6620. Organización de Computadoras
Trabajo Práctico 2:
Programación SIMD, cómputo paralelo y profiling

Riesgo, Daniela, *Padrón Nro. 95557*
danielap.riesgo@gmail.com

Martin, Débora, *Padrón Nro. 90934*
debbie1mes.world@gmail.com

Constantino, Guillermo, *Padrón Nro. 89776*
guilleconstantino@gmail.com

2do. Cuatrimestre de 2014
66.20 Organización de Computadoras − Prática Martes
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

El presente trabajo tiene como objetivo familiarizarse con assembly x86 SSE mientras se trata con optimizaciones usando programación en paralelo (con *threading*) además de la programación SIMD.

Índice

Universidad de Buenos Aires - FIUBA
66.20 Organización de Computadoras
Trabajo práctico 2
2^o cuatrimestre de 2014

`$Date: 2014/10/05 18:17:07 $`

1. Objetivos

Ejercitar algunas de las técnicas de optimización de software vistas en el curso. Familiarizarse con elementos de assembler i386 y programación SIMD (single instruction, multiple data); y con el uso de herramientas de análisis, medición y profiling de sistemas de cómputo.

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes, un informe impreso de acuerdo con lo que mencionaremos en la sección 5, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

4. Descripción

Se trata de un modificar un programa de tal forma que dibuje el conjunto Multibrot de orden 3 [1] [2] [3] y sus vecindades, en el cual la lógica de de cómputo del fractal deberá tener soporte nativo para procesadores x86.

El código fuente con la versión inicial del programa, se encuentra disponible en [4]. El mismo deberá ser considerado como punto de partida de todas las implementaciones. En particular notar que este código implementa polinomios de orden 2, para lo cual cada grupo deberá realizar las modificaciones a la lógica de cómputo del programa descriptas en las sección 4.4.

4.1. Programa

El programa recibe, por línea de comando, la descripción de la región del plano complejo y las características del archivo imagen a generar. No interactúa con el usuario, ya que no se trata de un programa interactivo, sino más bien de una herramienta de procesamiento *batch*. Al finalizar la ejecución, y volver al sistema operativo, el programa habrá dibujado el fractal en el archivo de salida.

El formato gráfico a usar es PGM o *portable gray map* [5], un formato simple para describir imágenes digitales monocromáticas.

4.2. Algoritmo

.

El algoritmo básico es simple: para algunos puntos c de la región del plano que estamos procesando haremos un cálculo repetitivo. Terminado el cálculo, asignamos el nivel de intensidad del pixel en base a la condición de corte de ese cálculo.

El color de cada punto representa la "velocidad de escape" asociada con ese número complejo: blanco para aquellos puntos que pertenecen al conjunto (y por ende la "cuenta" permanece acotada), y tonos gradualmente más oscuros para los puntos divergentes, que no pertenezcan al conjunto.

Más específicamente: para cada pixel de la pantalla, tomaremos su punto medio, expresado en coordenadas complejas, $c = c_{re} + c_{im}i$. A continuación, iteramos sobre $f_{i+1}(c) = f_i(c)^3 + c$, con $f_0(c) = c$. Cortamos la iteración cuando $|f_i(c)| > 2$, o después de `N` iteraciones.

En pseudo código:

```
para cada pixel $p {
        $f = $c = complejo asociado a $p;
        for ($i = 0; $i < $N - 1; ++$i) {
                if (abs($f) > 2)
                        break;
                $f = $f * $f * $f + $c;
        }
        dibujar el punto p con brillo $i;
}
```

Así tendremos, al finalizar, una representación visual de la cantidad de ciclos de cómputo realizados hasta alcanzar la condición de escape (ver figura 1).

4.3. Interfaz

A fin de facilitar el intercambio de código *ad-hoc*, normalizaremos algunas de las opciones que deberán ser provistas por el programa:

- `-r`, o `--resolution`, permite cambiar la resolución de la imagen generada. El valor por defecto será de 640x480 puntos.
- `-c`, o `--center`, para especificar el punto central de la porción del plano complejo dibujada, expresado en forma binómica (i.e. $a + bi$). Por defecto usaremos `-0.60 + 0.25i`.
- `-w`, o `--width`, especifica el ancho del rectángulo que contiene la región del plano complejo que estamos por dibujar. Valor por defecto: `0.1`.

- `-H`, o `--height` sirve, en forma similar, para especificar el alto del rectángulo a dibujar. Valor por defecto: `0.1`.
- `-o`, o `--output`, coloca la imagen de salida, (en formato PGM [5]) en el archivo pasado como argumento; o por salida estándar `-stdout-` si el argumento es "-".
- `-m`, o `--method`, permite seleccionar dinámicamente la implementación a usar para generar la imagen de salida: `generic` para seleccionar la implementación genérica, en lenguaje C; y `sse` para usar, cuando esté disponible, la implementación con soporte para hardware SSE. Por defecto, la implementación de referencia utiliza `generic`.
- `-n`, o `--nthreads`, permite fijar la cantiad de threads que intervienen en el cómputo del fractal. El valor por defecto de la implementación de referencia es `1`, es decir, un único hilo de cómputo.

