Programación de GPUs en CUDA: Optimización de aplicaciones 2

Pedro Alonso

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Contenido

Memoria del dispositivo: sincronización

Sincronización de threads dentro del bloque

- Compartir datos entre threads puede suponer la existencia de condiciones de carrera ya que no podemos asegurar que se ejecuten todos los threads físicamente al mismo tiempo.
- El siguiente código, que da la vuelta a un vector de 64 elementos, puede conducir a error ya que es posible que threads de un *Warp* lean datos de la memoria *shared* (linea 6) antes de que sean escritos por los threads correspondientes (linea 5).

```
1  __global__ void reverse( int *d, int n ) {
2    __shared__ int s[64];
3    int t = threadIdx.x;
4    int tr = n-t-1;
5    s[t] = d[t];
6    d[t] = s[tr];
7  }
8  reverse<<< 1, 64 >>>( d_d, 64 );
```

- El código correcto consistiría en realizar la modificación siguiente:

```
...
s[t] = d[t];
__syncthreads()
d[t] = s[tr];
...
```

Sincronización de threads fuera del bloque

- Extendemos el ejemplo anterior a vectores largos (múltiplos del tamaño de bloque por simplicidad) donde el vector `e` será el reverso de `d`.

```
1  #define BLOCKSIZE 64
2  __global__ void globalreverse(int *d, int *e, int n) {
3    __shared__ int s[BLOCKSIZE];
4    /* Intercambio dentro del bloque de threads */
5    int t = threadIdx.x + blockIdx.x * blockDim.x;
6    s[threadIdx.x] = d[t];
7    __syncthreads();
8    d[t] = s[blockDim.x-threadIdx.x-1];
9    /* Intercambio de bloques */
10   int b = threadIdx.x + ( gridDim.x - blockIdx.x - 1 ) * blockDim.x;
11   e[t] = d[b];
12 }
13 ...
14 globalreverse<<< n/BLOCKSIZE, BLOCKSIZE >>>(d_d, n);
```

- El ejemplo anterior no funciona: los kernels asumen que existe una sincronización en la línea 9 que no existe.

Sincronización de threads fuera del bloque

- La solución está en dividir el procedimiento en dos kernels dado que, si existe dependencia entre kernels, esta se respeta.

```
1  #define BLOCKSIZE 64
2  __global__ void globalreverse1(int *d, int n) {
3    __shared__ int s[BLOCKSIZE];
4    /* Intercambio dentro del bloque de threads */
5    int t = threadIdx.x + blockIdx.x * blockDim.x;
6    s[threadIdx.x] = d[t];
7    __syncthreads();
8    d[t] = s[blockDim.x-threadIdx.x-1];
9  }
10
11 __global__ void globalreverse2(int *d, int *e, int n) {
12 int t = threadIdx.x + blockIdx.x * blockDim.x;
13 /* Intercambio de bloques */
14 int b = threadIdx.x + ( gridDim.x - blockIdx.x - 1 ) * blockDim.x;
15 e[t] = d[b];
16}
17 ...
18 globalreverse1<<< n/BLOCKSIZE, BLOCKSIZE >>>(d_d, n);
19 globalreverse2<<< n/BLOCKSIZE, BLOCKSIZE >>>(d_d, n);
```

Primitivas a nivel de *Warp* (Warp-level primitives)

- Intercambio de datos sincronizado
- Consulta de máscara activa
- Sincronización de *Warp*

Primitivas a nivel de *Warp*

- Las GPU de NVIDIA ejecutan grupos de threads conocidos como *Warps* en modo *SIMT*.
- Es posible incrementar el rendimiento aprovechando la ejecución a nivel de *Warp*.
- En este tema mostramos cómo usar las primitivas introducidas en CUDA 9[1].
- Las GPU de NVIDIA y el modelo de programación CUDA emplean un modelo de ejecución denominado *SIMT* (Single Instruction, Multiple Thread).
- *SIMT* extiende la taxonomía de arquitecturas informáticas de Flynn a una quinta categoría.
- Una de las cuatro clases de Flynn, SIMD (Single Instruction, Multiple Data), se utiliza habitualmente para describir arquitecturas como las GPU. Sin embargo, existe una diferencia sutil pero importante.
- En una arquitectura **SIMD**, cada instrucción aplica la misma operación en paralelo a muchos elementos de datos, mientras que en una arquitectura ***SIMT***, en lugar de que un único thread emita instrucciones vectoriales aplicadas a vectores de datos, varios threads emiten instrucciones comunes a datos arbitrarios.

[1] Este contenido ha sido obtenido de este blog.

