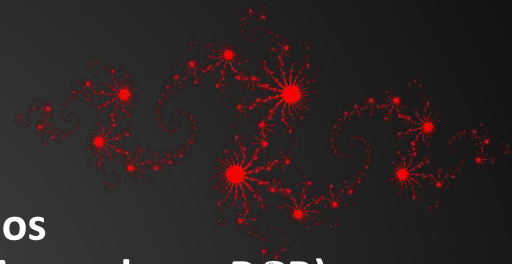


CUDA Threads

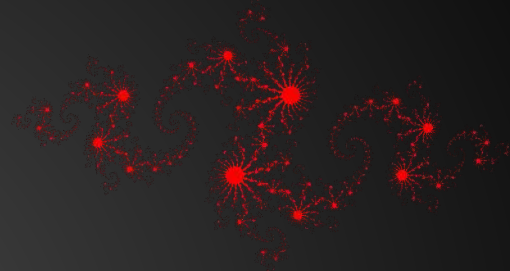
Marc-Antoine Le Guen

Segunda programa - Conjunto de Julia

- Crear imagenes procedurales
- Libreria de imagenes para visualizar los resultados
- unsigned char (0-255), 1 o 4 canales(Nivel de gris o colores RGB).
- Cada pixel tiene una coordenada (x,y) en la imagen
 - En nuestro caso, la posición del píxel en la imagen representa la posición x,y en el espacio complejo
 - x=número real y=número imaginario



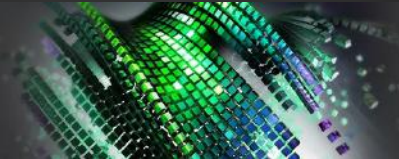
Segunda programa - Conjunto de Julia



- **Algoritmo**

- **Para cada pixel**

- Julia set $Z_{n+1} = Z_n^2 + C$ (1)
 - $\text{complex}(x,y) * \text{complex}(x,y) + C$
 - Calcular iteraciones de la fórmula (1) y estudiar su comportamiento
 - Calculamos 200 iteraciones
 - Si diverge estamos **fuera** del conjunto de julia $\sum_{n=0}^{200} Z_n > 1000$
 - fondo negro
 - Si no diverge estamos **dentro** del conjunto de julio $\sum_{n=0}^{200} Z_n \leq 1000$
 - Atribución de un color



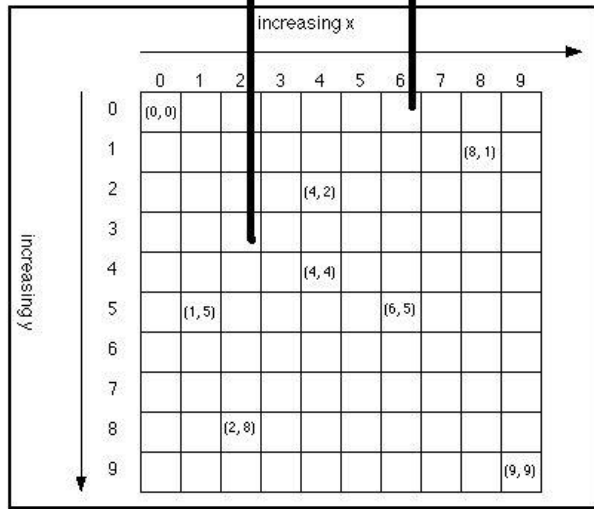
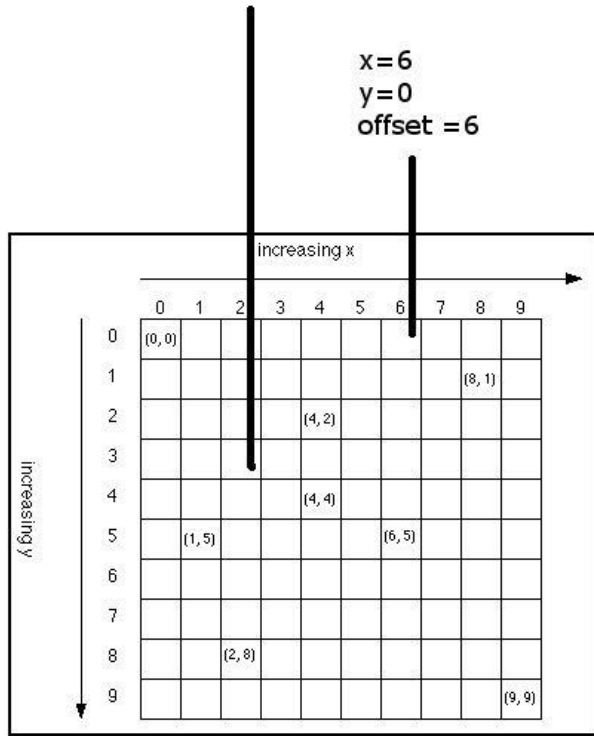
de Julia

