Optimizaciones**	**t-min**	**t-medio**	**t-max**
Mandel_J	size: 1000X1000 iter: 5000	Opt. previas + Cómputo en GPU	0,179 s	0,193 s	0,207 s

Speedup: 126,72

Conclusiones

1. **Profiling exhaustivo:**
   Realizar un profiling exhaustivo del código antes de iniciar las optimizaciones y utilizar herramientas como `perf`, `valgrind`, y `VTune` para identificar áreas críticas y comprender el comportamiento del programa en términos de acceso a memoria, caché y hotspots es imprescindible para la eficiente optimización de un programa.

2. **Entendimiento profundo del código:**
   Comprender a fondo el código, su lógica y las dependencias entre las funciones es crucial. Un análisis detallado del código puede revelar oportunidades de optimización que podrían pasar desapercibidas con un enfoque más superficial.

3. **Optimizaciones incrementales:**
   Se debe adoptar un enfoque incremental para las optimizaciones. Es decir, realizar cambios pequeños y medir el impacto en el rendimiento antes de pasar a optimizaciones más sustanciales. Esto facilita la identificación de las modificaciones más efectivas.

4. **Estrategias de vectorización:**
   Utilizar estrategias de vectorización, como `#pragma omp simd` para aprovechar las capacidades de procesamiento vectorial del hardware. La vectorización puede mejorar significativamente el rendimiento al realizar operaciones en paralelo.

5. **Paralelización eficiente:**
   Seleccionar cuidadosamente las estrategias de paralelización basándose en la naturaleza del problema y la arquitectura del hardware. La combinación de directivas OpenMP, MPI o cómputo en GPU puede proporcionar beneficios muy significativos.

6. **Cómputo en GPU con CUDA:**
   Considerar la realización del cómputo intensivo en la GPU utilizando tecnologías como CUDA. La GPU puede manejar eficientemente operaciones paralelizables y mejorar el rendimiento global, especialmente en problemas altamente paralelizables, como puede ser éste, ya que hay múltiples operaciones del mismo tipo.

7. **Adaptabilidad a Hardware Variado:**
   Adaptar las estrategias de optimización al hardware subyacente. Lo que funciona bien en una arquitectura de CPU puede no ser óptimo para una GPU (en nuestra práctica, por ejemplo, el uso de librerías que diesen soporte para números complejos). Mantenerse al tanto de las características específicas del hardware y ajustar las optimizaciones en consecuencia es esencial para una optimización fiable y precisa.

8. **Pruebas Rigurosas:**
   Realizar pruebas rigurosas después de cada optimización para evaluar el impacto en el rendimiento y verificar la estabilidad del código. Las mejoras locales pueden tener consecuencias no deseadas, y es esencial asegurarse de que el código optimizado sigue siendo funcional y preciso. Además, es importante la repetición de pruebas cambiando los parámetros de configuración (por ejemplo, los tipos de schedule en la paralelización o el número de threads por bloque en CUDA) puesto que pueden suponer mejoras importantes en términos de rendimiento.

9. **Documentación Clara:**
   Mantener una documentación clara de las optimizaciones realizadas, los resultados obtenidos y las decisiones tomadas. Esto facilita la colaboración con otros desarrolladores y proporciona un historial valioso para futuras mejoras o mantenimiento del código. En esta práctica, la memoria y las carpetas con cada fase de optimización representan esta documentación valiosa para otros programadores.

Observaciones: Comandos de compilación y/o ejecución usados para cada parte de la práctica

1. **Profiling y optimizaciones previas del código secuencial**
   a. perf y valgrind: gcc -g Mandel_J.c -o Mandel_J -lm
   b. VTune: icx -g Mandel_J.c -o Mandel_J -lm
2. **Vectorización y paralelización**
   a. icx -qopenmp Mandel_J.c -o Mandel_J -lm
3. **MPI**
   a. mpicc -fopenmp -o Mandel_J.intel Mandel_J.c -lm
   b. mpirun -np 2 -hostfile hosts.mpich ./Mandel_J.intel 1000 5000
4. **CUDA**
   a. nvcc -arch=sm_35 Mandel_J.cu -o Mandel_J -lm

Comentarios generales e impresiones sobre las prácticas de Computación de Alto Rendimiento

En general, las dos prácticas realizadas en el curso de Computación de Alto Rendimiento han sido experiencias enriquecedoras que me han permitido profundizar en el mundo de la optimización de programas. Aunque ambas prácticas abordaron problemas distintos, la esencia de mejorar la eficiencia y rendimiento fue el hilo conductor que las unió.

En la primera práctica, centrada en la optimización de un programa de multiplicación de matrices, experimenté una curva de aprendizaje pronunciada. Al inicio, la tarea de identificar y

corregir cuellos de botella en el código parecía desafiante. Sin embargo, a medida que avanzaba, pude apreciar la importancia de estrategias como la vectorización y la paralelización para mejorar significativamente la velocidad de ejecución. Fue gratificante ver cómo pequeños cambios en el código podían tener un impacto tan significativo en el rendimiento. Es una pena que debido a la falta de planificación, no pudiese dar lo mejor de mí a la hora de la realización de los apartados de MPI y CUDA, sin embargo, es algo que corregí en la realización de la segunda práctica, que fue sin duda mi favorita.

La segunda práctica, enfocada en la optimización de un programa para generar imágenes del conjunto fractal de Mandelbrot basado en Julia, proporcionó un cambio refrescante en términos de aplicación práctica. La exploración de algoritmos más eficientes y la implementación de técnicas para reducir la carga computacional me llevaron a apreciar la belleza de combinar la teoría matemática con la programación de alto rendimiento. Ver cómo se generaban las imágenes fractales a la vez que el aumento de eficiencia del código incrementaba exponencialmente (casi 127 de speedup me parece una locura) fue, sin duda, una experiencia visualmente impactante.

Ambas prácticas me enseñaron la importancia de considerar no solo la lógica y la funcionalidad del código, sino también su rendimiento y eficiencia. A lo largo del proceso, también gané habilidades en el manejo de herramientas de profiling y análisis de rendimiento, lo cual considero esencial para cualquier programador de alto rendimiento.

En resumen, estas prácticas han fortalecido mi comprensión de la optimización de código y me han inspirado a seguir explorando nuevas técnicas y enfoques para mejorar la eficiencia en proyectos futuros. El curso de Computación de Alto Rendimiento ha demostrado ser fundamental para mi desarrollo como estudiante de informática, proporcionándome las herramientas necesarias para enfrentar desafíos técnicos y lograr soluciones más eficientes. Incluso cuando no estaba muy seguro de escoger esta asignatura, puesto que cuando entré en la carrera el hardware era algo que no me llamaba nada la atención, junto con Estructura y Arquitectura de Computadores, esta asignatura ha conseguido sacar a la luz una parte de mí que no conocía. Me lo he pasado realmente bien aprendiendo las técnicas de optimización y probablemente intente especializarme en esto, o al menos, seguir aprendiendo más sobre el tema.

Por último, quería dar las gracias a todos los profesores de la asignatura, que han sido esenciales para que me acabara gustando tanto. Sin su ayuda y su profesionalidad, no se habría despertado en mí esta pasión por la computación de alto rendimiento. Si tuviese que criticar algo, quizás me ha faltado adentrarnos más en aspectos de optimización a nivel de ensamblador o muy bajo nivel en general, aunque entiendo que el tiempo es limitado y no se puede abarcar todo lo que nos gustaría.