4.4. Entregables

El entregable producido en este trabajo deberá implementar la lógica de cómputo del fractal (Multibrot de orden 3) en assembly SSE, es decir deberá tener soporte nativo para procesadores intel.

Para ello, cada grupo deberá tomar el código fuente de base para este TP, [4], y reescribir la función `sse_plot()` de tal forma de que permita generar un Multibrot de orden 3 (la implementación provista implementa polinomios de grado 2). Esta función está ubicada en el archivo `sse_plot.c`.

Asimismo, cada grupo deberá proponer e implementar un esquema alternativo para el modelo de cómputo paralelo.

Por otro lado, si bien este trabajo no tiene requerimientos específicos de *speedup*, nos interesa conocer el efecto que tienen estos cambios en relación a la versión de base *single-threaded* codificada en C (opción `generic`).

Para ello, cada grupo deberá realizar las mediciones que considere necesarias, fundamentándolas en el TP.

4.5. Casos de prueba

El informe trabajo práctico deberá incluir una sección dedicada a verificar el funcionamiento del código implementado. Para ello, será necesario escribir pruebas orientadas a probar el programa completo, ejercitando los casos más comunes de funcionamiento, los casos de borde, y también casos de error.

4.6. Ejemplos

Generamos un dibujo usando los valores por defecto, barriendo la región rectangular del plano comprendida entre los vértices `-0.65 + 0.30i` y `-0.55 + 0.20i`.

```
$ tp1-2014-2-bin -o uno.pgm
```

La figura 1 muestra la imagen `uno.pgm`.

A continuación, hacemos *zoom* sobre la región centrada en `-0.6 + 0.6i`, usando un rectángulo de `0.05` unidades de lado. El resultado podemos observarlo en la figura 2.

```
$ tp1-2014-2-bin --center -0.6+0.6i --width 0.05 --height 0.05 -o dos.pgm
```

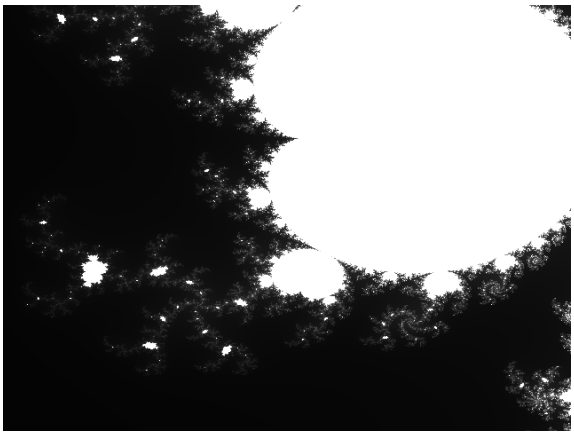

Figura 1: Región barrida por defecto.

Figura 2: Región comprendida entre `-0.625 + 0.625i` y `-0.575 + 0.575i`.

5. Informe

El informe deberá incluir:

- Documentación relevante al diseño e implementación del programa.
- Documentación relevante al proceso de compilación: cómo obtener el ejecutable a partir de los archivos fuente.
- Indicadores de performance de la aplicación, que permitan cuantificar el efecto de los cambios introducidos tomando como base de comparación la implementación *single-threaded* en lenguaje C.
- Las corridas de prueba, con los comentarios pertinentes.
- El código fuente, en lenguaje C.
- Este enunciado.

6. Fecha de entrega

La última fecha de entrega y presentación sería el martes 25/11.

Referencias

[1] `http://en.wikipedia.org/wiki/Mandelbrot_set` (Wikipedia).

[2] Introduction to the Mandelbrot Set.
`http://www.olympus.net/personal/dewey/mandelbrot.html`.

[3] Smooth shading for the Mandelbrot exterior.
`http://linas.org/art-gallery/escape/smooth.html`. Linas Vepstas. October, 1997.

[4] Código fuente con el esqueleto del trabajo práctico.
`http://www.fiuba7504.com.ar/tp2-2014-2-src.tar.gz`.

[5] PGM format specification.
`http://netpbm.sourceforge.net/doc/pgm.html`.

2. Introducción

Como primer objetivo, nos proponemos adentrarnos en el funcionamiento del assembly x86 SSE y la programación SIMD, en un entorno Intel. Averiguando sobre cómo el procesamiento SIMD permite trabajar con 4 variables flotantes de simple precisión en paralelo y cuáles son las instrucciones que se presentan para manejarlo en entorno Intel, implementándolo desde C con Assembler Inline, que maneja formato AT&T.
Luego, aprovechando la programación SIMD y el *threading* probar distintas formas de procesar la imagen para lograr mejorar el tiempo de ejecución mejorando la parte de plotting de la imagen.

3. Implementación del programa

3.1. Primera parte

Se trata de alterar el archivo *sse_plot.c* y el *generic_plot.c* para graficar el conjunto de Multibrot de orden 3 en vez del conjunto de Mandelbrot que ahora generan.
En el archivo *generic_plot.c* se cambió dentro del ciclo la forma de calcular los nuevos zr y zi. De la misma forma es lo único que debía cambiarse en el *sse_plot.c*. El único inconveniente con este segundo archivo es que los registros no alcanzaban para también recibir un vector de 4 números flotantes de simple precisión que contengan el 3 que debe ser usado en el cálculo de los nuevos zr y zi, pero por suerte podía ser obtenido a partir de restar el vector de 4s y 1's que ya se recibían.
Finalmente, para que la región barrida por defecto sea la especificada en el enunciado, también se debieron cambiar los parámetros por defecto del archivo *main.c* sobre las coordenadas de los puntos del extremo superior izquierdo y el extremo inferior derecho.

