CEIG - Spanish Computer Graphics Conference (2017)
F. J. Melero and N. Pelechano (Editors)

An Improved Parallel Technique for Neighbour Search on CUDA

J.J. Perea and J.M. Cordero

University of Seville, Spain

Abstract

In Computer Graphics is usual the modelling of dynamic systems through particles. The simulation of liquids, cloths, gas, smoke... are highlighted examples of that modelling. In this scope, is particularly relevant the procedure of neighbour particles searching, which represents a bottleneck in terms of computational cost. One of the most used searching techniques is the cell–based spatial division by cubes, where each cell is tagged by a hash value. Thus, all particles located into each cell have the same tag and are the candidate to be neighbours. The most useful feature of this technique is that it can be easily parallelized, what reduces the computational costs. Nevertheless, the parallelizing process has some drawbacks associated with data memory management. Also, during the process of neighbour search, it is necessary to trace into the adjacent cells to find neighbour particles, as a consequence, the computational cost is increased. To solve these shortcomings, we have developed a method that reduces the search space by considering the relative position of each particle in its own cell. This method, parallelized using CUDA, shows improvements in processing time and memory management over other "standard" spatial division techniques. (see http://www.acm.org/about/class/class/2012)

CCS Concepts

•***Computing methodologies*** → ***Distributed computing methodologies;*** *Physical simulation;*

1. Introducción

En Computer Graphics es habitual el modelado de medios continuos, mediante sistemas de partículas. En este ámbito, es destacable el hecho de que, para obtener simulaciones realistas, el sistema de partículas debe mostrar un alto nivel de cohesión. Este comportamiento se modela mediante la interacción de cada partícula del sistema, con las partículas más cercanas, usualmente designadas cómo *partículas vecinas*. El número de partículas vecinas está condicionado por la distancia de interacción, comúnmente denominada *radio de influencia*. En términos de coste computacional, la búsqueda de vecinas representa un cuello de botella en la simulación, lo que requiere de técnicas que favorezcan el óptimo procesamiento. En caso contrario, el coste computacional se incrementa exponencialmente [GDNB10].

La técnica de búsqueda de vecinas más básica es la exhaustiva [GDB08], comúnmente denominada fuerza bruta. Consiste en calcular la distancia de cada partícula con todas las partículas del sistema, seleccionando sólo aquellas que están a una distancia menor o igual que el radio de influencia. Aunque esta técnica permite conocer las partículas vecinas, no es apropiada debido a su alto coste de computacional, del orden de $\mathcal{O}(n^2)$.

Debido a este alto coste computacional, se han desarrollado diversas técnicas para reducirlo. Las más utilizadas se basan en la división del espacio de simulación en celdas. En todas ellas, aunque con ciertas diferencias, se pueden distinguir dos etapas: la de *división espacial* y la de *análisis–asignación*. En la etapa de división espacial el espacio se divide en cubículos o *celdas*, y se relaciona cada partícula con la celda que la contiene. En la etapa de análisis–asignación se comprueba que la distancia entre las partículas contenidas en cada celda y las partículas de las celdas adyacentes, sea igual o menor al radio de influencia.

Aunque estas técnicas representan una mejora, en comparación con la búsqueda exhaustiva, requieren que las celdas estén organizadas en una estructura ordenada. Dos son las comúnmente utilizadas: la estructura en árbol [Ben75] y la ordenada mediante función hash [IABT11]. En síntesis, la estructura en árbol, consiste en subdividir el espacio en celdas de tamaño decreciente, desde una celda inicial, que podría abarcar a todo el espacio de simulación, hasta celdas cuyo volumen viene restringido por el radio de influencia. Con esta estructura es posible rastrear cada celda y sus adyacentes en un período de tiempo relativamente corto. Por otro lado, en cuanto a la ordenación basada en la función hash, se utiliza una función que "*genera*" un número entero, idealmente único, a partir del centro de cada celda. Gracias a los valores obtenidos de la función hash, se establece una organización que permite la localización rápida de cada celda y sus adyacentes.

Ambas técnicas son adecuadas para el proceso de búsqueda de vecinas, y permiten obtener resultados satisfactorios. Sin embargo, comparativamente existen dos características de la técnica basada en la función hash que destacan sobre la estructura en árbol. La primera es que se genera una estructura de búsqueda estática, que

DOI: 10.2312/ceig.20171201

no requiere ser recalculada en cada paso de la simulación. La segunda es que el cálculo del valor hash de cada celda se realiza individualmente, haciéndolo adecuado para su procesamiento paralelo [LH06].

En referencia al procesamiento en paralelo, existen diferentes arquitecturas, siendo la más destacada la tecnología CUDA, desarrollada por NVIDIA e incorporada en unidades de procesamiento gráfico (GPU). La piedra angular de esta tecnología es el gran número de núcleos de procesamiento. Cada núcleo de procesamiento puede ejecutar múltiples tareas en paralelo a altas velocidades. Aprovechando la potencia computacional ofrecida por CUDA, el coste computacional de la búsqueda de partículas vecinas se puede reducir drásticamente, haciendo viable la simulación en tiempo real [Kno09, RBG*12], incluso con un alto número de partículas. A pesar de las ventajas que ofrece CUDA, el manejo de la memoria de la GPU puede reducir la eficiencia del proceso de búsqueda de partículas en las celdas adyacentes, ya que la información puede encontrarse muy dispersa en la memoria.

Para evitar este problema de dispersión, se suelen emplear dos técnicas que optimizan la gestión de memoria: la primera se basa en establecer una relación entre el valor hash y la posición en la memoria. A partir de esta relación, se puede establecer un criterio de orden que permita aglutinar los datos dispersos. No obstante, aunque es posible reducir la dispersión de la información, se induce una pérdida de eficiencia computacional cuando se realiza la búsqueda de vecinas entre celdas adyacentes. Por otro lado, la segunda técnica de optimización se basa en reducir el flujo de información, durante el proceso de análisis, tanto como sea posible. El problema que se plantea es que existen procesos, como la concurrencia en la inclusión de datos de cada partícula, que requiere del uso de "*semáforos*", lo que aumenta el coste computacional.

Para mejorar la eficiencia del proceso de búsqueda de vecinas y evitar los problemas de la gestión de memoria, presentamos nuestra técnica. Se trata de un método basado en la división del espacio mediante celdas cúbicas, en las que utilizamos una función hash para etiquetarlas. En nuestra propuesta, utilizamos la posición relativa de cada partícula, dentro de su celda contenedora, para reducir el número de celdas adyacentes donde buscar partículas vecinas. De este modo, logramos dos ventajas: la primera es que reducimos el número de consultas a los datos contenidos en la memoria, con lo que reducimos el flujo; la segunda es que optimizamos la memoria consumida ya que evitamos la redundancia de información que se requiere cuando la búsqueda se extiende a todas las celdas adyacentes a una dada.

