Práctica 04-06

Programación Concurrente y Paralela (PCP) # GIITIN

16/12/22

Rodríguez López, Alejandro # UO281827

Tabla de Contenido

Tabla de Ilustraciones

Tabla de Gráficas

Tabla de Funciones

Tabla de Tablas

Introducción

La programación en procesadores multinúcleo reduce enormemente el tiempo de ejecución de aquellos programas escritos con el cuidado suficiente para aprovechar las capacidades del hardware actual. En ese momento, habrá parecido casi imposible la necesidad de esperar por la finalización de un programa gracias a los avances en el paralelismo.

"There aren't likely to be any final answers. Both because the problems are hard and because as we find solutions we find even more ambitious objectives."[1]

Y efectivamente, así fue. Los problemas se volvieron más complicados debido al aumento en ambición y pronto fue necesario encontrar otra forma de aumentar la velocidad a la que se ejecutasen los programas.

¿Qué vendría ahora? ¿FPGA? ¿Aumentar los relojes de las CPU? ¿Más cores? ¿Más sockets?

A pesar de que todas estas soluciones acabaron llegando tarde o temprano (algunas de ellas con más éxito que otras), serán las GPU sobre las que nos centramos en este proyecto.

Gracias a este nuevo hardware, la posibilidad de acelerar los programas aún más volvería a existir. El salto de paralelismo en CPU a paralelismo en GPU sería a una escala mucho mayor, pero conllevaría también numerosos cambios en la forma de resolver los problemas.

Rodríguez López, Alejandro.
UO281827.

1 *"No es probable que hayan soluciones finales [al problema]. Tanto porque los problemas son difíciles y porque a medida que encontramos soluciones, hallamos objetivos aun más ambiciosos." ~ Victor A. Vyssotsky (https://www.youtube.com/watch?v=tc4ROCJYbm0)*

Propósito y motivaciones

Como ya se adelantó en la introducción, nos centramos en el uso de las GPU para el cálculo de problemas.

Se realizará un análisis teórico del problema a resolver (el cálculo de una fractal) y se comparará con algunas soluciones que se aprovechen del cálculo multinúcleo en CPU. Posteriormente, se introducirá el concepto de GPU CUDA y se resolverá el mismo problema.

Durante el proceso, se analizará el funcionamiento de las GPU de nVidia a la hora de ejecutar código CUDA, y finalmente se compararán los avances realizados con el nuevo hardware frente a los datos originales de CPU.

Preámbulo

Las ejecuciones con CPU se han realizado en los equipos de la universidad, mientras que las ejecuciones con GPU se han realizado en 2 sistemas distintos, el de la universidad (1660Ti) y el propio del alumno (1050Ti).

Las tablas con tiempos de respuesta han sido rellenadas con datos de la 1660Ti, ya que son parte obligatoria de la entrega, todos los datos que se obtengan como consecuencia de dichos tiempos también son resultado de la misma GPU.

El resto de las tareas opcionales, particularmente aquellas más complicadas (como el cálculo heterogéneo) han sido ejecutadas en la 1050Ti.

No obstante, siempre que se utiliza la GPU 1050Ti se indica al inicio de la sección.

Todas las funciones presentadas se encuentran junto con un Makefile personalizado (Para la razón de ello véase: AtomicAdd) al final del documento en el Anexo.

Análisis Teórico

Sistemas utilizados

$$TPP_{dp} = chasis \cdot nodos_{chasis} \cdot sockets_{nodo} \cdot cores_{socket} \cdot clock_{GHz} \cdot \frac{n^o\ flop}{ciclo}_{dp}$$

Function 1: TPP_{dp}

De acuerdo con ésta fórmula y sabiendo los datos de cada procesador, podemos calcular el TPP de cada sistema:

$TPP_{i3} = 1*1*1*2*3.1*8 = 49.6\,(GFlop)$
$TPP_{Ryzen} = 1*1*1*8*3.6*16 = 460.8\,(GFlop)$
$TPP_{Xeon} = 1*1*2*4*2.4*8 = 153.6\,(GFlop)$

Sabiendo el TPP (número de flops por segundo) del sistema, podemos conocer el tiempo que tarda en realziar un flop.

$$t_c = \frac{1}{TPP_{dp}}$$

Function 2: t_c

Table 1: TPP_{dp}

T_c Paralelo	T_c Secuencial
$t_{ci3} = \frac{1}{TPP_{i3} \cdot 10^9} = 2.01613\text{E-}11\,(seg)$	$t_{ci3} = \frac{2}{TPP_{i3} \cdot 10^9} = 4.0323\text{E-}11\,(seg)$
$t_{cRyzen} = \frac{1}{TPP_{Ryzen} \cdot 10^9} = 2.1701\text{E-}12\,(seg)$	$t_{cRyzen} = \frac{8}{TPP_{Ryzen} \cdot 10^9} = 1.7361\text{E-}12\,(seg)$
$t_{cXeon} = \frac{1}{TPP_{Xeon} \cdot 10^9} = 6.5104\text{E-}12\,(seg)$	$t_{cXeon} = \frac{8}{TPP_{Xeon} \cdot 10^9} = 5.2083\text{E-}11\,(seg)$

Problema a resolver

El problema del cálculo de la fractal y de su equivalente binarizada se divide en 3 partes:

1. Cálculo de la fractal.
2. Cálculo de la media.
3. Binarización

Estas tres partes deberán ser resueltas individualmente. Cada una de ellas se podría tratar como un problema independiente.

Complejidad temporal y espacial de los problemas

Cálculo de la fractal

Descripción.

Sin lugar a dudas el problema que presenta mayor dificultad matemática. La fractal del conjunto de Mandelbrot se define por:

$$\begin{aligned} z_0 &= 0 \\ z_{k+1} &= z_k^2 + c, c \in \mathbb{C} \end{aligned}$$

Function 3: Mandelbrot.

O en el código equivalente:

```
double    dx = (xmax-xmin)/xres,
          dy = (ymax-ymin)/yres,
          u = 0, v = 0, u_old = 0,
          paso_x = i*dx+xmin,
          paso_y = j*dy+ymin;

int       k = 1;

while (k < maxiter && (u*u+v*v) < 4)
{
    u_old = u;
    u = u_old*u_old - v*v + paso_x;
    v = 2*u_old*v + paso_y;
    k = k + 1;
}

if (k ≥ maxiter)     *(A+i+j*xres) = 0;
else                 *(A+i+j*xres) = k;
```

Siendo xres e yres los tamaños en pixeles (o posiciones del vector) del eje x e y respectivamente. Y siendo xmin, ymin, xmax e ymax las coordenadas de las esquinas superior izquierda e inferior derecha de la zona a mapear en el vector.

Ya que se trata de una serie, se utilizará un parámetro *maxiter* para limitar el número de iteraciones a ejecutar.

Complejidad computacional.

Debido a la condición (u*u+v*v) < 4, el número de iteraciones puede no ser el mismo para todos los píxeles, por ello, para el cálculo de la complejidad del problema, nos centraremos en el peor caso.

```
double    dx = (xmax-xmin)/xres,          // 1 FLOP
          dy = (ymax-ymin)/yres,          // 1 FLOP
          u = 0, v = 0, u_old = 0,
          paso_x = i*dx+xmin,             // 1 FLOP
          paso_y = j*dy+ymin;             // 1 FLOP
int       k = 1;

while (k < maxiter && (u*u+v*v) < 4)    // maxiter iteraciones, 2 FLOP.
{
    u_old = u;
    u = u_old*u_old - v*v + paso_x;      // 2 FLOP
    v = 2*u_old*v + paso_y;              // 2 FLOP
    k = k + 1;
}
if (k ≥ maxiter)     *(A+i+j*xres) = 0; // 1 FLOP
else                 *(A+i+j*xres) = k; // 1 FLOP
```

Tras analizar el coste del problema mostrado en pantalla, concluimos que la complejidad ronda los $6 \cdot maxiter + 5$ FLOP por píxel. Ya que hay $xres \times yres$ píxeles, la complejidad en el peor caso sería:

$$\begin{aligned} T_{mandel_{monohilo}}(xres, yres, maxiter) &= xres \cdot yres \cdot (6 \cdot maxiter + 5) \cdot t_c \\ O_{mandel_{monohilo}} &\in n^2 \end{aligned}$$

Function 4: Complejidad mandel

Ya que se puede asignar un núcleo a cada fila y en cada fila hay el mismo número de píxeles, el problema sería balanceado, por lo que:

$$T_{mandel_{multihilo}}(xres, yres, maxiter, p) = \frac{xres \cdot yres \cdot (6 \cdot maxiter + 5)}{p} \cdot t_c$$

Function 5: Complejidad mandel, multihilo

Complejidad espacial.

La complejidad espacial del problema se reduce al tamaño de la fractal, esto es:

$$xres \cdot yres \cdot sizeof(double) = xres \cdot yres \cdot 8\,B$$

Cálculo de la media

Descripción.

El cálculo de la media es notablemente más simple matemáticamente que el de la fractal en sí. Dado el vector de datos *A* de tamaño *n*, la media se define como:

$$\mu_A = \frac{\sum_{i=0}^{n} A_i}{n}$$

Function 6: Cálculo media

O lo que es lo mismo algorítmicamente:

```
int i;
double suma = 0.0;

for (i = 0 ; I < n ; i++)
{
    suma+=*(A+i);      // 1 FLOP
}

return suma/n;  // 1 FLOP
```

Complejidad computacional.

Entonces, la complejidad se define por:

$$T(n) = (n+1)\cdot t_c$$

Function 7: Complejidad media

Respecto al equivalente paralelo, es donde se encuentran la mayoría de los dilemas. El problema del cálculo de la media contiene una reducción suma+=*(A+i), una tarea compleja de paralelizar. En el ámbito de paralelización en CPU con OpenMP, se puede resolver a incluir

#pragma omp parallel for reduction(+:suma)

Mientras que en Cuda, se pueden utilizar librerías como CUBLAS para realizar el sumatorio (cublasDasum()).

Complejidad espacial.

La complejidad espacial es más simple de calcular ya que los únicos datos en el cálculo de la media son el vector original.

$$xres \cdot yres \cdot sizeof(double) = xres \cdot yres \cdot 8\,B$$

Binarización

Descripción.

El cálculo de la fractal binarizada a partir de la fractal original no tiene más misterio que iterar por cada posición de la fractal original y revisar si el pixel tiene un valor mayor o menor a la media.

```
int i;

for (i = 0 ; i < n ; i++)
{
    if (*(A+i) < media)     *(A+i) = 0;
    else                    *(A+i) = 255;
}

return;
```

Complejidad computacional.

```
int i;

for (i = 0 ; i < n ; i++)
{
    if (*(A+i) < media)     *(A+i) = 0;        // 1 FLOP
    else                    *(A+i) = 255;      // 1 FLOP
}

return;
```

Esto es:

$$T(n)=n\cdot t_c$$

Function 8: Complejidad binarización

Ya que cada pixel es independiente de los demás, el problema está equilibrado y es balanceado, por lo tanto:

$$T(n,p)=\frac{n}{p}\cdot t_c$$

Function 9: Complejidad binarización, multihilo

Complejidad espacial.

Respecto a la complejidad espacial, al igual que en la media sería únicamente el tamaño del vector.

$$xres\cdot yres\cdot sizeof(double)=xres\cdot yres\cdot 8\,B$$

A menos que el algoritmo realice una copia para preservar la fractal original, en cuyo caso, la complejidad espacial sería el doble.

CPU

Código fuente

Mandel

```
void mandel(double xmin, double ymin, double xmax, double ymax, int maxiter, int
xres, int yres, double* A)
{
    double dx, dy, u = 0, v = 0, u_old = 0, paso_x, paso_y;
    dx = (xmax-xmin)/xres;
    dy = (ymax-ymin)/yres;
    int i = 0, j = 0, k = 0;
    #pragma omp parallel private(i, j, u, v, k, u_old, paso_x, paso_y)
    #pragma omp single
    for (i = 0 ; i < xres ; i++)
          for (j = 0 ; j < yres ; j++)
                #pragma omp task
                {
                      paso_x = i*dx+xmin;
                      paso_y = j*dy+ymin;
                      u = 0;
                      v = 0;
                      k = 1;
                      while (k < maxiter && (u*u+v*v) < 4)
                      {
                            u_old = u;
                            u = u_old*u_old - v*v + paso_x;
                            v = 2*u_old*v + paso_y;
                            k = k + 1;
                      }
                      if (k ≥ maxiter) *(A+j*xres+i) = 0;
                      else             *(A+j*xres+i) = k;
                }
    return;
}
```

En este caso se ha utilizado como estrategia la creación de una task para cada pixel. En el caso de omitir las tasks, hubiese sido posible también realizar el cálculo dentro del propio bucle for, y en este caso sería interesante utilizar algún scheduling, en particular el schedule(dynamic).

La razón para elegir schedule dynamic (o guided que sería otra buena opción), es por lo dicho en el estudio teórico de esta función: "*Debido a la condición (u*u+v*v) < 4, el número de iteraciones puede no ser el mismo para todos los píxeles*". Entonces nos interesa un scheduling que asigne una carga a los hilos a medida que estos finalizan con la carga previa. Encontramos dos opciones:

- dynamic: Cada hilo solicita una carga nueva en cuanto finaliza su carga anterior.
- guided: Cada hilo solicita una carga nueva en cuanto finaliza su carga anterior, la carga nueva decrece exponencialmente.

