Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXXII, págs. 971-987 (artículo completo)
Carlos G. García Garino, Aníbal E. Mirasso, Mario A. Storti, Miguel E. Tornello (Eds.)
Mendoza, Argentina, 19-22 Noviembre 2013

IMPLEMENTACION PARALELA DE SPH USANDO DIRECTCOMPUTE

Pablo Sebastián Rojas Fredini y Alejandro César Limache

Centro de Investigación de Métodos Computacionales (CIMEC) UNL-CONICET. Santa Fe, Argentina.,
http://www.cimec.gov.ar/

Palabras Clave: partículas, smoothed particle hydrodynamics, tiempo real, GPGPU.

Resumen. La disponibilidad para consumo masivo de procesadores multinúcleos y placas de vídeo de propósito general han provocado una migración desde el punto de vista del diseño de los algoritmos numéricos en general. Se ha pasado de un paradigma serial a uno paralelo. Esto ha hecho necesario repensar los algoritmos y métodos clásicos y también posibilitó que métodos menos populares vuelvan a cobrar importancia. Un método que promete mostrar buena escalabilidad en las nuevas arquitecturas es Smoothed Particle Hydrodynamics, cuya popularidad se ha incrementado en los últimos años. En el presente trabajo se presenta un algoritmo y estructuras de datos convenientes para la implementación en GPGPU. La implementación se realiza utilizando la tecnología DirectCompute, que permite ejecutar el algoritmo en GPUs de diversos fabricantes e incluso en chipsets integrados. Finalmente se muestran resultados de pruebas de escalabilidad y performance usando formulaciones SPH para resolver problemas clásicos de fluidos.

1. INTRODUCCIÓN

La simulación de fluidos en tiempo real es una tarea computacionalmente muy costosa por lo que para llevarla a cabo, es necesario aprovechar al máximo las nuevas arquitecturas de hardware paralelas. Para ello ha sido necesario adaptar y reformular métodos clásicos para lograr buena escalabilidad y desempeño. Por otro lado este cambio en el paradigma de desarrollo ha permitido que métodos menos populares recobraran interés debido a poseer formulaciones que son paralelizables por naturaleza. Entre estos métodos podemos mencionar Laticce-Boltzman (Obrecht et al., 2013; Succi, 2001), Diferencias Finitas(Micikevicius, 2009) y SPH(Smoothed Particle Hydrodynamics) (Monaghan, 2005; Rojas Fredini y Limache, 2013; Liu y Liu, 2003).

SPH cuenta con numerosas aplicaciones en diferentes campos de la ingeniería(Monaghan, 2005; Rook et al., 2008; Jeong et al., 2003; Patrick et al., 1997; Cleary, 1997; Vesterlund, 2004), en los cuales el factor primordial es la precisión del método y no así la velocidad de cómputo. Por otro lado ha sido aplicado en la industria del entretenimiento de manera exitosa (Charypar et al., 2003; Gourlay, 2012; Green et al., 2011; Nuli y Kulkarni, 2012), teniendo en este caso como objetivo primordial la robustez y la velocidad, sacrificando en cierta medida la precisión. Actualmente una de las líneas de investigación más activas consiste en lograr que confluyan las formulaciones más precisas desde el punto de vista ingenieril, con las técnicas utilizadas en la industria del entretenimiento para lograr gran velocidad de cómputo.

En el presente trabajo se introduce un algoritmo para simular fluido en tiempo real mediante el método SPH. Dicho algoritmo se ejecuta íntegramente en la GPU (Graphics Processing Unit) eliminando las copias de memoria entre la CPU y la GPU durante los pasos de la simulación. La implementación se realiza sobre la nueva plataforma DirectCompute(Microsoft, 2013; Nvidia, 2013b) que es la más utilizada en la industria de los videojuegos, y tiene la ventaja de ser compatible con gran cantidad de hardware, incluso con chipsets de video integrados.

2. ALGORITMO

Para la simulación del fluido se utiliza una formulación SPH pseudo compresible(WCSPH) (Rojas Fredini y Limache, 2013) con un esquema explícito de integración temporal. En las próximas secciones se detalla el algoritmo.

2.1. Plataforma

Para la implementación del método se eligió la biblioteca DirectCompute, que es parte de DirectX 11. La misma presenta una funcionalidad similar a CUDA(Nvidia, 2013a) y OpenCL (Group, 2013) permitiendo al usuario utilizar el paralelismo masivo del hardware de video para cálculo de propósito general. A diferencia de CUDA que es dependiente del fabricante, DirectCompute funciona sobre la mayoría del hardware de video que ha salido al mercado en los últimos meses, incluso se ejecuta perfectamente en GPU integradas. Por otro lado, a diferencia de OpenCL DirectCompute es una tecnología que sólo funciona sobre plataformas Windows (incluyendo sus consolas de videojuegos y teléfonos inteligentes).

DirectCompute utiliza el lenguaje HLSL para programar las rutinas a ejecutarse en paralelo. Cada rutina de estas se denomina *compute shader* para diferenciarlo de los demás shaders de DirectX. Al ser parte de la misma API, permite un gran nivel de integración entre el cálculo y la visualización de manera transparente para el desarrollador.

Para el almacenamiento de la información de las partículas se utilizan buffers estructurados y para accederlos se lo realiza mediante vistas. Las vistas pueden ser de sólo lectura (Shader Resource View -SRV-) o de acceso aleatorio (Uniform Access View -UAV-). Estas vistas son las

que se enlazan a cada *compute shader* durante su ejecución. Por otro lado también se utilizan buffers constantes (*Constant Buffers*) para almacenar parámetros de la simulación que varían poco a lo largo de la ejecución.