Primitivas a nivel de *Warp*

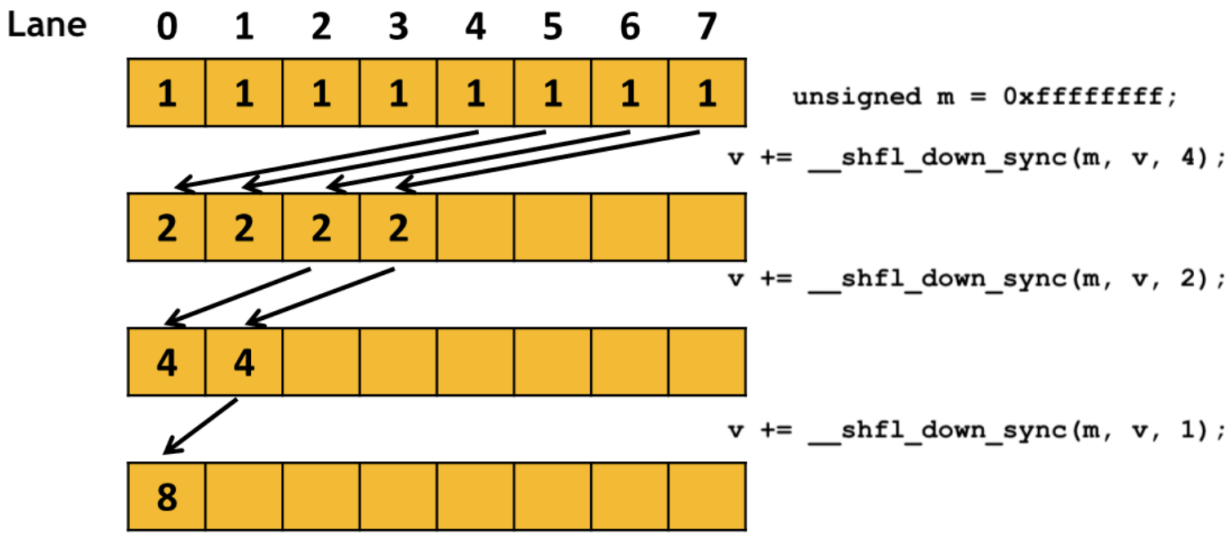
- Las GPU NVIDIA ejecutan *Warps* de 32 threads paralelos mediante SIMT.
- Cada thread accede a sus propios registros, carga y almacena datos desde direcciones divergentes y sigue rutas de flujo de control divergentes.
- El compilador de CUDA y la GPU trabajan para garantizar que los threads de un *Warp* ejecuten las mismas secuencias de instrucciones con la mayor frecuencia posible.
- CUDA admite operaciones colectivas al proporcionar primitivas a nivel *Warp* y operaciones colectivas de *Grupos Cooperativos* (*Cooperative Groups*).

Primitivas a nivel de *Warp*

- El código siguiente muestra un ejemplo del uso de primitivas a nivel de *Warp*.

```
#define FULL_MASK 0xffffffff
for (int offset = 16; offset > 0; offset /= 2)
    val += __shfl_down_sync(FULL_MASK, val, offset);
```

- Utiliza `__shfl_down_sync()` para realizar una reducción en árbol y calcular la suma de la variable `val` de cada thread en un *Warp*.
- Al final del bucle, el valor `val` del primer thread del *Warp* contiene la suma.

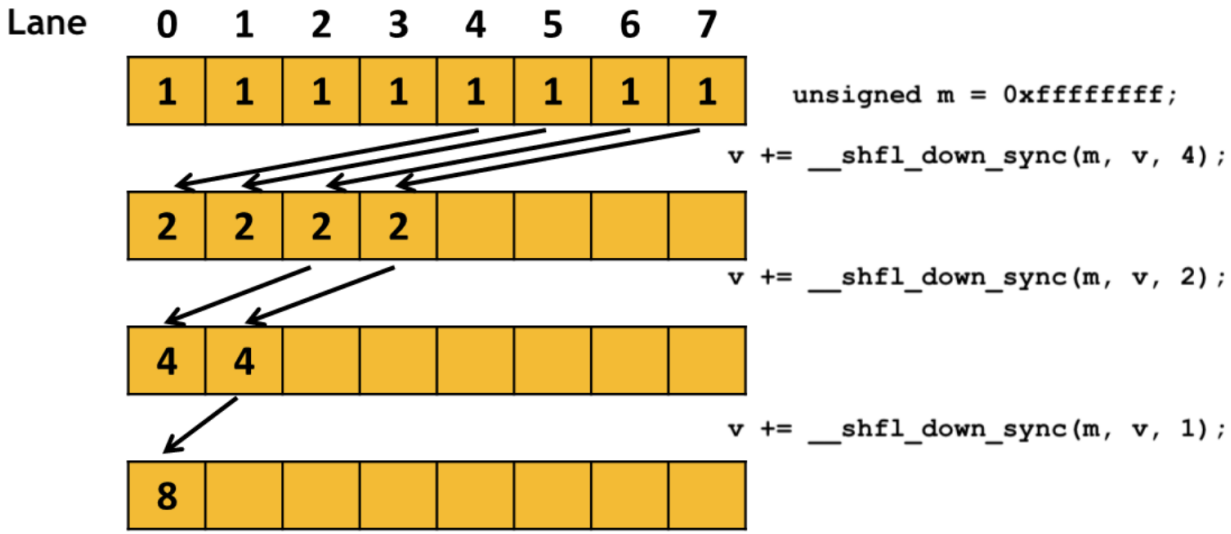


Primitivas a nivel de *Warp*

- Un *Warp* consta de 32 pistas (*lane*), y cada thread ocupa uno.
- Para un thread en la pista `X` del *Warp*,

```
__shfl_down_sync(FULL_MASK, val, offset)
```

  obtiene el valor de la variable `val` del thread en la pista `X+offset` del mismo *Warp*.
- El intercambio de datos se realiza entre registros y es más eficiente que usar memoria compartida.



Primitivas a nivel de *Warp*

CUDA 9 introdujo tres categorías de primitivas de nivel *Warp*:

1. Intercambio sincronizado de datos: intercambio de datos entre threads en *Warp*.
   - `__all_sync`, `__any_sync`, `__uni_sync`, `__ballot_sync`
   - `__shfl_sync`, `__shfl_up_sync`, `__shfl_down_sync`, `__shfl_xor_sync`
   - `__match_any_sync`, `__match_all_sync`
2. Consulta de máscara activa: devuelve una máscara de 32 bits que indica qué threads en un *Warp* están activos, es decir, con el thread en ejecución.
   - `__activemask`
3. Sincronización de threads: sincroniza threads en un *Warp* y proporciona una barrera de memoria.
   - `__syncwarp`

Primitivas a nivel de *Warp*: intercambio de datos sincronizado

- Cada una de las primitivas de "intercambio sincronizado de datos" realiza una operación colectiva entre un conjunto de threads en un *Warp*.

```
int __shfl_sync(unsigned mask, int val, int src_line, int width=warpSize
    );
int __shfl_down_sync(unsigned mask, int var, unsigned delta, int width=
    warpSize);
int __ballot_sync(unsigned mask, int predicate);
```

- Por ejemplo, cada thread que llama a `__shfl_sync()` o `__shfl_down_sync()` recibe datos de un thread en el mismo *Warp*, y cada thread que llama a `__ballot_sync()` recibe una máscara de bits que representa todos los threads en el *Warp* que pasan un valor verdadero para el argumento de predicado.
- El conjunto de threads que participan en la invocación de cada primitiva se especifica mediante una máscara de 32 bits, que constituye el primer argumento de estas primitivas.
- Todos los threads participantes deben estar sincronizados para que la operación colectiva funcione correctamente, por lo tanto, estas primitivas sincronizan primero los threads si aún no lo están.