Si bien ambas podrían ser una buena solución al problema, guided funcionaría mejor si la carga computacional se viese incrementada a medida que aumenta el número de iteración. Como no es el caso, sería más lógico seleccionar dynamic.

promedio

```
double promedio(int xres, int yres, double* A){
    int i;
    double s;
    s = 0;
    #pragma omp parallel for reduction(+:s)
    for (i = 0 ; i < xres*yres ; i++)
          s+=*(A+i);
   return s/(xres*yres);
}
```

Utilizamos #pragma omp parallel for reduction(+:s) para realizar el sumatorio.

binarizado

```
void binariza(int xres, int yres, double* A, double med){
    int i;
    #pragma omp parallel for
    for (i = 0 ; i < xres*yres ; i++)
    {
          if (*(A+i) ≥ med)*(A+i) = 255;
          else              *(A+i) = 0;
    }
    return;
}
```

Tabla

Table 2: Tiempos I3

Yres	Teórico		Empírico					
				C				
					Paralelo			
	Secuencial	Paralelo	Python	Secuencial	Tasks	Collapse	Schedule	
							Dynamic	Static
256	7.93E-03	3.97E-03	7.84E+00	1.43E-01	8.74E-02	8.22E-02	6.88E-02	8.54E-02
512	3.17E-02	1.59E-02	3.13E+01	5.22E-01	2.68E-01	2.72E-01	2.67E-01	3.01E-01
1024	1.27E-01	6.35E-02	1.25E+02	2.00E+00	1.09E+00	9.69E-01	9.32E-01	1.09E+00
2048	5.08E-01	2.54E-01	5.01E+02	7.99E+00	4.31E+00	3.81E+00	3.67E+00	4.24E+00
4096	2.03E+00	1.02E+00	N/A	3.19E+01	1.72E+01	1.53E+01	1.47E+01	1.70E+01
8192	8.12E+00	4.06E+00	N/A	1.28E+02	6.88E+01	6.09E+01	5.87E+01	6.78E+01

Table 3: Tiempos Ryzen

Yres	Teórico		Empírico					
	Secuencial	Paralelo	Python	C				
				Secuencial	Paralelo			
					Tasks	Collapse	Schedule	
							Dynamic	Static
256	8.54E-04	1.07E-04	4.96E+00	1.51E-01	8.82E-02	1.23E-02	2.24E-02	1.48E-02
512	3.42E-03	4.27E-04	1.98E+01	3.52E-01	2.58E-01	6.95E-02	8.06E-02	1.13E-01
1024	1.37E-02	1.71E-03	7.90E+01	1.41E+00	1.07E+00	2.22E-01	1.97E-01	2.62E-01
2048	5.47E-02	6.83E-03	3.16E+02	5.63E+00	4.36E+00	7.87E-01	7.41E-01	9.62E-01
4096	2.19E-01	2.73E-02	N/A	2.27E+01	1.67E+01	3.18E+00	2.95E+00	3.84E+00
8192	8.75E-01	1.09E-01	N/A	9.01E+01	6.70E+01	1.28E+01	1.19E+01	1.54E+01

Table 4: Tiempos Xeon

Yres	Teórico		Empírico					
	Secuencial	Paralelo	Python	C				
				Secuencial	Paralelo			
					Tasks	Collapse	Schedule	
							Dynamic	Static
256	2.56E-03	3.20E-04	1.14E+01	1.80E-01	2.37E-01	4.34E-02	3.07E-02	3.21E-02
512	1.02E-02	1.28E-03	4.54E+01	6.49E-01	8.87E-01	1.31E-01	1.29E-01	1.82E-01
1024	4.10E-02	5.12E-03	2.73E+02	3.90E+00	4.01E+00	3.72E-01	3.23E-01	4.19E-01
2048	1.64E-01	2.05E-02	1.09E+03	1.56E+01	1.55E+01	1.50E+00	1.29E+00	1.76E+00
4096	6.56E-01	8.20E-02	N/A	6.24E+01	5.74E+01	6.02E+00	5.16E+00	7.00E+00
8192	2.62E+00	3.28E-01	N/A	2.49E+02	2.37E+02	2.41E+01	2.07E+01	2.78E+01

Comparativas

Comprobación de datos

¿Tienen los resultados obtenidos en los 3 sistemas alguna correlación con los datos teóricos? Es probable que las métricas no coincidan exactamente, en parte debido a numerosos factores como las aproximaciones en el cálculo o las condiciones de cada máquina, pero sí que deberían de tener una similar tendencia.

Como simplificación, se debería de cumplir la siguiente relación hasta cierta medida.

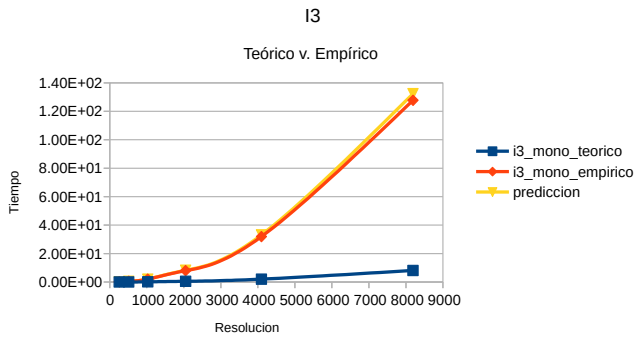
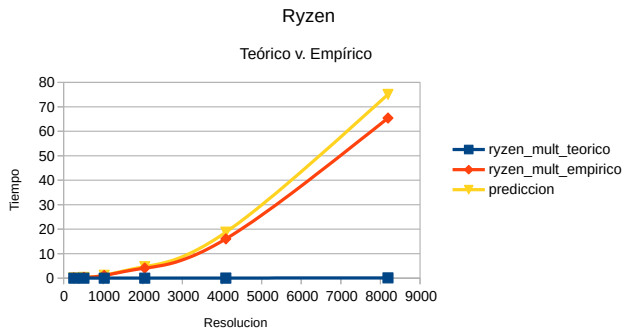
$$T_{empírico} \approx k_{cte} \cdot T_{teórico}$$

$$\approx \frac{T_{Teórico}}{\frac{t_{Empírico}}{\sum_{i=0}^{N} \frac{T_{empírico_i}}{N}}}$$

Function 10: Función de correlación

Al realizar las operaciones pertinentes, se obtienen las siguientes gráficas que muestran que la predicción (el tiempo calculado utilizando la función de correlación) es consistente para todos los equipos. La única excepción de ello podría ser el Ryzen, que parece tener un comportamiento algo más esporádico, particularmente el multihilo.

Table 5: Resultados de correlación

I3 Monohilo	Ryzen Paralelo
	

Otro método más formal para demostrar la similar tendencia de las métricas sería realizar una interpolación de los nodos para obtener una función, calcular la primera derivada de cada una y compararlas.

Ante la falta de nodos, se podría calcular la derivada mediante la siugiente definición minimizando h lo máximo posible.

$$f'(x) \approx \lim_{h \to 0} \frac{|f(x) - f(x+h)|}{|h|}$$

Function 11: f'(x)

Comparación algoritmos

Se han diseñado 4 distintos algoritmos multihilo además del monohilo (que corresponde a uno de los multihilo con OMP_NUM_THREADS=0). Se han calculado las medias de las aceleraciones de cada uno de los algoritmos frente al monohilo en el Ryzen y se han obtenido los siguientes resultados:

Table 6: Aceleraciones algoritmos CPU

Algoritmo	Descripción	Aceleración
Tasks	Para cada pixel crea una task.	1’31819888238952
Static	#pragma omp parallel for	5’32143284497037
Dynamic	#pragma omp parallel for schedule(dynamic)	6’32942825612002
Collapse	#prgama omp parallel for collapse(2)	5’95539224383319

Como era de esperar, Dynamic resulta ser el algoritmo con mejor aceleración debido a su comportamiento con la asignación de tareas a los hilos.

Tasks resulta ser el peor de todos, a penas sobrepasando el tiempo del secuencial, este comportamiento puede deberse en parte a la sobrecarga de tareas ya que uno de los hilos las crea a mayor velocidad que los demás pueden resolverlas.

Inconsistencias

Durante el estudio de los tiempos de respuesta, se encontró una notable inconsistencia. Particularmente en el Xeon, los tiempos de respuesta de los algoritmos paralelos eran superiores a aquellos de los secuenciales.

El script en cuestión que ejecutaba los algoritmos envia primero los paralelos y tras su finalización, los secuenciales. Probáblemente, debido al *caching* producido por los algoritmos paralelos los secuenciales se vean afectados por una injusta ventaja.

Para resolver esta sitaución, se ha modificado el script, de forma que ejecute los algoritmos paralelos, enviando sus tiempos de respuesta a */dev/null*, hecho esto repite la ejecución de los mismos algoritmos y finaliza con los secuenciales. Esto es, se toma una ejecución completa de los algoritmos paralelos como transitorio inicial.

Al realizar una primera iteración de las funciones, se produce el *caching* en una ejecución que no se tiene en cuenta, y tras ella, tanto la ejecución paralela como la secuencial se aprovechan la ventaja.

GPU

Un kernel debe ser llamado desde C de forma similar a la llamada de una función, pero se deben proporcionar parámetros extra que describen la cantidad de hilos a utilizar entre otras cosas. Generalmente[2]:

```
nombre_kernel <<< num_bloques, num_hilos_por_bloque[, bytes_memoria_shared,
stream ]>>> ([params]);
```

Utilización de memoria[3]

Como ya se sabe, gran parte de la diferencia entre diseñar un programa enfocado en la ejecución desde CPU y otro para GPU se encuentra en la organización de la memoria. A la hora de asignar memoria en un programa 'normal' (ejecutado en CPU) se utiliza la RAM, cuando al procesador le haga falta el dato en cuestión se cacheará y en algunos casos particulares se podrán traspasar algunos datos a memoria de paginación o *swap*.

Este sistema puede parecer complejo a simple vista debido a las variedades de memorias que posee un computador y las formas en las que interactúan entre ellas, pero a la hora de diseñar un programa, quien haga el desarrollo tiene poco que tener en cuenta. Los datos se almacenan en la memoria principal y del resto se encarga el sistema.

Al intervenir una GPU, la situación cambia notablemente, ya que el dispositivo no tiene acceso a la memoria RAM de la CPU. Para resolver la situación, una GPU tiene su propia memoria VRAM que a efectos prácticos se puede comprender como la RAM de la GPU.

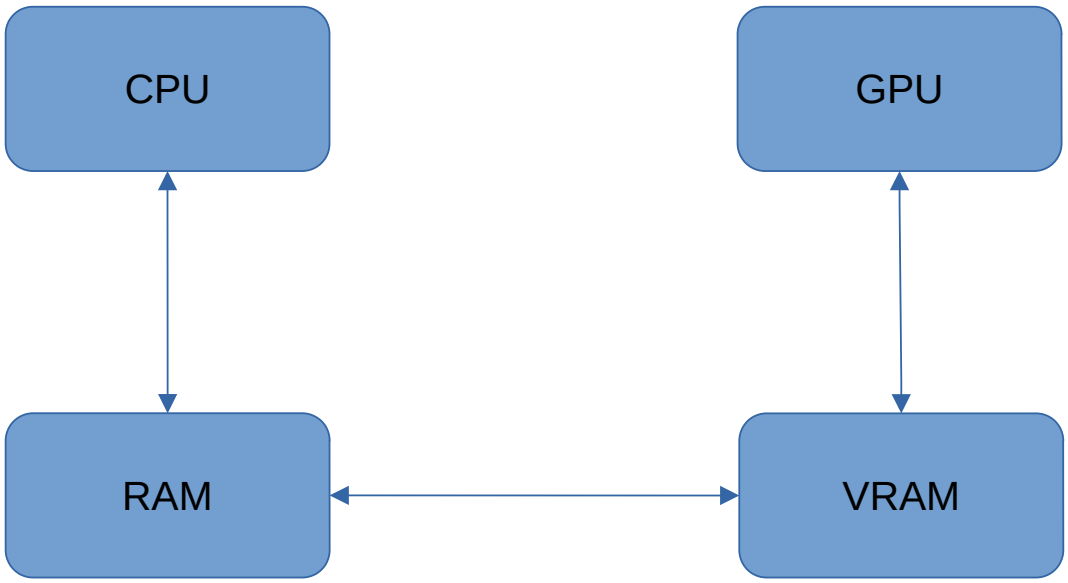


Illustration 1: Diagrama memoria GPU, 1

2 En el caso de que la organización sea superior a 1D, se podrán utilizar estructuras dim3 para facilitar la configuración.

3 Las explicacaciones y particularmente los diagramas a continuación siguen un patrón evolutivo y contienen notables simplificaciones. Más adelante algunas de las simplificaciones serán perfeccionadas para aproximarse más a la realidad.

Si buscamos que la GPU procese un dato cualsea, este se debe encontrar en la VRAM. Esto no es lo ideal, ya que hasta el momento, se almacenaba todo en la RAM sin más. De ahora en adelante, será necesario tener en cuenta dónde se almacena cada dato, e invertir tiempo en copiar información de una memoria a otra.

Existen otras dos opciones. La primera de ellas, el uso de la memoria pinned, en la que podemos proveer a la GPU de un puntero a datos que no están en VRAM (device pointer). La otra es la memoria unificada (managed) que constituye una memoria compartida entre la CPU y la GPU.

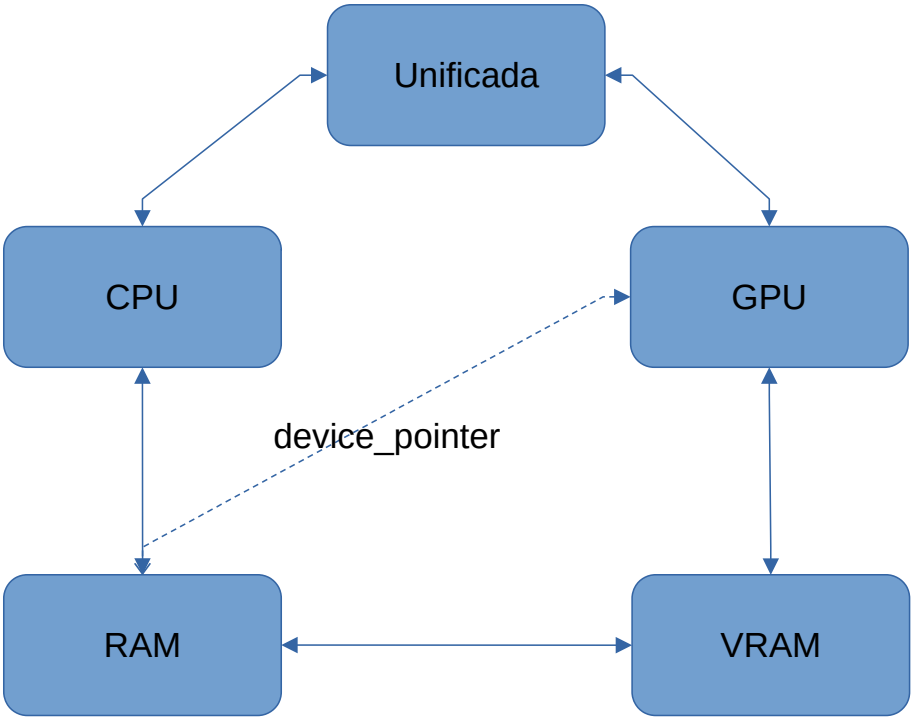


Illustration 2: Diagrama memoria GPU, 2

Utilización de hilos

La otra tarea que se debe cumplir al aprovechar la capacidad de CUDA, es la organización de los hilos. Una GPU está organizada en bloques, los bloques en warps y los warps en hilos. Concretamente, un bloque puede tener un máximo de 1024 hilos, y un warp 32.