2.2. Descripción general

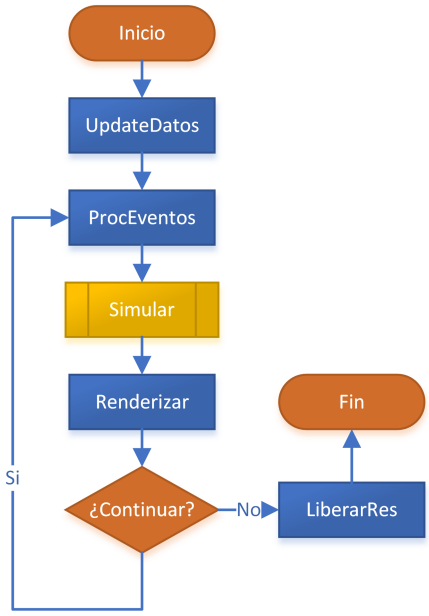


Figura 1: Loop principal de simulación

El lazo principal que puede verse en la Fig.1. Al comenzar el método *UpdateDatos* inicializa DirectCompute y copia los datos iniciales de simulación a la GPU. En este paso se crean las estructuras de datos principales: los buffers de partículas (*ParticleBuffer*), de densidad(*DensityBuffer*) y de fuerzas (*ForceBuffer*), en la Fig. 2 se pueden observar los tres buffers mencionados y la estructura de cada elemento. La división de estas estructuras fue hecha en función de los buffers utilizados en cada paso de algoritmo. Además se crea un buffer para almacenar las condiciones de borde y la forma del mismo, dicho buffer denominado BoundariesBuffer puede observarse en la Fig.3

Adicionalmente se crean buffers constantes que almacenan la información de la simulación que no varía en cada paso de tiempo. Un detalle de dicho buffer puede observarse en la Fig.4

En segundo lugar se ejecuta el procesamiento de eventos y la ejecución de scripts, que permiten controlar la simulación y cambiar los parámetros en tiempo real sin necesidad de reiniciar la simulación. Luego se realiza la simulación propiamente dicha -que será explicada en la si-

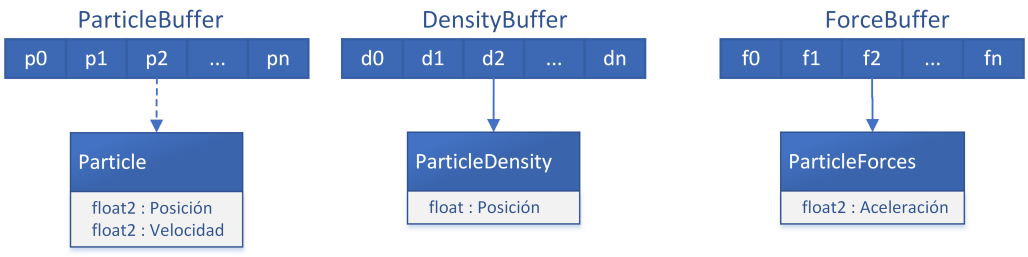


Figura 2: Buffers de partículas

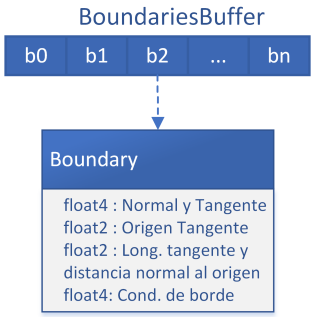


Figura 3: Buffer de condiciones de borde

Figura 4: Buffer constante de simulación

guiente sección- y finalmente se renderiza el resultado usando una técnica de *point splatting* muy popular en sistemas de partículas.

2.3. Loop de simulación

En la Fig.5 se puede observar un diagrama de flujo del lazo principal de simulación. En las secciones siguientes se realizará una descripción de cada una de las partes involucradas y las estructuras de datos utilizadas.

2.4. Búsqueda espacial

Para calcular las cantidades físicas en cada partícula es necesario conocer cúales son sus vecinas espaciales, ya que la sumatoria de SPH se realizará sólamente sobre las partículas que se encuentren en un radio h a su alrededor, siendo h el soporte del kernel SPH utilizado.

Existen varias alternativas para lograr realizar una búsqueda espacial rápida, teniendo en cuenta que la arquitectura en GPU es muy distinta a una arquitectura de CPU tradicional. En este trabajo se utiliza una técnica de hash espacial también conocida como *Z-indexing*(Prashant et al., 2010) . En la misma las partículas se distribuyen en una grilla que divide al espacio en $N_x \times N_y$ celdas, siendo N_* la cantidad de celdas en cada dirección. Cada celda es de un tamaño mínimo h en ambas direcciones. Dada una partícula P_i para determinar sus vecinas es necesario buscar todas las partículas cuya distancia a P_i sea menor que h, al ser el tamaño de las celdas igual o mayor que h sólo hace falta examinar la celda en la cual se encuentra P_i y sus 8 celdas vecinas. Esto se puede observar en la Fig.6