Primitivas a nivel de *Warp*: intercambio de datos sincronizado

Ejemplo: Calcular la suma de todos los elementos de una matriz `input[]`, cuyo tamaño `NUM_ELEMENTS` es menor que el número de threads en el bloque de threads.

Reducción de un vector en un bloque de threads

```
unsigned mask = __ballot_sync(FULL_MASK, threadIdx.x < NUM_ELEMENTS);
if (threadIdx.x < NUM_ELEMENTS) {
    val = input[threadIdx.x];
    for (int offset = 16; offset > 0; offset /= 2)
        val += __shfl_down_sync(mask, val, offset);
}
```

- El código utiliza la condición `thread.idx.x < NUM_ELEMENTS` para determinar si un thread participará o no en la reducción.
- `__ballot_sync()` se utiliza para calcular la máscara de pertenencia para la operación `__shfl_down_sync()`.
- `__ballot_sync()` en sí utiliza `FULL_MASK` (`0xffffffff` para 32 threads) porque asumimos que todos los threads lo ejecutarán.

Primitivas a nivel de *Warp*: consulta de máscara activa

- `__activemask()` devuelve una máscara de 32 bits (`unsigned int`) de todos los threads actualmente activos en el *Warp* que realiza la llamada.
- El problema del *modelo de ejecución de CUDA*:

Ejemplo incorrecto: Reducción de un vector en un bloque de threads

```
if (threadIdx.x < NUM_ELEMENTS) {
    unsigned mask = __activemask();
    val = input[threadIdx.x];
    for (int offset = 16; offset > 0; offset /= 2)
        val += __shfl_down_sync(mask, val, offset);
}
```

- El problema está en que no se puede asegurar que el `if` sea ejecutado por todos los threads al mismo tiempo.

Primitivas a nivel de *Warp*: Sincronización de *Warps*

- Cuando los threads de un *Warp* necesitan realizar comunicaciones u operaciones colectivas más complejas que las que ofrecen las primitivas de intercambio de datos, se puede usar la primitiva `__syncwarp()` para sincronizar los threads de un *Warp* (similar a la primitiva `__syncthreads()`).

```
void __syncwarp(unsigned mask=FULL_MASK);
```

- La primitiva `__syncwarp()` hace que el thread en ejecución espere hasta que todos los threads especificados en la máscara hayan ejecutado la misma primitiva.

Ejemplo: Transposición de una matriz de 4×8

```
float val = get_value(...);
__shared__ float smem[4][8];
int x1 = threadIdx.x % 8;
int y1 = threadIdx.x / 8;
int x2 = threadIdx.x / 4;
int y2 = threadIdx.x % 4;
smem[y1][x1] = val;
__syncwarp();
val = smem[y2][x2];
```

Primitivas a nivel de *Warp*: Sincronización de *Warps*

Consejos de uso:

- Asegurarse de que se utilizan las primitivas propuestas en CUDA 9.0. (Utilizan una máscara como primer argumento y llevan sufijo `_sync`).
- Pensar como alternativa en la utilización de Grupos Cooperativos (*Cooperative Groups*).

Operaciones atómicas

Operaciones atómicas

- Una función atómica realiza una operación atómica de lectura, modificación y escritura en una palabra de 32 o 64 bits que reside en la memoria global o compartida.
- La operación es atómica en el sentido de que se garantiza su ejecución sin interferencias de otros threads.
- Se aplica la exclusión mutua, lo que elimina la posibilidad de que se produzcan actualizaciones perdidas como resultado de condiciones de carrera.
- CUDA ofrece un conjunto completo de funciones atómicas para realizar operaciones aritméticas.
- Las operaciones atómicas de memoria son un mecanismo que alivia las condiciones de carrera y los problemas de sincronización en el acceso a memoria.
- El rendimiento se reduce cuando muchos threads intentan realizar operaciones atómicas en un número reducido de ubicaciones.
- Muy útiles en aplicaciones típicas como el cálculo de histogramas.

Operaciones atómicas: `atomicAdd`

- Lee el valor `old` de 16, 32 o 64 bits ubicado en la dirección de la memoria global o compartida, calcula (`old + val`) y almacena el resultado en la memoria en la misma dirección.
- Las tres operaciones se realizan en una sola transacción atómica.
- La función devuelve `old`.