Esta capacidad de organizar los hilos en distintos grupos nos permitirá crear algoritmos que funcionen en 1D, 2D o 3D.

Organización 1D, organización de memoria.

Kernels 1D

A continuación, se muestra la versión más simple del programa en GPU.

Mandel

```
__global__ void kernelMandel(double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A)
{
    int   i = threadIdx.x+blockIdx.x*blockDim.x,
          j;

    if (i < xres)
    {
          for (j = 0 ; j < yres ; j++)
          {
                double      dx = (xmax-xmin)/xres,
                            dy = (ymax-ymin)/yres,
                            u = 0, v = 0, u_old = 0,
                            paso_x = i*dx+xmin,
                            paso_y = j*dy+ymin;
                int         k = 1;

                while (k < maxiter && (u*u+v*v) < 4)
                {
                      u_old = u;
                      u = u_old*u_old - v*v + paso_x;
                      v = 2*u_old*v + paso_y;
                      k = k + 1;
                }
                if (k ≥ maxiter) *(A+i+j*xres) = 0;
                else             *(A+i+j*xres) = k;
          }
    }
    return;
}
```

El algoritmo utiliza una estructura 1D para resolver el problema, esto significa que de la solución original hemos podido eliminar uno de los bucles for.

promedio

```
// NO KERNEL
extern "C" double promedioGPU(int xres, int yres, double* A, int ThpBlk)
{
    double      avg = 0, *Dev_a = NULL;
    int   size = xres*yres*sizeof(double);
    cublasHandle_t handle;

    CUDAERR(cudaMalloc((void**) &Dev_a, size));
    CUDAERR(cudaMemcpy(Dev_a, A, size, cudaMemcpyHostToDevice));

    cublasCreate(&handle);
    cublasDasum(handle, size/sizeof(double), Dev_a, 1, &avg);
    cublasDestroy(handle);

    cudaFree(Dev_a);
    return avg/size*sizeof(double);
}
```

El algoritmo para el promedio no muestra ningún kernel en particular, se utiliza la librería CUBLAS para resolver el problema, más adelante se mostrará una versión hecha a mano con CUDA.

binarizado

```
__global__ void kernelBinariza(int xres, int yres, double* A, double med)
{
    int   i = threadIdx.x + blockIdx.x * blockDim.x;

    if (i < xres*yres)
    {
          if (*(A+i) > med) *(A+i) = 255;
          else              *(A+i) = 0;
    }
    return;
}
```

El algoritmo para el binarizado es muy simple, utiliza una única dimensión para eliminar el único bucle for que había en el equivalente secuencial.

Llamadas a Kernels 1D

Como se ha explicado en el apartado Utilización de memoria, será necesario organizar la memoria de alguna de las formas descritas antes de la llamada al kernel. A continuación, se muestran varias organizaciones de memoria en la llamada al kernel Mandel.

Device

La matriz original (A) residen en RAM, creamos una estructura igual (Dev_a) que almacenamos en la GPU (cudaMalloc()). En este caso particular A no tiene ningún dato de entrada, sino que es una estructura de destino, por lo que no es necesario copiar ningún dato de A a Dev_a para llamar al kernel.

Tras la ejecución del kernel copiamos los datos de Dev_a a A utilizando cudaMemcpyDeviceToHost y finalizamos con cudaFree(Dev_a) para liberar la memoria utilizada.

```
extern "C" void mandelGPU(double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A, int ThpBlk)
{
    double      *Dev_a = NULL;
    int         size = xres*yres*sizeof(double),
                n_blks = (int) (yres/ThpBlk)+1;

    CUDAERR(cudaMalloc((void **)&Dev_a, size));

    kernelMandel <<<n_blks, ThpBlk>>> (xmin, ymin, xmax,  ymax, maxiter, xres, yres,
Dev_a);

    cudaDeviceSynchronize();
    CHECKLASTERR();

    CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
    cudaFree(Dev_a);
}
```

Unificada

En esta otra versión utilizamos la memoria compartida para almacenar los datos (cudaMallocManaged()). Tras la ejecución del kernel traemos los datos de Dev_a a A para preservarlos en RAM.

```
extern "C" void managed_mandelGPU(double xmin, double ymin, double xmax, double
ymax, int maxiter, int xres, int yres, double* A, int ThpBlk)
{
    double      *Dev_a = NULL;
    int   size = xres*yres*sizeof(double),
          n_blks = (int) (yres/ThpBlk)+1;

    CUDAERR(cudaMallocManaged((void**)&Dev_a, size, cudaMemAttachGlobal));

    kernelMandel <<<n_blks, ThpBlk>>> (xmin, ymin, xmax,  ymax, maxiter, xres, yres,
Dev_a);

    cudaDeviceSynchronize();
    CHECKLASTERR();

    CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
    cudaFree(Dev_a);
}
```

Pinned

Finalmente, utilizamos la memoria pinned. Creamos Dev_a (cudaHostAlloc) y solicitamos el puntero. Tras llamar al kernel copiamos los datos de Dev_a a A.

```
extern "C" void pinned_mandelGPU(double xmin, double ymin, double xmax, double ymax,
int maxiter, int xres, int yres, double* A, int ThpBlk)
{
    double      *Dev_a = NULL, *ptr_Dev_a = NULL;
    int   size = xres*yres*sizeof(double),
          n_blks = (int) (yres/ThpBlk)+1;

    CUDAERR(cudaHostAlloc((void**)&Dev_a, size, cudaHostAllocMapped));
    CUDAERR(cudaHostGetDevicePointer((void**) &ptr_Dev_a, (void*)Dev_a, 0));

    kernelMandel <<<n_blks, ThpBlk>>> (xmin, ymin, xmax,  ymax, maxiter, xres, yres,
ptr_Dev_a);

    cudaDeviceSynchronize();
    CHECKLASTERR();

    CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
    cudaFreeHost(Dev_a);
}
```

Si la memoria pinned consiste de un puntero al Host de los datos, ¿no se podría solicitar un puntero a la estructura A para ahorrarnos la creación de Dev_a?

Parecería no haber ningún problema, A se encuentra en RAM, podríamos utilizar:

```
cudaHostGetDevicePointer((void**) &ptr_dev_a, (void*)A, 0));
```

para almacenar en ptr_Dev_a un puntero a A. De esta forma, ahorramos por completo el uso de Dev_a, ya que el kernel, al utilizar ptr_dev_a para hacer las modificaciones, estaría accediendo a la memoria Host. Incluso, ahorraríamos la copia de datos tras el kernel porque los datos ya se encontrarían en A.

Resulta ser que este no es el caso[4], ya que la memoria pinned no es exactamente la misma RAM a la que accede la CPU. El diagrama precisamente se asimilaría más al siguiente:

Entonces es necesario crear una estructura (Dev_a) en la memoria pinned, solicitar un ptr_dev_a a dicha estructura y finalmente volver a copiar Dev_a a A desde la pinned a la RAM.

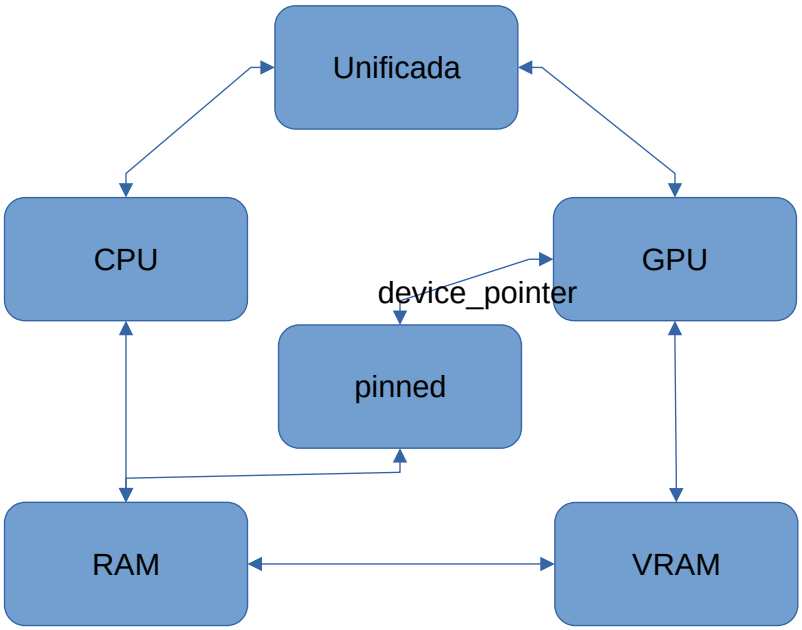


Illustration 3: Diagrama memoria GPU, 3

Dicho esto, parecería que el único beneficio de la pinned era ahorrarnos copias de una memoria a otra, pero ya que no es el caso, ¿en qué caso usaríamos pinned en lugar de otra?.

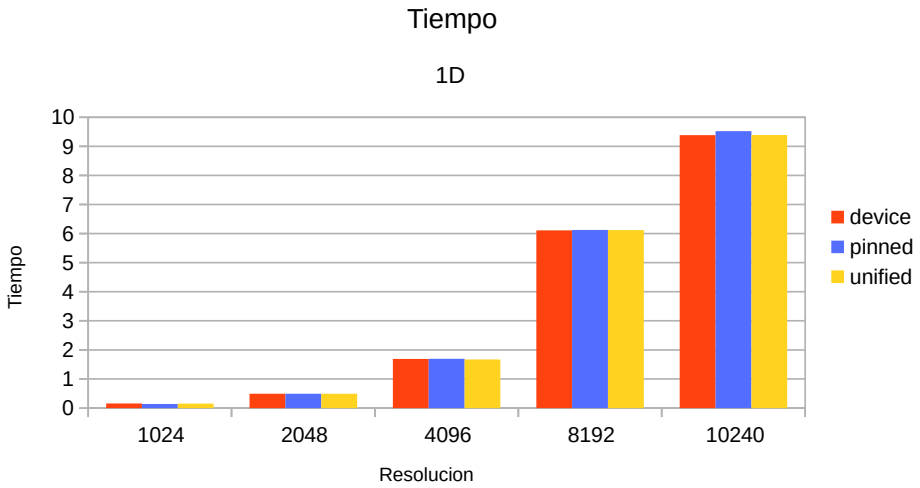
El dilema se reduce a la capacidad de los sistemas. Es común encontrar GPUs con entre 4-8 GB de VRAM, algunas específicas para HPC pueden tener hasta 24, 40 o incluso 80GB en las gamas más altas, mientras que los equipos hoy en día suelen tener entre 8 y 16 o hasta 32 y 64 GB de RAM. Aunque la memoria pinned no sea accesible por la CPU, sigue siendo RAM, por lo que en equipos que no estén diseñados para HPC, tendremos acceso a mucha más pinned que VRAM.

4 Véase https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

Comparativas

Para las siguientes comparativas, se ha utilizado la siguiente GPU: msi 1050Ti ‘Aero’ ITX.

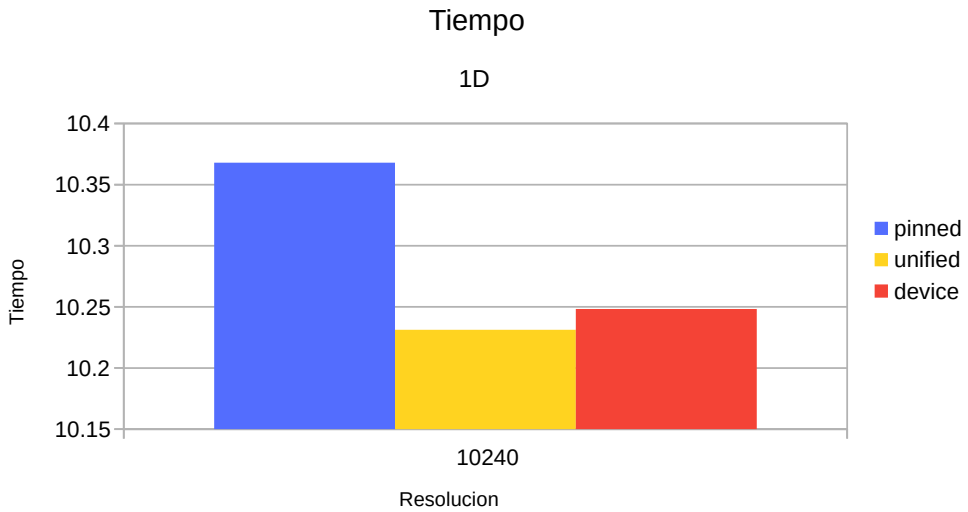
A pesar de que a nivel de desarrollo existe una notable diferencia entre las formas de organizar la memoria en GPU, a la hora de comparar las distintas opciones empíricamente no son tan diferenciables entre ellas.



Graph 1: Tiempo ejecución, memorias

Con esto, se puede concluir que el problema evaluado no se encuentra limitado debido a las copias de datos. Pues es muy probable que un problema distinto que dependa de una mayor cantidad de información encuentre un mayor valor en la organización de su memoria.

Si buscásemos el mejor de todos aunque fuese por conocerlo en el ámbito de este problema, podríamos centrarnos particularmente en una de las resoluciones. Si observamos la ejecución con resolución 10240, observamos los siguientes resultados que denotan que la memoria unificada es la mejor.



Graph 2: Tiempo ejecución, memorias (res=10240)

Organización 2D, organización de hilos

La otra herramienta que se puede modificar es la forma en la que los hilos se organizan. Resumidamente, se reduce a organización en 1D, 2D o 3D. Hasta el momento se han visto las variaciones 1D de los algoritmos. A continuación se muestran tanto los kernel 2D como sus respectivas llamadas.