CPU - Kernel

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM;
            int jv = julia( x, y );
            ptr[offset*4 + 0] = 255 * jv;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
}
```

```
x=2
y=3
offset = 2 + 3 * 10
offset = 32
```

```
x=6
y=0
offset = 6
```

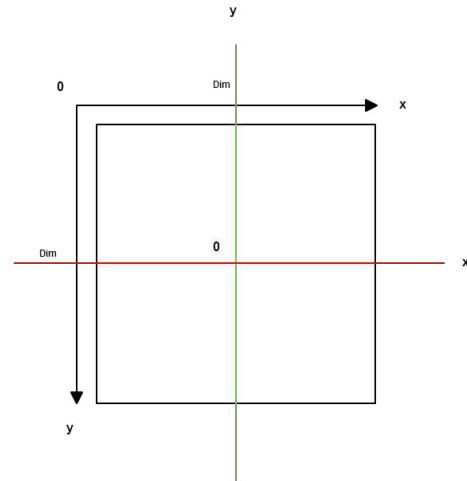


Segundo programa - Conjunto de Julia

- Cambiar de espacio imagen clásico hacia el espacio
 - `(float)(DIM/2 - x)/(DIM/2)` espacio entre `[-1.0, 1.0]`
 - Scale factor de zoom (1.5 unzoom) `[-1.5, 1.5]`

CPU - JuliaValue

