Arquitecturas unificadas en GPUs: CUDA

- Arquitecturas unificadas en GPUs
- Introducción a la tecnología CUDA de NVIDIA
- Gestión de hebras: bloques y grids
- Gestión de memoria y transferencia de datos CPU-GPU
- Ejecución de kernels
- Casos de estudio: investigación en el GGGJ

Arquitecturas especializadas y unificadas

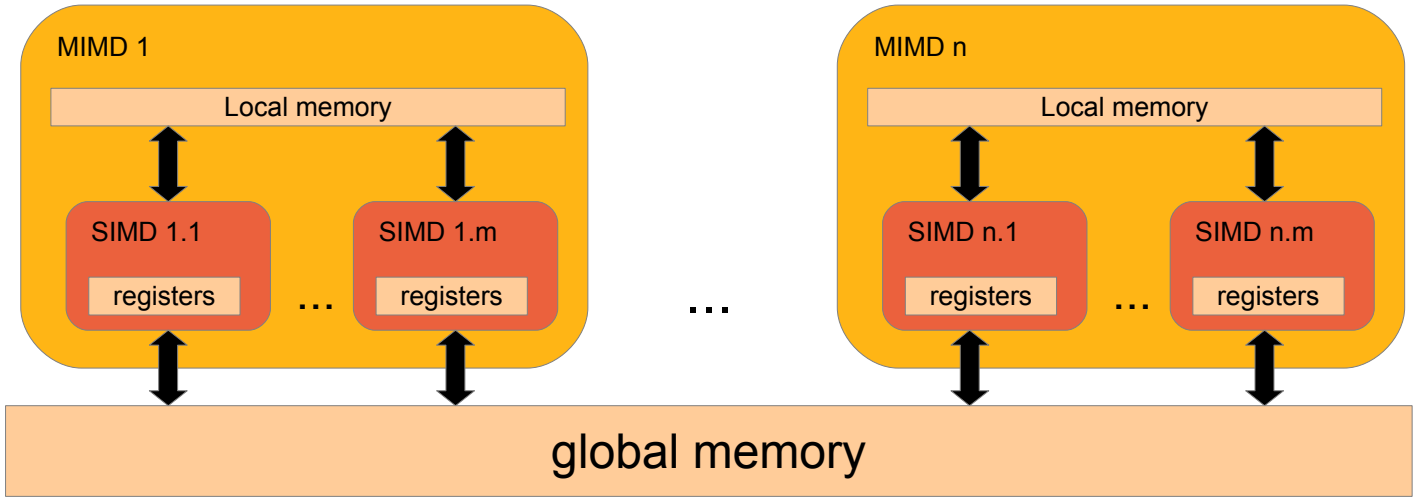
- Las primeras generaciones de GPUs tenían un número fijo de procesadores de shader:

Nvidia GF 6800GT
16 procesadores fragment shader
6 procesadores vertex shader

- A partir de la serie GF 8xxx de Nvidia y HD 4xxx de AMD/ATI, la **arquitectura es unificada**: existe un conjunto de procesadores que pueden trabajar como vertex/geometry/fragment shader indistintamente

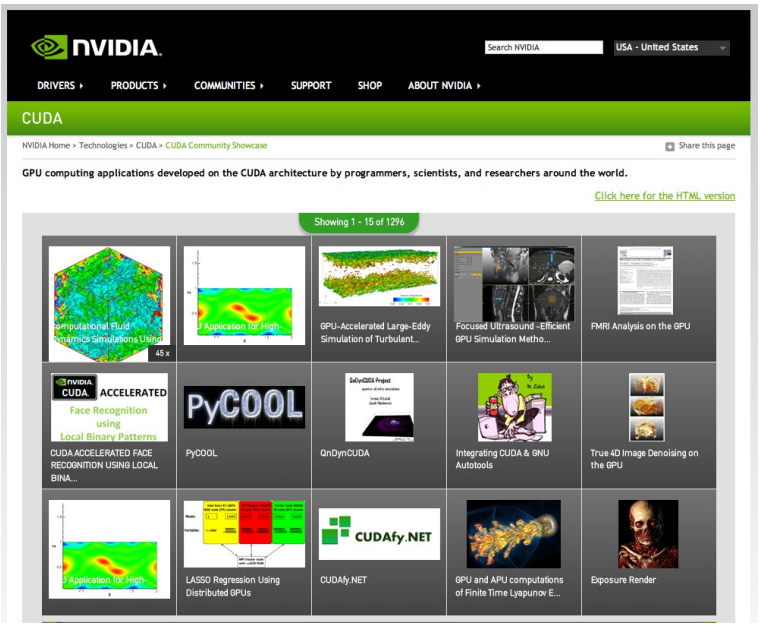
Arquitecturas unificadas

- Las GPUs con arquitecturas unificadas están evolucionando hacia un multiprocesador de propósito general
- Gran número de procesadores SIMD
- Memoria local (muy rápida) y global



GPGPU en arquitecturas unificadas

- Han revolucionado la GPGPU poniéndola al alcance de un público mucho mayor
- Hay ya más de 1000 aplicaciones documentadas para estas arquitecturas



http://www.nvidia.com/object/cuda-apps-flash-new.html

Ventajas

- Las GPUs con arquitecturas unificadas se programan de una manera mucho más sencilla
- No es necesario trabajar con el pipeline gráfico: la programación es más parecida a la tradicional en el host
- La curva de aprendizaje es mucho menor

Desventajas

- Requiere la instalación de un toolkit especial (CUDA, OpenCL)
- La tecnología CUDA sólo funciona en GPUs de Nvidia
- La tecnología OpenCL funciona en todas las plataformas, aunque el rendimiento es inferior a CUDA en GPUs Nvidia
- Algunos problemas todavía pueden ser resueltos de manera más eficiente con shaders