3.2. Segunda parte

Usando la programación SIMD ya de por si se logra una mejora en tiempo de ejecución ya que se puede procesar 4 pixeles de la imagen a la vez. Sin embargo, la programación con *threading* puede en ciertos casos mejorar aún más la ejecución, dependiendo de la cantidad de threads a usar y los cores que se tengan disponibles para ejecutarlos.
El programa recibido del curso implementa programación en paralelo haciendo threads que se encargan de bandas horizontales de la imagen. Esto no siempre es lo más conveniente porque la imagen puede requerir más cálculo en cierta banda o zona de la imagen que otra. Por esto primero se decidió probar con procesamiento de bandas verticales de la imagen, esperando que el resultado no fuera tan distinto y dependiera de la zona a dibujar del fractal. Luego se intentó hacer cómputo paralelo de distintas zonas rectangulares de la imagen, una para cada thread. Nuevamente se espera que dependa de la imagen pero de todas formas se espera un tiempo de cómputo para cada thread equilibrado.

Asique por último se implementó cómputo paralelo en donde se tomara a la imagen como un conjunto de rectángulos y donde cada thread se encargara

de calcular varios rectángulos pero que pertenezcan a bandas horizontales y verticales distintas.

Primero, para compilar y obtener el ejecutable del programa se usa la siguiente sentencia:

```
$ make -f Makefile.init Makefiles CCARGS="-g -DUSE_SSE_ASSEMBLY -msse"

make -f Makefile.in MAKELEVEL= Makefiles
(echo "# Do not edit -- this file documents how this program was built for your machine.";
set +e;                                 \
if cmp makedefs.tmp makedefs.out; then \
    rm makedefs.tmp;                    \
else                                    \
    mv makedefs.tmp makedefs.out;       \
fi >/dev/null 2>/dev/null
rm -f Makefile; (cat makedefs.out Makefile.in) >Makefile
$ make
```

Para testear performance se usó el comando time que devuelve el tiempo real, de user y de system. Es decir, *real*: tiempo desde que se llamó a la función hasta que terminó; esto incluye el tiempo que haya gastado en otros procesos. *user*: tiempo de CPU que tarda el proceso especificado en modo de usuario y no kernel. *sys*: tiempo de CPU que tarda el proceso especificado en modo kernel.

Con esto nos importa la parte de cómputo del fractal que es en la cual nos enfocamos en mejorar para obtener un alto SpeedUp por lo que el tiempo *user* es el que rescataremos de los resultados.

Luego se probó la versión generic y con la función time se obtuvo lo siguiente. Para estas corridas se utilizaron las versiones 2 de los archivos que corresponden a los mismos archivos de la cátedra pero para el conjunto de Multibrot de orden 3.

```
$ time ./tp2-2014-2-bin -o uno.pgm

user 0m0.420s
```

Con la versión sse se obtuvo:

```
$ time ./tp2-2014-2-bin -o uno.pgm -m sse

user 0m0.200s
```

Si se usan por ejemplo dos threads: con versión generic

```
$ time ./tp2-2014-2-bin -o uno.pgm -n 2

user 0m0.404s
```

con versión sse

```
$ time ./tp2-2014-2-bin -o uno.pgm -m sse -n 2

user 0m0.168s
```

Pero ya si se usan demasiados threads, esto es contraproductivo al tiempo que le toma al sistema el proceso:

```
$ time ./tp2-2014-2-bin -o uno.pgm -m sse -n 10

user 0m0.192s


$ time ./tp2-2014-2-bin -o uno.pgm -m sse -n 20

user 0m0.208s
```

Más adelante se pensó en probar con bandas verticales en vez de horizontales, esperando que esto dependa de cómo se distribuyan los cálculos en la imagen pero sin poder asegurar nada para cualquier caso general. Para esto se usó la versión 3 de todos los archivos.

Para 3 threads,

```
time ./tp2-2014-2-bin -o uno.pgm -n 3

user 0m0.420s
```

como el caso visto. Pero veamos dos nuevas imagénes y comparemos mejor

Versión 3: (Bandas verticales)

```
$ time ./tp2-2014-2-bin -o mejor_horizontal.pgm -c -0.5520+0.2001i -n 2

user 0m0.548s


$ time ./tp2-2014-2-bin -o mejor_vertical.pgm -c -0.015+1.2i -H 0.5 -w 0.5 -n 2

user 0m0.348s
```

Versión 2: (Bandas horizontales)

```
$ time ./tp2-2014-2-bin -o mejor_horizontal.pgm -c -0.5520+0.2001i -H 0.5 -w 0.5 -n 2

user 0m0.456s


$ time ./tp2-2014-2-bin -o mejor_vertical.pgm -c -0.015+1.2i -H 0.5 -w 0.5 -n 2

user 0m0.376s
```