```
int julia( int x, int y ) {  
    const float scale = 1.5;  
    float jx = scale * (float)(x - DIM/2)/(DIM/2);  
    float jy = scale * (float)(y - DIM/2)/(DIM/2);  
    cuComplex c (-0.8, 0.156);  
    cuComplex a (jx, jy);  
    int i = 0;  
    for (i=0; i<200; i++) {  
        a = a * a + c;  
        if (a.magnitude2 () > 1000)  
            return 0;  
    }  
    return 1;  
}
```



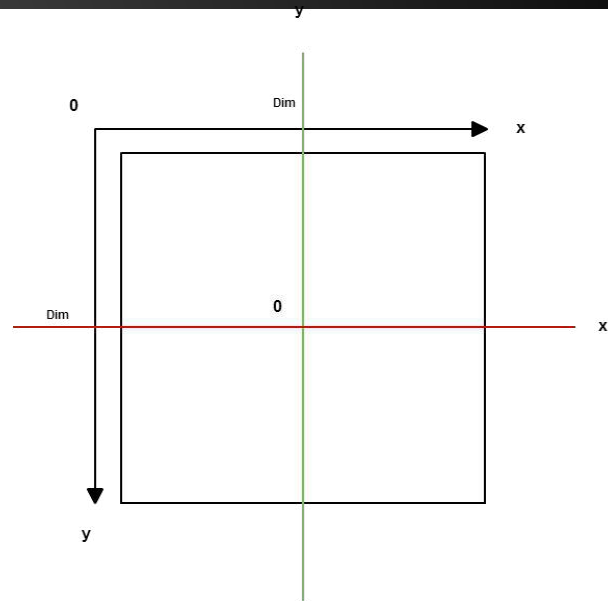
Segundo programa - Conjunto de Julia

- Complex number

$$C = -0.8 + 0.158 \times i$$

CPU - JuliaValue

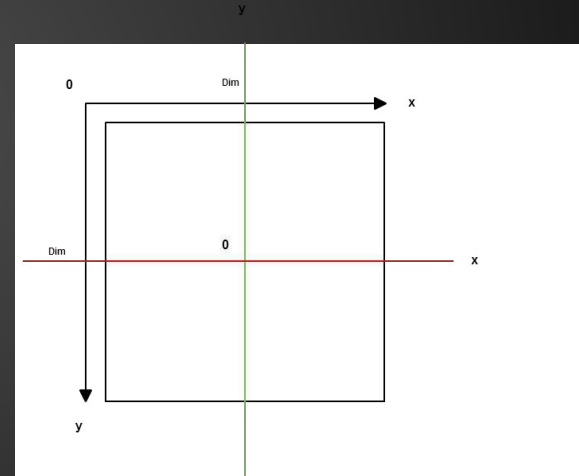
```
int julia( int x, int y ) {  
    const float scale = 1.5;  
    float jx = scale * (float)(x - DIM/2)/(DIM/2);  
    float jy = scale * (float)(y - DIM/2)/(DIM/2);  
    cuComplex c(-0.8, 0.156);  
    cuComplex a(jx, jy);  
    int i = 0;  
    for (i=0; i<200; i++) {  
        a = a * a + c;  
        if (a.magnitude2() > 1000)  
            return 0;  
    }  
    return 1;  
}
```



Segundo programa - Conjunto de Julia

CPU - JuliaValue

```
int julia( int x, int y ) {  
    const float scale = 1.5;  
    float jx = scale * (float)(x - DIM/2)/(DIM/2);  
    float jy = scale * (float)(y - DIM/2)/(DIM/2);  
    cuComplex c (-0.8, 0.156);  
    cuComplex a (jx, jy);  
    int i = 0;  
    for (i=0; i<200; i++) {  
        a = a * a + c;  
        if (a.magnitude2() > 1000)  
            return 0;  
    }  
    return 1;  
}
```



Segundo programa - Conjunto de Julia

CPU - Complex structure

```
struct cuComplex {  
    float r;  
    float i;  
    cuComplex( float a, float b ) : r(a), i(b) {}  
    float magnitude2( void ) {  
        return r * r + i * i;  
    }  
    cuComplex operator*(const cuComplex& a) {  
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);  
    }  
    cuComplex operator+(const cuComplex& a) {  
        return cuComplex(r+a.r, i+a.i);  
    }  
};
```



Segundo programa - Conjunto de Julia

- Crear un imagen para el GPU y una para el CPU (visualizar el resultado)
- `dim3 grid(DIM,DIM)` el tercer parámetro está por defecto en 1
 - definimos una cuadrícula en 2D de blocks del mismo tamaño de la imagen
 - 1 pixel = 1 block = 1 copia del kernel
 - `kernel<<<grid,1>>>(gpu_bitmap);`
 - Copiamos el resultado en `cpu_bitmap` para usarlo...

GPU - main

```
unsigned char *cpu_bitmap=(unsigned char*)malloc(sizeof(unsigned char)*DIM*DIM*4);  
cudaMalloc( &gpu_bitmap,  
u( (void**)&gpu_bitmap, DIM*DIM*4 );
```

```
dim3 grid(DIM,DIM);  
kernel<<<grid,1>>>( gpu_bitmap );
```

```
cudaMemcpy( cpu_bitmap, gpu_bitmap,DIM*DIM*4 , cudaMemcpyDeviceToHost );  
cudaFree( gpu_bitmap );
```