Introducción a Nvidia CUDA

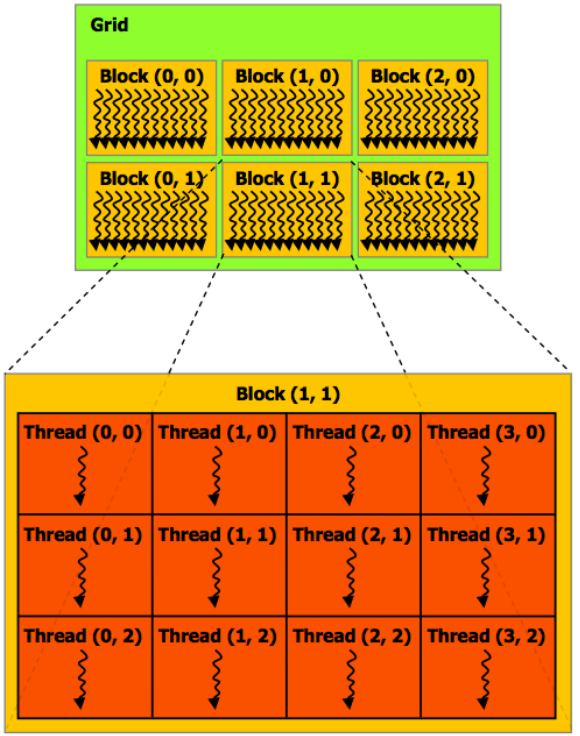
- Aparece en 2006 con las nuevas generaciones de GPUs con arquitectura unificada de Nvidia
- Programable en C/C++ con algunas extensiones
- Toolkit, SDK y documentación gratuitos
- Amplia comunidad de desarrollo en Internet y multitud de aplicaciones existentes

Modelo de programación: hebras, bloques y grids

- CUDA es capaz de ejecutar un número arbitrario de **hebras** (threads)
- Las hebras se agrupan en **bloques** de un número de hebras determinado. Las hebras dentro de un bloque se ejecutan en paralelo en un multiprocesador SIMD
- La ejecución consiste en lanzar a la GPU un conjunto de bloques de hebras (**grid**)

Modelo de programación: hebras, bloques y grids

- Cada hebra tiene un índice único dentro del bloque (1, 2 o 3 dimensiones)
- Cada bloque tiene un índice único dentro del grid (1, 2 o 3 dimensiones)



Modelo de programación: hebras, bloques y grids

- Dentro de una hebra podemos conocer su identificador mediante la variable **threadIdx**
- Podemos conocer el bloque al que pertenece la hebra mediante la variable **blockIdx**
- Las dimensiones del bloque y del grid vienen dadas por las variables **blockDim** y **gridDim**
- Todos tienen tres componentes (x, y, z)

Tipos de memoria

- Las hebras disponen de **memoria local** limitada (muy rápida)
- Las hebras dentro de un bloque disponen de un espacio de **memoria compartida** (rápida)
- Existe también un área de **memoria global** accesible desde cualquier hebra (lenta)
- ¡Ojo con las colisiones entre hebras al acceder a memoria compartida o global!

Interfaz de programación C/C++

- Las aplicaciones CUDA se programan en C/C++ estándar con algunas etiquetas especiales
- El compilador nvcc que viene con el CUDA Toolkit genera el código CPU y GPU correspondiente de manera transparente
- Los ficheros deben tener la extensión .cu

Interfaz de programación: kernels

- Son funciones ejecutadas por cada hebra de la GPU
- Se marcan con el prefijo __global__
- Cada kernel debe usar el identificador de hebra y bloque para acceder a datos diferentes

```
// Suma paralela de dos vectores

__global__ void threadSumaGPU(float *op1, float *op2,
  float *res, int tam)
{
     // Calcular la posición que le corresponde al thread
     int pos = blockIdx.x * blockDim.x + threadIdx.x;

     if (pos < tam)
          res[pos] = op1[pos] + op2[pos];
}
```

Interfaz de programación: kernels

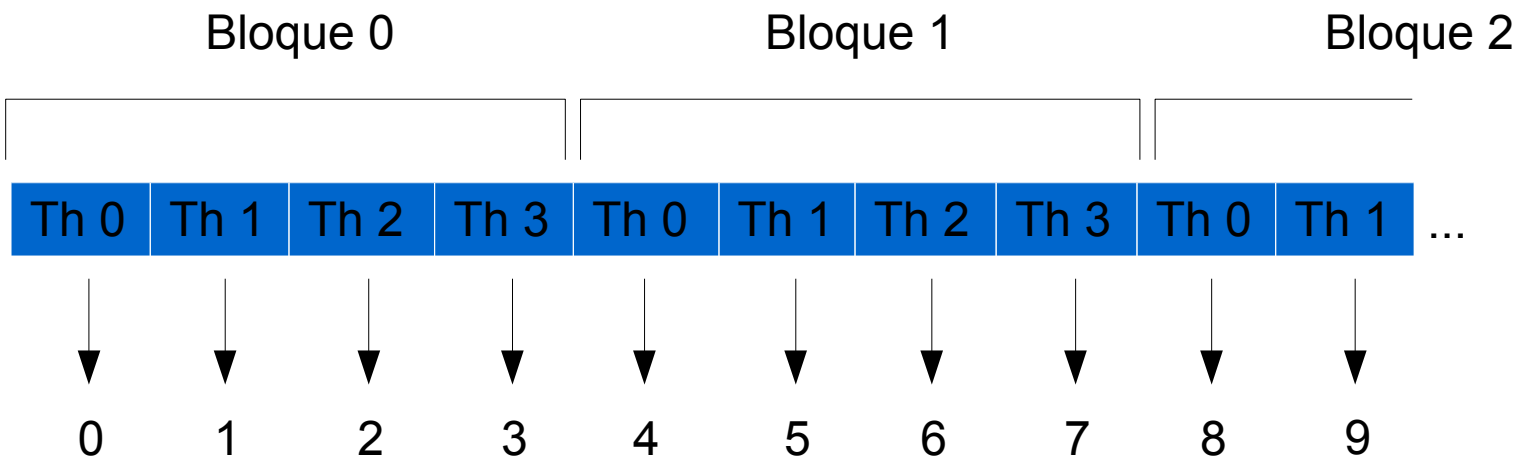
- Son funciones ejecutadas por cada hebra de la GPU
- Se marcan con el prefijo __global__
- Cada kernel debe usar el identificador de hebra y bloque para acceder a datos diferentes

```
// Suma paralela de dos vectores

__global__ void threadSumaGPU(float *op1, float *op2,
  float *res, int tam)
{
     // Calcular la posición que le corresponde al thread
     int pos = blockIdx.x * blockDim.x + threadIdx.x;

     if (pos < tam)
          res[pos] = op1[pos] + op2[pos];
}
```