El resto del artículo está organizado del siguiente modo: En la sección 2, describiremos las investigaciones más relevantes en el campo de la búsqueda de partículas vecinas. Estas investigaciones establecen el ámbito en el que se sitúa nuestra técnica. En la sección 3, describiremos los fundamentos de la arquitectura CUDA, centrándonos en sus capacidades y limitaciones, ya que son importantes para el desarrollo de nuestra propuesta. En la sección 4, describiremos los fundamentos de la técnica estándar, como la usada por Green [Gre10], para la división del espacio en celdas, el etiquetado mediante la función hash, así cómo las limitaciones de la metodología estándar. Así, estableceremos el entorno conceptual para describir nuestra técnica. En la sección 5, describiremos los fundamentos de la técnica propuesta, sus capacidades y las bases para su implementación en CUDA. En la sección 6, mostraremos los resultados que se obtienen con nuestra técnica y las ventajas que ésta proporciona. Para ello, la compararemos con la técnica de división espacial estándar [Gre10]. Finalmente, en la sección 7, expondremos las conclusiones a partir de los resultados obtenidos.

2. Trabajos Relacionados

En la búsqueda de vecinas mediante división espacial, destaca la investigación pionera desarrollada por Bentley [Ben75], en la que se describe la división jerárquica del espacio de simulación y la relación, mediante una estructura en árbol, de cada una de las jerarquías obtenidas. Importantes mejoras se han ido introduciendo tomando como base esta propuesta. Una de las más importantes, es la presentada por Kumar et al. [KZN08]. En ella, se propone que cada subdivisión esté alineada con los ejes del espacio de simulación; de este modo, según los autores, se reduce el número de "consultas" a realizar durante el proceso de búsqueda. Aunque diversos estudios destacan las ventajas que ofrecen las técnicas basadas en estructura en árbol [BP11, XQ16], es sabido que presentan importantes limitaciones si se requiere que el árbol tenga un gran número de niveles y tenga que recalcularse y balancearse continuamente [PDC*03], como suele ser habitual en la simulación de fluidos.

Usualmente, la alternativa para superar las limitaciones de la estructura en árbol es el uso de métodos de discretización no jerárquica, en los que el espacio de simulación se divide en celdas de igual tamaño, conexas y no solapadas. Estas técnicas son especialmente eficientes en el procesamiento en paralelo [IABT11]. En este ámbito, especial mención merece el estudio desarrollado por Harada et al. [HSK07], donde la relación partícula–celda contenedora se realiza a través de la coordenada de textura. Por lo tanto, cada celda es codificada mediante un píxel y se le asigna al espacio computacional tridimensional. Garcia et al. [GDB08] realiza un exhaustivo estudio de esta técnica y en sus conclusiones destaca dos importantes características, estas son, el acceso rápido a los datos de la memoria y la posibilidad de ser paralelizada. Rozen et al. [RBA08] proponen un método más óptimo basado en la utilización de la memoria de textura ubicada en el caché del procesador. En una linea de investigación alternativa, se sitúa el estudio llevado a cabo por Ihmsen et al. [IABT11]. En esta investigación se describe cómo cada partícula del sistema se etiqueta, en base a su posición, mediante una función hash, lo que representa, según los autores, una ventaja a la hora de la velocidad de procesamiento.

Diversos estudios destacan las ventajas de las técnicas basadas en el etiquetado mediante función hash [THM*03, WBK07, GDB08, FWZS11]. En estas investigaciones se advierte de un requisito fundamental: la función hash debe garantizar la unicidad de cada uno de los valores obtenidos, sin que ello induzca un excesivo coste computacional. Testler et al. [THM*03] presentan una función basada en operaciones lógicas excluyentes. Aunque su formulación permite obtener una gestión adecuada de los datos, presenta dos limitaciones: la primera es que no garantiza totalmente unicidad de los resultado, pudiendo aparecer problemas de colisión, la segunda es que los valores obtenidos son muy grandes. Este hecho induce la dispersión de los datos en la memoria, lo que reduce efi-

ciencia a la hora de gestionarlos. Un modo habitual de reducir esta dispersión, es modular los valores obtenidos de la función hash. Esta opción, además, reduce el consumo de memoria. Sin embargo, con este proceso el número de valores que colisionan aumenta exponencialmente, lo que requiere de un proceso selectivo que detecte estos casos, los filtre y los gestione de manera alternativa. Con todo ello, se puede mejorar la gestión y el consumo de memoria, pero se reduce la eficiencia [Cay12]. Con el objetivo de resolver este problema, Fan et al. [FWZS11] propone una función hash que reduce los casos de colisión y cuyos resultados no requieren ser modulados para optimizar el uso de la memoria. Aunque en muchos casos la propuesta cumple sus objetivos, en situaciones donde existe una alta densidad de partículas en unas celdas y muy baja en otras próximas, como ocurre en las interfases y contornos cuando se simulan fluidos, se produce una pérdida de información que afecta al realismo de la simulación [Cay12].

Uno de los factores a tener en cuenta al implementar técnicas basadas en el etiquetado mediante función hash, es el tamaño de las celdas. Wróblewski et al. [WBK07] desarrollan un estudio detallado sobre la relación entre el tamaño de la celda y la eficiencia. Su estudio se centra en dos tamaños de arista de la celda, estos son, R y $2R$, donde R es el radio de influencia. En una línea de investigación similar, Viccione et al. [VBC08], estudian cómo afecta el tamaño de la celda a la eficiencia y, además, analizan las características de la función hash utilizada. Estos estudio alertan de un hecho fundamental que afecta a la eficiencia del proceso: la búsqueda de partículas vecinas no sólo se circumscribe a la celda que contiene cada partícula, sino a las adyacentes, 26 en el caso tridimensional. Ihmsen et al. [IABT11] desarrolla una técnica para optimizar el proceso de búsqueda de partículas vecinas en las celdas adyacentes. Su propuesta se basa en utilizar celdas cuya valor de arista es R. Con este tamaño, Ihmsen et al., aseguran que se reduce el número de celdas adyacentes en las que buscar vecinas. Sin embargo, con este tamaño arista, se plantean dos problemas: el primero es el incremento del uso de memoria, ya que la disminución del tamaño de la celda incrementa el número de celdas a gestionar, el segundo es que no se garantiza que todas las partículas vecinas sean detectadas como tales. De sus conclusiones se deduce que el aumento del tamaño de la celda mejora los resultados.