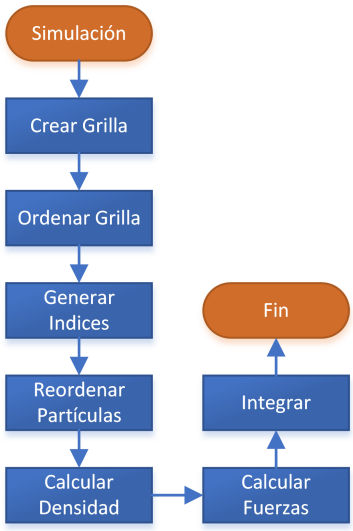


Figura 5: Loop de simulación

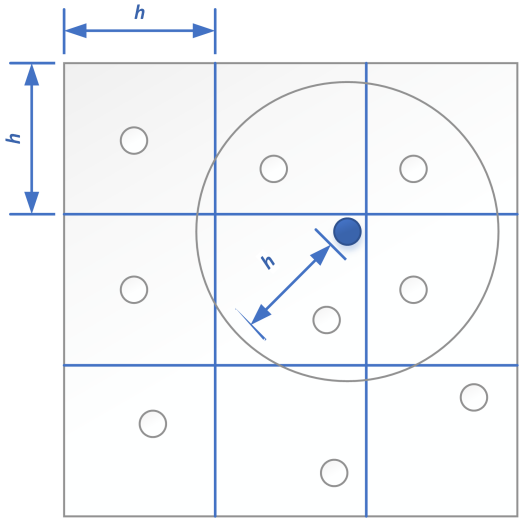


Figura 6: Grilla para el ordenamiento espacial

2.4.1. Crear Grilla

Si bien el algoritmo utiliza una grilla como estructura de datos subyacente, en GPU nunca se crea la grilla propiamente dicha, ya que no es una estructura eficiente para el acceso a memoria. Para implementar esta técnica en GPU se reformula el algoritmo de CPU. En CPU se suelen insertar las partículas en las celdas de la grilla creada en memoria, en cambio en GPU, cada partícula almacena el índice de grilla en el cual se encuentra mediante una función de hash. El índice de cada celda consiste en un par de enteros X e Y que indican la posición de la celda dentro de la grilla en la dirección x e y respectivamente. En la estructura GridBuffer mostrada en la Fig.7 se almacenan enteros de 32 bits compuestos por los índices X e Y de la celda que ocupa dicha partícula y además se almacena el número de la partícula. En este paso el número de partícula es redundante ya que existe una correspondencia 1-1 con su posición dentro de la estructura pero luego el buffer será reordenado, por lo que es necesario mantener información del lugar que ocupaba cada partícula al principio del paso de tiempo. Esta elección impone dos

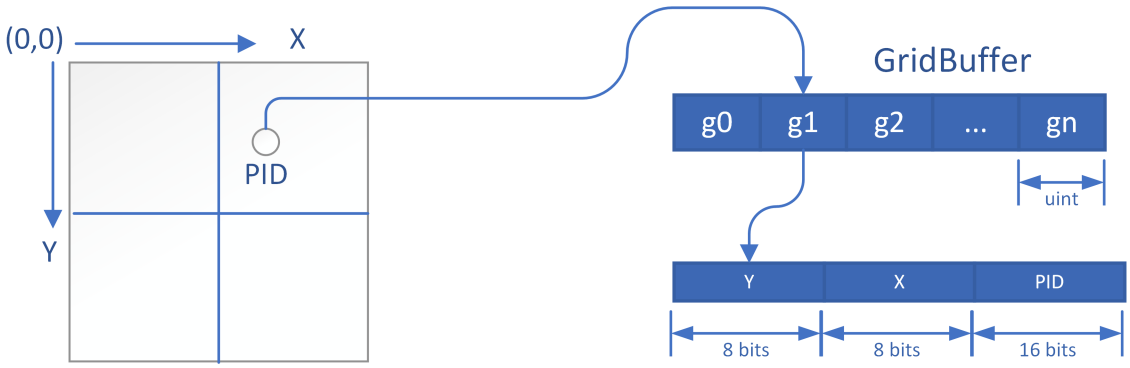


Figura 7: Linealización de la grilla

cotas al algoritmo. La primera limita la cantidad de partículas que es posible simular, ya que al usar un índice de 16 bits para la misma, no será posible simular más de 2^{16} partículas. La segunda cota tiene que ver con la cantidad de celdas que podrá tener la grilla. Al usar índices de 8 bits para cada dimensión se tendrán como máximo $2^8 \times 2^8$ celdas, esto es importante ya que pone una cota indirectamente al soporte de la función de peso de SPH. Claro es, que simplemente cambiando el tipo de dato usado para la grilla se pueden extender estas cotas, por ejemplo de uint a uint2.

A continuación se detalla el código y funciones auxiliares para la generación de la grilla

```
float2 GridCalculateCell(float2 position)
{
    return clamp(position * g_vGridDim.xy + g_vGridDim.zw, float2(0, 0), float2(255, 255));
}
unsigned int GridConstuctKey(uint2 xy)
{
    // Bit pack [------UNUSED------][-----Y----][-----X----]
    //                16-bit          8-bit      8-bit
    return dot(xy.yx, uint2(256, 1));
}
unsigned int GridConstuctKeyValuePair(uint2 xy, uint value)
{
    // Bit pack [-----Y----][-----X----][-------VALUE-------]
    //             8-bit      8-bit          16-bit
    return dot(uint3(xy.yx, value), uint3(256*256*256, 256*256, 1));
}
unsigned int GridGetKey(unsigned int keyvaluepair)
{
    return (keyvaluepair >> 16);
}
unsigned int GridGetValue(unsigned int keyvaluepair)
{
    return (keyvaluepair & 0xFFFF);
}
//--------------------------------------------------------------------------------------
// Build Grid
//--------------------------------------------------------------------------------------
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void BuildGridCS( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 GTid : ↩
    SV_GroupThreadID, uint GI : SV_GroupIndex )
{
    const unsigned int P_ID = DTid.x; // Particle ID to operate on
    float2 position = ParticlesRO[P_ID].position;
    float2 grid_xy = GridCalculateCell( position );
    GridRW[P_ID] = GridConstuctKeyValuePair((uint2)grid_xy, P_ID);
}
```

Alli GridRW es el UAV correspondiente al GridBuffer, y ParticlesRO es el SRV del Particle-