Si bien nos centraremos en las diferencias entre las organizaciones de los hilos, no se debe olvidar la organización de la memoria vista en Organización 1D, organización de memoria pues sigue siendo una herramienta que coexiste con la organización de hilos.

Kernels 2D

Mandel

```
__global__ void kernelMandel2D(double xmin, double ymin, double xmax, double ymax,
int maxiter, int xres, int yres, double* A)
{
    int   i = threadIdx.x+blockIdx.x*blockDim.x,
          j = threadIdx.y+blockIdx.y*blockDim.y;

    if (i < xres && j < yres)
    {
          double      dx = (xmax-xmin)/xres,
                      dy = (ymax-ymin)/yres,
                      u = 0, v = 0, u_old = 0,
                      paso_x = i*dx+xmin,
                      paso_y = j*dy+ymin;
          int         k = 1;

          while (k < maxiter && (u*u+v*v) < 4)
          {
                u_old = u;
                u = u_old*u_old - v*v + paso_x;
                v = 2*u_old*v + paso_y;
                k = k + 1;
          }
          if (k ≥ maxiter) *(A+i+j*xres) = 0;
          else             *(A+i+j*xres) = k;
    }
    return;
}
```

El algoritmo utiliza una estructura 2D para resolver el problema, esto significa que de la solución original hemos podido eliminar ambos bucles for.

Promedio

A diferencia de los demás kernels, el promedio utilizando CUBLAS no tiene versión 2D, ya que no se llama a ningún kernel.

binarizado

```
__global__ void kernelBinariza2D(int xres, int yres, double* A, double med)
{
    int   i = threadIdx.x + blockIdx.x * blockDim.x,
          j = threadIdx.y + blockIdx.y * blockDim.y;

    if (i < xres && j < yres)
    {
          if (*(A+i+j*xres) > med)      *(A+i+j*xres) = 255;
          else                          *(A+i+j*xres) = 0;
    }
    return;
}
```

Llamadas a Kernels 2D

En este caso, no se estudiarán particularmente las diferentes configuraciones de memoria ya que se han visto en el apartado Llamadas a Kernels 1D y en este caso serían las mismas. Sin embargo, todas ellas han sido implementadas y se encuentran adjuntas en el Anexo 1, funciones GPU. En su lugar, se describirá la estructura que utilizan las GPUs para organizar sus hilos.

Como ya se adelantó en Utilización de hilos, una GPU consta de bloques que contienen hilos agrupados en warps, de una forma similar a la descrita en el siguiente diagrama:

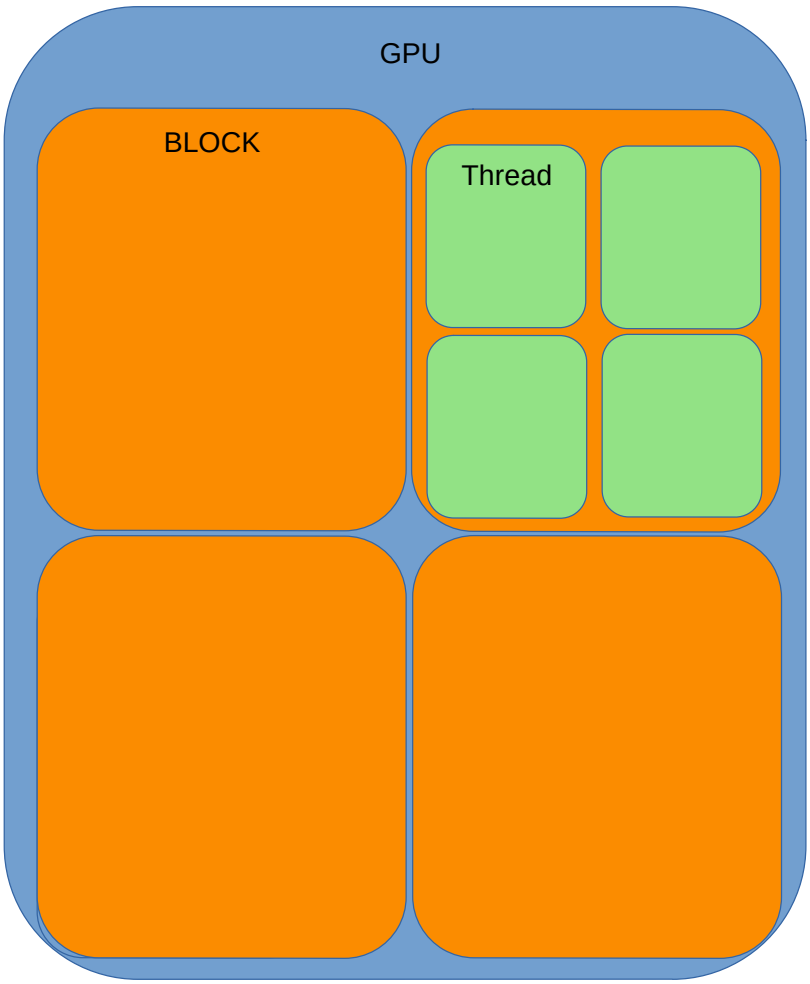


Illustration 4: Organización hilos GPU

```
extern "C" void managed_mandelGPU2D(double xmin, double ymin, double xmax, double
ymax, int maxiter, int xres, int yres, double* A, int ThpBlk)
{
    double      *Dev_a = NULL;
    int   size = xres*yres*sizeof(double);
    dim3  dim_block   (ThpBlk, ThpBlk),
          dim_grid    ((xres+dim_block.x-1)/dim_block.x,
(yres+dim_block.y-1)/dim_block.y);

    CUDAERR(cudaMallocManaged((void**)&Dev_a, size, cudaMemAttachGlobal));

    kernelMandel2D <<<dim_grid, dim_block>>> (xmin, ymin, xmax,   ymax, maxiter,
xres, yres, Dev_a);

    cudaDeviceSynchronize();
    CHECKLASTERR();

    CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
    cudaFree(Dev_a);
}
```

Préstese particular atención a las líneas marcadas de rojo. Hasta el momento se definía un int para el número de bloques a utilizar y otro para el número de hilos por cada bloque, estos datos se pasaban en la configuración en la primera y segunda posición respectivamente.

Ahora, que es posible tener 2 dimensiones, deberemos pasar el doble de atributos de configuración, pues habrá uno para el eje x y otro para el y. El tipo de dato dim3 permite asociar 3 datos para pasarlos a la configuración más cómodamente.

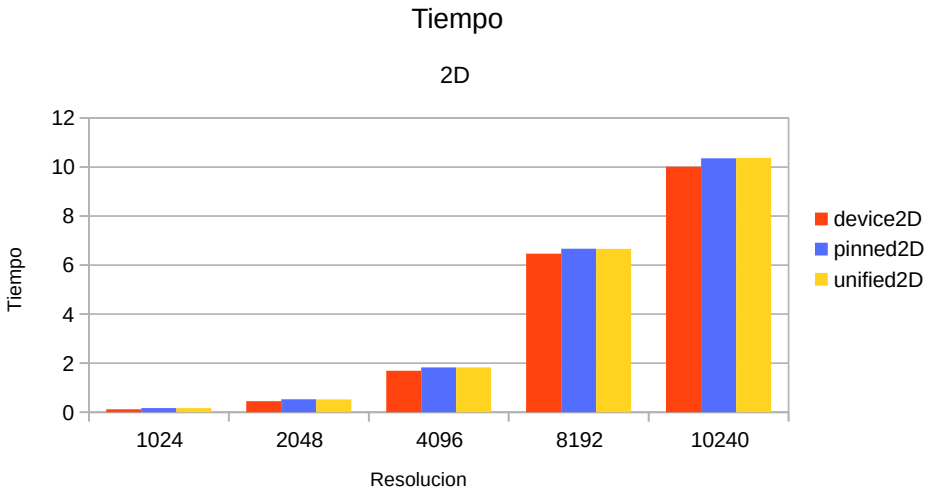
En este caso dim_block será el tamaño de los bloques o el número de hilos en cada bloque. Se puede ver que independientemente de la dimensión, serán siempre ThpBlk. Por otro lado, dim_grid será el número de bloques en cada dimensión, en la x tendremos (xres+dim_block.x-1)/dim_block.x mientras que en la y serán (yres+dim_block.y-1)/dim_block.y.

Como ya se ha podido deducir por la declaración de dim_grid, para acceder a los distintos datos de un dim3, utilizamos dim3.dimensión. Muy similar es el comportamiento de threadIdx que utilizamos para obtener el id de cada hilo, por lo que se puede concluir que threadIdx es también un dim3.

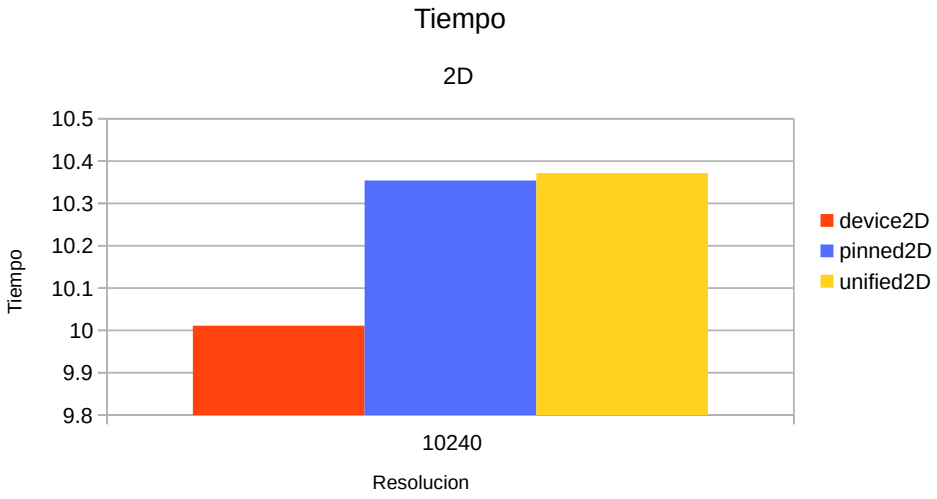
Comparativas 1D v. 2D

Para las siguientes comparativas, se ha utilizado la siguiente GPU: msi 1050Ti ‘Aero’ ITX.

Podríamos esperar que al igual que en 1D, la memoria unificada sea la que mejor rendimiento muestre. Sin embargo, resulta ser todo lo contrario. La memoria unificada es la que peor resultados muestra mientras que el acercamiento más básico de copiar los datos a VRAM aparenta ser el más interesante.

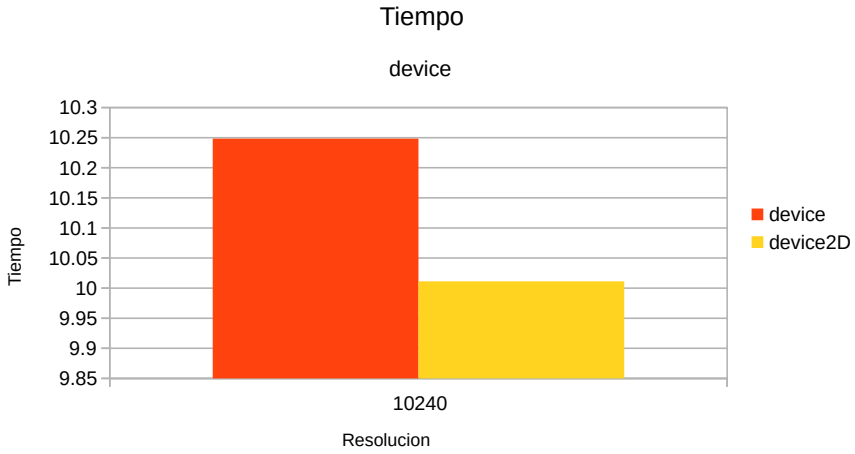


Graph 3: Tiempo ejecución, device 2D



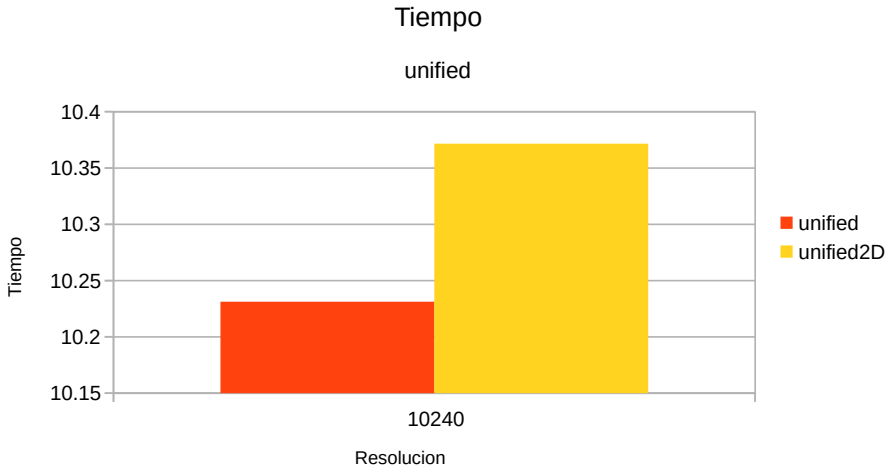
Graph 4: Tiempo ejecución, 2D (res=10240)

Comparamos el rendimiento de los algoritmos organizados en 1D frente a sus respectivos 2D.



Graph 5: 1D v. 2D, device (res=10240)

Tanto si se almacenan los datos en VRAM como en la pinned, el rendimiento parece ser mejorado por la organización 2D.



Graph 6: 1D v. 2D, unified (res=10240)

Sin embargo, en la memoria unificada el efecto es el opuesto, el rendimiento era mejor con los hilos organizados en 1D.

De acuerdo con nvidia, el objetivo de la organización en 1D/2D/3D es facilitar la indexación para estructuras que frecuentemente se procesarán en la GPU (vectores, matrices y volúmenes)[5]. Entonces, teóricamente no debería haber ninguna diferencia entre las ejecuciones de un programa organizado en 1D, 2D y 3D.