Una de las principales ventajas que presenta la gestión de la búsqueda de vecinas mediante etiquetado hash es la posibilidad de ser paralelizada. En este ámbito varias investigaciones aprovechan las capacidades del procesamiento paralelo mediante GPU [SGS10, Cud12]. Especial relevancia presenta la tecnología CUDA, vinculada al hardware de NVIDIA. La principal característica que presenta esta tecnología es su alta capacidad de procesamiento en paralelo a un coste relativamente bajo. Sin embargo, la paralelización con CUDA puede implicar problemas de gestión de la memoria, especialmente los errores relacionados con la caché [IABT11]. Para superar esta barrera, Goswani et al. [GSSP10] desarrollan una investigación en la que se describe el problema de la sobrecarga de memoria. Para reducirlo, proponen utilizar una técnica basada en la denominada *index–z*, también conocida como *curva–z* [BMJFS01]. Asimismo, Domínguez et al. [DCGG13] realizan una descripción exhaustiva de los cuellos de botella que aparecen en el proceso de búsqueda de vecinas. Establecen que una relación entre las partículas y la celda que las contiene, mediante la función hash, puede reducir los problemas de sobrecarga de la memoria. Además, destacan que, la ordenación en la memoria de los valores hash obtenidos, mejora la eficiencia del proceso de búsqueda.

Diversos estudios se han centrado en determinar los mejores algoritmos de ordenación utilizados para agrupar la información almacenada en memoria [Akl14]. Algunos de los más relevantes son Bitonic Sort [PSHL10] o el Sample Sort [LOS10], entre otros. Si bien, en CUDA, la versión más reciente del algoritmo Radix Sort [SHG09, Hwu11] es una de las más utilizadas. Ello se debe a su alta velocidad de ordenación que no requiere de una clasificación previa de los datos a ordenar.

3. Fundamentos de la Arquitectura CUDA

CUDA (Compute Unified Device Architecture) es una plataforma de programación lanzada por NVIDIA, que permite el cálculo intensivo utilizando sus GPUs. Con CUDA se utiliza la capacidad de procesamiento paralelo de la GPU para reducir los tiempos de procesamiento, soportando diferentes marcos de programación. Se puede utilizar en una amplia gama de software gráfico de NVIDIA, a partir de la serie GeForce 8 (arquitectura Tesla).

La estructura CUDA se organiza jerárquicamente tanto para los elementos de proceso como para la memoria. Los elementos de proceso son: multiprocesadores (*multiprocessors*), núcleos (*cores*) e hilos (*threads*), Mientras que los tipos de memoria son: memoria global (*global memory*), memoria compartida (*shared memory*) y memoria local (*local memory*). Dentro de los elementos de proceso, cada multiprocesador está compuesto de varios núcleos CUDA que, a su vez, contienen grupos de hilos de ejecución. Cada hilo tendrá acceso a su propia memoria local, que no puede ser accedida por cualquier otro hilo del mismo grupo. Si los hilos del mismo grupo necesitan compartir información durante el procesamiento, debe utilizarse la memoria compartida. La memoria compartida tiene un ancho de banda muy superior a la global. Sin embargo, la latencia de la memoria global es muy superior en comparación con la de la memoria compartida [NVI15]. La consecuencia es que la escritura y el acceso a los datos contenidos en la memoria compartida, es más lento que en la memoria local, por lo que puede ralentizar el procesamiento [Mic12].

En referencia al procesamiento, éste se realiza en grupos de 32 hilos, llamados tramas (*warps*). Una trama es la unidad de datos mínima manejada por un multiprocesador. Esta estructura mínima en tramas favorece el optimizado del número de operaciones, lo que inducirá una mejora en el rendimiento. Una característica importante es que no sólo los hilos del mismo grupo pueden compartir información, sino que las diferentes tramas también pueden intercambiar datos usando la memoria global. La importancia de la memoria global es esencial, ya que el tamaño de los grupos de hilos es limitado y requiere la sincronización entre diferentes grupos para lograr un paralelismo óptimo y masivo [Cud12]. Esta sincronización sólo puede llevarse a cabo en la memoria global. No obstante no debe abusarse del uso de la memoria global, ya que el acceso a la información contenida en ella ralentiza el procesamiento.

Para optimizar el rendimiento en las implementaciones de CUDA, deben tenerse en cuenta las limitaciones de su arquitectura, en cuanto a gestión de memoria y ejecución de GPU. Para una

gestión óptima de la memoria, los datos almacenados en la memoria deben estar muy próximos [IABT11]. Esta limitación está relacionada con la caché, así que, cuanta más información se transfiera a la caché, en cada solicitud de datos, menos solicitudes serán necesarias realizar y, por lo tanto, más rápido será el procesamiento paralelo [Ros13]. Para que este proceso se realice de manera eficiente, es necesario que los datos, en la memoria compartida, estén almacenados secuencialmente, de manera que en única petición se transfiera la máxima información posible. Esta accesibilidad secuencial, denominada *acceso coalescente*, es uno de los puntos más importantes a la hora de operar con CUDA. El método más habitual para evitar la dispersión de datos en la memoria es el uso de algoritmos de clasificación, de modo que, partiendo de la información dispersa en la memoria, se establezca una vinculación entre los datos y, con esta vinculación, se fije un criterio de ordenación que disponga todos los datos consecutivamente. Existen varios algoritmos de clasificación, como se describe en [SHG09,Hwu11], pero el que se considera más eficiente es el denominado Radix Sort [MG10].

Por otro lado, el flujo de ejecución de CUDA depende del procesamiento de instrucciones condicionales ejecutadas en una misma trama, ya que éstas introducen diferentes vías de ejecución. Como CUDA pasa por todas estas posibles vías de ejecución de modo consecutivo [Cud12], se establece un periodo de "stand–by" de todos los hilos de esa trama hasta que se alcanza la via adecuada para cada hilo. Así, un uso excesivo de estas instrucciones condicionales reduce el nivel de paralelismo de las funciones que las utilizan [FSYA07].

4. Búsqueda de Vecinas

La técnica de la búsqueda de partículas vecinas, basada en la división espacial, se asienta en un hecho fundamental: todas las partículas tienen sus vecinas dentro de su celda contenedora y, eventualmente, en las celdas adyacentes. En este contexto, es necesario relacionar el radio de influencia de la partícula y el tamaño de la celda. Además, es relevante conocer, la estructura de la arquitectura que se va a emplear en el procesamiento, ya que la gestión del flujo de datos y de la memoria, condicionan la eficiencia obtenida. En referencia a esta última consideración, una de las técnicas más empleadas, utilizando la tecnología NVIDIA, es la descrita en [Gre10].