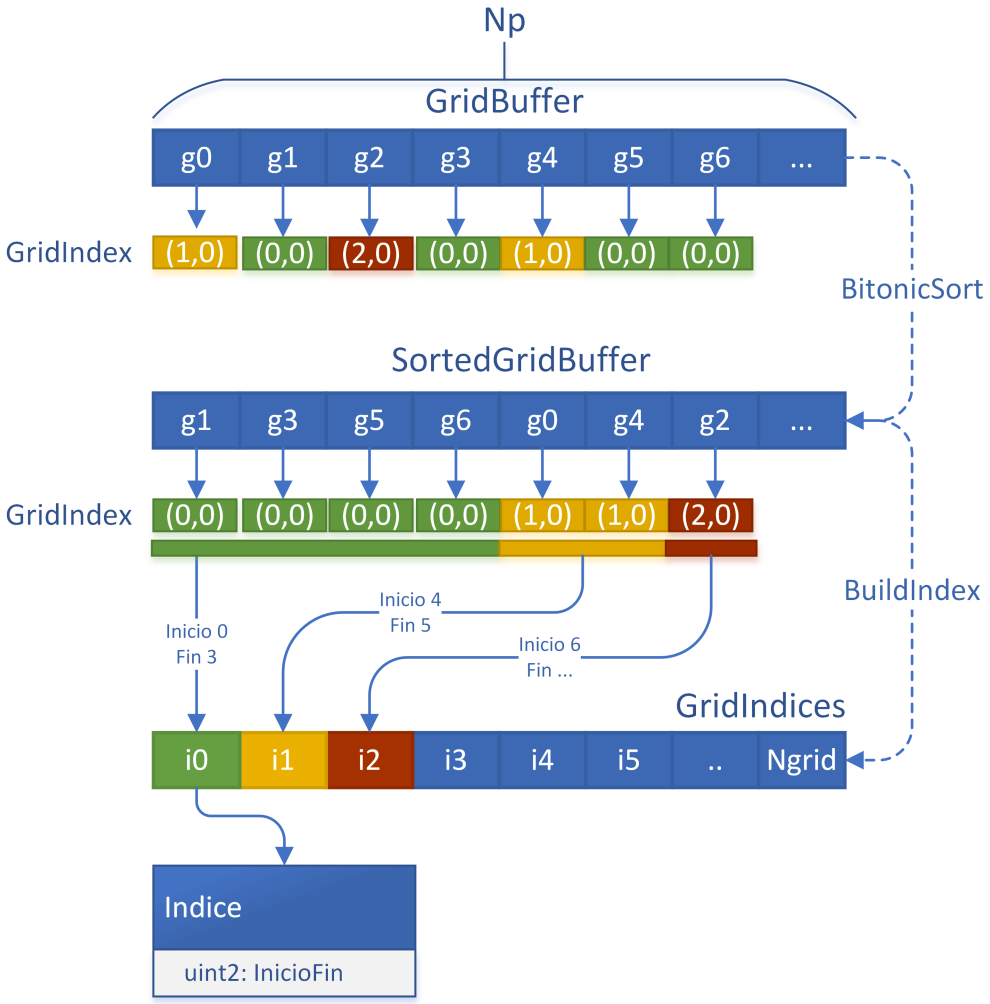


Figura 8: Ordenamiento de la grilla y generación de los índices

Buffer.

2.4.2. Ordenar Grilla

El siguiente paso consiste en ordenar las partículas (representadas en el buffer GridBuffer) de acuerdo a la posición de grilla que ocupan, de manera que todas las partículas pertenecientes a una misma grilla se ubiquen consecutivas en memoria, como se muestra en la primer parte de la Fig.8. Luego se construye un buffer llamado GridIndices cuyos elementos son uint2 que indican en que lugar del arreglo de partículas ordenado comienzan y terminan las partículas pertenecientes a cada índice de grilla. Esto último se puede observar en la segunda parte de la Fig.8, siendo Np la cantidad de partículas y Ngrid la cantidad de celdas en la simulación.

Para el ordenamiento se utiliza un algoritmo de ordenamiento que es independiente de los datos, lo cual lo hace muy apropiado para ser implementado en GPU. En particular se implementó el algoritmo *bitonic merge sort*(Kipfer y Westermann, 2005).

A continuación se muestra la implementación de la limpieza y la construcción de índices dejando de lado el ordenamiento para no perder claridad en la exposición.

```
//
```

```
// Build Grid Indices
//------------------------------------------------------------------------------
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void ClearGridIndicesCS( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 ↩
    GTid : SV_GroupThreadID, uint GI : SV_GroupIndex )
{
    GridIndicesRW[DTid.x] = uint2(0, 0);
}
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void BuildGridIndicesCS( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 ↩
    GTid : SV_GroupThreadID, uint GI : SV_GroupIndex )
{
    const unsigned int G_ID = DTid.x; // Grid ID to operate on
    unsigned int G_ID_PREV = (G_ID == 0)? g_iNumParticles : G_ID; G_ID_PREV--;
    unsigned int G_ID_NEXT = G_ID + 1; if (G_ID_NEXT == g_iNumParticles) { G_ID_NEXT = 0; }

    unsigned int cell = GridGetKey( GridRO[G_ID] );
    unsigned int cell_prev = GridGetKey( GridRO[G_ID_PREV] );
    unsigned int cell_next = GridGetKey( GridRO[G_ID_NEXT] );
    if (cell != cell_prev)
    {
        // I am the start of a cell
        GridIndicesRW[cell].x = G_ID;
    }
    if (cell != cell_next)
    {
        // I am the end of a cell
        GridIndicesRW[cell].y = G_ID + 1;
    }
}
```

Alli GridIndicesRW es el UAV del buffer de indices de grilla, y GridRO es el SRV del buffer de partículas ordenado según la posición en la grilla.

2.4.3. Reordenar Partículas

En este paso del algoritmo se realiza un reordenamiento del buffer ParticleBuffer que hasta el momento no había sido modificado y se almacena en un buffer auxiliar llamado SortedParticleBuffer. Este reordenamiento se realiza para que coincidan las posiciones del GridBuffer reordenado con las partículas.

Este es el último paso de generación de la estructura de ordenamiento espacial. De aquí en adelante cada hilo de ejecución que desee acceder a las partículas ubicadas en una coordenada determinada de la grilla puede hacerlo recorriendo el vector de partículas ordenadas mediante los índices almacenados en GridIndices.

El código es muy sencillo y se lista a continuación por completitud.

```
//------------------------------------------------------------------------------
// Rearrange Particles
//------------------------------------------------------------------------------
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void RearrangeParticlesCS( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 ↩
    GTid : SV_GroupThreadID, uint GI : SV_GroupIndex )
{
    const unsigned int ID = DTid.x; // Particle ID to operate on
    const unsigned int G_ID = GridGetValue( GridRO[ ID ] );
    ParticlesRW[ID] = ParticlesRO[ G_ID ];
}
```

Alli ParticlesRW es el UAV del buffer SortedParticleBuffer y ParticlesRO es el SRV de las partículas sin ordenar ParticleBuffer.