```
//Visualizar cpu_bitmap
```

Segundo programa - Conjunto de Julia

- Crear un imagen para el GPU y una para el CPU (visualizar el resultado)
- `dim3 grid(DIM,DIM)` el tercer parámetro está por defecto en 1
 - definimos una cuadrícula en 2D de blocks del mismo tamaño de la imagen
 - 1 pixel = 1 block = 1 copia del kernel
 - `kernel<<<grid,1>>>(gpu_bitmap);`
 - Copiamos el resultado en `cpu_bitmap` para usarlo...

GPU - main

```
unsigned char *gpu_bitmap;
unsigned char *cpu_bitmap=(unsigned char*)malloc(sizeof(unsigned char)*DIM*DIM*4);
cudaMalloc( (void**)&gpu_bitmap, DIM*DIM*4 );

dim3 grid(DIM,DIM);
kernel<<<grid,1>>>( gpu_bitmap );

cudaMemcpy( cpu_bitmap, gpu_bitmap,DIM*DIM*4 , cudaMemcpyDeviceToHost );
cudaFree( gpu_bitmap );

//Visualizar cpu_bitmap
```

Segundo programa - Conjunto de Julia

- Crear un imagen para el GPU y una para el CPU (visualizar el resultado)
- `dim3 grid(DIM,DIM)` el tercer parámetro está por defecto en 1
 - definimos una cuadrícula en 2D de blocks del mismo tamaño de la imagen
 - 1 pixel = 1 block = 1 copia del kernel
 - `kernel<<<grid,1>>>(gpu_bitmap);`
 - Copiamos el resultado en `cpu_bitmap` para usarlo...

GPU - main

```
unsigned char *gpu_bitmap;  
unsigned char *cpu_bitmap=(unsigned char*)malloc(sizeof(unsigned char)*DIM*DIM);  
cudaMalloc ( (void**)&gpu_bitmap , DIM*DIM);  
  
dim3 grid(DIM,DIM);  
kernel<<<grid,1>>>( gpu_bitmap );  
  
cudaMemcpy ( cpu_bitmap , gpu_bitmap ,DIM*DIM , cudaMemcpyDeviceToHost );  
cudaFree ( gpu_bitmap );  
  
//Visualizar cpu_bitmap
```

Segundo programa - Conjunto de Julia

- Conversión simple del cálculo de Julia y de la estructura de datos
 - __device__
 - Código ejecutado en el device


GPU - JuliaValue

```
__device__ int julia( int x, int y ) {  
    const float scale = 1.5;  
    float jx = scale * (float)( x - DIM/2)/(DIM/2);  
    float jy = scale * (float)(y - DIM/2)/(DIM/2);  
    cuComplex c(-0.8, 0.156);  
    cuComplex a(jx, jy);  
  
    int i = 0;  
    for (i=0; i<200; i++) {  
        a = a * a + c;  
        if (a.magnitude2() > 1000)  
            return 0;  
    }  
    return 1;  
}
```

GPU - Complex structure

```
struct cuComplex {  
    float r;  
    float i;  
    __device__ cuComplex( float a, float b ) : r(a), i(b) {}  
    __device__ float magnitude2( void ) {  
        return r * r + i * i;  
    }  
    __device__ cuComplex operator*(const cuComplex& a) {  
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);  
    }  
    __device__ cuComplex operator+(const cuComplex& a) {  
        return cuComplex(r+a.r, i+a.i);  
    }  
};
```

Segundo programa - Conjunto de Julia

- Desaparecen los dos bucles de iteración
- Usamos los índices del blockIdx corriente
 - posición del píxel = posición del block en el grid
- Pasar de C  CUDA puede ser así de fácil.

GPU - kernel

```
__global__ void kernel( unsigned char *ptr ) {  
    // map from blockIdx to pixel position  
    int x = blockIdx.x;  
    int y = blockIdx.y;  
    int offset = x + y * gridDim.x;  
  
    // now calculate the value at that position  
    int juliaValue = julia( x, y );  
    ptr[offset] = 255 * juliaValue;  
  