Para que esta técnica funcione, es necesario establecer una conexión, uno a uno, entre las partículas y las celdas que las contienen. Así mismo, es necesario fijar un criterio de clasificación entre diferentes celdas. La forma habitual de relacionar celda–partícula es el etiquetado mediante funciones hash. De esta manera, las celdas contenedoras y las partículas contenidas están unívocamente relacionadas, como se ilustra en la Figura 1. Además, a partir de los valores hash asociados a cada celda, se puede establecer una estructura de ordenación.

Descriptivamente, esta técnica se puede dividir en dos etapas: la de división espacial y la de análisis–asignación.

En la etapa de división espacial, se establece el tamaño óptimo para cada celda, por lo general todas de igual tamaño; en Wróblewski et al. [WBK07] se desarrolla un estudio sobre el

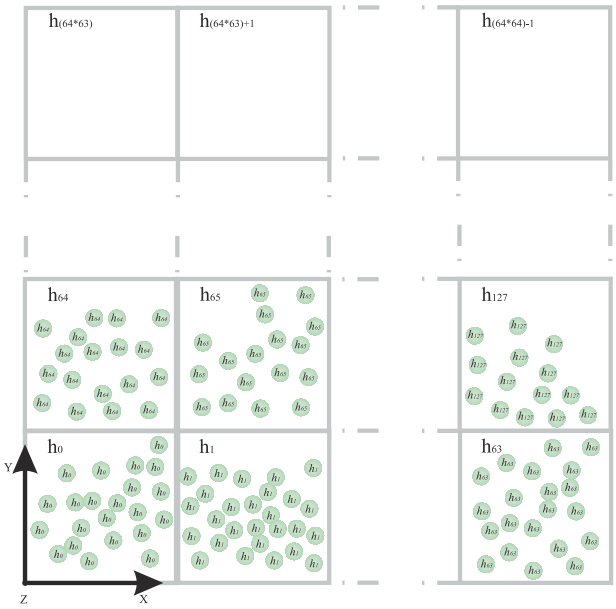


Figura 1: *Representación de las partículas y de celdas en las que se divide el espacio. El valor hash de cada partícula coincide con el de la celda que la contiene.*

tamaño óptimo que deben tener las celdas. Seguidamente, se localiza el centro de cada celda y se obtiene un valor entero utilizando una función hash. Este valor, idealmente único, permite "etiquetar" la celda. La misma función hash, se utiliza con las partículas. De este modo, a partir de su posición, las partículas y su celda contenedora tienen el mismo valor hash. Cabe destacar que, en esta etapa se debe comprobar cuales son las celdas que están ocupadas, ya que, para evitar un desbordamiento de memoria, sólo los valores hash de estas celdas ocupadas deben almacenarse en la memoria.

En la etapa de análisis–asignación, se determinan las vecinas de cada partícula. En esta fase, se calcula la distancia, de cada partícula, con las otras partículas que se encuentran dentro de la misma celda, esto es, las partículas con el mismo valor hash. Si la distancia es la igual o inferior que el radio de influencia establecido, entonces las partículas son vecinas. No obstante, cómo el radio de influencia define una esfera cerrada centrada en la partícula y la celda es generalmente cúbica, es necesario completar la búsqueda de vecinas en celdas adyacentes.

Como el proceso de etiquetado y asignación de vecinas son procesos independientes para cada partícula, es fácilmente paralelizable, por ejemplo utilizando CUDA. A pesar de las ventajas que su uso puede presentar, existen limitaciones en la arquitectura de CUDA relacionadas con la gestión de memoria. Especialmente, en lo referente a la dispersión de datos. El objetivo es que esta dispersión sea lo mínima posible para favorecer accesos coalescentes a los datos almacenados en la memoria. Esta restricción condiciona la función hash que debe utilizarse, ya que los valores obtenidos pueden inducir, en mayor o menor medida, a la dispersión de los datos. Esta dispersión puede tener un efecto aún más "nocivo", en la eficiencia del proceso, ya que hay que considerar la búsqueda de

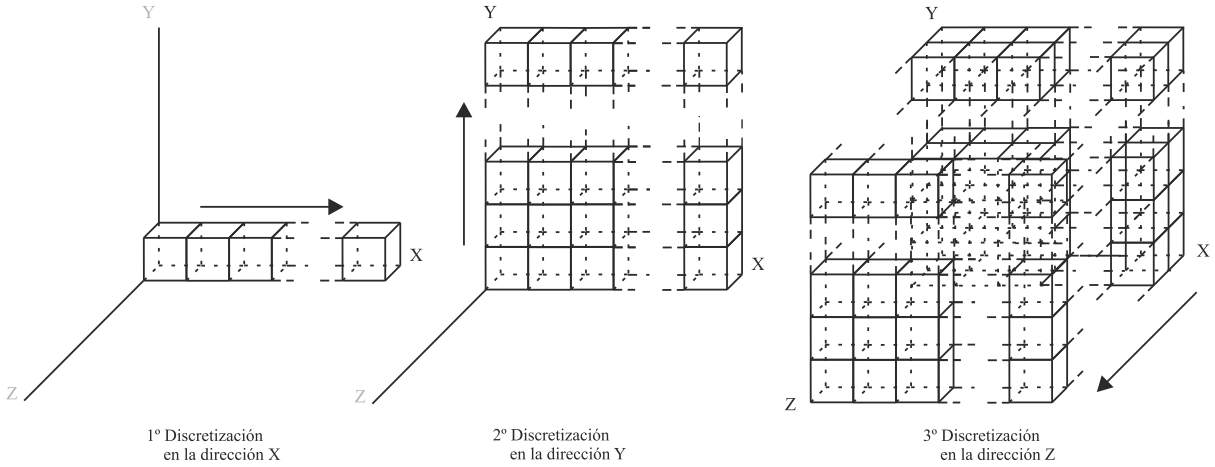


Figura 2: *Esquema del orden de discretización del espacio de simulación. Este orden favorece la coalescencia de los datos en la memoria.*

partículas vecinas en las 26 celdas adyacentes, para el caso tridimensional. Por esta razón, se requiere tanto el uso de una función hash que induzca la minima dispersión posible, como el desarrollar una metodología que reduzca el número de celdas adyacentes en las que buscar partículas vecinas. A partir de estas dos premisas formulamos nuestra técnica de búsqueda de vecinas que se explicará en la siguiente sección.

5. Modelo Propuesto

En esta sección vamos a describir nuestra técnica propuesta para la búsqueda de partículas vecinas. Describiremos el proceso de discretización espacial seguido, el hash utilizado y la técnica por la que reducimos el número de celdas adyacentes en las que buscar vecinas. Todo ello optimizado para su implementación en la arquitectura CUDA.