2.5. Calcular Densidad

2.5.1. Formulación

Para el cálculo de la densidad se utilizó la formulación

$$\rho_i = \sum_j m_j W_{ij} \tag{1}$$

donde ρ es la densidad del fluido en la partícula i, m_j es la masa de las partículas vecinas j y $W_{ij} = W(x_i - x_j, h)$ es el kernel de SPH, siendo x la posición de las partículas y h el soporte de dicha función.

2.5.2. Estructuras de datos

Durante el cálculo de la densidad es necesario conocer la posición de las partículas, por lo que se enlaza el SRV de SortedParticleBuffer, el UAV de DensityBuffer que es donde se escribirá la salida y el buffer constante SimulationConstants del cual se obtiene la masa de las partículas. Además se enlaza el buffer GridIndices que permite realizar la búsqueda espacial de manera eficiente. Este último buffer se enlazará en todas las etapas que sea necesario hacer cálculos basados en la vecindad.

2.5.3. Implementación

La implementación puede observarse a continuación

```
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void DensityCS_Grid( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 GTid : ↩
    SV_GroupThreadID, uint GI : SV_GroupIndex )
{
    const unsigned int P_ID = DTid.x;
    const float h_sq = g_fSmoothlen * g_fSmoothlen;
    float2 P_position = ParticlesRO[P_ID].position;
    float2 P_velocity = ParticlesRO[P_ID].velocity;
    float density = CalculateDensity(0.0); ;

    // Calculate the density based on neighbors from the 8 adjacent cells + current cell
    int2 G_XY = (int2)GridCalculateCell( P_position );
    for (int Y = max(G_XY.y - 1, 0) ; Y <= min(G_XY.y + 1, 255) ; Y++)
    {
        for (int X = max(G_XY.x - 1, 0) ; X <= min(G_XY.x + 1, 255) ; X++)
        {
            unsigned int G_CELL = GridConstuctKey(uint2(X, Y));
            uint2 G_START_END = GridIndicesRO[G_CELL];

            for (unsigned int N_ID = G_START_END.x ; N_ID < G_START_END.y ; N_ID++)
            {
                float2 N_position = ParticlesRO[N_ID].position;
                float2 N_velocity = ParticlesRO[N_ID].velocity;

                float2 diff = N_position - P_position;
                float r_sq = dot(diff, diff);

                if (r_sq < h_sq && P_ID != N_ID)
                {
                    density += CalculateDensity(r_sq); //Density kenel

                }
            }
        }
    }
```

```
    ParticlesDensityRW[P_ID].density = density;
  }
```

ParticlesRO es el SRV del buffer ordenado de partículas SortedParticleBuffer, GridIndices es el SRV de los índices de grilla y CalculateDensity es una función que implementa la Eq.(1) y ParticlesDensityRW es el UAV del DensityBuffer.

2.6. Calcular Fuerzas

2.6.1. Formulación

Para el calculo de momento alrededor de una partícula i se utilizan las siguientes formulaciones de SPH cuyo desarrollo puede consultarse en Rojas Fredini y Limache (2013):

$$m_i \frac{D\boldsymbol{v}_i}{Dt} = \boldsymbol{F}_i^{\text{press}} + \boldsymbol{F}_i^{\text{visc}} \tag{2}$$

donde:

$$\boldsymbol{F}_i^{\text{press}} = \frac{m_i}{\rho_i} < \nabla p_i >= \sum_j m_i m_j \left(\frac{p_j + p_i}{\rho_i \rho_j} \right) \nabla_i W_{ij} \tag{3}$$

y

$$\boldsymbol{F}_i^{\text{visc}} = \frac{m_i}{\rho_i} < \nabla \cdot \boldsymbol{\tau}_i >= \sum_j m_i m_j \left(\frac{\boldsymbol{\tau}_j + \boldsymbol{\tau}_i}{\rho_i \rho_j} \right) \cdot \nabla_i W_{ij} \tag{4}$$

siendo $\nabla_i W_{ij} = \frac{\partial W(x_i - x_j)}{\partial x_i}$.

Para computar la presión se utilizó la ecuación de estado

$$p = \kappa \left(\left(\frac{\rho}{\rho_0} \right)^{\gamma} - 1 \right) \tag{5}$$

en función de la velocidad del sonido c_s, κ esta dado por

$$\kappa = \frac{\rho_0 c_s^2}{\gamma} \tag{6}$$

para más detalles ver Monaghan (2005).

2.6.2. Estructuras de datos

Para el cálculo de fuerzas se enlaza el buffer constante SimulationConstants del cual se obtienen valores de densidad, masa, etc. En este paso, como buffers de entrada se enlaza el SRV SortedParticleBuffer que permite obtener la velocidad de cada partícula durante el cálculo del término viscoso. Además se vincula el SRV DensityBuffer, GridIndices y BoundariesBuffer. Este último contiene información de la forma del dominio de cálculo, usado para calcular la contribución del borde a la ecuación de momento.

Para generar la salida de este shader se enlaza el UAV ParticleForces, en donde cada partícula almacena la fuerza/aceleración a la cual se ve sometida.