Si se revisan los datos obtenidos con mayor atención, en particular calculando las aceleraciones de los algoritmos 2D frente a sus correspondientes 1D, obtenemos los siguientes resultados.

$$\begin{aligned} Accel_{\frac{A}{B}} &= \frac{Productividad_A}{Productividad_B} \\ &= \frac{\frac{1}{T_A}}{\frac{1}{T_B}} \\ &= \frac{T_B}{T_A} \end{aligned}$$

Organización Memoria	Aceleración 2D/1D
Device	1’02368094182374
Pinned	1’00130765993193
Unified	0’986469312524219

Como se puede ver, a pesar de existir una aceleración, es bastante pequeña, casi atribuible al hecho de que se trata de distintas ejecuciones y la variación en el tiempo se encuentra dentro de un margen de error.

Es probable incluso, que el compilador *nvcc* convierta las tareas 2D y 3D en 1D de forma que a la hora de la ejecución, el código sea casi el mismo en los 3 casos.

5 Véase: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Otros algoritmos

Promedio

A continuación, estudiaremos algunas variantes de la función utilizada para el cálculo del promedio. Como ya se mencionó en Cálculo de la media, matemáticamente la función se reduce a:

$$\frac{\sum_{i=0}^{N} t_i}{N}$$

Function 12: Promedio

Computacionalmente, el grueso del problema se encuentra en el sumatorio, pues se trata de una reducción de un array.

La solución presentada inicialmente se basaba completamente en el uso de la librería CUBLAS para el cálculo del sumatorio. Otra opción más interesante, sería diseñar desde cero un algoritmo que complete el problema sin necesidad de la intervención de otras librerías.

Divide y vencerás

Un acercamiento clásico y al mismo tiempo inteligente sería aplicar *Divide y Vencerás* para resolver la reducción. La solución consiste en dividir el problema en trozos iguales (bloques), resolver el problema en cada trozo y continuar resolviendo el mismo problema con las soluciones de los trozos hasta alcanzar el final.

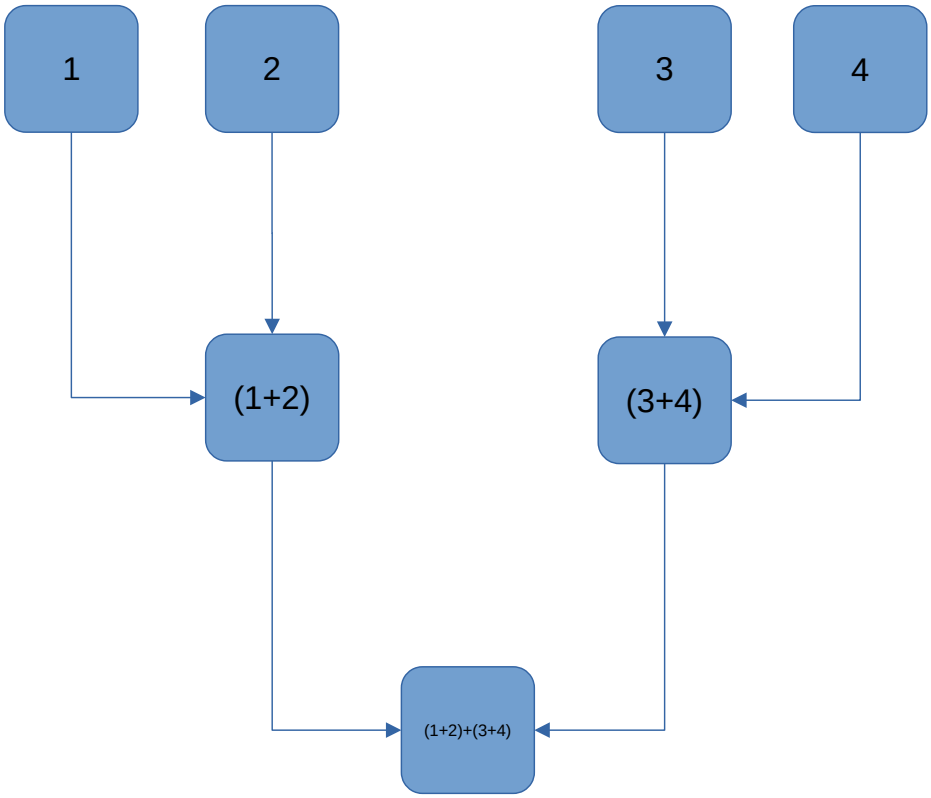


Illustration 5: Divide y Vencerás (DyV)

```
extern "C" double promedioGPUSum(int xres, int yres, double* A, int ThpBlk)
{
    double      *avg = NULL,
                *Dev_blks = NULL,
                *Dev_avg = NULL,
                *Dev_a = NULL;

    int   size = xres*yres*sizeof(double),
          n_blks = (xres*yres+ThpBlk-1)/ThpBlk;

    avg = (double*) malloc (sizeof(double));

    CUDAERR(cudaMalloc((void**) &Dev_blks, n_blks*sizeof(double)));    // midway
    CUDAERR(cudaMalloc((void**) &Dev_avg, sizeof(double)));      // dst
    CUDAERR(cudaMalloc((void**) &Dev_a, size));      // src data

    CUDAERR(cudaMemcpy(Dev_a, A, size, cudaMemcpyHostToDevice));

    sum_blks  <<<  n_blks,  ThpBlk,  ThpBlk*sizeof(double)  >>>  (xres*yres,  Dev_a,
Dev_blks);
    sum <<< 1, 1024, 1024*sizeof(double) >>> (Dev_blks, Dev_avg, n_blks);

    CHECKLASTERR();
    CUDAERR(cudaMemcpy(avg, Dev_avg, sizeof(double), cudaMemcpyDeviceToHost));

    cudaFree(Dev_blks);
    cudaFree(Dev_a);
    cudaFree(Dev_avg);

    return *avg/size*sizeof(double);
}
```

Nótese como la función de C llama a dos kernels para asegurar la sincronización.

Atomic

Otra opción sería utilizar *Atomic*, funciones que realizan operaciones muy simples en un sólo instante, de forma que a pesar de ser bloqueante, no dejan al sistema detenido durante un tiempo importante.

Se trata de un código muy simple, pero el rendimiento dejará que esperar ya que gran parte de la función ha sido 'serializada' por *atomicAdd*.

```
__global__ void sum_atomic (int size, double* data, double* dst)
{
    int   i = threadIdx.x + blockIdx.x * blockDim.x;

    if (i < size)
          atomicAdd(dst, *(data+i));

    return;
}

// (...)

extern "C" double promedioGPUAtomic(int xres, int yres, double* A, int ThpBlk)
{
    double      *avg = (double*) malloc (sizeof(double));,
                *Dev_a = NULL,
                *Dev_avg = NULL;
    int         size = xres*yres*sizeof(double),
                n_blks = (xres*yres+ThpBlk-1)/ThpBlk;

    CUDAERR(cudaMalloc((void**) &Dev_avg, sizeof(double)));      // dst
    CUDAERR(cudaMalloc((void**) &Dev_a, size));                  // src data
    CUDAERR(cudaMemcpy(Dev_a, A, size, cudaMemcpyHostToDevice));

    sum_atomic <<< n_blks, ThpBlk >>> (xres*yres, Dev_a, Dev_avg);

    CHECKLASTERR();
    CUDAERR(cudaMemcpy(avg, Dev_avg, sizeof(double), cudaMemcpyDeviceToHost));

    cudaFree(Dev_a);
    cudaFree(Dev_avg);

    return *avg/size*sizeof(double);
}
```

Es probable que con el *Makefile* proporcionado originalmente no se compile adecuadamente este kernel. A pesar de que *double atomicAdd(double* address, double val)* se encuentra en la página de nvidia[6], sólo se puede utilizar en hardware que soporte SM_6x GPUs[7][8]. Esto es, cualquier GPU con CudaCapability superior a 6.x puede utilizar la función, pero deberá añadir los flags *--gpu-architecture=compute_61 –gpu-code=sm_61* a su *Makefile*[9]. Por ello, este kernel y su correspondiente función en C se encuentran comentados en el código fuente adjunto a este documento.

Otra opción para ejecutar una solución con *Atomic* sin modificar el *Makefile* sería utilizar la versión *unsigned long long int atomicAdd(unsigned long long int*, unsigned long long int);*.

6 Véase: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicadd

7 La GPU 1660Ti no parece soportar esta característica.

8 Origen: https://forums.developer.nvidia.com/t/why-does-atomicadd-not-work-with-doubles-as-input/56429/2

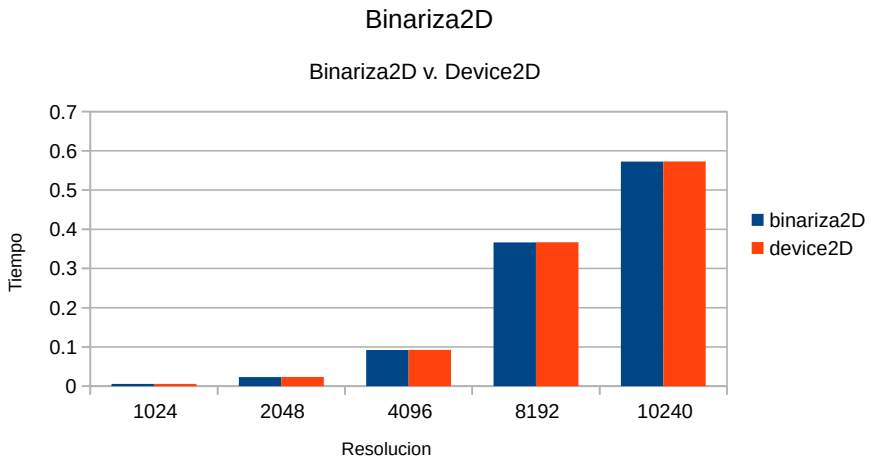
9 Véase Anexo 3, Makefile

Binarizado 2D

La función del binarizado se puede ver como un caso particular en el sentido que sólo poseía un bucle for debido a la organización de la matriz (en forma de estructura 1D), por lo que inicialmente parecería que con la organización 2D no sería posible debido a la falta de un segundo bucle for.

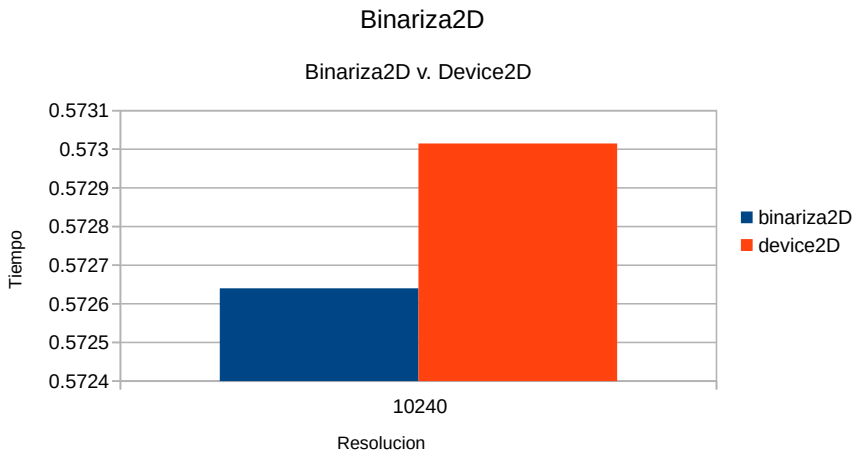
Sin embargo, esto es posible ya que también se puede iterar por la estructura como si fuese bidimensional. Pudiendo hacer lo mismo utilizando la organización 2D.

Si realizamos una comparación para cada resolución entre el binarizado 2D y su correspondiente de la librería CUBLAS, obtenemos la siguiente gráfica:



Graph 7: Binarizado 2D

Como se puede ver, ambos algoritmos van a la par, siendo necesario utilizar una gráfica menos general para poder apreciar una diferencia. Observando la medición realizada en 10240:



Graph 8: Binarizado 2D (res=10240)

Obviamente existe una diferencia entre ambos, pero es muy pequeña como para atribuirla al cambio de 1D a 2D, por lo que se refuerza lo sugerido en Comparativas 1D v. 2D.

Maximización del paralelismo

Hasta el momento se han observado las mejoras obtenidas al aprovechar el paralelismo en la CPU y GPU como dos herramientas independientes. Esto es, en el apartado CPU, la CPU computaba todo el problema y lo análogo para GPU.

Algoritmo heterogéneo

Sin embargo, ¿no es posible que ambas trabajen en conjunto, ya que pertenecen al mismo sistema?

Si asignamos una parte del cálculo de mandel a la GPU, la parte restante a la CPU y ambas trabajan sobre la misma estructura en memoria deberíamos obtener el resultado final, pero utilizando ahora sí el 100% (aproximadamente) de la capacidad del sistema.

¿Qué organización de memoria deberíamos escoger para resolver este problema? La opción que menos modifica los algoritmos ya creados sería utilizar la memoria compartida. De esta forma, podremos trabajar en la misma estructura sin preocupaciones de quién accede a qué. Si siguiésemos cualquier otra organización, nos veríamos obligados a copiar parte de la solución de un lado para otro, lo que puede llevar a conflictos innecesarios.

Entonces tendremos una única estructura sobre la cual la GPU calculara un porcentaje *perc* y la CPU calculará el resto $(1-perc)$. Para ello, reducimos el número de bloques a utilizar en la GPU, para que en lugar de utilizar la cantidad necesaria para calcular *yres* píxeles, calcule $yres \cdot perc$ píxeles. Una vez hecho esto, modificamos la función de OpenMP, añadiéndole un nuevo parámetro *double perc* que corresponderá al porcentaje en el que la GPU deja de trabajar o lo que es lo mismo, aquel en el que la CPU debe iniciar el cálculo. Utilizamos este porcentaje para modificar el punto de inicio de uno de los bucles *for* del algoritmo.