Descriptivamente, como toda técnica basada en la división espacial, distinguimos dos etapas. En la etapa de división espacial, desarrollamos una discretización en celdas hexaédricas, todas ellas de igual tamaño. Cada una de estas celdas, queda etiquetada a través de una función hash, cuyas características serán descritas posteriormente. Vamos a utilizar la misma función hash para etiquetar las partículas y las celdas, de este modo, conseguimos emparejar cada partícula con la celda que la contiene. En la segunda etapa, la de análisis–asignación, vamos a localizar la ubicación relativa de cada partícula dentro de la celda. Esta posición relativa nos permitirá reducir, significativamente, el proceso de búsqueda en las celdas adyacentes. La consecuencia directa es que disminuiremos las peticiones a la memoria global, que es la más lenta en el procesamiento [LCT14].

En el algoritmo 1 se describen los principales pasos para la implementación de la técnica propuesta. En las secciones 5.1 y 5.2 realizaremos una descripción más detallada de este algoritmo.

5.1. Etapa de División Espacial

Está demostrado que tanto el tamaño de las celdas, cómo el orden seguido en la búsqueda de partículas vecinas en las celdas adyacentes, condiciona la eficiencia del proceso [WBK07]. En base a este estudio, nuestra propuesta es discretizar todo el espacio con celdas de igual tamaño y en sentido dextrógiro, es decir, primero discretizamos en la dirección X, luego en la Y y finalmente en la Z, ver Figura 2.

Esta elección ha estado condicionada por la arquitectura CUDA; donde los bloques de hilos, alcanzan las instrucciones de manera similar a las instrucciones simples en el código lineal, y, por lo tanto, la distancia entre los datos en memoria –que son creados, modificados y consultados– es tal, que hace que el orden $X \rightarrow Y \rightarrow Z$ sea el más óptimo para nuestro propósito. Este hecho es válido siempre que se use un orden dextrógiro de discretización, es decir, $X \rightarrow Y \rightarrow Z$, $Z \rightarrow X \rightarrow Y$ o $Y \rightarrow Z \rightarrow X$ [MCPC15].

Algorithm 1 Proceso propuesto para la búsqueda de vecinas.

Require: El espacio debe ser discretizado en celdas de longitud l. Cada celda está etiquetada mediante función hash (1).
Ensure: Cada partícula conoce todas sus partículas vecinas y sus posiciones.
Input: Posiciones de las partículas, Radio de influencia.

```
1:  for Cada partícula do
2:      Evaluar su valor hash, que coincide con el valor hash de su
        celda.
3:      Etiquetar la partícula con el valor de hash de su celda.
4:  end for
5:  Ordenar los valores hash
6:  for Cada partícula a do
7:      Determinar el octante donde se ubica la partícula.
8:      Calcular el subconjunto de celdas adyacentes N_Adj.
9:      for Cada partícula b contenida en una celda de N_Adj do
10:         Calcular la distancia entre a y b.
11:         if (Distancia <= Radio de Influencia) then
12:             b es vecina de a
13:         end if
14:     end for
15: end for
```

El siguiente paso es etiquetar cada celda mediante la función hash. En nuestro caso, hemos formulado la función (1):

$$Hash(x,y,z) = x_{trunc}(\varepsilon_z - 1) + y_{trunc}\,\varepsilon_y\,\varepsilon_x + z_{trunc}\,\varepsilon_x\,\varepsilon_z \quad (1)$$

donde ε_x, ε_y, ε_z son el número de subdivisiones en las direcciones x, y y z respectivamente, y x_{trunc}, y_{trunc} y z_{trunc} son las coordenadas truncadas de cada partícula, en base a la ecuación (2).

$$x_{jtrunc} = \left\lfloor \frac{x_j}{R} \right\rfloor \&\&(\varepsilon_{x_j} - 1) \quad \forall j = 1,2,3; \quad (2)$$

donde && es el operador lógico AND, x_{1trunc}, x_{2trunc}, x_{3trunc} hacen referencia a x_{trunc}, y_{trunc}, z_{trunc} y, por otro lado, ε_{x_1}, ε_{x_2} y ε_{x_3} se refiere a ε_x, ε_y y ε_z respectivamente.

La expresión (1), ha sido obtenida a partir de la función hash formulada por [FWZS11]. Hemos realizado algunos cambios, sobre todo en los multiplicadores de x_{trunc} y z_{trunc}, con el fin de garantizar la unicidad de los valores hash obtenidos y favorecer que las celdas próximas en el espacio tengan asociados valores hash que se sitúen cercanos en la memoria. Una vez que las celdas quedan etiquetadas con el valor hash, el siguiente paso es determinar el número de partículas contenidas en cada celda. Para ello, utilizamos la ecuación (1), con lo que podemos emparejar celdas y partículas.

Llegados a este punto, cabe destacar que, aunque partículas y celdas estén emparejadas, esta "información" no se almacena, generalmente, de manera ordenada y consecutiva en la memoria. Esto tiene un impacto negativo a la hora de realizar los accesos a la memoria, ya que se necesitan múltiples solicitudes a la memoria, para obtener toda la información de las partículas que se encuentran en una celda, y mucho más para la "consulta" sobre las celdas adyacentes. Esto se debe a que la eficiencia de los modelos implementados en CUDA dependen, en gran medida, de las transferencias de información desde, y hacia, la memoria. En consecuencia, se obtienen mejoras significativas de rendimiento minimizando la cantidad de accesos. En nuestro caso, vamos a reducir la cantidad de peticiones a la memoria, mediante la ordenación de los datos a través de su valor hash asociado. Existen diversos algoritmos de clasificación adecuados para su implementación en paralelo, en nuestro modelo hemos optado el algoritmo de Radix Sort de NVIDIA [HB10]. Esta elección viene condicionada por la discretización espacial utilizada, mostrada en Figura 2, donde los hash de los bloques de discretización en las direcciones X, Y y Z están muy próximos. La consecuencia es que en cada solicitud de datos a la memoria obtenemos información relevante de cada celda vecina. Este hecho representa una mejora respecto de otras técnicas que llevan a cabo una indexación basada curvas fractales, tales como la curva de Hilbert [SA08] o la curva–z [BMJFS01]. Aunque la indexación puede ubicar datos muy proximos en la memoria, existen situaciones en las que celdas próximas en el espacio están alejadas en la memoria [XQ16]. Además, el proceso de indexación de los hash en la memoria es más lento que la simple ordenación, lo que al final induce una pérdida de eficiencia [LCT14].