}
```

CPU - kernel

```
void kernel( unsigned char *ptr ){  
    for (int y=0; y<DIM; y++) {  
        for (int x=0; x<DIM; x++) {  
            int offset = x + y * DIM;  
            int juliaValue = julia( x, y );  
            ptr[offset] = 255 * juliaValue;  
        }  
    }  
}
```

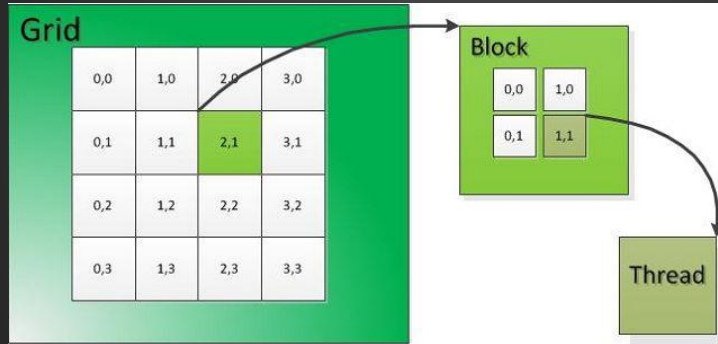


Threads

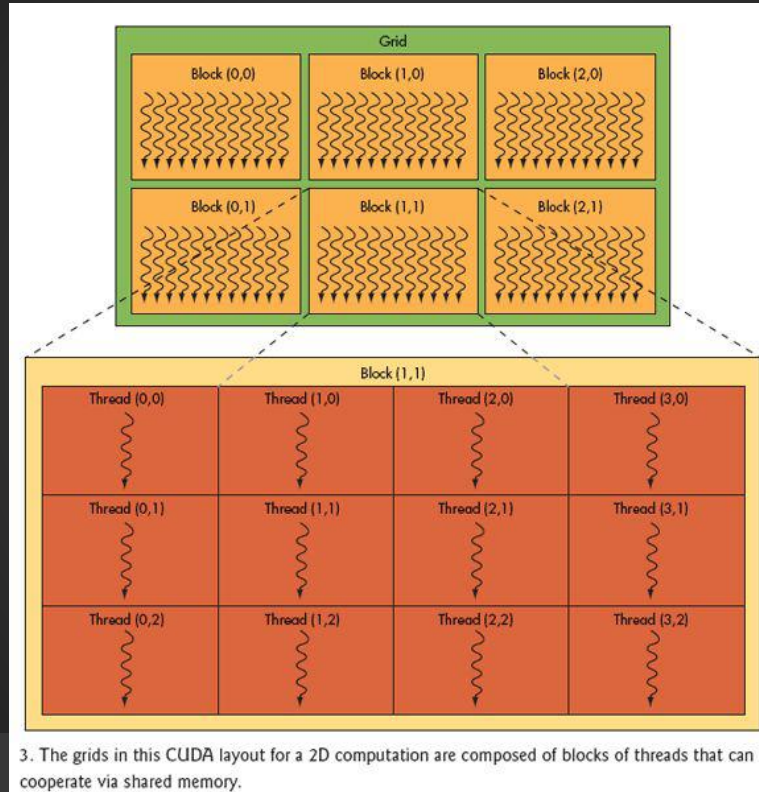
- N blocks es lo mismo que N copias de kernels
- `add<<<N,1>>>(dev_a, dev_b, dev_c);`
 - N blocks
 - 1 Thread
- N blocks x 1 thread/block = N parallel threads
- N/2 blocks x 2 threads/block = N parallel threads
- N/4 blocks x 4 threads/block = N parallel threads
- ...



Threads, Arquitectura



Threads



Threads

- N blocks
 - blockIdx
- N threads
 - threadIdx

```
add<<<N,1>>>>(vec_a,vec_b,vec_c);
```

```
__global__ void add( int *a, int *b, int *c
)
{
    int tid = blockIdx.x;
    if(tid < N)
        c[tid] = a[tid] + b[tid];
}
```