Identificador de una hebra

- Esta expresión es muy utilizada y permite obtener un índice global único por hebra empezando en cero:

```
int pos = blockIdx.x * blockDim.x + threadIdx.x;
```



Funciones auxiliares en kernels

- Pueden definirse funciones auxiliares en la GPU mediante el prefijo __device__

```
__device__ float suma(float a, float b) {
     return a + b;
}

__global__ void threadSumaGPU(float *op1, float *op2,
  float *res, int tam)
{
     // Calcular la posición que le corresponde al thread
     int pos = blockIdx.x * blockDim.x + threadIdx.x;

     if (pos < tam)
          res[pos] = suma(op1[pos], op2[pos]);
}
```

Acceso a memoria desde kernels

- Las variables locales de un kernel son creadas en memoria local
- Las variables locales de un kernel con el prefijo **__shared__** son creadas en la memoria compartida del multiprocesador
- Los punteros siempre apuntan a bloques en memoria global

```
__global__ void miThread(float *puntGlobal)
{
     int varLocal;
     __shared__ int varCompartida;
}
```

Memoria compartida

- La memoria compartida es mucho más rápida que la memoria global
- Su utilidad es cachear datos de memoria global que deben ser usados por varias hebras de un mismo bloque
- El que pueda ser utilizada depende de la naturaleza del algoritmo

Sincronización de hebras

- El comando __syncthreads() permite asegurar que todas las hebras de un bloque están sincronizadas en un punto
- Útil para saber que todas las hebras han completado una cierta tarea, como por ejemplo copiar datos de memoria global a memoria compartida

Accesos atómicos

- Permite que varios threads operen sobre una posición en memoria global de forma segura:
  - atomicAdd()
  - atomicSub()
  - atomicInc()
  - atomicDec()
- Inconveniente: son ineficientes

```
__global__ void miThread(float *puntGlobal)
{
     // Inseguro!!
     puntGlobal[0] = puntGlobal[0] + 1;
     // Seguro
     atomicInc(puntGlobal[0]);
}
```

Asignación de memoria global en GPU

- Sólo puede hacerse desde el host, nunca desde un kernel
- Similares a la funciones C malloc() y free() para asignar memoria en el host

```
cudaError_t cudaMalloc(void **gpuPtr, size_t numBytes)
cudaError_t cudaFree(void **gpuPtr)
```

Transferencia de datos

- Siempre se realizan en el host
- Similar a la función memcpy() de C

```
cudaError_t cudaMemcpy(void *dst, const *void src, size_t count,
enum cudaMemcpyKind kind)
```

- El parámetro cudaMemcpyKind puede tomar los valores:
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToDevice
  - cudaMemcpyDeviceToHost

Ejecución

- Se realiza siempre en el host
- Primero debe especificarse el tamaño del bloque y del grid (número de bloques)
- A continuación se hace la llamada indicando el nombre del kernel y los argumentos

```
// Especificar el número de hebras por bloque
dim3 threadsPerBlock(64);
// Calcular el número de bloques necesarios en función
// del número de datos a procesar
dim3 numBlocks(ceil((float) tam / threadsPerBlock.x));

// Lanzar ejecución
threadSumaGPU<<<numBlocks, threadsPerBlock>>>(gpuA, gpuB, gpuRes, tam);
```

Resumen

- Pasos a seguir para implementar un algoritmo en CUDA
  1. Definir el kernel
  2. Asignar memoria global en el dispositivo
  3. Transferir datos al dispositivo desde la memoria del host
  4. Lanzar la ejecución
  5. Transferir resultados desde el dispositivo a memoria del host

Ejemplo: suma paralela

```
__global__ void threadSumaGPU(float *op1, float *op2,
  float *res, int tam)
{
  // Calcular la posición que le corresponde al thread
  int pos = blockIdx.x * blockDim.x + threadIdx.x;

  if (pos < tam)
    res[pos] = op1[pos] + op2[pos];
}

void sumaCUDA(float *a, float *b, float *res, int tam)
{
  float *gpuA, *gpuB, *gpuRes;
  int tamBytes = sizeof(float) * tam;

  cudaMalloc(&gpuA, tamBytes);
  cudaMalloc(&gpuB, tamBytes);
  cudaMalloc(&gpuRes, tamBytes);

  cudaMemcpy(gpuA, a, tamBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(gpuB, b, tamBytes, cudaMemcpyHostToDevice);

  dim3 threadsPerBlock(64);
  dim3 numBlocks(ceil((float) tam / threadsPerBlock.x));

  threadSumaGPU<<<numBlocks, threadsPerBlock>>>(gpuA, gpuB, gpuRes, tam);

  cudaMemcpy(res, gpuRes, tamBytes, cudaMemcpyDeviceToHost);
  cudaFree(gpuA);  cudaFree(gpuB); cudaFree(gpuRes);
}
```

Paralelismo dinámico

- Permite lanzar una nueva computación (grid) desde un thread (en lugar de desde el host)
- La ejecución del thread padre termina cuando acaba la del hijo

```
__global__ void kernelHijo(float *datos, int numDatos, int val)
{
     int pos = blockIdx.x * blockDim.x + threadIdx.x;
     if (pos < numDatos)
          datos[pos] = val;
}

__global__ void kernelPadre(float *datos, int numDatos, int val)
{
     kernelHijo<<floor(numDatos/16), 16>>>(datos, numDatos, val);
}

void main() {
     kernelPadre<<<1, 1>>>(datos, numDatos, val);
}
```

Paralelismo dinámico

- Aspectos a tener en cuenta:
  - La memora local y compartida del grid hijo no es visible desde el padre
  - La operación *cudaDeviceSynchronize()* permite sincronizar un grid hijo con el thread padre que lo ha lanzado

```
__global__ void kernelPadre(float *datos, int numDatos, int val)
{
     kernelHijo<<floor(numDatos/16), 16>>>(datos, numDatos, val);
     CudaDeviceSynchronize();

     // La ejecución del grid de kernels kernelHijo ha terminado aquí
     …

}
```

Consejos para obtener un buen rendimiento

- A menor cantidad de datos a transferir a la GPU y cálculo más intensivo computacionalmente, mejor rendimiento
- Para procesamientos sencillos la GPU da un resultado mucho peor que la CPU
- Evitar colisiones: acceso simultáneo de varias hebras a la misma posición de memoria
- Copiar a memoria compartida datos que sean necesarios para varias hebras del bloque