Una vez descrita la primera etapa de la técnica propuesta, esto es, la función hash y el proceso para optimizar la distribución de los datos en la memoria, vamos a describir la segunda etapa. En ella explicaremos cómo localizamos la partícula dentro de cada celda y cómo su posición relativa va a reducir el número de celdas adyacentes en las que buscar partículas vecinas.

5.2. Etapa de Análisis–Asignación

El objetivo primordial en esta segunda etapa, es ubicar cada partícula dentro de su celda contenedora, que denominaremos C. Para ello, en el caso tridimensional, vamos a utilizar tres planos secantes perpendiculares entre sí que se cortan en el centro de C. De este modo, definimos 8 octantes, de lado $l/2$, cada uno de los cuales lo designamos cómo $C'i \;\; \forall i = 1,\ldots,8$. En esta estructura, definimos un origen local, que denominaremos cómo O, situado en vértice mínimo de la celda, es decir, $O = (x_{min}, y_{min}, z_{min})$. Esta descomposición no afecta a la celda original C, que mantiene su tamaño para el resto de operaciones, y sólo se ha realizado para mejorar el proceso de localización relativa de cada partícula. La división "*virtual*" descrita se muestra en la Figura 3.

La existencia de los octantes, y más concretamente de los límites de cada octante, va a permitir detectar la posición relativa de la partículas. Cuantitativamente, el proceso de ubicación lo realizamos a través de la ecuación (3).

$$\left.\begin{array}{l} x_{jmin} \le x_j < x_{jmin} + (l/2) \\ x_{jmin} + (l/2) \le x_j \le x_{jmin} + l \end{array}\right\} \forall j = 1,2,3 \quad (3)$$

donde $x_1 = x$, $x_2 = y$ y $x_3 = z$ son las coordenadas de las partículas.

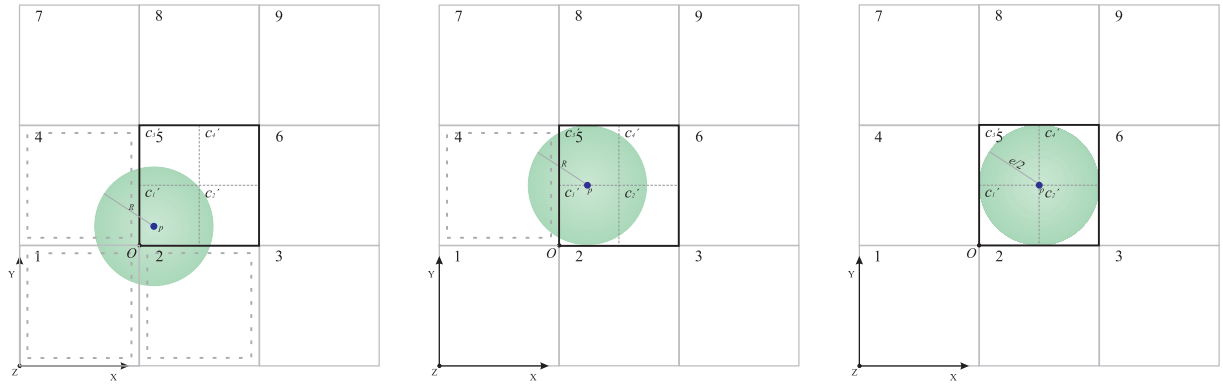
Una vez que la partícula queda ubicada dentro de C, seleccionamos las celdas adyacentes donde buscar partículas vecinas. A partir del orden seguido en la división espacial, descrito en la sección 5.1, establecemos el rango de celdas donde realizar la búsqueda, según la ecuación:

$$\alpha_{x_j} R^{\delta(j-1)} \varepsilon_{x_j}^{(j-1)} \le C_{x_j} \le \beta_{x_j} R^{\delta(j-1)} \varepsilon_{x_j}^{(j-1)} \quad (4)$$

donde δ es la función delta de Dirac, C_{x_j} es la celda incrementada en la dirección x_j y α_{x_j} y β_{x_j} son coeficientes que satisfacen la ecuación:

$$\begin{array}{llll} \alpha_{x_j} = 0; & \beta_j = 1 & \text{si} & x_{jmin} \le x_j < x_{jmin} + (l/2) \\ \alpha_{x_j} = -1; & \beta_j = 0 & \text{si} & x_{jmin} + (l/2) \le x_j \le x_{jmin} + l \end{array}$$

El proceso descrito se utiliza en el caso más general, véase Figura 3a, sin embargo, existen otros dos casos en los que el número de celdas adyacentes a considerar es menor. En el primer caso, que denominamos *caso secante*, la partícula se encuentra en un plano secante, ver Figura 3b, aquí las celdas adyacentes en las que buscar, son las "cortadas" por el plano secante. En el segundo caso, que designamos *caso central*, la partícula está situada en la intersección de los planos secantes, véase Figura 3c. Este es el caso más simple, ya que sólo necesitamos buscar las partículas vecinas en su propia celda. A partir de la ecuación (4), podemos deducir, para el caso tridimensional, el número de celdas adyacentes en las que buscar partículas vecinas. En el caso general, mostrado para $2D$ en la Figura 3a, serían 9 celdas adyacentes; para el caso secante,



(a) *Caso general.* **(b)** *Caso secante.* **(c)** *Caso central.*

Figura 3: *Posiciones relativas, en proyección bidimensional, de la partícula dentro de la celda. El contorno discontinuo indica las celdas adyacentes donde deben buscarse las partículas vecinas.*

Figura 3b, el número de celdas adyacentes serían 3, mientras que en el caso central, Figura 3b, sólo se opera con la celda que contiene a la partícula. Comparativamente, estos valores son muy inferiores a las 26 celdas adyacentes, a la que hay que añadir la celda contenedora, que hay que considerar en los modelos existentes.

Una vez filtrado el subconjunto de celdas candidatas a contener partículas vecinas, el siguiente paso es comprobar si la distancia, con las partículas contenidas en estas celdas, es menor, o igual, que el radio de influencia. De ser así, asignamos esas partículas como vecinas.

A pesar de los cálculos adicionales para localizar la posición relativa de las partículas, conseguimos reducir el tiempo de procesamiento y ganar eficiencia, ya que es menor el número de celdas adyacentes en las que buscar. Esta afirmación se evidenciará en los resultados mostrados en la sección 6.