Imagen	Version 2	Version 3
mejor_horizontal.pgm	0,456 s	0,548 s
mejor_vertical.pgm	0,376 s	0,348 s

Cuadro 1: Valores de tiempo medidos para dos threads

Como se ve, en una imagen en donde lo negro y lo blanco está concentrado en bandas verticales distintas tarda más para esa versión porque se debe esperar a que termine el thread que más tarde en computar (que más negro tenga). Mientras que si lo negro está concentrado en bandas horizontales ese gran cómputo se distribuye en distintos threads disminuyendo el tiempo total. Lo mismo aunque de forma invertida se esperaría en el caso contrario cuando se usa la versión de bandas horizontales, pero las pruebas demuestran lo contrario.f Sin embargo sí podemos decir que el tiempo que tarda en computar el fractal de la imagen mejor_horizontal es mejor en la versión de bandas horizontales que en la de bandas verticales. Y el mismo resultado esperado se da para la otra imagen.

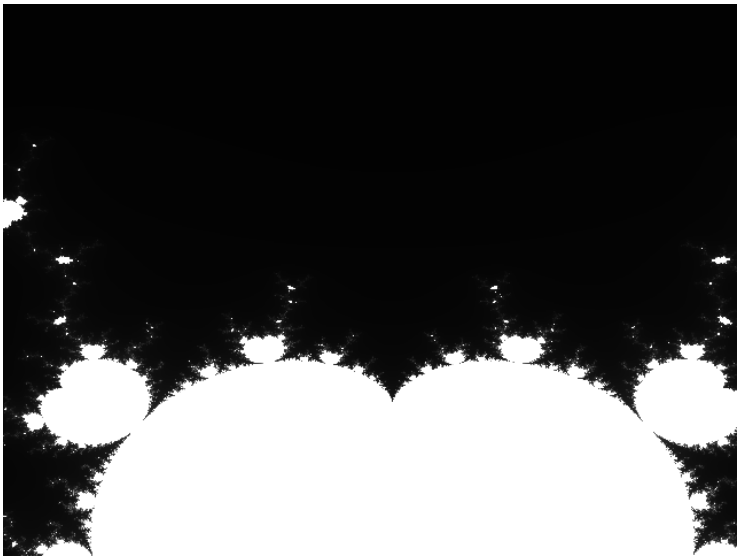

Figura 1: Resultado de $./tp2 - 2014 - 2 - bin - omejor\_vertical.pgm - c - 0{,}015 + 1{,}2i - H0{,}5 - w0{,}5 - n2$

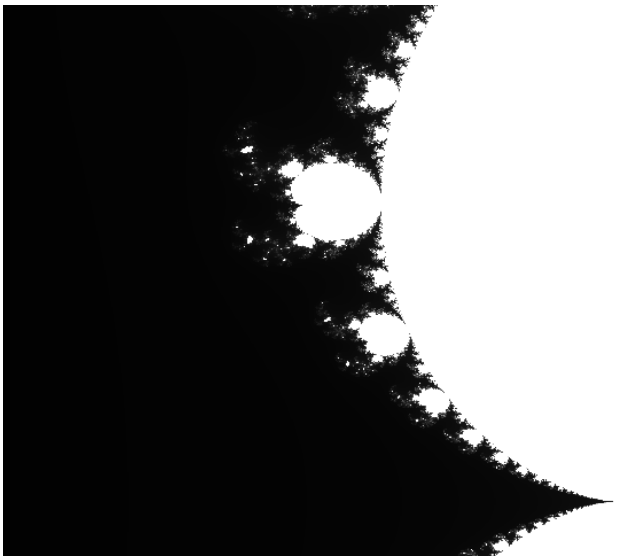

Figura 2: Resultado de $./tp2 - 2014 - 2 - bin - omejor\_horizontal.pgm - c - 0{,}5520 + 0{,}2001i - H0{,}5 - w0{,}5 - n2$

Finalmente se ideó una versión donde la imagen se divida en rectángulos, la cantidad es según especifique el usuario, y se le asigna un thread a cada uno. Corresponde a la versión 4 de los archivos.

```
$ time ./tp2-2014-2-bin -o uno.pgm -m sse
user 0m0.208s

$ time ./tp2-2014-2-bin -o uno.pgm
user 0m0.423s

$ time ./tp2-2014-2-bin -o uno.pgm -m sse -n 2x3
user 0m0.200s

$ time ./tp2-2014-2-bin -o uno.pgm -n 2x3
user 0m0.404s

$ time ./tp2-2014-2-bin -o uno.pgm -m sse -n 3x5
user 0m0.192s

$ time ./tp2-2014-2-bin -o uno.pgm -n 3x5
user 0m0.440s

$ time ./tp2-2014-2-bin -o uno.pgm -c -0.5520+0.2001i -H 0.5 -w 0.5 -n 2x2
user 0m0.520s

$ time ./tp2-2014-2-bin -o uno.pgm -c -0.015+1.2i -H 0.5 -w 0.5 -n 2x2
user 0m0.364s
```

Se puede ver como las mejoras no son muy notables, pero sin embargo en comparación con los casos anteriores se ve que esta implementación siempre está dentro de lo aceptable sin ser tener la peor performance pero tampoco la mejor. Esto podemos suponer que se debe a que usa varios threads a la vez y la computadora no puede manejarlos eficientemente.