```
add<<<1,N>>>>(vec_a,vec_b,vec_c);
```

```
__global__ void add( int *a, int *b, int *c
)
{
    int tid = threadIdx.x;
    if(tid < N)
        c[tid] = a[tid] + b[tid];
}
```

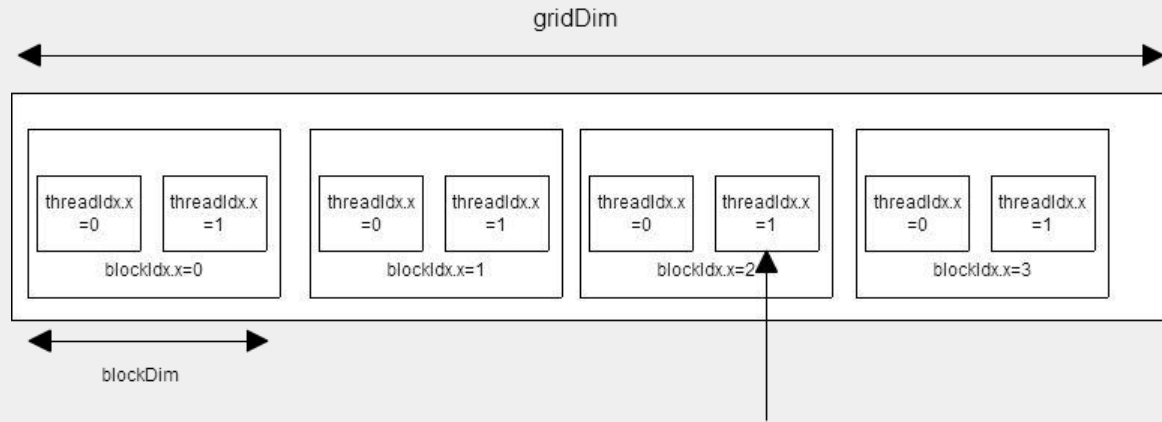


Threads, composición

- N blocks max : **65 535** blocks por dimensión
- N threads : *maxThreadsPerBlock* **512 - 1024**
- Definimos una cantidad máxima de thread a utilizar : *MaxThreads*
 - N cantidad de datos
 - Utilizaremos $(N / \text{MaxThreads})$ blocks con cada uno *MaxThreads* threads.
 - Problema $N < \text{MaxThreads}$
 - $\text{MaxThreads} = 128$
 - $N=100$
 - $N/\text{MaxThreads} = 0$
 - $((N+\text{MaxThreads}-1) / \text{MaxThreads})$ blocks
 - Mínimo utilizaremos un block
 - $N=128$
 - $(N+\text{MaxThreads}-1) / \text{MaxThreads} = 255/128 = 1$
 - $N=100$
 - $(N+\text{MaxThreads}-1) / \text{MaxThreads} = 227/128 = 1$
 - $N = 130$
 - $(N+\text{MaxThreads}-1) / \text{MaxThreads} = 257/128 = 2$
 - Habrá más threads que datos $128 > 100$ y $256 > 130$
 - `if (tid < N)`



Threads, composición



$$\text{offset} = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim} = 1 + 2 * 2 = 5$$



Threads, composición

```
add<<((N+(Nthreads-1))/Nthread,Nthread)>>(vec_a,vec_b,vec_c);
```

```
__global__ void add( int *a, int *b, int *c )  
{  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    if(tid < N)  
        c[tid] = a[tid] + b[tid];  
}
```



Threads, composición

- $(N + (N_{\text{threads}} - 1)) / N_{\text{thread}} < 65\,535$ (Max Blocks Per Dimension)
- $N_{\text{thread}} < \text{maxThreadsPerBlock}$
- Problema
 - $N > 65\,535 * \text{maxThreadsPerBlock} (1024)$
 - $N > 67\,107\,840$

Problema