Desarrollo con CUDA

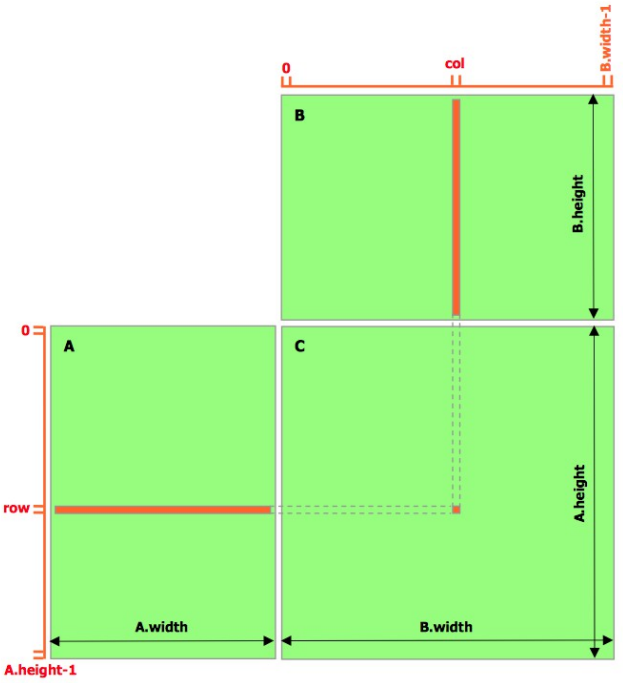
- Es necesario una GPU Nvidia con los drivers recientes
- Instalar el toolkit de CUDA (http://developer.nvidia.com/cuda-downloads)
- Los .cu se compilan con el compilador nvcc y se enlazan con la biblioteca libcudart

```
nvcc -o sumaCUDA sumaCUDA.cu -L/usr/local/cuda/lib -lcudart
```

- Usar makefiles o configurar proyecto para hacer esto automáticamente

Caso de estudio: multiplicación de matrices

- Solución directa: cada thread calcula una posición de la matriz resultante C multiplicando los valores de una fila de A por los de una columna de B y acumulando el resultado



Caso de estudio: multiplicación de matrices

- La implementación es muy sencilla

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];

    C.elements[row * C.width + col] = Cvalue;
}
```

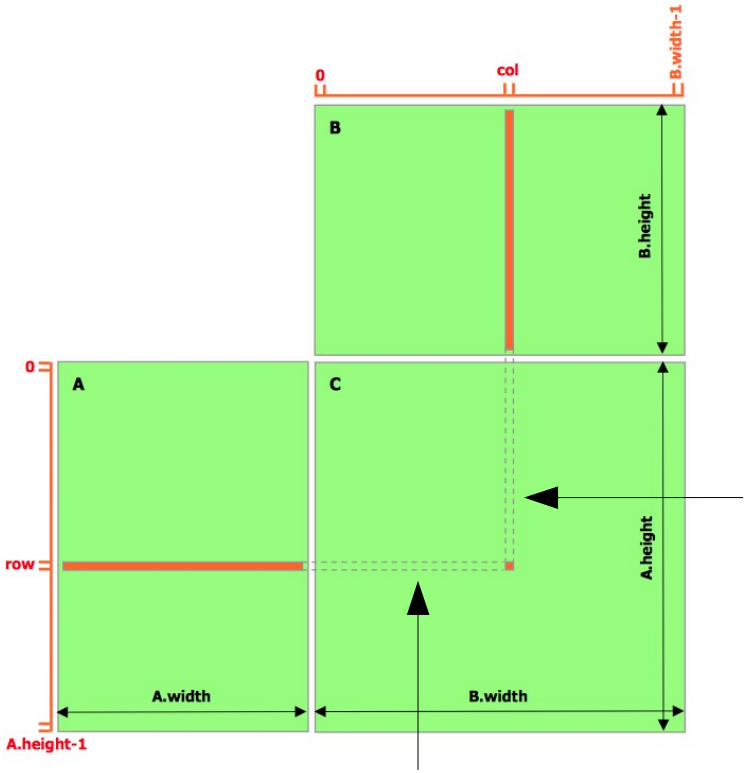
Caso de estudio: multiplicación de matrices

- La implementación es muy sencilla

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];

    C.elements[row * C.width + col] = Cvalue;
}
```

Caso de estudio: multiplicación de matrices

- Mejorable: muchas hebras tienen que acceder a las mismas posiciones en memoria global

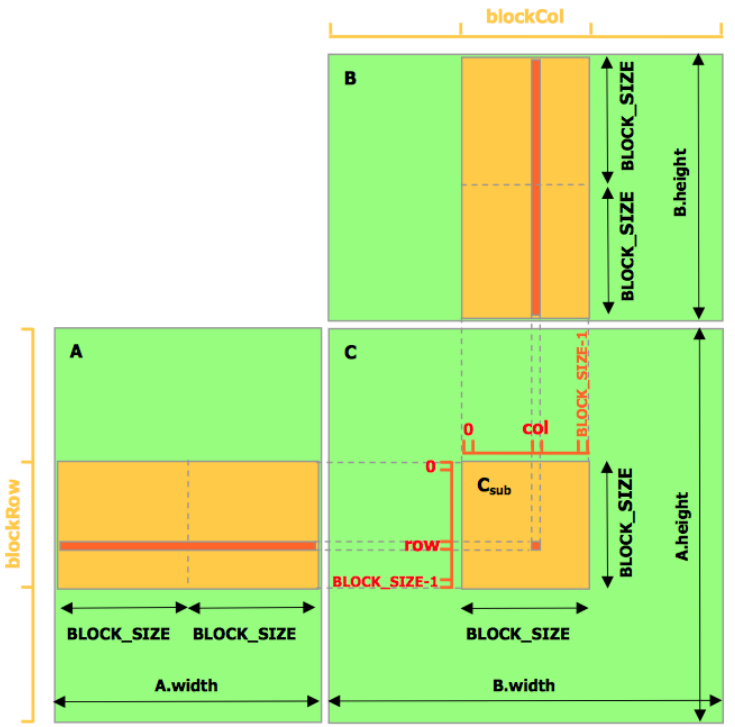


Estos threads requieren la misma columna de B

Estos threads requieren la misma fila de A

Caso de estudio: multiplicación de matrices

- Versión mejorada que usa memoria compartida
  - Cada bloque calcula una submatriz
  - Las matrices A y B también se dividen en bloques