Imagen Default	Version 2		Version 4	
Threads	generic (s)	sse (s)	generic (s)	sse (s)
1	0,420	0,200	0,423	0,208
2	0,404	0,168		
3	0,420		0,404	0,200
5			0,440	0,192
10		0,192		
20		0,208		

Cuadro 2: Valores de tiempo medidos para la imagen default

4. Corridas de prueba

Para cada una de las versiones se realizaron las pruebas normales para verificar el funcionamiento correcto.

1. Generamos una imagen de 1 punto de lado, centrada en el orígen del plano complejo:

```
> ./tp0 -c 0+0i -r 1x1 -o -
P2
1
1
255
255
```

Notar que el resultado es correcto, ya que este punto pertenece al conjunto de Mandelbrot.

2. Repetimos el experimento, pero nos centramos ahora en un punto que seguro no pertenece al conjunto:

```
> ./tp0 -c 10+0i -r 1x1 -o -
P2
1
1
255
0
```

3. Imagen PGM

```
> ./tp2-2014-2-bin -o por_default.pgm
```

Genera la imagen siguiente imagen:

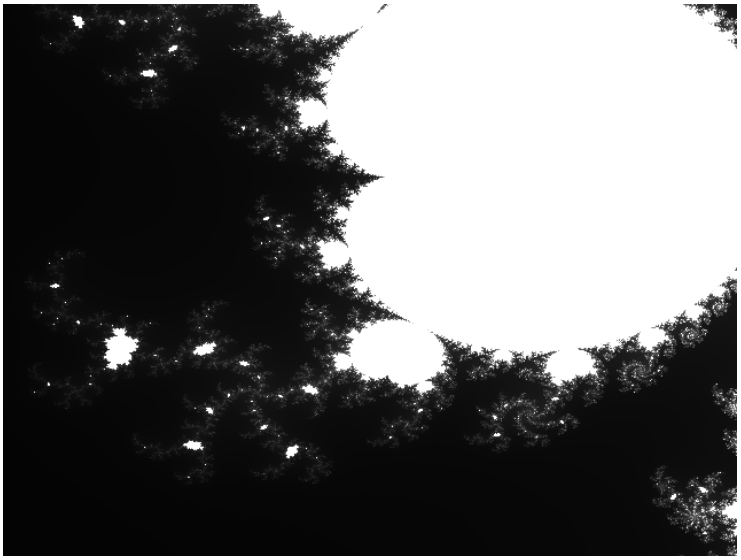

Figura 3: Region barrida por defecto

4. Imagen PGM con región no centrada y un rectángulo de 0,005 unidades de lado.

```
> ./tp2-2014-2-bin --center -0.6+0.6i --width 0.05 --height 0.05 -o zoom.pgm
```

Genera la siguiente imagen:

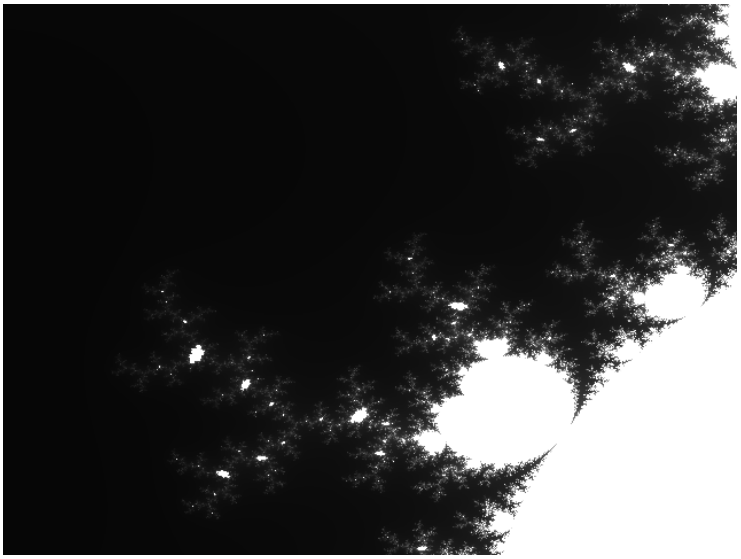

Figura 4: Región comprendida entre -0.625 + 0.625i y -0.575 + 0.575i.

5. Código fuente C

Para cada versión se explicitaran sólo los cambios a partir de los archivos ofrecidos por la cátedra para facilitar la lectura.