2.6.3. Implementación

```
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void ForceCS_Grid( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 GTid : SV_GroupThreadID, uint GI : SV_GroupIndex )
{
    const unsigned int P_ID = DTid.x; // Particle ID to operate on

    float2 P_position = ParticlesRO[P_ID].position;
    float2 P_velocity = ParticlesRO[P_ID].velocity;
    float P_density = ParticlesDensityRO[P_ID].density;
    float P_pressure = CalculatePressure(P_density);

    const float h_sq = g_fSmoothlen * g_fSmoothlen;

    float2 accelerationP = float2(0, 0);
    float2 accelerationV = float2(0, 0);

    // Calculate the acceleration based on neighbors from the 8 adjacent cells + current cell
    int2 G_XY = (int2)GridCalculateCell( P_position );
    for (int Y = max(G_XY.y - 1, 0) ; Y <= min(G_XY.y + 1, 255) ; Y++)
    {
        for (int X = max(G_XY.x - 1, 0) ; X <= min(G_XY.x + 1, 255) ; X++)
        {
            unsigned int G_CELL = GridConstuctKey(uint2(X, Y));
            uint2 G_START_END = GridIndicesRO[G_CELL];

      for (unsigned int N_ID = G_START_END.x ; N_ID < G_START_END.y ; N_ID++)
            {
                float2 N_position = ParticlesRO[N_ID].position;
                float2 diff = P_position - N_position;

                float r_sq = dot(diff, diff);
                if (r_sq < h_sq && P_ID != N_ID)
                {

                    float2 N_velocity = ParticlesRO[N_ID].velocity;
                    float N_density = ParticlesDensityRO[N_ID].density;
                    float N_pressure = CalculatePressure(N_density);
                    float r = sqrt(r_sq);
                    float2 vij = P_velocity-N_velocity;

                    //Pressure Term
                    accelerationP += CalculateGradPressure(r, P_pressure, N_pressure, P_density, N_density, vij,diff);
                    //Viscosity Term
                    accelerationV += CalculateLapVelocity(r, P_velocity, N_velocity, P_density,N_density,diff);

                }
            }
        }
    }

  float Wmax=5/(g_fSmoothlen*3.14159265);

  //Should compute the viscous contribution due to wall truncation
  for (unsigned int j = 0 ; j < g_iWallCount ; j++)
  {
    accelerationP+=ComputeWallPressureForce();
    accelerationV+=ComputeWallViscouseForce();

  }

  ParticlesForcesRW[P_ID].acceleration = accelerationP+ accelerationV ;

}
```

Aqui ParticleForcesRW es el UAV donde se almacenan las fuerzas de partículas, ParticlesRO es el SRV del buffer SortedParticleBuffer y ParticlesDensityRO es el SRV de densidad de partículas. La funcion CalculatePressure() implementa la Eq.(5), mientras que CalculateGradPressure() y CalculateLapVelocity implementan las Eqs.(2) y (4) respectivamente. Finalmente las funciones ComputeWallPressureForce() y ComputeWallViscouseForce() calcula la contribución de las paredes debido al truncamiento del kernel.

2.7. Integrar

2.7.1. Formulación

El último paso del lazo de simulación consiste en realizar la integración temporal, para lo cual se utiliza un esquema de Euler semi-implícito:

$$\begin{aligned} v_{n+1} &= v_n + g(t_n, x_n)\Delta t \\ x_{n+1} &= x_n + f(t_n, v_{n+1})\Delta t \end{aligned} \tag{7}$$

2.7.2. Estructuras de datos

Durante esta etapa se enlaza el SRV ParticleForces calculado en la etapa anterior y el SRV BoundariesBuffer para chequear que ninguna partícula viola la condición de contorno. Además se enlaza el buffer de partículas ordenado SortedParticleBuffer. El resultado de la integración se escribe en el buffer de partículas sin ordenar sobreescribiéndolo, de manera que al recomenzar el lazo se disponga de la información actualizada y en el orden correcto.

2.7.3. Implementación

```
[numthreads(SIMULATION_BLOCK_SIZE, 1, 1)]
void IntegrateCS( uint3 Gid : SV_GroupID, uint3 DTid : SV_DispatchThreadID, uint3 GTid : ↩
    SV_GroupThreadID, uint GI : SV_GroupIndex )
{
    const unsigned int P_ID = DTid.x; // Particle ID to operate on
    float2 position = ParticlesRO[P_ID].position;
    float2 velocity = ParticlesRO[P_ID].velocity;
    float2 acceleration = ParticlesForcesRO[P_ID].acceleration;

    // Apply gravity
    acceleration += g_vGravity;

    // Integrate
    velocity += g_fTimeStep * acceleration;
    position += g_fTimeStep * velocity;
    ParticlesRW[P_ID].velocity=velocity;
    ParticlesRW[P_ID].position = position;

}
```

Aqui ParticlesRW es el SRV de partículas sin ordenar y ParticlesRO es el SRV auxiliar de partículas ordenado. Finalmente ParticleForcesRO es el SRV del buffer de momento.

2.8. Visualización

Para la visualización se utiliza un algoritmo sencillo de *point splatting*. El mismo se ejecuta en tres etapas y utiliza la generación de geometría en GPU a través de implementar un *geometry shader*. A continuación se describen las etapas involucradas en la visualización. Las estructuras utilizadas en el renderizado son expuestas a continuación:

```
cbuffer cbRenderConstants : register( b0 )
{
    matrix g_mViewProjection;
    float g_fParticleSize;
};

struct VSParticleOut
{
    float2 position : POSITION;
    float4 color : COLOR;
};

struct GSParticleOut
{
    float4 position : SV_Position;
    float4 color : COLOR;
    float2 texcoord : TEXCOORD;
};
```

La estructura cbRenderConstants posee información de las transformaciones afines y del tamaño de dibujo de las partículas. Las otras dos estructuras son utilizadas para el pasaje de información entre las diferentes etapas del pipeline gráfico.

2.8.1. Vertex Shader

Para generar los puntos no se introduce geometría adicional a la ya existente, proveniente de los buffers de simulación. En particular, en esta etapa se extrae la información del buffer de partículas y se la colorea con la información del buffer de densidad. Como dichos buffers fueron creados y manipulados siempre en GPU no es necesario realizar ningún movimiento de memoria. A continuación se expone el primer shader del pipeline.

```
VSParticleOut ParticleVS(uint ID : SV_VertexID)
{
    VSParticleOut Out = (VSParticleOut)0;
    Out.position = ParticlesRO[ID].position;
    Out.color = VisualizeNumber(ParticleDensityRO[ID].density, 1000.0f, 2000.0f);

    return Out;
}
```

Como se puede observar de esta etapa se obtiene como salida la posición transformada de cada partícula y su color asociado, de acuerdo a la densidad. La función VisualizeNumber() realiza una interpolación lineal en el espacio rgb.