```
add<<<((N+(Nthreads-1))/Nthread,Nthread>>>)(vec_a,vec_b,vec_c);
```

```
__global__ void add( int *a, int *b, int *c )  
{  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    if(tid < N)  
        c[tid] = a[tid] + b[tid];  
}
```

Threads, composición

- $((N + (N_{\text{threads}} - 1)) / N_{\text{thread}}) < 65\,535$ (Max Blocks Per Dimension)
- $N_{\text{thread}} < \text{maxThreadsPerBlock}$
- Problema
 - $N > 65\,535 * \text{maxThreadsPerBlock} (1024)$
 - $N > 67\,107\,840$

Solución

```
add<<<((N+(Nthreads-1))/Nthread,Nthread)>>>(vec_a,vec_b,vec_c);
```

```
__global__ void add( int *a, int *b, int *c )  
{  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    while(tid < N){  
        c[tid] = a[tid] + b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
}
```

Threads, composición

- $\text{blockDim} * \text{gridDim}(\text{copias del kernel}) < N$
- $\text{tid } n^\circ 0$ trata el elemento $n^\circ 0 + \text{elemento } n^\circ \text{tid} + \text{blockDim.x} * \text{gridDim.x}$
- Ejemplo, : $N = 1000$, $\text{BlockDim} * \text{gridDim} = 100$
 - $\text{tid } n^\circ 1 \rightarrow$ elemento $n^\circ 1, 101, 201, 301, 401, 501, 601, 701, 801, 901$
 - $\text{tid } n^\circ 6 \rightarrow$ elemento $n^\circ 6, 106, 206, 306, 406, 506, 606, 706, 806, 906$

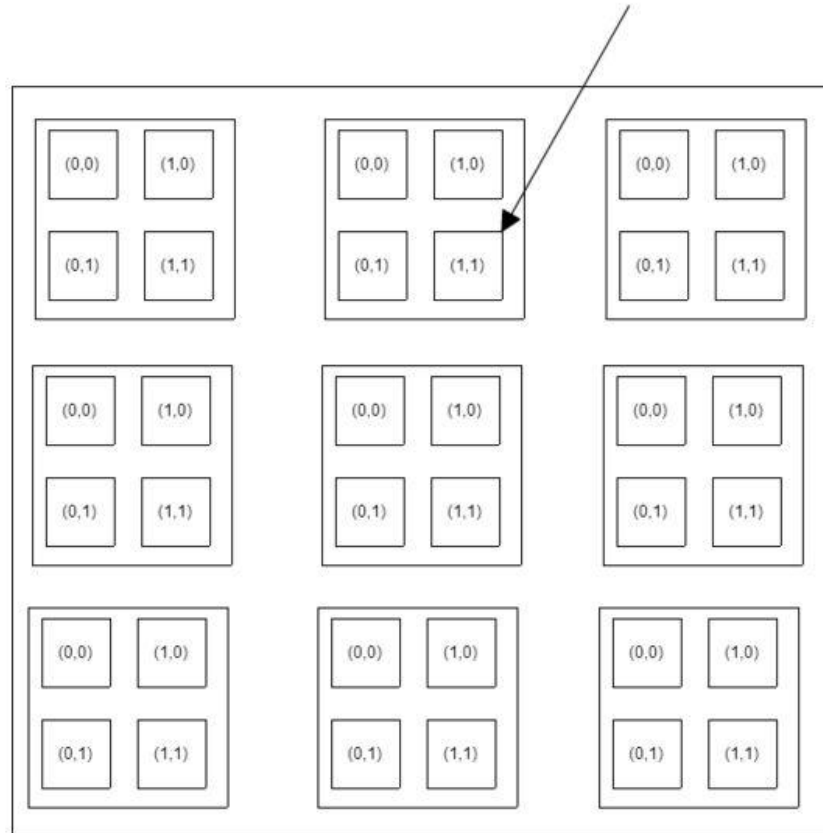
```
__global__ void add( int *a, int *b, int *c )  
{  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    while(tid < N){  
        c[tid] = a[tid] + b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
}
```



Threads, Gener

- Dividimos la imágenes en bloques de threads