Respecto al código, es crucial llamar primero al kernel y después ejecutar la sección de OpenMP. Ya que si se hiciese al revés, el kernel no se lanzaría hasta que no haya finalizado la sección de CPU, serializando parcialmente el trabajo.

```
extern "C" void mandel_omp (double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A, double perc)
{
    double dx, dy, u = 0, v = 0, u_old = 0, paso_x, paso_y;
    dx = (xmax-xmin)/xres;
    dy = (ymax-ymin)/yres;
    int i = 0, j = 0, k = 0;
    #pragma omp parallel for private (i, j, u, v, k, u_old, paso_x, paso_y)
schedule(dynamic)
    for (i = xres*perc ; i < xres ; i++)
          for (j = 0 ; j < yres ; j++)
          {
                paso_x = i*dx+xmin;
                paso_y = j*dy+ymin;
                u = 0;
                v = 0;
                k = 1;
                while (k < maxiter && (u*u+v*v) < 4)
                {
                      u_old = u;
                      u = u_old*u_old - v*v + paso_x;
                      v = 2*u_old*v + paso_y;
                      k = k+1;
                }
                if (k ≥ maxiter) *(A+j*xres+i) = 0;
                else             *(A+j*xres+i) = k;
          }
    return;
}

// (...)

extern "C" void mandelHetero(double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A, int ThpBlk)
{
    double      *Dev_a = NULL,
                perc_gpu = 0.9;
    int         size = xres*yres*sizeof(double),
                n_blks = (int) (yres*perc_gpu+ThpBlk-1)/ThpBlk;

    CUDAERR(cudaMallocManaged((void **)&Dev_a, size, cudaMemAttachGlobal));
    CUDAERR(cudaMemcpy(Dev_a, A, size, cudaMemcpyHostToDevice));

    kernelMandel <<<n_blks, ThpBlk>>> (xmin, ymin, xmax, ymax, maxiter, xres, yres,
Dev_a);
    mandel_omp(xmin, ymin, xmax, ymax, maxiter, xres, yres, Dev_a, perc_gpu);

    cudaDeviceSynchronize();
    CHECKLASTERR();

    CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
    cudaFree(Dev_a);
}
```

Tras ejecutar obtenemos un error en el cálculo. En muchos casos es interesante observar la imagen resultado como punto de inicio en la depuración ya que en algunos casos a través del error en el dibujo, se puede intuir su causa. Obsérvese la imágen.

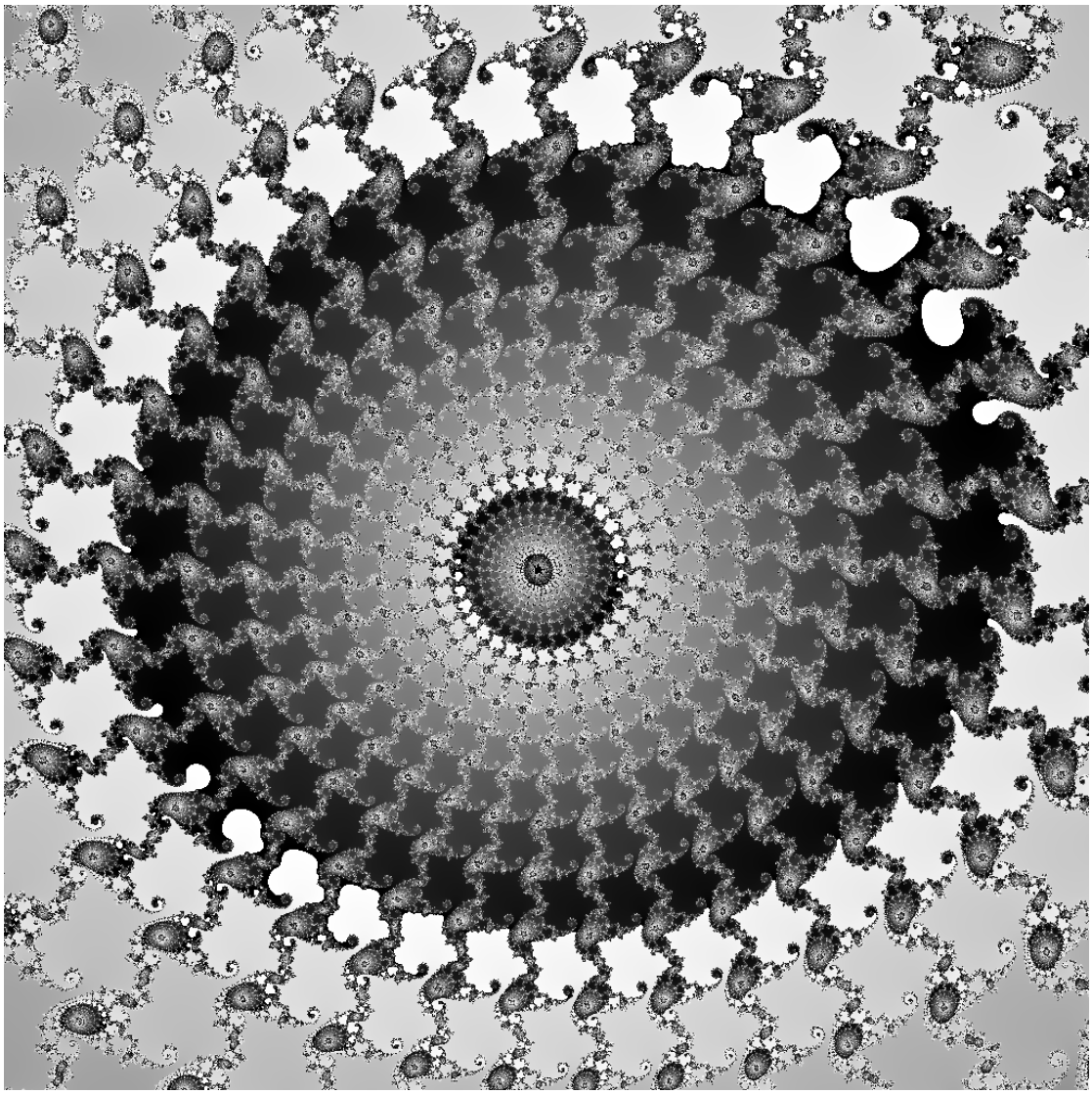


Illustration 6: Resultado cálculo heterogéneo.

No parece tener ningún error a simple vista. Implementamos un algoritmo que busque diferencias entre esta fractal y aquella que usamos de referencia. Esta función retornará una estructura posteriormente interpretable como imágen marcando los pixeles distintos en blanco.

```
def diffs (size, A, B, dst):
    for i in range (0, size, 1):
        dst[i] = 0 if A[i] == B[i] else 255;
    return dst
```

Ejecutemos una vez más el programa y observamos la imágen producida por *diffs*.

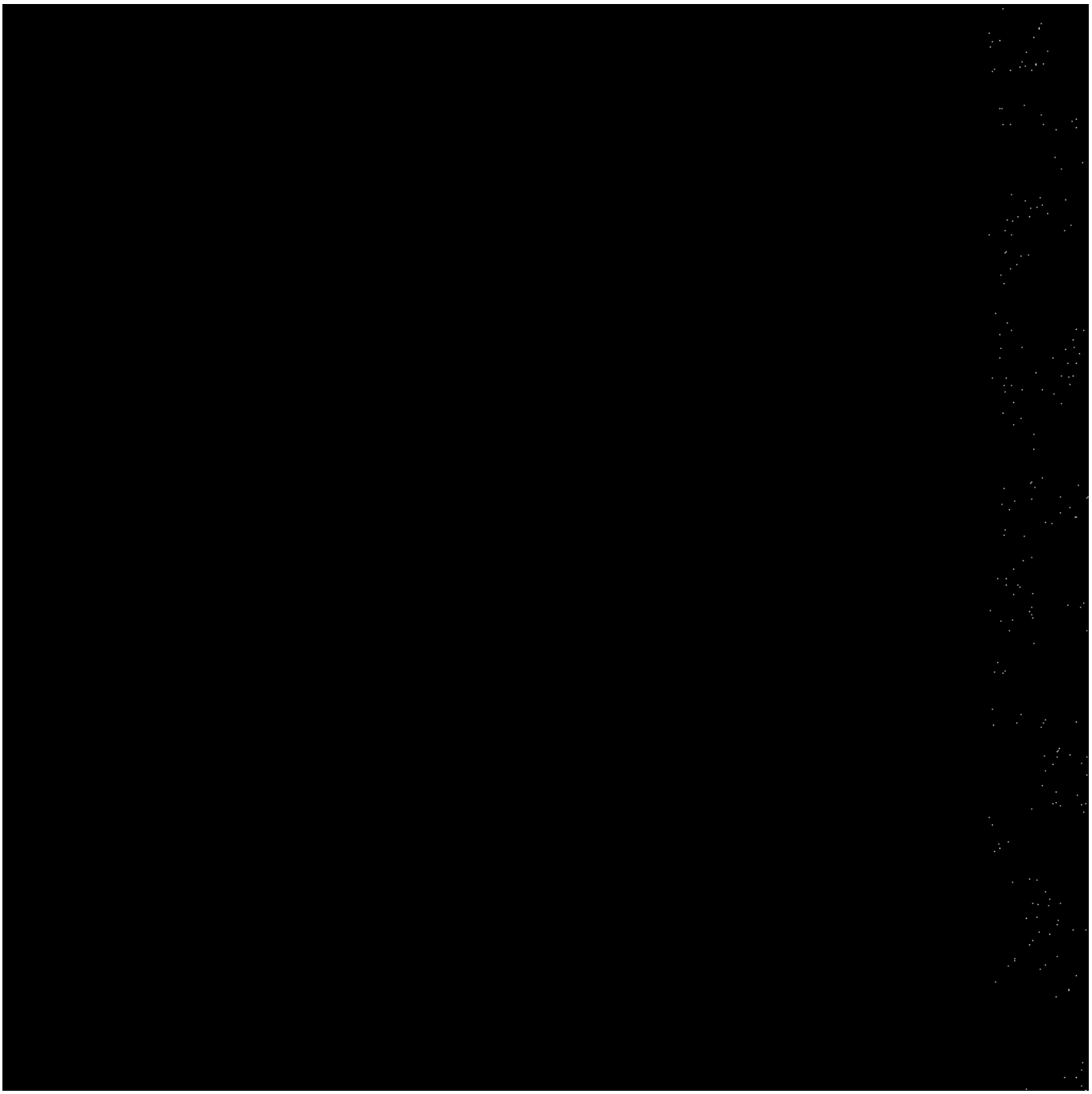


Illustration 7: diffs

Se podría suponer que la sección con errores (banda vertical derecha) es el cálculo de la CPU debido a su inferior tamaño (10%).

Esta teoría se confirma si eliminamos el cálculo realizado por la GPU del código fuente. Se obtiene sólo el trozo derecho marcado en diffs de la fractal, que tampoco presenta errores percebibles a simple vista.

¿Qué sucede? Se sabe que la imagen de referencia con la que se compara el resultado es una fractal calculado en su totalidad por la GPU. Comparemos la fractal de referencia utilizada en CPU (que había sido calculada en su totalidad con la CPU) con la fractal de referencia de GPU. Teóricamente, no debería de haber ningún error entre ellas.

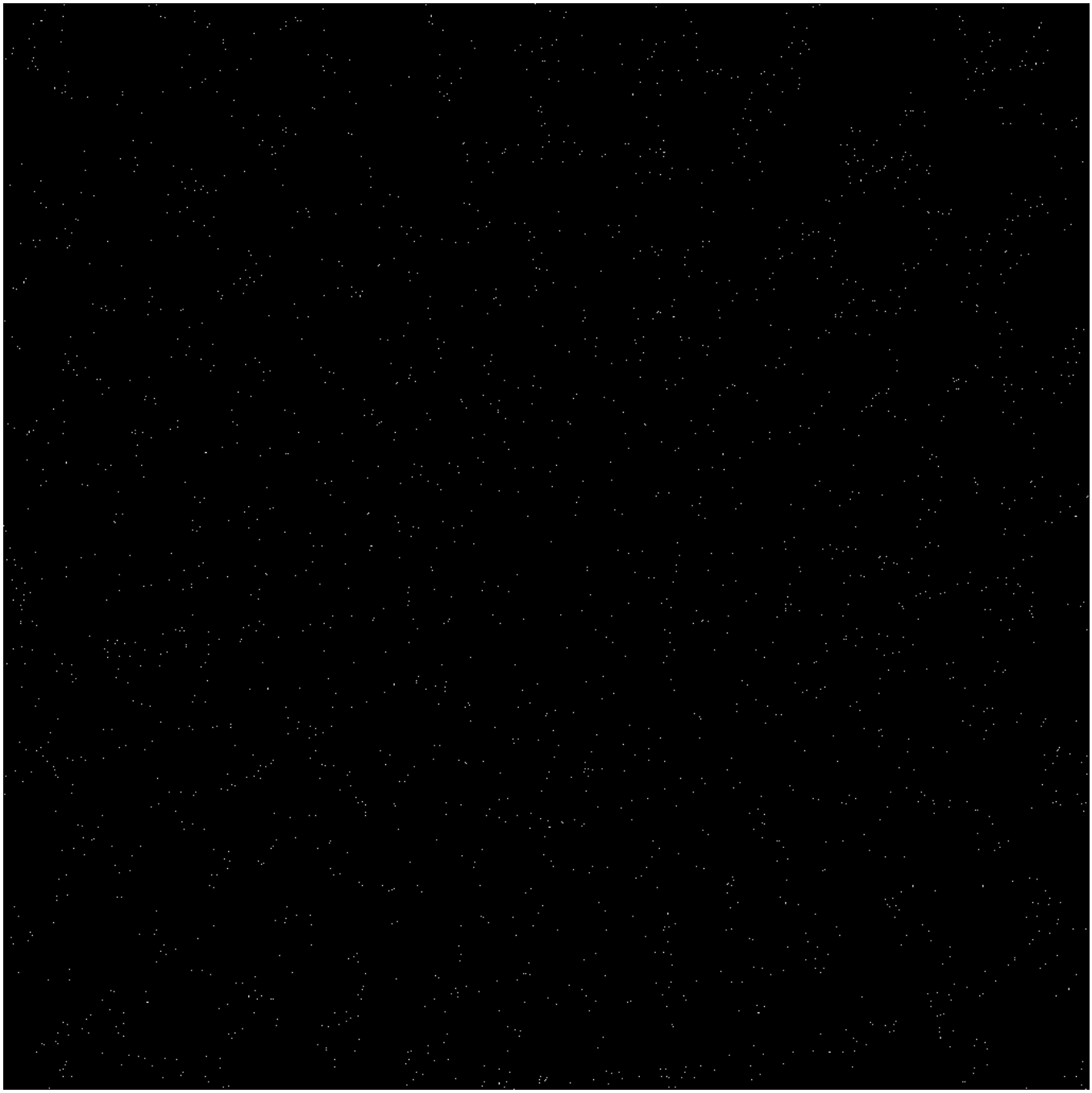


Illustration 8: diffs, CPU v. GPU

Errores esporádicos por toda la fractal. De acuerdo con varios documentos de nvidia y otros usuarios, a pesar de la existencia de estándares como el IEEE-754, aún hay inconsistencias entre los cálculos realizados por GPU y CPU.[10]

10 Véase: https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf

Optimización del algoritmo heterogéneo

El algoritmo ya funciona, a pesar de que sus resultados se encuentran acotados a un error. Pero aún se puede obtener algo más de rendimiento, ¿cuál es el porcentaje óptimo para que la CPU y la GPU tarden exactamente lo mismo?.