5.1. Versión 2

5.1.1. main2.c

```
#include <param2.h>

float upper_left_re = -0.65; /* Extremo superior izquierzo (re). */
float upper_left_im = +0.30; /* Extremo superior izquierzo (im). */
float lower_right_re = -0.55; /* Extremo inferior derecho (re). */
float lower_right_im = +0.20; /* Extremo inferior derecho (im). */
```

5.1.2. generic_plot2.c

```
#include <param2.h>

for (c = 0; c < parms->shades; ++c) {
if ((absz = zr*zr + zi*zi) >= 4.0f)
break;
sr = zr * zr * zr
   - 3 * zr * zi * zi
   + cr;
si = 3 * zr * zr * zi
   - zi * zi * zi
   + ci;
zr = sr;
zi = si;
}
```

5.1.3. sse_plot2.c

```
#include <param2.h>


/* Calculate Z = Z^3 + C. */
"Z_eq_Z3_plus_C:         \n\t"
"movaps   %6, %%xmm6     \n\t" /* xmm6: FOUR */
"subps    %%xmm1, %%xmm6 \n\t" /* xmm6: THREE */

"movaps   %%xmm5, %%xmm7 \n\t" /* xmm7: ZI^2 */
"mulps    %%xmm2, %%xmm7 \n\t" /* xmm7: ZR * ZI^2 */
"mulps    %%xmm6, %%xmm7 \n\t" /* xmm7: 3 * ZR * ZI^2 */
"mulps    %%xmm4, %%xmm2 \n\t" /* xmm2: ZR^3 */
"subps    %%xmm7, %%xmm2 \n\t" /* xmm2: ZR^3 - 3 * ZR * ZI^2 */

"mulps    %%xmm4, %%xmm6 \n\t" /* xmm6: 3 * ZR^2 */
"mulps    %%xmm3, %%xmm6 \n\t" /* xmm6: 3 * ZR^2 * ZI */
```

```
"mulps    %%xmm3, %%xmm5 \n\t" /* xmm5: ZI^3 */
"subps    %%xmm5, %%xmm6 \n\t" /* xmm6:  3 * ZR^2 * ZI - ZI^3 */

"addps    %3, %%xmm2     \n\t" /* xmm2: += CR */
"addps    %4, %%xmm6     \n\t" /* xmm6: += CI */
   /* xmm2: new ZR */
"movaps   %%xmm6, %%xmm3 \n\t" /* xmm3: new ZI */

"jmp      loop           \n\t"
```

5.2. Versión 3

5.2.1. main3.c

```
#include <param3.h>

float upper_left_re = -0.65; /* Extremo superior izquierzo (re). */
float upper_left_im = +0.30; /* Extremo superior izquierzo (im). */
float lower_right_re = -0.55; /* Extremo inferior derecho (re). */
float lower_right_im = +0.20; /* Extremo inferior derecho (im). */

parms.x0 = 0;
parms.x1 = x_res;
size_t x0 = 0;
size_t xd = x_res / nthreads;

for (tn = 0; tn < nthreads; ++tn) {
memcpy(&tinfo[tn].parms, &parms, SZ(parms));
tinfo[tn].parms.x0 = x0;
tinfo[tn].parms.x1 = (tn < (nthreads - 1))
                     ? (x0 = x0 + xd)
                     : (x_res);

if (pthread_create(&tinfo[tn].tid,
                   NULL,
                   (void *)plot,
                   &tinfo[tn].parms) != 0) {
fprintf(stderr, "cannot create thread.\n");
exit(1);
}
}
```

5.2.2. param3.h

```
typedef struct {
float UL_re;
float UL_im;
float LR_re;
float LR_im;
```

```
float d_re;
float d_im;

size_t x_res;
size_t y_res;
size_t shades;

size_t x0;
size_t x1;

uint8_t *bitmap;
} param_t;
```

5.2.3. generic_plot3.c

```
#include <param3.h>

ci = parms->UL_im;
u8 = parms->bitmap + parms->x0;

for (y = 0; y < parms->y_res; ++y) {
cr = parms->UL_re + parms->x0 * parms->d_re;

for (x = parms->x0; x < parms->x1; ++x) {

zr = cr;
zi = ci;

/*
 * Determinamos el nivel de brillo asociado al punto
 * (cr, ci), usando la fórmula compleja recurrente
 * f = f^3 + c.
 */
for (c = 0; c < parms->shades; ++c) {
if ((absz = zr*zr + zi*zi) >= 4.0f)
break;
sr = zr * zr * zr
   - 3 * zr * zi * zi
   + cr;
si = 3 * zr * zr * zi
   - zi * zi * zi
   + ci;
zr = sr;
zi = si;
}

/* Escribimos la intensidad del px. */
*u8++ = (uint8_t)c;

/* Calculamos la siguiente parte real. */
```

```
cr += parms->d_re;
}
/* Calculamos el lugar en el bitmap de la proxima linea. */
u8 += (parms->x_res - (parms->x1 - parms->x0));

/* Calculamos la siguiente parte compleja. */
ci -= parms->d_im;

}
```