`atomicAdd`

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                 unsigned long long int val);
float atomicAdd(float* address, float val);
double atomicAdd(double* address, double val);
__half2 atomicAdd(__half2 *address, __half2 val);
__half atomicAdd(__half *address, __half val);
__nv_bfloat162 atomicAdd(__nv_bfloat162 *address, __nv_bfloat162 val);
__nv_bfloat16 atomicAdd(__nv_bfloat16 *address, __nv_bfloat16 val);
float2 atomicAdd(float2* address, float2 val);
float4 atomicAdd(float4* address, float4 val);
```

Operaciones atómicas: ejemplo de uso

Suma de elementos de un vector (con condición de carrera)

```
__global__ void vectSumRace( int* d_vect, size_t size, int* result ) {
    size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
    while( tid < size ) {
        *result + = d_vect[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

Suma de elementos de un vector correcta

```
__global__ void vectSumAtomic( int* d_vect, size_t size, int* result ) {
    size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
    while( tid < size ) {
        atomicAdd( result, d_vect[tid] );
        tid += blockDim.x * gridDim.x;
    }
}
```

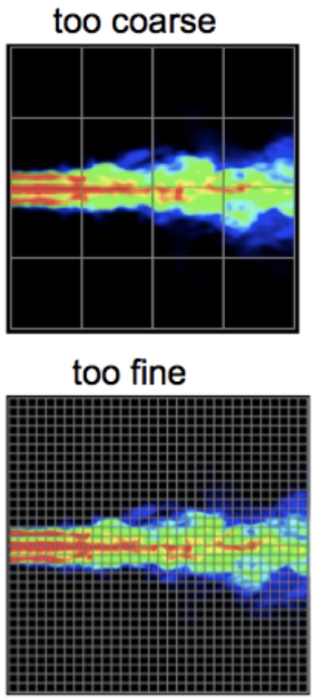
Operaciones atómicas: ejemplo de uso

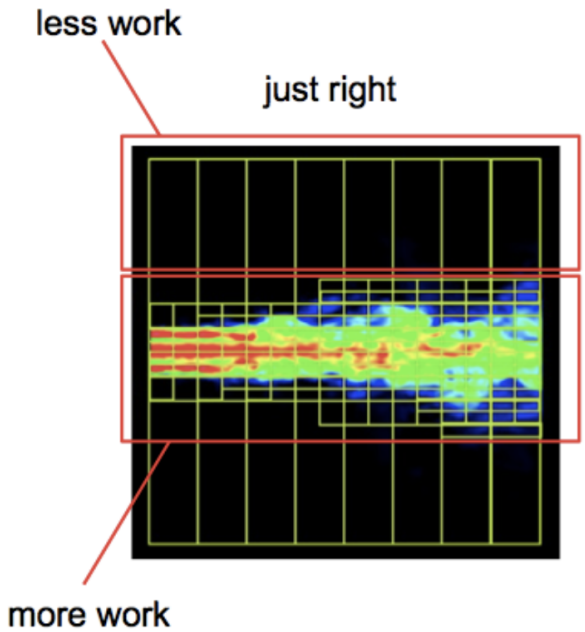
- Las operaciones atómicas pueden aplicarse a memoria del dispositivo y memoria *shared*.
- Existen varias aritméticas: `atomicSub`, `atomicMin`, `atomicMax`, `atomicInc`, `atomicDec`.
- `int atomicExch( int* address, int val )`: Lee la palabra `old` de 32 bits o 64 bits ubicada en la dirección `address` en la memoria global o compartida y almacena `val` en la memoria en la misma dirección.
- `int atomicCAS(int* address, int compare, int val)`: Lee la palabra `old` de 32 o 64 bits ubicada en la dirección `address` en la memoria global o compartida, calcula `old == compare ? val : old` y almacena el resultado en la memoria en la misma dirección. La función devuelve `old` (Comparar e intercambiar).
- También existen operaciones atómicas a nivel de bit: `atomicAnd`, `atomicOr`, `atomicOr`, `atomicXor`.

Paralelismo dinámico

Paralelismo dinámico: introducción

- Los programas CUDA deben ajustarse a un modelo de programación paralela masiva y plana.
- Algunos patrones de paralelismo, como el paralelismo anidado, no se pueden expresar con facilidad.
- El paralelismo anidado surge de forma natural en muchas aplicaciones, como las que utilizan cuadrículas adaptativas, que se utilizan en aplicaciones para reducir la complejidad computacional y capturar el nivel de detalle relevante.
- Las aplicaciones paralelas masivas y planas deben utilizar una cuadrícula fina y realizar cálculos no deseados, o una cuadrícula gruesa y perder detalles más finos.

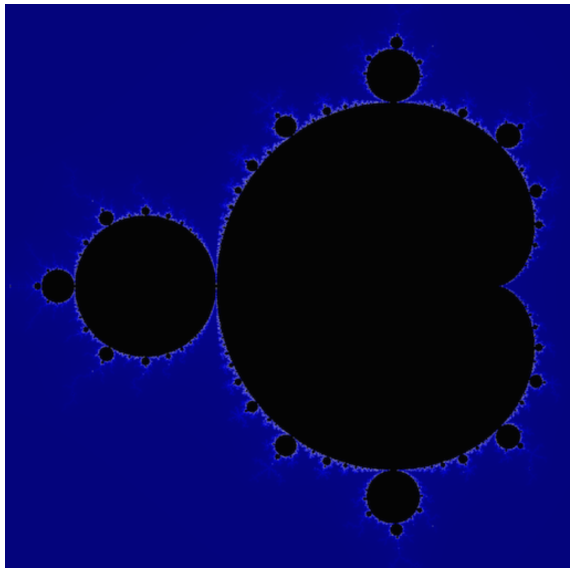




- El *Paralelismo Dinámico* permite ejecutar kernels desde threads que se ejecutan en el dispositivo; los threads pueden ejecutar más threads.

Paralelismo dinámico: introducción

- El paralelismo dinámico suele ser útil para problemas donde no se puede evitar el paralelismo anidado. Por ejemplo:
  - Algoritmos que utilizan estructuras de datos jerárquicas, como las cuadrículas adaptativas;
  - Algoritmos que utilizan recursión, donde cada nivel de recursión tiene paralelismo, como quicksort;
  - Algoritmos donde el trabajo se divide naturalmente en lotes independientes, donde cada lote implica un procesamiento paralelo complejo, pero no puede utilizar completamente una sola GPU.
- El *paralelismo dinámico* fue introducido en **CUDA 5.0** y puede utilizarse en dispositivos con capacidad de cómputo 3.5 o superior.
- Vamos a utilizar un caso de estudio: *El conjunto de Mandelbrot*[a].