6. Resultados

En esta sección vamos a llevar a cabo dos tests de eficiencia. El objetivo es mostrar las capacidades y mejoras que se obtienen con el modelo propuesto. En el primer test cuantificamos el tiempo de procesamiento asociado a la búsqueda de partículas vecinas entre las celdas adyacentes. En el segundo test, se compararán los tiempos de procesamiento obtenidos con el modelo estándar [Gre10] y con el modelo propuesto. Ambas pruebas se realizarán teniendo en cuenta el aumento del número de partículas, desde 1000 a 200000, y para diferentes tamaños de la celda contenedora.

Para llevar a cabo estas pruebas, hemos simulado un sistemas de partículas, cuya interacción es modelada mediante una fuerza que disminuye, proporcionalmente, con la distancia. Esta fuerza está limitada por el radio de influencia. Además, las partículas están sometidas a la fuerza de gravedad. Este modelo tiene características similares al desarrollado por Green [Gre10]. Sin embargo, la implementación de Green no está diseñada para pruebas de rendimiento. Para la integración temporal, utilizamos el Método de Euler de segundo orden. La implementación se ha realizado en CUDA, utilizando una GPU de características técnicas relativamente limitadas, como es una NVIDIA GTX780. Así mostramos las mejoras que ofrece nuestra propuesta aún en un equipo limitado.

Los valores de los parámetros comunes utilizados en los dos tests son los siguientes: masa de partículas $m = 3 \cdot 10^{-4}\,kg$, constante de proporcionalidad de fuerza $k = 3600\,N/m$ y paso de tiempo $\Delta t = 2 \cdot 10^{-4}\,s$. Trabajaremos con tres longitudes de arista, l, diferentes: $2R$, $2.52R$ y $4R$, donde R es el radio de influencia. El objetivo de trabajar con diferentes tamaños, es mostrar cómo este parámetro condiciona la eficiencia. Hemos optado por un tamaño mínimo de celda de $2R$ para que, al menos, contenga una esfera de radio R. Para tamaños inferiores, lo que ocurrirá es que el número de casos generales (como el mostrado en la Figura 4a) será mayor, lo que aumenta el número de consultas a las 9 celdas adyacentes. La consecuencia es que no se obtendrán, necesariamente, valores de tiempo más cortos. No obstante, si el tamaño de la celda es muy grande, el número de partículas candidatas aumenta, lo que inducirá una ralentización del proceso.

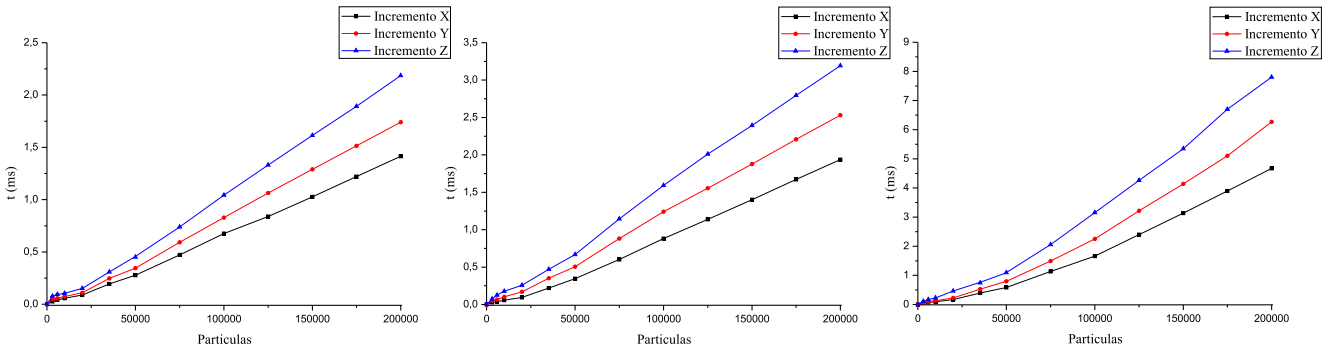
6.1. Dependencia Temporal para la Búsqueda en Celdas Adyacentes

En esta primera prueba mostramos el "consumo" de tiempo al buscar partículas vecinas entre las celdas adyacentes. El proceso que hemos seguido es el siguiente: en primer lugar, medimos el tiempo consumido en la búsqueda de vecinas en la celda contenedora. A continuación, rastreamos en las celdas adyacentes, en cada dirección y de manera ordenada, es decir, primero se incrementa en la dirección X –designado como incremento X en la Figura 4–, después la Y –designado como incremento Y en la Figura 4– y finalmente la Z –designado como incremento Z en la Figura 4–. Para no falsear los resultados, y mantener su coherencia, imponemos que en todas las celdas adyacentes, se ubiquen el mismo valor medio de partículas. Estos valores se muestran en la Tabla 1. Con estas condiciones, los resultados obtenidos se muestran en las gráficas de la Figura 4.

Con esta prueba se justifica el orden de división considerado y la función de hash utilizada, ya que el incremento en el tiempo de

	1000	2000	5000	10000	20000	50000	100000	200000
$\overline{N}(i)$ $[l = 2R]$	10	13	17	21	30	41	50	62
$\overline{N}(i)$ $[l = 2.52R]$	15	21	31	43	57	69	78	83
$\overline{N}(i)$ $[l = 4R]$	26	31	45	56	65	74	81	96

Table 1: *Número medio de partículas vecinas, $\overline{N}(i)$, obtenido para diferentes tamaños de celda. Estos son los valores asociados al tiempo de procesamiento en cada dirección, mostrados en las gráficas de la Figura 4*



(a) $l = 2R$ **(b)** $l = 2.52R$ **(c)** $l = 4R$

Figura 4: *Gráficas del tiempo de procesamiento para la segmentación (o incremento) en cada dirección del espacio. El incremento se realiza según al esquema de discretización de la Figura 2.*

ejecución es lineal, independientemente de la posición ocupada por las celdas adyacentes.

6.2. Dependencia Temporal del Tamaño de la Celda

Con esta prueba comparamos los tiempos de procesamiento obtenidos con el modelo estándar y con nuestra propuesta. Al igual que en el test anterior, las pruebas se realizan variando el tamaño de la celda. A partir de esta comparativa, se podrá analizar las mejoras de nuestra propuesta. En este caso, el número medio de partículas en cada celda es el mostrado en la Tabla 2. Con este número medio de partículas, los tiempos de procesamiento dan como resultado las gráficas de la Figura 5.