Caso de estudio: multiplicación de matrices

- Se van trayendo bloques de las matrices A y B a memoria compartida
- Se usa su información para calcular un resultado parcial que se acumula
- Cada thread sólo trae dos valores a memoria compartida

Caso de estudio: multiplicación de matrices

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Loop over all the sub-matrices of A and B
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        // Get sub-matrices Asub and Bsub
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);

        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();
```

Caso de estudio: multiplicación de matrices

```
        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
}

    // Write Csub to device memory
    // Each thread writes one element
    SetElement(Csub, row, col, Cvalue);
}
```

Cálculo usando datos en memoria compartida

Caso de estudio: multiplicación de matrices

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
   return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value)
{
   A.elements[row * A.stride + col] = value;
}

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
   Matrix Asub;
   Asub.width = BLOCK_SIZE;
   Asub.height = BLOCK_SIZE;
   Asub.stride = A.stride;
   Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
   return Asub;
}
```

Salto desde el comienzo de una fila a otra

Consejos para obtener un buen rendimiento

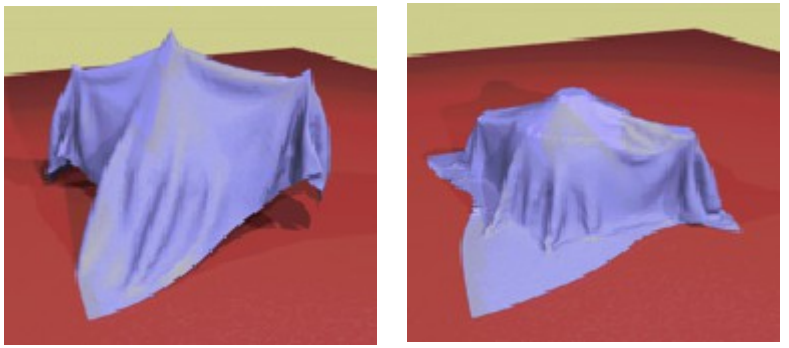
- A menor cantidad de datos a transferir a la GPU y cálculo más intensivo computacionalmente, mejor rendimiento
- Para procesamientos sencillos la GPU da un resultado mucho peor que la CPU
- Evitar colisiones: acceso simultáneo de varias hebras a la misma posición de memoria
- Copiar a memoria compartida datos que sean necesarios para varias hebras del bloque

Casos de estudio: investigación en el GGGJ

- Autointersección de mallas
- Cálculo de redes de drenaje en mapas de elevación
- Algoritmos de simplificación 3D para cálculo de la dimensión fractal

Autointersección de mallas

- Definición del problema:
  - En animación de modelos deformables es necesario detectar las autointersecciones para imponer restricciones de movimiento (ej: telas)
  - Es frecuente que muchas mallas contengan defectos en forma de autointersecciones



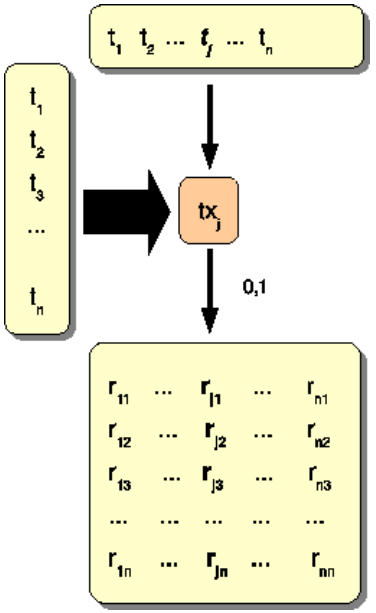
Solución trivial

- Definición del problema:
  - Calcula todas las intersecciones posibles (matriz de intersecciones)
  - Tiempo de proceso: $O(n^2)$
  - Mejorable utilizando técnicas mucho más complejas

```
for (int i = 0; i < numTri * numTri; ++i) intersec[i] = 0;

for (int i = 0; i < numTri; ++i) {
   for (int j = i + 1; j < numTri; ++j) {
      if (triTriIntersect(i,j)) {
         intersec[i][j] = intersec[j][i] = 1;
      }
   }
}
```

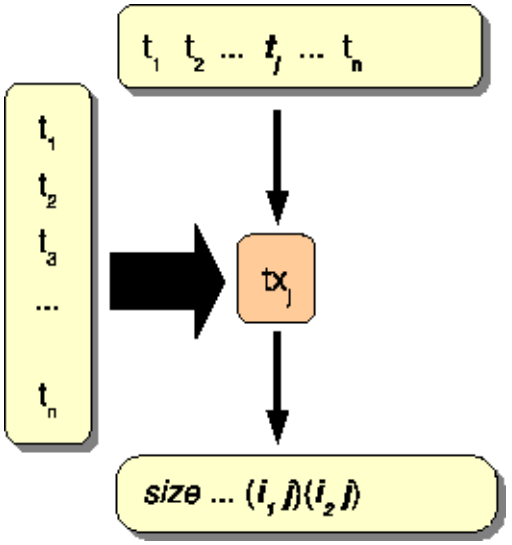
Solución CUDA 1

- Cada thread comprueba la intersección de 1 triángulo frente al resto
- El resultado puede ser almacenado en una matriz de dimensiones numTri x numTri
- Ineficiente



Solución CUDA 2

- Devolver una lista de parejas de triángulos conflictivos en un buffer de resultados
- Requiere el uso de operaciones atómicas para el acceso concurrente a la lista