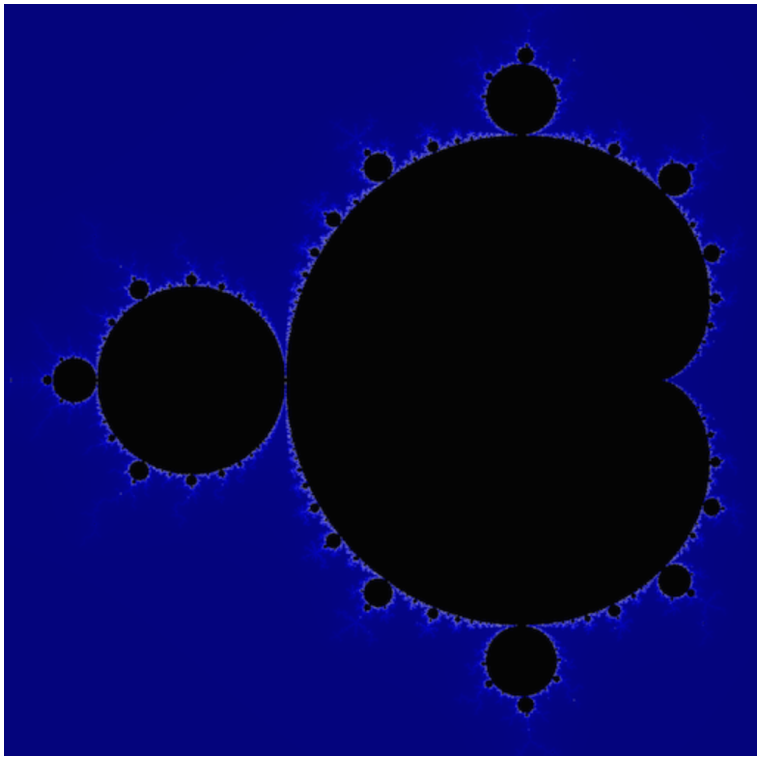


[a] La información de estas transparencias ha sido obtenida de aquí.

Paralelismo dinámico: Mandelbrot

El conjunto de Mandelbrot se define como:

$$\begin{aligned} z_0 &= c \\ z_{n+1} &= z_n^2 + c \\ M &= \{c \in \mathbb{C} : \exists\ R\ \forall n : |z_n| < R\} \end{aligned}$$



Paralelismo dinámico: Mandelbrot

El Algoritmo Escape Time

```
#define MAX_DWELL 512
// w, h       --- width and hight of the image, in pixels
// cmin, cmax --- coordinates of bottom-left and top-right image corners
// x, y       --- coordinates of the pixel
__host__ __device__ int pixel_dwell( int w, int h, complex cmin, complex cmax, int x, int y ) {
  complex dc = cmax - cmin;
  float fx = (float)x / w, fy = (float)y / h;
  complex c = cmin + complex( fx * dc.re, fy * dc.im );
  complex z = c;
  int dwell = 0;

  while( dwell < MAX_DWELL && abs2(z) < 2 * 2 ) {
    z = z * z + c;
    dwell++;
  }

  return dwell;
}
```

- Si el valor de **dwell** es igual a **MAX_DWELL**, el punto pertenece al conjunto y suele ser de color negro.
- Si el valor de **dwell** es menor que **MAX_DWELL**, el punto no pertenece al conjunto y se usa el valor de **dwell** para colorear el píxel.

Paralelismo dinámico: Mandelbrot

Kernel Por-Pixel para el conjunto de Mandelbrot

```
__global__ void mandelbrot_k( int *dwells, int w, int h, complex cmin, complex cmax ) {
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
  if( x < w && y < h )
    dwells[y * w + x] = pixel_dwell( w, h, cmin, cmax, x, y );
}

int main(void) {

  // ...
  int w = 4096, h = 4096;
  dim3 bs(64, 4), grid(divup(w, bs.x), divup(h, bs.y));
  mandelbrot_k<<< grid, bs >>>( d_dwells, w, h, complex(-1.5, 1), complex(0.5, 1));
  // ...
}
```

- Este algoritmo desperdicia muchos recursos computacionales.
- Existen grandes regiones dentro del conjunto que podrían simplemente colorearse de negro. Dado que se realizan `MAX_DWELL` iteraciones para cada píxel negro, es aquí donde el algoritmo dedica la mayor parte del tiempo de cálculo.
- También existen grandes regiones con un `dwell` constante pero bajo fuera del conjunto de Mandelbrot.
- Las únicas áreas donde se requiere un cálculo de alta resolución se encuentran a lo largo del límite fractal del conjunto.