5.2.4. sse_plot3.c

```
#include <param3.h>

CR0[0] = parms->UL_re + (parms->x0 + 0.5f) * parms->d_re;
CR0[1] = parms->UL_re + (parms->x0 + 1.5f) * parms->d_re;
CR0[2] = parms->UL_re + (parms->x0 + 2.5f) * parms->d_re;
CR0[3] = parms->UL_re + (parms->x0 + 3.5f) * parms->d_re;
INIT4(CI0, parms->UL_im - 0.5f * parms->d_im);

u8 =  parms->bitmap + parms->x0;

/* Calculate Z = Z^3 + C. */
"Z_eq_Z3_plus_C:         \n\t"
"movaps   %6, %%xmm6     \n\t" /* xmm6: FOUR */
"subps    %%xmm1, %%xmm6 \n\t" /* xmm6: THREE */

"movaps   %%xmm5, %%xmm7 \n\t" /* xmm7: ZI^2 */
"mulps    %%xmm2, %%xmm7 \n\t" /* xmm7: ZR * ZI^2 */
"mulps    %%xmm6, %%xmm7 \n\t" /* xmm7: 3 * ZR * ZI^2 */
"mulps    %%xmm4, %%xmm2 \n\t" /* xmm2: ZR^3 */
"subps    %%xmm7, %%xmm2 \n\t" /* xmm2: ZR^3 - 3 * ZR * ZI^2 */

"mulps    %%xmm4, %%xmm6 \n\t" /* xmm6: 3 * ZR^2 */
"mulps    %%xmm3, %%xmm6 \n\t" /* xmm6: 3 * ZR^2 * ZI */
"mulps    %%xmm3, %%xmm5 \n\t" /* xmm5: ZI^3 */
"subps    %%xmm5, %%xmm6 \n\t" /* xmm6:  3 * ZR^2 * ZI - ZI^3 */

"addps    %3, %%xmm2     \n\t" /* xmm2: += CR */
"addps    %4, %%xmm6     \n\t" /* xmm6: += CI */
   /* xmm2: new ZR */
"movaps   %%xmm6, %%xmm3 \n\t" /* xmm3: new ZI */

"jmp      loop           \n\t"



size_t suma_u8 = parms->x_res - (parms->x1 - parms->x0);
```

```
__asm__ volatile (
"U8_avanza_una_linea:    \n\t"
#if _LP64
"addq     %2, %0         \n\t"
#else
"addl     %2, %0         \n\t"
#endif
: "=r" (u8) /* %0 */
: "0" (u8),/* %1 */
  "m" (suma_u8)/* %2 */
: "cc"
);
```

5.3. Versión 4

5.3.1. main4.c

```
#include <param4.h>

float upper_left_re = -0.65; /* Extremo superior izquierzo (re). */
float upper_left_im = +0.30; /* Extremo superior izquierzo (im). */
float lower_right_re = -0.55; /* Extremo inferior derecho (re). */
float lower_right_im = +0.20; /* Extremo inferior derecho (im). */
size_t nthreads_x = 1;            /* Cantidad de threads de cómputo en x. */
size_t nthreads_y = 1;            /* Cantidad de threads de cómputo en y. */

static void
do_nthreads(const char *name, const char *spec)
{
int nt_x, nt_y;
char x, c;

if (sscanf(spec, "%d%c%d %c", &nt_x, &x, &nt_y, &c) != 3
|| nt_x <= 0 || nt_y <= 0 || x != 'x')
do_usage(name, 1);

/* Set new threads. */
nthreads_x = nt_x;
nthreads_y = nt_y;
}

size_t y0, x0;
size_t yd, xd;
size_t tn, nt;

parms.y0 = 0;
parms.y1 = y_res;
parms.x0 = 0;
parms.x1 = x_res;
```

```
if ((tinfo = malloc(SZ(struct thread_info) * nthreads_x * nthreads_y)) == NULL) {
fprintf(stderr, "cannot allocate memory.\n");
exit(1);
}

y0 = 0;
yd = y_res / nthreads_y;
x0 = 0;
xd = x_res / nthreads_x;

for (tn = 0; tn < nthreads_y; ++tn) {
for (nt = 0; nt < nthreads_x; ++nt){
memcpy(&tinfo[tn * nthreads_x + nt].parms, &parms, SZ(parms));
tinfo[tn * nthreads_x + nt].parms.y0 = y0 + tn * yd ;
tinfo[tn * nthreads_x + nt].parms.y1 = tn < (nthreads_y - 1)
 ? (y0 + (tn + 1) * yd)
 : (y_res);
tinfo[tn * nthreads_x + nt].parms.x0 = x0 + nt * xd;
tinfo[tn * nthreads_x + nt].parms.x1 = nt < (nthreads_x - 1)
 ? (x0 + (nt + 1) * xd)
 : (x_res);

if (pthread_create(&tinfo[tn * nthreads_x + nt].tid,
   NULL,
   (void *)plot,
   &tinfo[tn * nthreads_x + nt].parms) != 0) {
fprintf(stderr, "cannot create thread.\n");
exit(1);
}
}
}
```

5.3.2. param4.h

```
typedef struct {
float UL_re;
float UL_im;
float LR_re;
float LR_im;
float d_re;
float d_im;

size_t x_res;
size_t y_res;
size_t shades;
```

```
size_t y0;
size_t y1;
size_t x0;
size_t x1;

uint8_t* bitmap;
} param_t;
```