Solución CUDA 2

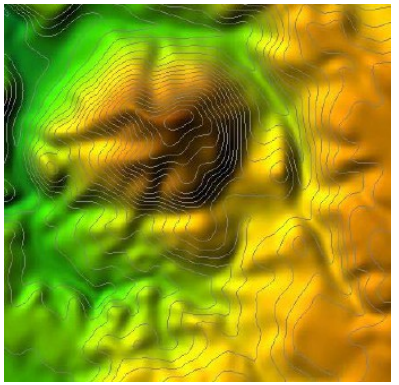
```
__global__ void selfIntersectionGPUThread(
  Triangle *dMesh,
  unsigned nTriangles,
  int *dTestResults)
{
  // Index of triangle to process
  int nt = blockIdx.x * BLOCK_SIZE + threadIdx.x;

  // Get triangle from global memory
  Vertex v0, v1, v2, u0, u1, u2;
  vertexCopy(v0, dMesh[nt][0]);
  vertexCopy(v1, dMesh[nt][1]);
  vertexCopy(v2, dMesh[nt][2]);

  // Start testing triangles from index nt + 1
  int ntt = nt + 1;

  while (ntt < nTriangles) {
    // Get triangle to test
    vertexCopy(u0, dMesh[ntt][0]);
    vertexCopy(u1, dMesh[ntt][1]);
    vertexCopy(u2, dMesh[ntt][2]);

    // Test triangles for intersection
    if (triTriIntersect(v0,v1,v2,u0,u1,u2)) {
      if (dTestResults[0] < MAX_RESULTS) {
        // Get and move current position of
        // the list. Store the indices of triangles
        int pos = atomicAdd(&dTestResults[0], 2);
        dTestResults[pos] = nt;
        dTestResults[pos + 1] = ntt;
      }
    }
    ++ntt;
  }
}
```

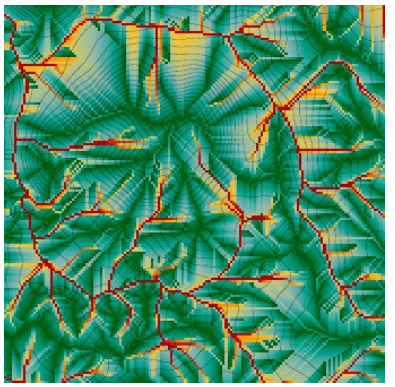
Resultados

- Uso de memoria eficiente
- Hasta 42X con modelos grandes
- Implementación extremadamente sencilla
- Referencias:
  - Rueda, A., Ortega, L. Geometric Algorithms on CUDA. Actas de GRAPP'2008, Fuchal - Madeira (Portugal)

Cálculo de redes de drenaje

- Problema clásico en Hidrología
  - A partir de un mapa de elevaciones (DEM) calcular la red hídrica del mismo
- Aplicaciones
  - Simulación y prevención de inundaciones
  - Estudios de contaminación en ríos y lagos



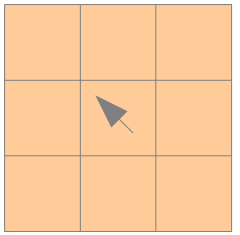


Algoritmo clásico

- O'Callaghan y Mark (1984)
  - Calcular matriz de direcciones: para cada celda del DEM, estudiar las 8 celdas adyacentes
  - Decidir direcciones de salida en mínimos locales y zonas planas (complejo)

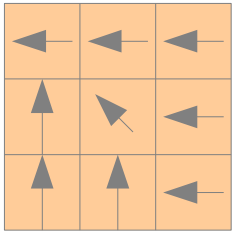
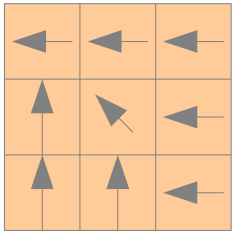
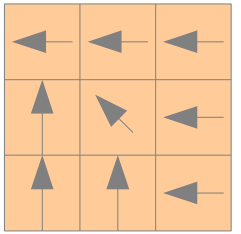
90	95	110
95	101	115
95	101	110

DEM



Direcciones

Acumulación de agua

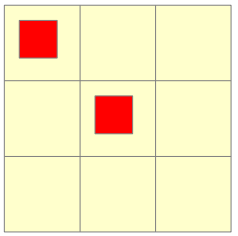
	Cumulo	Cumulo Acum.	Direcciones
Estado inicial	1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0	
Iteración 1	3 1 0 1 2 0 0 1 0	3 1 0 1 2 0 0 1 0	
Iteración 2	4 0 0 0 1 0 0 0 0	7 1 0 1 3 0 0 1 0	

Cálculo de la red

- La acumulación termina cuando toda el agua sale del DEM
- La red de drenaje viene dada por el conjunto de celdas de la matriz de cúmulos cuyo valor supera un umbral preestablecido

7	1	0
1	3	0
0	1	0

Cumulo Acum.



Red (cumulo > 2)

Implementación CUDA

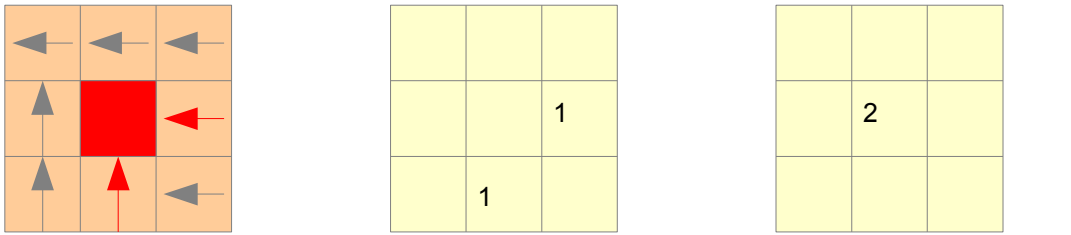
- Asignación de direcciones y eliminación de mínimos locales como preprocesamiento en CPU
- Asociamos un thread a cada celda, que transfiere el cúmulo a otra celda vecina siguiendo su dirección de desplazamiento
- Son necesarias operaciones atómicas: a una celda puede llegar agua desde varias celdas
- 1 ejecución CUDA completa por cada iteración