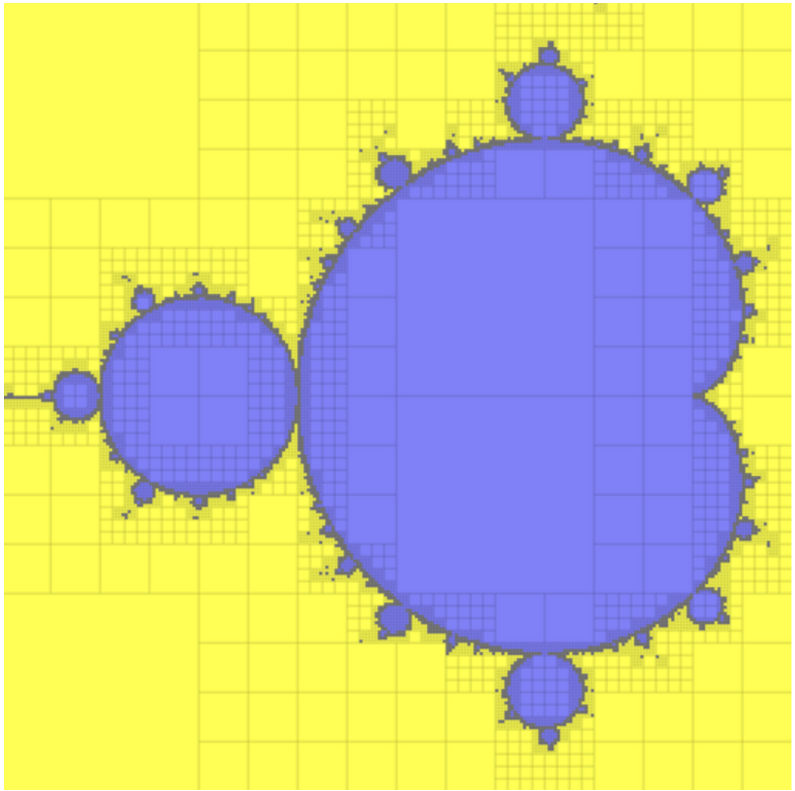
Paralelismo dinámico: Mandelbrot

- Podemos aprovechar las grandes regiones de `dwell` uniforme mediante el algoritmo jerárquico de Mariani-Silver para concentrar el cálculo donde más se necesita.
- Este algoritmo se basa en que el conjunto de Mandelbrot es **conexo**: *existe un camino entre dos puntos cualesquiera que pertenezcan al conjunto*, por tanto, si existe una región (de cualquier forma) cuyo borde se encuentra completamente dentro del conjunto de Mandelbrot, toda la región pertenece a él.
- Si el borde de la región tiene `dwell` constante, cada píxel de la región tiene el mismo `dwell`, por tanto, si el `dwell` es uniforme, basta con evaluarlo en el borde.
- El algoritmo de Mariani-Silver combina este principio con la subdivisión recursiva de regiones con `dwell` no constante:

El Algoritmo de Mariani-Silver

```
mariani_silver(rectangle)
  if (border(rectangle) has common dwell)
        fill rectangle with common dwell
  else if (rectangle size < threshold)
    per-pixel evaluation of the rectangle
  else
    for each sub_rectangle in subdivide(rectangle)
        mariani_silver(sub_rectangle)
```

Paralelismo dinámico: Mandelbrot



- Este algoritmo se ajusta bien al Paralelismo Dinámico de CUDA.
- Cada llamada a `mariani_silver` se asigna a un bloque de threads del kernel `mandelbrot_k`.
- Los threads del bloque utilizan una *reducción paralela* para determinar si todos los píxeles del borde tienen el mismo `dwell`.
- El subproceso `0` en cada bloque decide si rellenar la región, subdividirla o evaluar el `dwell` para cada píxel del rectángulo.

Paralelismo dinámico: Fundamentos

- Necesitamos ejecutar kernels desde los threads que se ejecutan en el dispositivo.
- Los lanzamientos de kernel del dispositivo utilizan la misma sintaxis que los lanzamientos desde el host.
- Al igual que en el host, el lanzamiento del kernel del dispositivo es asíncrono.
- El *Paralelismo Dinámico* utiliza la biblioteca CUDA Device Runtime (`cudadevrt`).
- Todas las funciones de la API devuelven un código de error.

El Algoritmo de Mariani-Silver con Paralelismo Dinámico: `same_dwell`

```
#define NEUT_DWELL (MAX_DWELL + 1)
#define DIFF_DWELL (-1)

__device__ int same_dwell(int d1, int d2) {
  if (d1 == d2)
    return d1;
  else if (d1 == NEUT_DWELL || d2 == NEUT_DWELL)
    return min(d1, d2);
  else
    return DIFF_DWELL;
}
```

Paralelismo dinámico: Fundamentos

El Algoritmo de Mariani-Silver con Paralelismo Dinámico: el kernel

```
#define MAX_DWELL 512
#define BSX 64          /** block size along x direction */
#define BSY 4           /** block size along y direction */
#define MAX_DEPTH 4     /** maximum recursion depth */
#define MIN_SIZE 32     /** size below which we should call the per-pixel kernel */
#define SUBDIV 4        /** subdivision factor along each axis */
#define INIT_SUBDIV 32 /** subdivision factor when launched from the host */

__global__ void mandelbrot_block_k( int *dwells, int w, int h, complex cmin, complex cmax,
                                    int x0, int y0, int d, int depth ) {

  x0 += d * blockIdx.x, y0 += d * blockIdx.y;
  int common_dwell = border_dwell( w, h, cmin, cmax, x0, y0, d );
  if ( threadIdx.x == 0 && threadIdx.y == 0 ) {
    if ( common_dwell != DIFF_DWELL ) { // uniform dwell, just fill
      dim3 bs(BSX, BSY), grid(divup(d, BSX), divup(d, BSY));
      dwell_fill<<< grid, bs >>>( dwells, w, x0, y0, d, comm_dwell );
    } else if ( depth + 1 < MAX_DEPTH && d / SUBDIV > MIN_SIZE ) { // subdivide recursively
      dim3 bs(blockDim.x, blockDim.y), grid(SUBDIV, SUBDIV);
      mandelbrot_block_k<<< grid, bs >>>( dwells, w, h, cmin, cmax, x0, y0, d / SUBDIV, depth+1 );
    } else { // leaf, per-pixel kernel
      dim3 bs(BSX, BSY), grid(divup(d, BSX), divup(d, BSY));
      mandelbrot_pixel_k<<< grid, bs >>>( dwells, w, h, cmin, cmax, x0, y0, d );
    }
    cucheck_dev(cudaGetLastError());
  }
}  // mandelbrot_block_k
```

Paralelismo dinámico: Fundamentos

El kernel `mandelbrot_block_k` utiliza las siguientes funciones y kernels:

`border_dwell`: función que comprueba si el valor de `dwell` es el mismo a lo largo del borde de la región actual; realiza una reducción paralela dentro de un bloque de threads.