Para resolver esta situación, se podría aplicar cómodamente una variante del método de bisección. Dado que la función $f(x)$ que describe el tiempo de respuesta es de Clase 1 (Continua y derivable) en un intervalo dado $[0,1]$, si se conoce un intervalo $[a,b]$ que contiene un c que minimice $f(x)$, se podrá aplicar el método.

Originalmente, bisección busca un punto $x/f(x)=0$, en nuestro caso entonces, $f(x)=t_{GPU}(x)-t_{CPU}(x)$, pero ya que no obtenemos $t_{GPU}(x)$ y $t_{CPU}(x)$ por separado, minimizaremos $f(x)=t_{GPU+CPU}(x)$.

Se conoce que el tiempo en 90% es aproximadamente 4.98 segundos y en 70% 7.29 segundos.

Table 7: Pseudo-Bisección

% GPU	T	Acción
80	6.78	Aumentar %
85	7.05	Disminuir %
82.5	6.77	-

A pesar de que sería posible continuar con el algoritmo, a medida que se avanza, las mejores disminuyen maś y más, por lo que finalmente, podríamos aproximar que el porcentaje óptimo para los algoritmos y el hardware utilizado sería 82.5% para GPU.

Tabla Tiempos GPU

Table 8: Tiempos GPU

Yres	Tiempos 1D			Tiempos 2D			Heterogéneo
	Device	Pinned	Unified	Device	Pinned	Unified	
1024	1.22E-01	1.02E-01	1.01E-01	5.89E-02	1.02E-01	1.00E-01	1.36E-01
2048	2.76E-01	2.84E-01	2.76E-01	2.16E-01	2.84E-01	2.76E-01	3.08E-01
4096	9.22E-01	9.53E-01	9.23E-01	8.09E-01	9.53E-01	9.23E-01	9.64E-01
8192	3.27E+00	3.39E+00	3.27E+00	3.07E+00	3.39E+00	3.27E+00	3.88E+00
10240	5.03E+00	5.22E+00	5.04E+00	4.72E+00	5.22E+00	5.04E+00	6.77E+00

Comparaciones 1660Ti v. 1050Ti

La 1660Ti tiene una aceleración entre 1.38 y 1.87 dependiendo de la resolución, generalmente, a mayor resolución, mayor la aceleración.

Las siguientes aceleraciones se han calculado utilizando la memoria device, con los hilos organizados en 1D y el tiempo corresponde a la totalidad de la ejecución (mandel+promedio+binarizado). Respecto a los algoritmos se han utilizado los más simples.

Table 9: Aceleraciones 1660Ti v. 1050Ti

Resolución	Aceleración 1660Ti/1050Ti
1024	1.38
2048	1.76
4096	1.81
8192	1.85
10240	1.87

Este comportamiento es de esperar, ya que la 1660Ti es una tarjeta una generación más nueva que la 1050Ti, además de ser de una categoría superior. No obstante, ambas son capaces de ejecutar CUDA adecuadamente si bien no todos los programas que se puedan ejecutar en una serán compatibles para la otra (ver AtomicAdd).

Conclusión

La programación paralela en GPUs ha demostrado tener capacidades extraordinarias a la hora de realizar calculos. En la mayoría de casos un acercamiento a fuerza bruta resulta la mejor opción gracias a la cantidad de núcleos que poseen las GPUs.

Claro está que al igual que cualquier otra herramienta que utilice la paralelización, los problemas que una GPU puede resolver eficientemente son limitados. El cálculo de cualquier serie seguirá siendo monohilo, debido a las dependencias que incluye el problema.

A medida que pasa el tiempo es más común encontrar sistemas con una o incluso varias GPUs, por lo que poco a poco resulta más interesante aprovechar el paralelismo con estas herramientas gracias a su disponibilidad.

Como conclusión, el uso del paralelismo presenta un acercamiento prometedor a una gran cantidad de problemas. Y es probable que el rendimiento, en particular de las GPU siga aumentando a medida que nuevas generaciones salen al mercado.

Instrumentación y recursos utilizados

Sitios online

Título	Sección	Fecha Edición	Autor(a)	Link
OpenMP Application Program Interface	-	Jul, 2013	-	https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
CUDA C++ Programming Guide	AtomicAdd	Dic, 2022	Nvidia	https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicadd
Why does atomicAdd not work with doubles as input?	-	Dic, 2017	cbuchner1	https://forums.developer.nvidia.com/t/why-does-atomicadd-not-work-with-doubles-as-input/56429/2
CUDA Refresher: The CUDA Programming Model	Cuda Kernel and thread hierarchy	Jun, 2020	Pradeep Gupta	https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/
How to Optimize Data Transfers in CUDA C/C++	Pinned Host Memory	Dec, 2012	Mark Harris	https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

Software

Nombre	Uso	Descripción	Versión	Link
Libreoffice Writer	Edición de texto	Editor de texto	7.4.3.2	https://www.libreoffice.org/discover/writer/
Libreoffice Calc	Recopilación de datos y cálculos	Hoja de cálculo	7.4.3.2	https://www.libreoffice.org/discover/calc/
Libreoffice Draw	Edición de ilustraciones	Editor gráfico	7.4.3.2	https://www.libreoffice.org/discover/draw/
Vim	Edición de código	Editor de texto	9.0.1046-1	https://github.com/vim/vim
CUDA	Ejecución de código	Toolkit	11.8.0-1	https://developer.nvidia.com/cuda-toolkit

Hardware

Nombre	Uso	Link Especificaciones
Nvidia GTX 1050Ti	Computación	https://www.techpowerup.com/gpu-specs/msi-gtx-1050-ti-aero-itx.b4262

Anexo

Anexo 0, funciones CPU

```
#include "Prototipos.h"
#include <omp.h>

void mandel(double xmin, double ymin, double xmax, double ymax, int maxiter, int
xres, int yres, double* A){
        double dx, dy, u = 0, v = 0, u_old = 0, paso_x, paso_y;
        dx = (xmax-xmin)/xres;
        dy = (ymax-ymin)/yres;
        int i = 0, j = 0, k = 0;
        #pragma omp parallel private(i, j, u, v, k, u_old, paso_x, paso_y)
        #pragma omp single
        for (i = 0 ; i < xres ; i++)
        {
                for (j = 0 ; j < yres ; j++)
                {
                        #pragma omp task
                        {
                                paso_x = i*dx+xmin;
                                paso_y = j*dy+ymin;
                                u = 0;
                                v = 0;
                                k = 1;
                                while (k < maxiter && (u*u+v*v) < 4)
                                {
                                        u_old = u;
                                        u = u_old*u_old - v*v + paso_x;
                                        v = 2*u_old*v + paso_y;
                                        k = k + 1;
                                }
                                if (k ≥ maxiter)       *(A+j*xres+i) = 0;
                                else                   *(A+j*xres+i) = k;
                        }
                }
        }
}
```

```
void mandel_schedule_static (double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A){
        double dx, dy, u = 0, v = 0, u_old = 0, paso_x, paso_y;
        dx = (xmax-xmin)/xres;
        dy = (ymax-ymin)/yres;
        int i = 0, j = 0, k = 0;
        #pragma omp parallel for private(i, j, u, v, k, u_old, paso_x, paso_y)
        for (i = 0 ; i < xres ; i++)
        {
                for (j = 0 ; j < yres ; j++)
                {
                        paso_x = i*dx+xmin;
                        paso_y = j*dy+ymin;
                        u = 0;
                        v = 0;
                        k = 1;
                        while (k < maxiter && (u*u+v*v) < 4)
                        {
                                u_old = u;
                                u = u_old*u_old - v*v + paso_x;
                                v = 2*u_old*v + paso_y;
                                k = k + 1;
                        }
                        if (k ≥ maxiter)       *(A+j*xres+i) = 0;
                        else                   *(A+j*xres+i) = k;
                }
        }
        return;
}

void mandel_schedule_dynamic (double xmin, double ymin, double xmax, double ymax,
int maxiter, int xres, int yres, double* A){
        double dx, dy, u = 0, v = 0, u_old = 0, paso_x, paso_y;
        dx = (xmax-xmin)/xres;
        dy = (ymax-ymin)/yres;
        int i = 0, j = 0, k = 0;
          #pragma omp parallel for private(i, j, u, v, k, u_old, paso_x, paso_y)
schedule(dynamic)
        for (i = 0 ; i < xres ; i++)
        {
                for (j = 0 ; j < yres ; j++)
                {
                        paso_x = i*dx+xmin;
                        paso_y = j*dy+ymin;
                        u = 0;
                        v = 0;
                        k = 1;
                        while (k < maxiter && (u*u+v*v) < 4)
                        {
                                u_old = u;
                                u = u_old*u_old - v*v + paso_x;
                                v = 2*u_old*v + paso_y;
                                k = k + 1;
                        }
                        if (k ≥ maxiter)       *(A+j*xres+i) = 0;
                        else                   *(A+j*xres+i) = k;
                }
        }
        return;
}
```

```
void mandel_collapse (double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A){
        double dx, dy, u = 0, v = 0, u_old = 0, paso_x, paso_y;
        dx = (xmax-xmin)/xres;
        dy = (ymax-ymin)/yres;
        int i = 0, j = 0, k = 0;
          #pragma omp parallel for private(i, j, u, v, k, u_old, paso_x, paso_y)
schedule(dynamic) collapse(2)
        for (i = 0 ; i < xres ; i++)
        {
                for (j = 0 ; j < yres ; j++)
                {
                        paso_x = i*dx+xmin;
                        paso_y = j*dy+ymin;
                        u = 0;
                        v = 0;
                        k = 1;
                        while (k < maxiter && (u*u+v*v) < 4)
                        {
                                u_old = u;
                                u = u_old*u_old - v*v + paso_x;
                                v = 2*u_old*v + paso_y;
                                k = k + 1;
                        }
                        if (k ≥ maxiter)       *(A+j*xres+i) = 0;
                        else                   *(A+j*xres+i) = k;
                }
        }
        return;
}

double promedio (int xres, int yres, double* A){
        int i;
        double s;
        s = 0;
        #pragma omp parallel for reduction(+:s)
        for (i = 0 ; i < xres*yres ; i++)
                s+=*(A+i);
   return s/(xres*yres);
}

double promedio_atomic (int xres, int yres, double* A){
        int     i;
        double  s = 0,
                p_s = 0;

        #pragma omp parallel firstprivate(p_s)
        {
                p_s = 0;
                #pragma omp single
                        s = 0;
                #pragma omp for
                for (i = 0 ; i < xres*yres ; i++)
                        p_s+=*(A+i);
                #pragma omp atomic update
                        s+=p_s;
        }
   return s/(xres*yres);
}
```

```
void binariza(int xres, int yres, double* A, double med){
        int i;
        #pragma omp parallel for
        for (i = 0 ; i < xres*yres ; i++)
        {
                if (*(A+i) ≥ med)      *(A+i) = 255;
                else    *(A+i) = 0;
        }
}
```