5.3.3. generic_plot4.c

```
#include <param4.h>

ci = parms->UL_im - parms->y0 * parms->d_im;
u8 = parms->bitmap + parms->y0 * parms->x_res + parms->x0;

for (y = parms->y0; y < parms->y1; ++y) {
cr = parms->UL_re + parms->x0 * parms->d_re;

for (x = parms->x0; x < parms->x1; ++x) {
zr = cr;
zi = ci;

/*
 * Determinamos el nivel de brillo asociado al punto
 * (cr, ci), usando la fórmula compleja recurrente
 * f = f^3 + c.
 */
for (c = 0; c < parms->shades; ++c) {
if ((absz = zr*zr + zi*zi) >= 4.0f)
break;
sr = zr * zr * zr
   - 3 * zr * zi * zi
   + cr;
si = 3 * zr * zr * zi
   - zi * zi * zi
   + ci;
zr = sr;
zi = si;
}

/* Escribimos la intensidad del px. */
*u8++ = (uint8_t)c;

/* Calculamos la siguiente parte real. */
cr += parms->d_re;
}
/* Calculamos el lugar en el bitmap de la proxima linea. */
u8 += (parms->x_res - (parms->x1 - parms->x0));
```

```
/* Calculamos la siguiente parte compleja. */
ci -= parms->d_im;
}
```

5.3.4. sse_plot4.c

```
#include <param4.h>

CR0[0] = parms->UL_re + (parms->x0 + 0.5f) * parms->d_re;
CR0[1] = parms->UL_re + (parms->x0 + 1.5f) * parms->d_re;
CR0[2] = parms->UL_re + (parms->x0 + 2.5f) * parms->d_re;
CR0[3] = parms->UL_re + (parms->x0 + 3.5f) * parms->d_re;
INIT4(CI0, parms->UL_im - (parms->y0 + 0.5f) * parms->d_im);

u8 = parms->bitmap + parms->y0 * parms->x_res + parms->x0;

/* Calculate Z = Z^3 + C. */
"Z_eq_Z3_plus_C:         \n\t"
"movaps   %6, %%xmm6     \n\t" /* xmm6: FOUR */
"subps    %%xmm1, %%xmm6 \n\t" /* xmm6: THREE */

"movaps   %%xmm5, %%xmm7 \n\t" /* xmm7: ZI^2 */
"mulps    %%xmm2, %%xmm7 \n\t" /* xmm7: ZR * ZI^2 */
"mulps    %%xmm6, %%xmm7 \n\t" /* xmm7: 3 * ZR * ZI^2 */
"mulps    %%xmm4, %%xmm2 \n\t" /* xmm2: ZR^3 */
"subps    %%xmm7, %%xmm2 \n\t" /* xmm2: ZR^3 - 3 * ZR * ZI^2 */

"mulps    %%xmm4, %%xmm6 \n\t" /* xmm6: 3 * ZR^2 */
"mulps    %%xmm3, %%xmm6 \n\t" /* xmm6: 3 * ZR^2 * ZI */
"mulps    %%xmm3, %%xmm5 \n\t" /* xmm5: ZI^3 */
"subps    %%xmm5, %%xmm6 \n\t" /* xmm6:  3 * ZR^2 * ZI - ZI^3 */

"addps    %3, %%xmm2     \n\t" /* xmm2: += CR */
"addps    %4, %%xmm6     \n\t" /* xmm6: += CI */
   /* xmm2: new ZR */
"movaps   %%xmm6, %%xmm3 \n\t" /* xmm3: new ZI */

"jmp      loop           \n\t"


size_t suma_u8 = parms->x_res - (parms->x1 - parms->x0);

__asm__ volatile (
"U8_avanza_una_linea:    \n\t"
#if _LP64
"addq     %2, %0         \n\t"
#else
"addl     %2, %0         \n\t"
#endif
```

```
: "=r" (u8) /* %0 */
: "0" (u8),/* %1 */
  "m" (suma_u8)/* %2 */
: "cc"
);
```

6. Conclusiones

Al medir los tiempos de cada corrida se hizo en una misma máquina para poder comparar los resultados entre sí. Si se observan los tiempos expuestos en la segunda tabla, se puede ver que la implementación genérica tarda más tiempo en los casos en que se comparan las corridas con poca cantidad de hilos. Esto puede deberse a que, al ser necesario intercambiar una mayor cantidad de hilos, el procesador pierde más tiempo que el que usaría al correrlo normalmente.

Por otro lado, también se puede ver que las implementaciones inline son más eficientes debido a que optimizan el trabajo más de lo que lo haría el compilador.

Al comparar las implementaciones con bandas verticales y horizontales se pudo notar que la eficiencia depende de la distribución del color en la imagen. Determinado método resultará mejor si la cantidad de pixeles blancos y su intencidad son equivalentes en cada segmento de la imagen, los cuales son procesados por distintos threads. Esto quedó demostrado para el caso de las imágenes expuestas.

Referencias

[1] Mandelbrot, http://en.wikipedia.org/wiki/Mandelbrot_set/

[2] Introduction to the Mandelbrot Set, http://www.olympus.net/personal/dewey/mandelbrot.html.

[3] Smooth shading for the Mandelbrot exterior. http://linas.org/art-gallery/escape/smooth.html. Linas Vepstas. October, 1997.

[4] PGM format specification. http://netpbm.sourceforge.net/doc/pgm.html

[5] Oetiker, Tobias, "The Not So Short Introduction To LaTeX2", http://www.physics.udel.edu/$\sim$dubois/lshort2e/