`dwell_fill`: establece cada píxel dentro del rectángulo de imagen dado con el valor de `dwell` especificado.

`mandelbrot_pixel_k`: el mismo kernel por píxel que usamos en nuestra primera implementación, pero aplicado solo dentro de un rectángulo de imagen específico.

El Algoritmo de Mariani-Silver con Paralelismo Dinámico: llamada al kernel

```
int main(void) {
  // ...
  int width = 8192, height = 8192;
  mandelbrot_block_k<<< dim3(init_subdiv, init_subdiv), dim3(bsx, bsy) >>>
          ( dwells, width, height, complex(-1.5, -1), complex(0.5, 1), 0, 0,
              width / INIT_SUBDIV );
  // ...
}
```

Paralelismo dinámico: compilación y enlazado

El kernel `mandelbrot_block_k` utiliza las siguientes funciones y kernels:

- Para usar *Paralelismo Dinámico* se debe compilar el código para una *Compute Capability* 3.5 o superior y enlazarlo con la biblioteca `cudadevrt`.
- Para compilar se debe especificar la opción `-rdc=true` (`-relocatable-device-code=true`) para generar el código del dispositivo reubicable, necesario para la vinculación posterior.

Generación del ejecutable

```
nvcc -arch=sm_35 -rdc=true myprog.cu -lcudadevrt -o myprog.o
```

Grupos cooperativos
(*Cooperative groups*)

Cooperative groups: Introducción

- En algoritmos paralelos eficientes, los threads cooperan y comparten datos para realizar cálculos colectivos.
- Para compartir datos, los threads deben sincronizarse.
- La granularidad de la compartición varía según el algoritmo, por lo que la sincronización de threads debe ser flexible.
- Incluir la sincronización explícitamente en el programa garantiza la seguridad, la mantenibilidad y la modularidad.
- **CUDA 9** introduce los *Grupos Cooperativos*, cuyo objetivo es satisfacer estas necesidades ampliando el modelo de programación de CUDA para permitir que los kernels organicen dinámicamente grupos de threads.
- El modelo de programación CUDA ha proporcionado una construcción única y sencilla para sincronizar threads cooperantes: la función `__syncthreads()`.
- Sin embargo, a menudo es necesario definir y sincronizar grupos de threads más pequeños que los bloques de threads para lograr un mayor rendimiento, flexibilidad de diseño y reutilización del software mediante interfaces de función colectivas para todo el grupo.

Cooperative groups: Introducción

- El modelo de programación de Grupos Cooperativos describe patrones de sincronización tanto dentro como entre bloques de threads de CUDA.
- Proporciona una API de código de dispositivo CUDA para definir, particionar y sincronizar grupos de threads.
- También proporciona API del lado del host para iniciar redes cuyos threads garantizan la ejecución concurrente, lo que permite la sincronización entre bloques de threads.
- Esta herramienta mejora la creación de software: las funciones colectivas pueden tomar un argumento explícito que representa el grupo de threads participantes.
- El modelo de programación de Grupos Cooperativos consta de los siguientes elementos:
  - **Tipos de datos** que representan grupos de threads cooperativos y sus propiedades;
  - **Grupos intrínsecos** definidos por la API de inicio de CUDA.
  - **Operaciones de partición** de grupos;
  - Una operación de **sincronización de barreras** de grupo;
  - **Operaciones colectivas específicas** de cada grupo.

Fuente: https://developer.nvidia.com/blog/cooperative-groups/

Cooperative groups: Fundamentos

- Para utilizar Grupos Cooperativos hay que incluir el archivo de cabecera.

```
#include <cooperative_groups.h>
```

- Los tipos e interfaces de grupos cooperativos se definen en el espacio de nombres de C++ `cooperative_groups`.

```
using namespace cooperative_groups;
```

- No es raro usar un alias más corto.

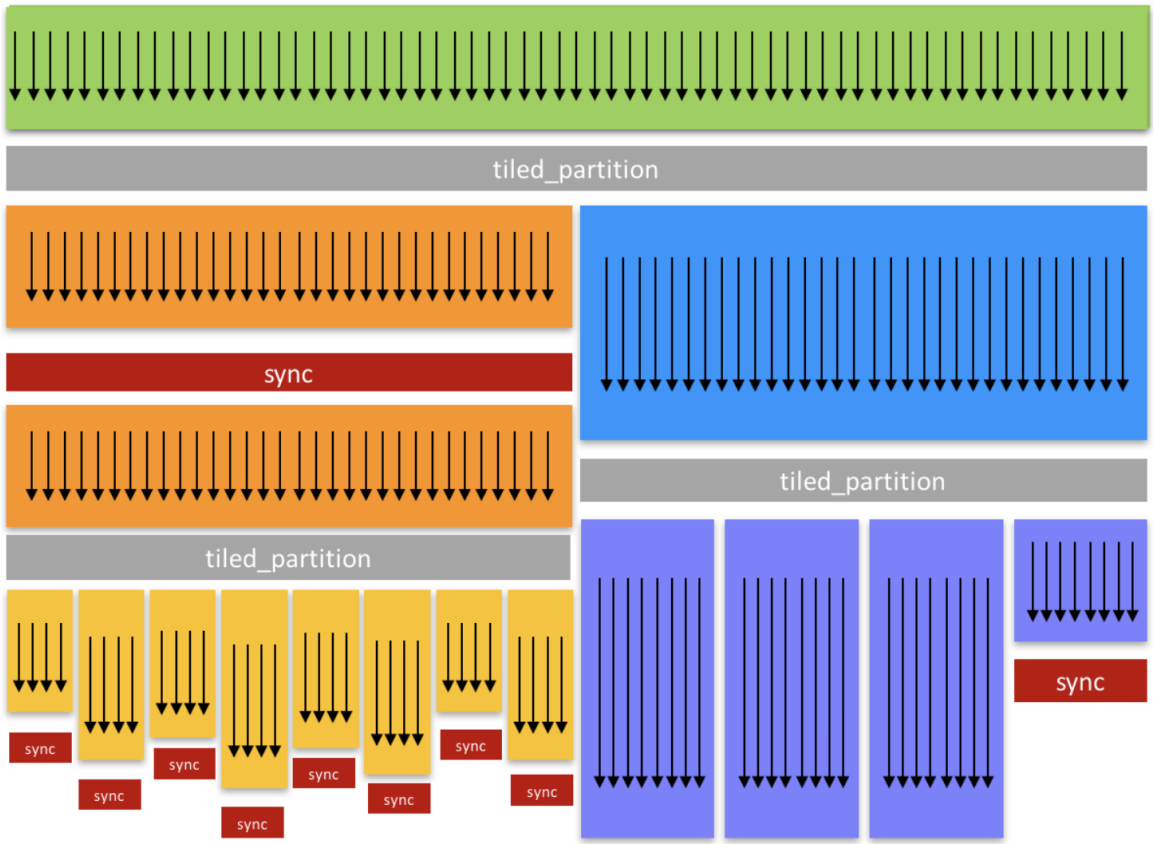
```
namespace cg = cooperative_groups;
```