```
x = threadIdx.x + blockDim.x * blockDim.x = 1 + 1 * 2 = 3  
y = threadIdx.y + blockDim.y * blockDim.y = 1 + 0 * 2 = 1  
offset = x + y * blockDim.x * gridDim.x = 3 + 1 * 2 * 3 = 9
```



Threads, Generación de imágenes

- Dividimos la imágenes en bloques de threads

```
dim3 blocks (DIM/16,DIM/16);  
dim3 threads (16,16);  
while(1){  
    kernel<<<blocks,threads>>>( gpu_bitmap , ticks );  
    ticks++;  
    //copy gpu_bitmap -> cpu_bitmap  
    //visualize cpu_bitmap  
}
```



Threads, Generación de imágenes

- Dividimos la imágenes en bloques de threads

```
__global__ void kernel( unsigned char *ptr, int ticks ) {  
    // map from threadIdx/BlockIdx to pixel position  
    int x = threadIdx.x + blockIdx.x * blockDim.x;  
    int y = threadIdx.y + blockIdx.y * blockDim.y;  
    int offset = x + y * blockDim.x * gridDim.x;  
  
    // now calculate the value at that position  
    float fx = x - DIM/2;  
    float fy = y - DIM/2;  
    float d = sqrtf( fx * fx + fy * fy );  
    float fact=16.0f;  
    if(d<fact)  
        fact=d;  
    unsigned char color = (unsigned char)((255.0f * cos(d/2.0f - ticks/8.0f))/(d/fact));  
    ptr[offset*4 + 0] = 0;  
    ptr[offset*4 + 1] = color*0.88f;  
    ptr[offset*4 + 2] = color*0.92f;  
    ptr[offset*4 + 3] = 255;  
}
```



Threads, Generación de imágenes

- Dividimos la imágenes en blocks de threads

```
__global__ void kernel( unsigned char *ptr, int ticks ) {  
    // map from threadIdx/BlockIdx to pixel position  
    int x = threadIdx.x + blockIdx.x * blockDim.x;  
    int y = threadIdx.y + blockIdx.y * blockDim.y;  
    int offset = x + y * blockDim.x * gridDim.x;  
  
    // now calculate the value at that position  
    float fx = x - DIM/2;  
    float fy = y - DIM/2;  
    float d = sqrtf( fx * fx + fy * fy );  
    float fact=16.0f;  
    if(d<fact)  
        fact=d;  
    unsigned char color = (unsigned char)((255.0f * cos(d/2.0f - ticks/8.0f))/(d/fact));  
    ptr[offset*4 + 0] = 0;  
    ptr[offset*4 + 1] = color*0.88f;  
    ptr[offset*4 + 2] = color*0.92f;  
    ptr[offset*4 + 3] = 255;  
}
```



Threads, Generación de imágenes

- Dividimos la imágenes en blocks de threads

```
__global__ void kernel( unsigned char *ptr, int ticks ) {  
    // map from threadIdx/BlockIdx to pixel position  
    int x = threadIdx.x + blockIdx.x * blockDim.x;  
    int y = threadIdx.y + blockIdx.y * blockDim.y;  
    int offset = x + y * blockDim.x * gridDim.x;  
  
    // now calculate the value at that position  
    float fx = x - DIM/2;  
    float fy = y - DIM/2;  
    float d = sqrtf( fx * fx + fy * fy );  
    float fact=16.0f;  
    if(d<fact)  
        fact=d;  
    unsigned char color = (unsigned char)((255.0f * cos(d/2.0f - ticks/8.0f))/(d/fact));  
    ptr[offset*4 + 0] = 0;  
    ptr[offset*4 + 1] = color*0.88f;  
    ptr[offset*4 + 2] = color*0.92f;  
    ptr[offset*4 + 3] = 255;  
}
```