2.8.2. Geometry Shader

En esta etapa se realiza la creación de los cuadrángulos que representarán cada partícula. Por cada vértice que ingresa se generan 2 triángulos nuevos y se descarta el vértice original.

```
[maxvertexcount(4)]
void ParticleGS(point VSParticleOut In[1], inout TriangleStream<GSParticleOut> SpriteStream)
{
    [unroll]
    for (int i = 0; i < 4; i++)
    {
        GSParticleOut Out = (GSParticleOut)0;

        float4 position = float4(In[0].position, 0.2, 1) + g_fParticleSize * float4(↩
            g_positions[i], 0, 0);
```

```
        Out.position = mul(position, g_mViewProjection);
        Out.color = In[0].color;
        Out.texcoord = g_texcoords[i];
        SpriteStream.Append(Out);
    }
    SpriteStream.RestartStrip();
}
```

2.8.3. Pixel Shader

Esta etapa es la más sencilla de todas y simplemente consiste en asignar el color de fragmento de acuerdo a la densidad.

```
float4 ParticlePS(GSParticleOut In) : SV_Target
{
    return In.color;

}
```

3. PERFORMANCE

Aquí se presentan los primeros resultados de performance del algoritmo propuesto. Para ello se simuló un problema de *dam break* en una cavidad cuadrada de lado unitario. En la Fig.9 se puede observar un esquema del problema. El mismo consiste en dejar caer una columna de agua hasta impactar con el lado opuesto de la cavidad, teniendo en las paredes una condición de no deslizamiento.

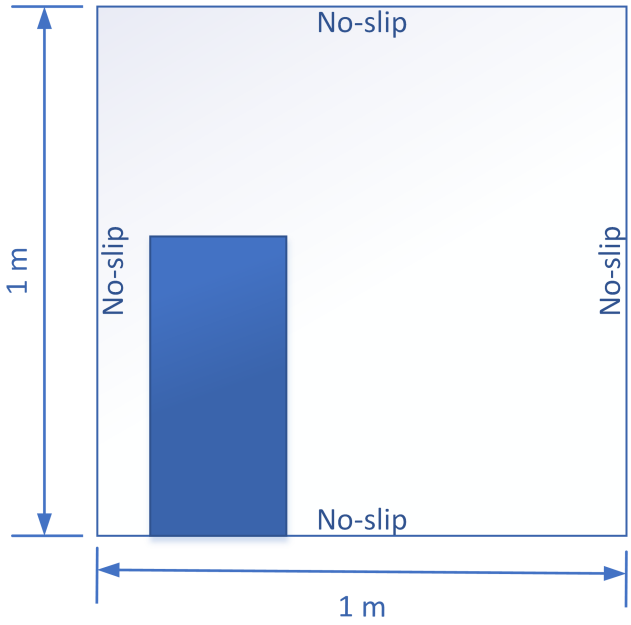
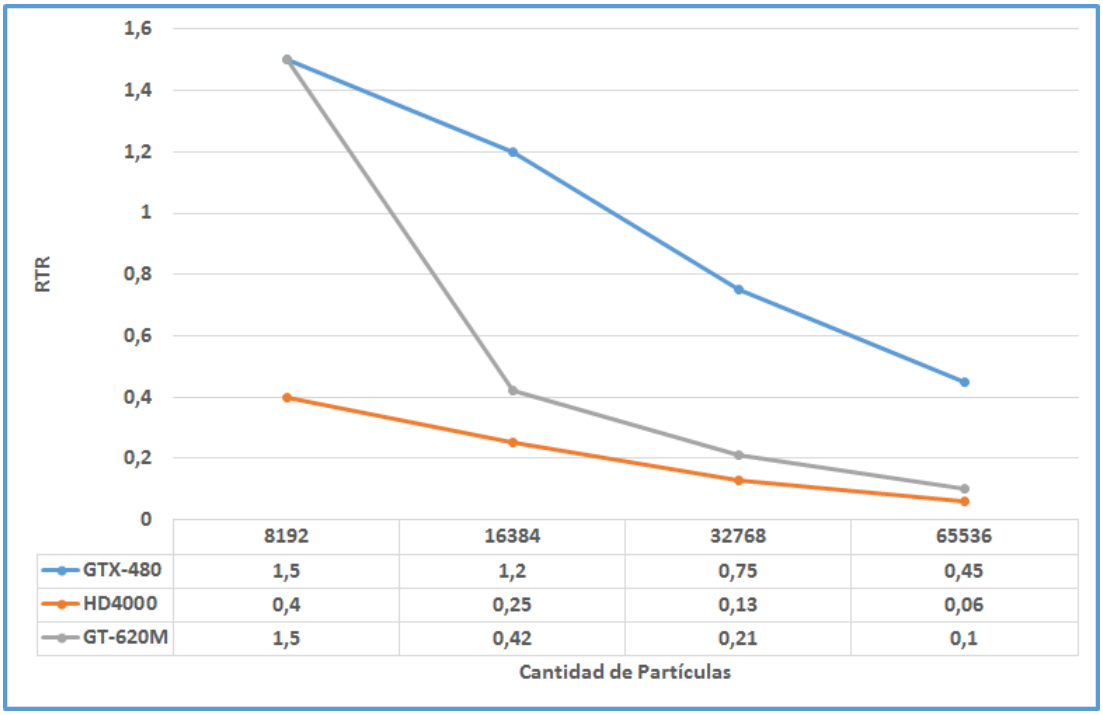