- El tipo fundamental en los Grupos Cooperativos es **`thread_group`**, que es un identificador de un grupo de threads. Este identificador solo es accesible para los miembros del grupo al que representa.
- Se puede obtener el número total de threads de un grupo con el método **`size()`**.
- **`thread_rank()`**: Devuelve el índice del thread dentro del grupo (entre `0` y `size()-1`).
- **`is_valid()`**: Comprueba la validez de un grupo.

Cooperative groups: Operaciones colectivas

- Los grupos de threads permiten realizar operaciones colectivas entre todos los threads de un grupo.
- Las operaciones colectivas son operaciones que necesitan sincronizarse o comunicarse entre un conjunto específico de threads.

- Debido a la necesidad de sincronización, cada thread identificado como participante en un colectivo debe realizar una llamada correspondiente a dicha operación.
- Se puede sincronizar un grupo llamando al método colectivo **`sync()`**.



Cooperative groups: Operaciones colectivas

- Ejemplo sencillo de una función de `device` para reducción paralela escrita con Grupos Cooperativos.
- Cuando los threads de un grupo la llaman, calculan cooperativamente la suma de los valores pasados por cada thread del grupo (argumento `val`).

Ejemplo de reducción paralela con grupos cooperativos

```
using namespace cooperative_groups;
__device__ int reduce_sum( thread_group g, int *temp, int val ) {
    int lane = g.thread_rank();

    // Each iteration halves the number of active threads
    // Each thread adds its partial sum[i] to sum[lane+i]
    for ( int i = g.size() / 2; i > 0; i /= 2 ) {
        temp[lane] = val;
        g.sync(); // wait for all threads to store
        if( lane < i ) val += temp[lane + i];
        g.sync(); // wait for all threads to load
    }
    return val; // note: only thread 0 will return full sum
}
```

Cooperative groups: Operaciones colectivas

- Los Grupos Cooperativos introducen un nuevo tipo de dato, **`thread_block`**, para representar explícitamente este concepto dentro de un kernel.
- Una instancia de **`thread_block`** es un identificador del grupo de threads en un bloque de threads CUDA que se inicializa de la siguiente manera.

```
thread_block block = this_thread_block();
```

- Cada thread que ejecuta esa línea tiene su propia instancia de la variable **`block`**.
- Los threads con el mismo valor de la variable CUDA, **`blockIdx`**, forman parte del mismo grupo de bloques de threads.
- Sincronizar un grupo **`thread_block`** es muy similar a llamar a **`__syncthreads()`**.
- Las siguientes líneas de código hacen lo mismo (suponiendo que todos los threads del bloque de threads las alcanzan).

```
__syncthreads();
block.sync();
cg::synchronize(block);
this_thread_block().sync();
cg::synchronize(this_thread_block());
```

Cooperative groups: Operaciones colectivas

- El tipo de datos **`thread_block`** extiende la interfaz **`thread_group`** con los siguientes métodos específicos del bloque.

```
dim3 group_index();  // 3-dimensional block index within the grid
dim3 thread_index(); // 3-dimensional thread index within the block
```

- Estos son equivalentes a **`blockIdx`** y **`threadIdx`** de CUDA, respectivamente.

Cooperative groups: Operaciones colectivas

Ejemplo de reducción paralela de todos los elementos de un array

```
__device__ int thread_sum( int *input, int n ) {
    int sum = 0;

    for( int i = blockIdx.x * blockDim.x + threadIdx.x;
        i < n;
        i += blockDim.x * gridDim.x ) {
        int in = input[i];
        sum += in;
    }
    return sum;
}

__global__ void sum_kernel_block( int *sum, int *input, int n ) {
    int my_sum = thread_sum( input, n );

    extern __shared__ int temp[];
    auto g = this_thread_block();
    int block_sum = reduce_sum( g, temp, my_sum );

    if ( g.thread_rank() == 0 ) atomicAdd( sum, block_sum );
}
```

Cooperative groups: Operaciones colectivas

El kernel `sum_kernel_block` es llamado según el código siguiente:

Ejemplo de reducción paralela de todos los elementos de un array

```
int n = 1 << 24;
int blockSize = 256;
int nBlocks = ( n + blockSize - 1 ) / blockSize;
int sharedBytes = blockSize * sizeof(int);

int *sum, *data;
cudaMallocManaged( &sum, sizeof(int) );
cudaMallocManaged( &data, n * sizeof(int) );
std::fill_n( data, n, 1 ); // initialize data
cudaMemset( sum, 0, sizeof(int) );

sum_kernel_block<<< nBlocks, blockSize, sharedBytes >>>( sum, data, n );
```