Implementación CUDA

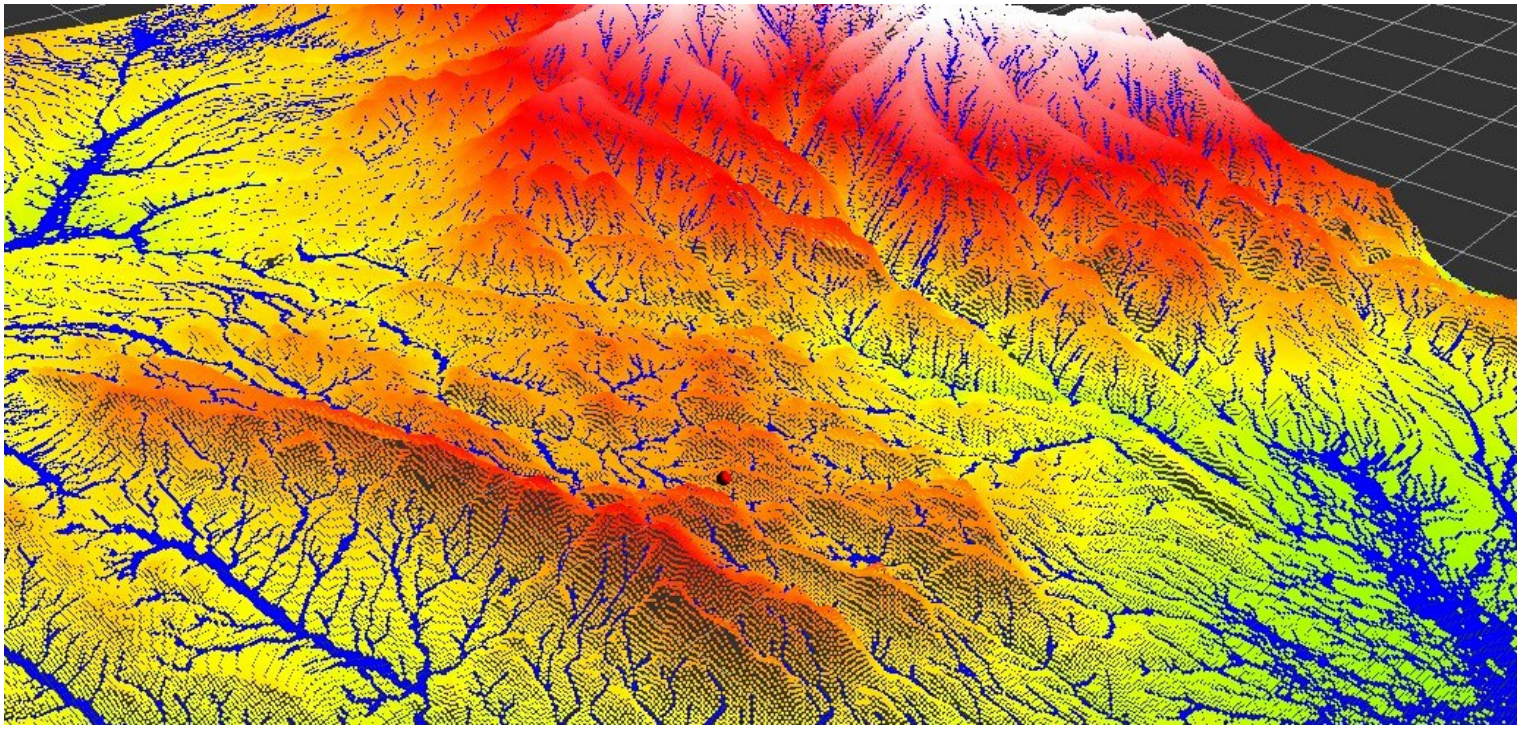
```
__global__ void gridRedDrenaje_Thread(Celda *celdas) {

  unsigned i = blockIdx.y*BLOCK_SIZE+threadIdx.y;
  unsigned j = blockIdx.x*BLOCK_SIZE+threadIdx.x;

  // Cúmulo de la celda
  int cumuloInstantaneo = celdas->cumuloActual[i][j];

  if (cumuloInstantaneo > 0) {
      // Obtener dirección
      int2 dir = celdas->dir[i][j];

      // Actualizar cúmulos
      atomicAdd(&celdas->cumuloNuevo[i + dir.y][j + dir.x], cumuloInstantaneo);
      atomicAdd(&celdas->cumuloAcum[i + dir.y][j + dir.x], cumuloInstantaneo);
    }
  }
}
```

Mejoras implementación CUDA

- Procesar más de 1 celda en cada thread (8x8)
- Evitar operaciones atómicas invirtiendo el sentido del procesamiento: dada una celda, acumular el agua procedente de las vecinas (implementación más compleja)



Resultados



Resultados

- Hasta 8X
- Mejores resultados con modelos de drenaje más sofisticados
- Referencias:
  - Lidia Ortega, Antonio J. Rueda. Parallel drainage network computation on CUDA. Computers & Geosciences, 36 (2010)

Algoritmos de simplificación 3D para el cálculo de la dimensión fractal

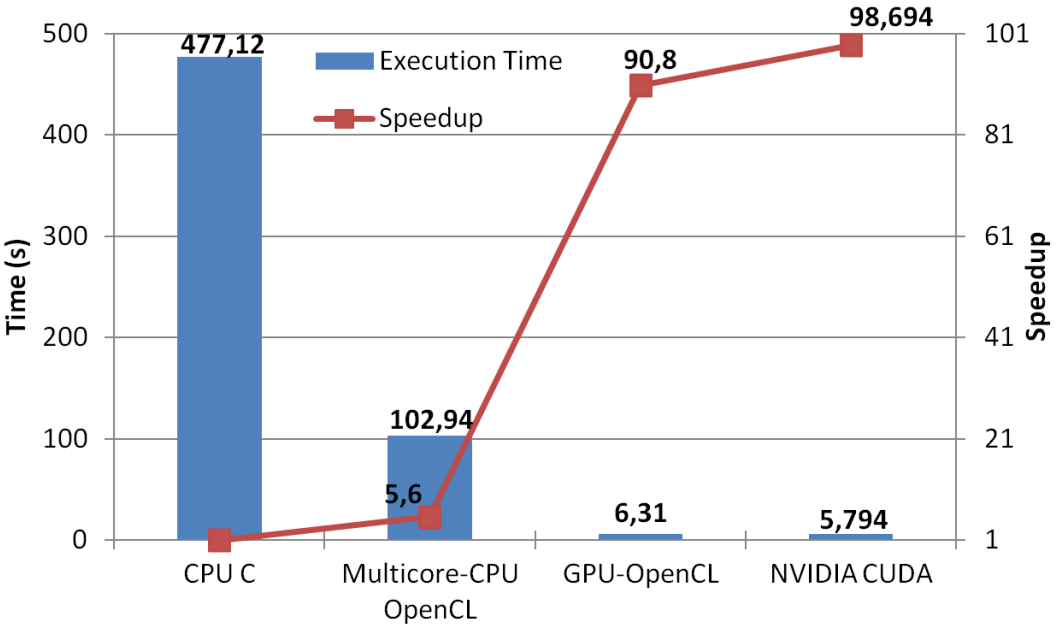
- Trabajo realizado por Juan Ruiz De Miras y Jesús Jiménez (tesis doctoral)
- La **dimensión fractal** del cerebro es una medida que permite detectar la aparición de lesiones en enfermedades neurodegenerativas antes de que sean visibles a simple vista por expertos médicos
- Dos operaciones básicas para el cálculo de la dimensión fractal:
  - Generación de esqueleto 3D
  - Box-counting 3D