Figura 9: Cavidad cuadrada

Siendo el factor adimensional RTR (Real Time Ratio) definido como la razón entre el tiempo de simulación y el tiempo físico un parámetro representativo de la performance de nuestro algoritmo, se realizaron simulaciones para diferentes cantidades de partículas. El primer equipo utilizado fue una notebook con procesador i5-3317U y con una placa de video integrada Intel

HD4000. La siguiente prueba se realizó en una notebook con procesador i5-3317U y con una placa de video discreta NVidia GT-620M. Finalmente se realizó un experimento en una computadora de escritorio con procesador Intel i7 980X y una placa de video NVidia GTX-480. En la Fig.10 se pueden observar los resultados obtenidos. En algunos casos se ha alcanzado la meta del tiempo real, incluso en hardware no tan potente como es el caso de la GT-620M. A pesar de haberse alcanzado para un número pequeño de partículas, dicha cantidad es más que suficiente para aplicaciones de animación y videojuegos. Además como se muestra en Rojas Fredini y Limache (2013) se pueden obtener buenos resultados para aplicaciones de ingeniería con esa cantidad baja de partículas.



	8192	16384	32768	65536
GTX-480	1,5	1,2	0,75	0,45
HD4000	0,4	0,25	0,13	0,06
GT-620M	1,5	0,42	0,21	0,1

Figura 10: Resultados

Además es menester destacar que el algoritmo propuesto se ejecutó correctamente en todo el hardware propuesto, sin mostrar problemas de compatiblidad.

4. CONCLUSIONES

En el presente trabajo se mostró una implementación novedosa de SPH sobre la nueva arquitectura de cómputo DirectCompute, la implementación se realizó en el lenguaje HLSL lo que permitió unificar el pipeline de renderizado con el de cálculo, sin necesidad de realizar copias extras de memoria. El algoritmo además se ejecuta íntegramente en GPU evitando el cuello de botella de las copias entre memoria principal del sistema y la memoria de video, esto ha permitido alcanzar órdenes de ejecución en tiempo real para determinados problemas como se mostro anteriormente.

Si bien el algoritmo desarrollado se implementó sobre DirectCompute los autores consideran factible implementarlo sobre otras tecnologías de GPGPU como CUDA y OpenCL. Sería esperable obtener rendimientos similares, al menos para las rutinas de cálculo. Sobre la visualización es más difícil realizar afirmaciones hasta no llevar adelante las pruebas correspondientes, ya que tanto CUDA como OpenCL utilizan una capa de software adicional que les permite realizar la

comunicación con OpenGL y con DirectX.

Finalmente si bien no se puede asegurar a priori que DirectCompute presenta ventajas de performance sobre las otras alternativas mencionadas, el alto nivel de integración entre la visualización y el cálculo presenta un entorno de trabajo muy ágil y fluido.

REFERENCIAS

Charypar D., Müller M., y Gross M. Particle-based fluid simulation for interactive applications. En M. Lin y D. Breen, editores, *SIGGRAPH Symposium on Computer Animation*. 2003.

Cleary P.W. Modelling confined multi-material heat and mass flows using sph. En *Inter. Conf. on CFD in Mineral & Metal Processing and Power Generation CSIRO*, páginas 1–23. 1997.

Gourlay D.M.J. Fluid simulation for video games. *Intel online articles*, 2012.

Green S., Tonge R., Sainz M., Johnston D., y Schoemehl D. Fluid simulation in alice: Madness returns. Informe Técnico, Intel, 2011.

Group K. Opencl. http://www.khronos.org/opencl/, 2013.

Jeong J., Jhona M., Halowb J., y van Osdol. Smoothed particle hydrodynamics: Applications to heat conduction. *Computer Physics Communications*, 153:71–84, 2003.

Kipfer P. y Westermann R. Improved gpu sorting. *GPU Gems 2*, 2005.

Liu M. y Liu G. *Smoothed Particle Hydrodynamics: A Meshfree Particle Method*. World Scientific Publishing Co. Pte. Ltd., 2003.

Micikevicius P. 3d finite difference computation on gpus using cuda. En *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, páginas 79–84. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-517-8. doi:10.1145/1513895.1513905.

Microsoft. Directcompute. http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331(v=vs.85).aspx, 2013.

Monaghan J. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68:1703–1759, 2005.

Nuli U.A. y Kulkarni P.J. Sph based fluid animation using cuda enabled gpu. *International Journal of Computer Graphics and Animation*, 2012.

Nvidia. Cuda. http://www.nvidia.com/object/cuda_home_new.html, 2013a.

Nvidia. Directcompute para nvidia. http://www.nvidia.es/object/directcompute_es.html, 2013b.

Obrecht C., Kuznik F., Tourancheau B., y Roux J.J. Scalable lattice boltzmann solvers for {CUDA} {GPU} clusters. *Parallel Computing*, 39(6-7):259 – 270, 2013. ISSN 0167-8191. doi:http://dx.doi.org/10.1016/j.parco.2013.04.001.

Patrick J., Zhu Y., y Morris J. Modeling low reynolds number incompressible flows using sph. *Journal of Computational Physics*, 136:214–226, 1997.

Prashant G., Philipp S., Barbara S., y Renato P. Interactive sph simulation and rendering on the gpu. En *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, páginas 55–64. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2010.

Rojas Fredini P.S. y Limache A.C. Evaluation of weakly compressible sph variants using derived analytical solutions of taylor-couette flows. *Comput. Math. Appl.*, 66(3):304–317, 2013. ISSN 0898-1221. doi:10.1016/j.camwa.2013.05.008.

Rook R., Yildiz M., y Dost S. Modeling transient heat transfer using sph and implicit time integration. *Numerical Heat Transfer, Part B: Fundamentals*, 51:1–23, 2008.

Succi S. *The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond*. Numerical Mathematics and Scientific Computation. Clarendon Press, 2001. ISBN 9780198503989.

Vesterlund M. *Simulation and Rendering of a Viscous Fluid Using Smoothed Particle Hydrodynamic*. Tesis de Doctorado, VRLab, Umea University, 2004.