Anexo 1, kernels & funciones GPU

```
#include "PrototiposGPU.h"

__global__ void kernelMandel(double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A)
{
        int     i = threadIdx.x+blockIdx.x*blockDim.x,
                j;

        if (i < xres)
        {
                for (j = 0 ; j < yres ; j++)
                {
                        double  dx = (xmax-xmin)/xres,
                                dy = (ymax-ymin)/yres,
                                u = 0, v = 0, u_old = 0,
                                paso_x = i*dx+xmin,
                                paso_y = j*dy+ymin;
                        int     k = 1;

                        while (k < maxiter && (u*u+v*v) < 4)
                        {
                                u_old = u;
                                u = u_old*u_old - v*v + paso_x;
                                v = 2*u_old*v + paso_y;
                                k = k + 1;
                        }
                        if (k ≥ maxiter)       *(A+i+j*xres) = 0;
                        else                   *(A+i+j*xres) = k;
                }
        }
        return;
}
```

```
__global__ void kernelMandel2D(double xmin, double ymin, double xmax, double ymax,
int maxiter, int xres, int yres, double* A)
{
        int     i = threadIdx.x+blockIdx.x*blockDim.x,
                j = threadIdx.y+blockIdx.y*blockDim.y;

        if (i < xres && j < yres)
        {
                double  dx = (xmax-xmin)/xres,
                        dy = (ymax-ymin)/yres,
                        u = 0, v = 0, u_old = 0,
                        paso_x = i*dx+xmin,
                        paso_y = j*dy+ymin;
                int     k = 1;

                while (k < maxiter && (u*u+v*v) < 4)
                {
                        u_old = u;
                        u = u_old*u_old - v*v + paso_x;
                        v = 2*u_old*v + paso_y;
                        k = k + 1;
                }
                if (k ≥ maxiter)       *(A+i+j*xres) = 0;
                else                   *(A+i+j*xres) = k;
        }
        return;
}


__global__ void kernelBinariza(int xres, int yres, double* A, double med)
{
        int     i = threadIdx.x + blockIdx.x * blockDim.x;

        if (i < xres*yres)
        {
                if (*(A+i) > med)       *(A+i) = 255;
                else                    *(A+i) = 0;
        }
        return;
}

__global__ void kernelBinariza2D(int xres, int yres, double* A, double med)
{
        int     i = threadIdx.x + blockIdx.x * blockDim.x,
                j = threadIdx.y + blockIdx.y * blockDim.y;

        if (i < xres && j < yres)
        {
                if (*(A+i+j*xres) > med)        *(A+i+j*xres) = 255;
                else                            *(A+i+j*xres) = 0;
        }
        return;
}
```

```
__global__ void sum_blks (int size, double* A, double* dst)
{
        extern __shared__ double shared_data [];
        int     i = threadIdx.x + blockIdx.x * blockDim.x,
                j = blockDim.x/2;

        double tmp = 0.0;

        // if (i < size)
        // {
                while (i < size)
                {
                        tmp += *(A+i);
                        i += blockDim.x * gridDim.x;
                }

                *(shared_data+threadIdx.x) = tmp;
                __syncthreads();

                while(j ≠ 0)
                {
                        if (threadIdx.x < j)
                                                        *(shared_data+threadIdx.x)  +=
*(shared_data+j+threadIdx.x);
                        __syncthreads();
                        j⁄=2;
                }

                if (threadIdx.x == 0)
                        *(dst+blockIdx.x) = *(shared_data);
        // }
        return;
}
```

```
__global__ void sum (double* data, double* dst, int n_blks)
{
        extern __shared__ double shared_data [];
        int     i = threadIdx.x + blockIdx.x * blockDim.x,
                j = blockDim.x/2,
                bpt = n_blks / blockDim.x,
                k;

        double tmp = 0.0;

        // if (i < size)
        // {
                for (k = 0 ; k < bpt ; k++)
                        tmp += *(data+bpt*i+k);

                *(shared_data+threadIdx.x) = tmp;
                __syncthreads();

                while(j ≠ 0)
                {
                        if (threadIdx.x < j)
                                                        *(shared_data+threadIdx.x)  +=
*(shared_data+j+threadIdx.x);
                        __syncthreads();
                        j⁄=2;
                }

                if (threadIdx.x == 0)
                        *(dst) = *(shared_data);
        // }
        return;
}

__global__ void sum_atomic (int size, double* data, double* dst)
{
        int     i = threadIdx.x + blockIdx.x * blockDim.x;

        if (i < size)
                // atomicAdd(dst, *(data+i));

        return;
}
```

```
extern "C" void mandel_omp (double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A, double perc)
{
        double dx, dy, u = 0, v = 0, u_old = 0, paso_x, paso_y;
        dx = (xmax-xmin)/xres;
        dy = (ymax-ymin)/yres;
        int i = 0, j = 0, k = 0;
          #pragma omp parallel for private (i, j, u, v, k, u_old, paso_x, paso_y)
schedule(dynamic)
        for (i = xres*perc ; i < xres ; i++)
                for (j = 0 ; j < yres ; j++)
                {
                        paso_x = i*dx+xmin;
                        paso_y = j*dy+ymin;
                        u = 0;
                        v = 0;
                        k = 1;
                        while (k < maxiter && (u*u+v*v) < 4)
                        {
                                u_old = u;
                                u = u_old*u_old - v*v + paso_x;
                                v = 2*u_old*v + paso_y;
                                k = k+1;
                        }
                        if (k ≥ maxiter)       *(A+j*xres+i) = 0;
                        else                   *(A+j*xres+i) = k;
                }
        return;
}

extern "C" void mandelHetero(double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A, int ThpBlk)
{
        double  *Dev_a = NULL;
        int     size = xres*yres*sizeof(double),
                perc_gpu = 0.852,
                n_blks = (int) (yres*perc_gpu+ThpBlk-1)/ThpBlk;

        CUDAERR(cudaMallocManaged((void **)&Dev_a, size, cudaMemAttachGlobal));
        CUDAERR(cudaMemcpy(Dev_a, A, size, cudaMemcpyHostToDevice));

          kernelMandel <<<n_blks, ThpBlk>>> (xmin, ymin, xmax, ymax, maxiter, xres,
yres, Dev_a);
        mandel_omp(xmin, ymin, xmax, ymax, maxiter, xres, yres, Dev_a, perc_gpu);

        cudaDeviceSynchronize();
        CHECKLASTERR();

        CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
        cudaFree(Dev_a);
}
```

```
extern "C" void mandelGPU(double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A, int ThpBlk)
{
        double  *Dev_a = NULL;
        int     size = xres*yres*sizeof(double),
                n_blks = (int) (yres+ThpBlk-1)/ThpBlk;

        CUDAERR(cudaMalloc((void **)&Dev_a, size));

         kernelMandel <<<n_blks, ThpBlk>>> (xmin, ymin, xmax,  ymax, maxiter, xres,
yres, Dev_a);

        cudaDeviceSynchronize();
        CHECKLASTERR();

        CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
        cudaFree(Dev_a);
}

extern "C" void managed_mandelGPU(double xmin, double ymin, double xmax, double
ymax, int maxiter, int xres, int yres, double* A, int ThpBlk)
{
        double  *Dev_a = NULL;
        int     size = xres*yres*sizeof(double),
                n_blks = (int) (yres+ThpBlk-1)/ThpBlk;

        CUDAERR(cudaMallocManaged((void**)&Dev_a, size, cudaMemAttachGlobal));

         kernelMandel <<<n_blks, ThpBlk>>> (xmin, ymin, xmax,  ymax, maxiter, xres,
yres, Dev_a);

        cudaDeviceSynchronize();
        CHECKLASTERR();

        CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
        cudaFree(Dev_a);
}

extern "C" void pinned_mandelGPU(double xmin, double ymin, double xmax, double ymax,
int maxiter, int xres, int yres, double* A, int ThpBlk)
{
        double  *Dev_a = NULL, *ptr_Dev_a = NULL;
        int     size = xres*yres*sizeof(double),
                n_blks = (int) (yres/ThpBlk)+1;

        CUDAERR(cudaHostAlloc((void**)&Dev_a, size, cudaHostAllocMapped));
        CUDAERR(cudaHostGetDevicePointer((void**) &ptr_Dev_a, (void*)Dev_a, 0));

         kernelMandel <<<n_blks, ThpBlk>>> (xmin, ymin, xmax,  ymax, maxiter, xres,
yres, ptr_Dev_a);

        cudaDeviceSynchronize();
        CHECKLASTERR();

        CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
        cudaFreeHost(Dev_a);
}
```

```
extern "C" void mandelGPU2D(double xmin, double ymin, double xmax, double ymax, int
maxiter, int xres, int yres, double* A, int ThpBlk)
{
        double  *Dev_a = NULL;
        int     size = xres*yres*sizeof(double);
        dim3    dim_block (ThpBlk, ThpBlk),
                                        dim_grid    ((xres+dim_block.x-1)/dim_block.x,
(yres+dim_block.y-1)/dim_block.y);

        CUDAERR(cudaMalloc((void **)&Dev_a, size));

         kernelMandel2D <<<dim_grid, dim_block>>> (xmin, ymin, xmax,  ymax, maxiter,
xres, yres, Dev_a);

        cudaDeviceSynchronize();
        CHECKLASTERR();

        CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
        cudaFree(Dev_a);
}

extern "C" void managed_mandelGPU2D(double xmin, double ymin, double xmax, double
ymax, int maxiter, int xres, int yres, double* A, int ThpBlk)
{
        double  *Dev_a = NULL;
        int     size = xres*yres*sizeof(double);
        dim3    dim_block (ThpBlk, ThpBlk),
                                        dim_grid    ((xres+dim_block.x-1)/dim_block.x,
(yres+dim_block.y-1)/dim_block.y);

        CUDAERR(cudaMallocManaged((void**)&Dev_a, size, cudaMemAttachGlobal));

         kernelMandel2D <<<dim_grid, dim_block>>> (xmin, ymin, xmax,  ymax, maxiter,
xres, yres, Dev_a);

        cudaDeviceSynchronize();
        CHECKLASTERR();

        CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
        cudaFree(Dev_a);
}
```

```
extern "C" void pinned_mandelGPU2D(double xmin, double ymin, double xmax, double
ymax, int maxiter, int xres, int yres, double* A, int ThpBlk)
{
        double  *Dev_a = NULL, *ptr_Dev_a = NULL;
        int     size = xres*yres*sizeof(double);
        dim3    dim_block (ThpBlk, ThpBlk),
                                        dim_grid    ((xres+dim_block.x-1)/dim_block.x,
(yres+dim_block.y-1)/dim_block.y);

        CUDAERR(cudaHostAlloc((void**)&Dev_a, size, cudaHostAllocMapped));
        CUDAERR(cudaHostGetDevicePointer((void**) &ptr_Dev_a, (void*)Dev_a, 0));

         kernelMandel2D <<<dim_grid, dim_block>>> (xmin, ymin, xmax,  ymax, maxiter,
xres, yres, ptr_Dev_a);

        cudaDeviceSynchronize();
        CHECKLASTERR();

        CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));
        cudaFreeHost(Dev_a);
}

extern "C" double promedioGPU(int xres, int yres, double* A, int ThpBlk)
{
        double  avg = 0, *Dev_a = NULL;
        int     size = xres*yres*sizeof(double);
        cublasHandle_t handle;

        CUDAERR(cudaMalloc((void**) &Dev_a, size));
        CUDAERR(cudaMemcpy(Dev_a, A, size, cudaMemcpyHostToDevice));

        cublasCreate(&handle);
        cublasDasum(handle, size/sizeof(double), Dev_a, 1, &avg);
        cublasDestroy(handle);

        cudaFree(Dev_a);
        return avg/size*sizeof(double);
}
```

```
extern "C" double promedioGPUSum(int xres, int yres, double* A, int ThpBlk)
{
        double  *avg = NULL,
                *Dev_blks = NULL,
                *Dev_avg = NULL,
                *Dev_a = NULL;

        int     size = xres*yres*sizeof(double),
                n_blks = (xres*yres+ThpBlk-1)/ThpBlk;

        avg = (double*) malloc (sizeof(double));

        CUDAERR(cudaMalloc((void**) &Dev_blks, n_blks*sizeof(double))); // midway
        CUDAERR(cudaMalloc((void**) &Dev_avg, sizeof(double))); // dst
        CUDAERR(cudaMalloc((void**) &Dev_a, size));     // src data

        CUDAERR(cudaMemcpy(Dev_a, A, size, cudaMemcpyHostToDevice));

          sum_blks <<< n_blks, ThpBlk, ThpBlk*sizeof(double) >>> (xres*yres, Dev_a,
Dev_blks);
        sum <<< 1, 1024, 1024*sizeof(double) >>> (Dev_blks, Dev_avg, n_blks);

        CHECKLASTERR();
        CUDAERR(cudaMemcpy(avg, Dev_avg, sizeof(double), cudaMemcpyDeviceToHost));

        cudaFree(Dev_blks);
        cudaFree(Dev_a);
        cudaFree(Dev_avg);

        return *avg/size*sizeof(double);
}

extern "C" double promedioGPUAtomic(int xres, int yres, double* A, int ThpBlk)
{
        double  *avg = NULL,
                *Dev_a = NULL,
                *Dev_avg = NULL;

        int     size = xres*yres*sizeof(double),
                n_blks = (xres*yres+ThpBlk-1)/ThpBlk;

        avg = (double*) malloc (sizeof(double));

        CUDAERR(cudaMalloc((void**) &Dev_avg, sizeof(double))); // dst
        CUDAERR(cudaMalloc((void**) &Dev_a, size));     // src data

        CUDAERR(cudaMemcpy(Dev_a, A, size, cudaMemcpyHostToDevice));

        sum_atomic <<< n_blks, ThpBlk >>> (xres*yres, Dev_a, Dev_avg);

        CHECKLASTERR();
        CUDAERR(cudaMemcpy(avg, Dev_avg, sizeof(double), cudaMemcpyDeviceToHost));

        cudaFree(Dev_a);
        cudaFree(Dev_avg);

        return *avg/size*sizeof(double);
}
```

```
extern "C" void binarizaGPU(int xres, int yres, double* A, double med, int ThpBlk)
{
        double *Dev_a = NULL;
        int     n_blks = (int) (xres*yres+ThpBlk-1)/ThpBlk,
                size = xres*yres*sizeof(double);

        CUDAERR(cudaMalloc((void **)&Dev_a, size));
        CUDAERR(cudaMemcpy(Dev_a, A, size, cudaMemcpyHostToDevice));

        kernelBinariza<<< n_blks, ThpBlk >>> (xres, yres, Dev_a, med);

        cudaDeviceSynchronize();
        CHECKLASTERR();

        CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));

        cudaFree(Dev_a);
}

extern "C" void binarizaGPU2D(int xres, int yres, double* A, double med, int ThpBlk)
{
        double  *Dev_a = NULL;
        int     size = xres*yres*sizeof(double);
        dim3    dim_block (ThpBlk, ThpBlk),
                                        dim_grid    ((xres+dim_block.x-1)/dim_block.x,
(yres+dim_block.y-1)/dim_block.y);

        CUDAERR(cudaMalloc((void **)&Dev_a, size));
        CUDAERR(cudaMemcpy(Dev_a, A, size, cudaMemcpyHostToDevice));

        kernelBinariza2D<<< dim_grid, dim_block>>> (xres, yres, Dev_a, med);

        cudaDeviceSynchronize();
        CHECKLASTERR();

        CUDAERR(cudaMemcpy(A, Dev_a, size, cudaMemcpyDeviceToHost));

        cudaFree(Dev_a);
}
```

Anexo 2, Makefile

```
cflags=-Xcompiler -fopenmp,-O3,-Wall,-fPIC,-L/opt/cuda/include  # To be used on
1660Ti's system
# cflags=-Xcompiler -fopenmp,-O3,-Wall,-fPIC,-L/opt/cuda/include --gpu-
architecture=compute_61 -gpu-code=sm_61 # To be used on 1050Ti's system

libs=-lcublas -lcudart -lgomp

all: cleanall mandelGPU

mandelGPU.o: mandelGPU.cu
    nvcc $(cflags) -c mandelGPU.cu -o mandelGPU.o

mandelGPU: mandelGPU.o
    g++ -shared -Wl,-soname,mandelGPU.so -o mandelGPU.so mandelGPU.o
-L$(CUDADIR)/lib64 $(libs   # To be used on 1660Ti's system
    # g++ -shared -Wl,-soname,mandelGPU.so -o mandelGPU.so mandelGPU.o
-L/opt/cuda/lib64 $(libs)   # To be used on 1050Ti's system
    rm -f mandelGPU.o

clean:
    rm -f *~ *.o core

cleanall: clean
    rm -f mandelGPU.so mandelGPU.o
```