Generación del esqueleto 3D

- Técnica de adelgazamiento similar a la usada en la esqueletonización 2D
- Paralelismo inherente con memoria compartida
- Implementado en CPU, CUDA y OpenCL

Generación del esqueleto 3D: resultados

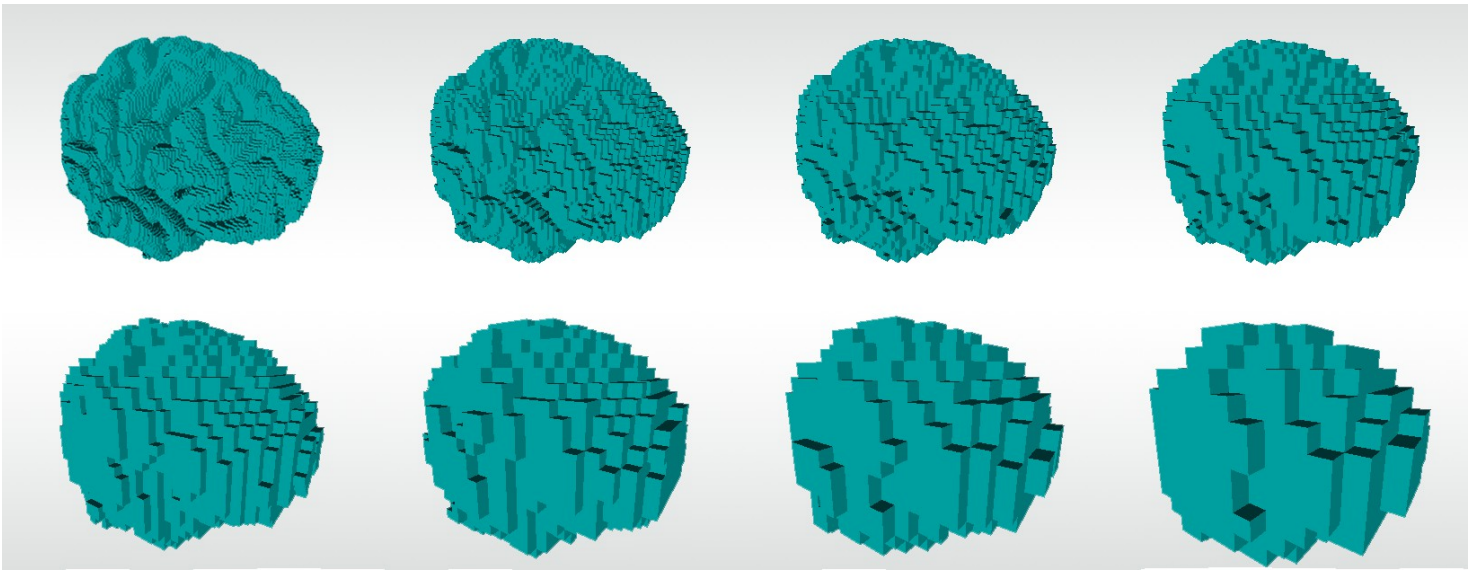
- Cerca de un 100x de aceleración



- Referencias:
  - Jesús Jiménez, Juan R. de Miras. Three-dimensional thinning algorithms on GPUs and multicore CPUs. Concurrency and Computation: Practice and Experience 24, 2012

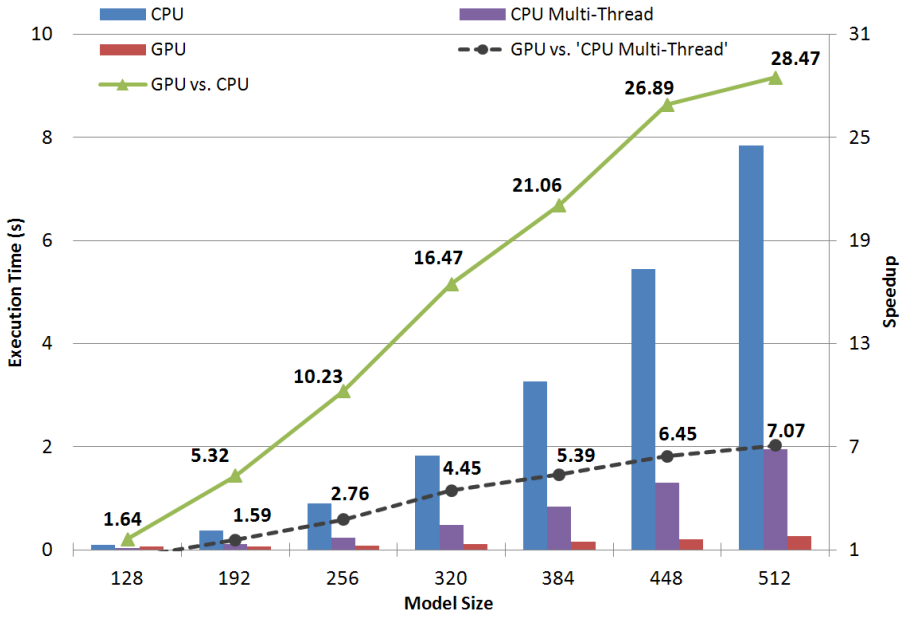
Algoritmo de Box-counting 3D

- Dividir el espacio en rejillas de cajas a distinta resolución para cubrir el objeto y contar el número de cajas resultante
  - Paralelismo inherente
  - Ciertas operaciones sin compartición de datos



Algoritmo de Box-counting 3D: resultados

- Hasta 30x de aceleración



- Referencias:
  - Jesús Jiménez, Juan R. de Miras. Fast box-counting algorithm on GPU. Computer Methods and Programs in Biomedicine 108, 2012