7. Conclusiones

En este artículo hemos analizado las principales características de la búsqueda de vecinas basada en la división espacial en celdas. Hemos destacado los condicionantes, más relevantes, que afectan al coste computacional cuando se utiliza la arquitectura CUDA. A partir de estas restricciones, hemos desarrollado un método para mejorar la eficiencia de la búsqueda de partículas vecinas. Hemos llevado a cabo un conjunto de pruebas con las que evidenciar las mejoras que ofrece nuestra técnica. A partir de los resultados obtenidos, podemos concluir que:

- La división ortogonal del espacio, en sentido horario, propuesta, junto con la función hash no normalizada que hemos reformulado, disminuye considerablemente la dispersión de los datos de la memoria y reduce la posibilidad de colisiones de los valores hash obtenidos. En consecuencia, el tiempo de procesamiento disminuye. Esta conclusión puede deducirse a partir de los resultados mostrados en las gráficas de la Figura 4, donde se comprueba el incremento lineal.
- Hemos reducido el tiempo de procesamiento y mejorado la eficiencia al utilizar la posición relativa de las partículas. Esta afirmación viene respaldada por los resultados mostrados en las gráficas de la Figura 5. Comparativamente, la mejora se sitúa en un rango entre un 85% y un 130%.
- El valor óptimo, para nuestro modelo, se obtiene con celdas cuya longitud de arista es $l = 2R$, es decir, para celdas cuyo tamaño es $V = 8R^3$. Esta conclusión se puede deducir a partir los resultados mostrados en las gráficas de Figuras 4 y 5.

8. Agradecimientos

Esta investigación ha sido apoyada por el proyecto Pololas (TIN2016-76953-C3-2-R), mediante el SoftPLM Network (TIN2015-71938-REDT) del Ministerio de Economía y Competitividad de España.

References

[Akl14] AKL S. G.: *Parallel sorting algorithms*. Academic press, 2014. 3

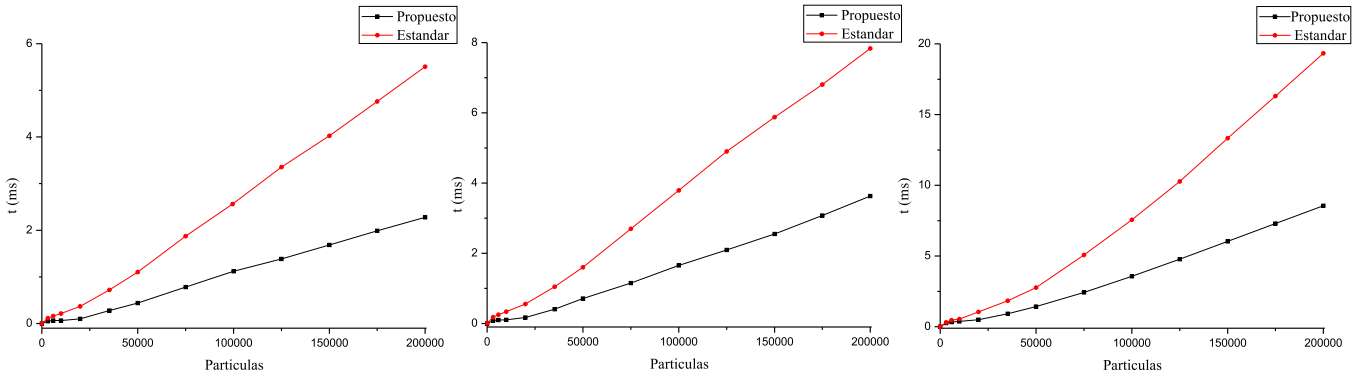
[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Community ACM 18*, 9 (Sep 1975), 509–517. doi:10.1145/361002.361007. 1, 2

[BMJFS01] B. MOON B., JAGADISH H. V., FALOUTSOS C., SALTZ J. H.: Analysis of the clustering properties of the hilbert space–filling curve. *IEEE Transactions on Knowledge and Data Engineering 13* (Jan 2001), 124–141. doi:10.1109/69.908985. 3, 6

[BP11] BURTSCHER M., PINGALI K.: An efficient cuda implementation of the tree–based barnes hut n–body algorithm. In *GPU Computing*

	1000	2000	5000	10000	20000	50000	100000	200000
$\overline{N}(i)$ $[l = 2R]$	12	13	19	24	31	44	53	66
$\overline{N}(i)$ $[l = 2.52R]$	18	25	36	48	61	71	79	90
$\overline{N}(i)$ $[l = 4R]$	31	40	51	59	68	77	89	115

Table 2: *Número medio de partículas vecinas, $\overline{N}(i)$, para diferentes tamaños de celda. Estos son los valores asociados al tiempo de procesamiento total, mostrados en las gráficas de la Figura 5.*



(a) $l = 2R$ **(b)** $l = 2.52R$ **(c)** $l = 4R$

Figura 5: *Gráficas comparativas del tiempo de procesamiento total utilizando la técnica de división espacial estándar y del método propuesto.*

Gems Emerald Edition, Hwu W.-m. W., (Ed.). Morgan Kaufmann, 2011, pp. 75–92. doi:10.1016/B978-0-12-384988-5.00006-1. 2

[Cay12] CAYTON L.: Accelerating nearest neighbor search on manycore systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International* (2012), IEEE, pp. 402–413. 3

[Cud12] CUDA C.: Programming guide. *NVIDIA Corporation, July* (2012). 3, 4

[DCGG13] DOMÍNGUEZ J., CRESPO A., GÓMEZ-GESTEIRA M.: Optimization strategies for cpu and gpu implementations of a smoothed particle hydrodynamics method. *Computer Physics Communications 184*, 3 (2013), 617–627. doi:10.1016/j.cpc.2012.10.015. 3

[FSYA07] FUNG W. W., SHAM I., YUAN G., AAMODT T. M.: Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (2007), IEEE Computer Society, pp. 407–420. 4

[FWZS11] FAN W., WANG B., ZHOU J., SUN J.: Parallel spatial hashing for collision detection of deformable surfaces. In *Computer-Aided Design and Computer Graphics (CAD/Graphics), 2011 12th International Conference on* (2011), IEEE, pp. 288–295. 2, 3, 6

[GDB08] GARCIA V., DEBREUVE E., BARLAUD M.: Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on* (2008), IEEE, pp. 1–6. 1, 2

[GDNB10] GARCIA V., DEBREUVE E., NIELSEN F., BARLAUD M.: K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *ICIP* (2010), pp. 3757–3760. 1

[Gre10] GREEN S.: Particle simulation using cuda. *NVIDIA whitepaper* (2010). 2, 4, 7

[GSSP10] GOSWAMI P., SCHLEGEL P., SOLENTHALER B., PAJAROLA R.: Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2010), Eurographics Association, pp. 55–64. 3

[HB10] HOBEROCK J., BELL N.: Thrust: A parallel template library, 2010. 6

[HSK07] HARADA T., SEIICHI K. Y. K.: Smoothed particle hydrodynamics on gpus. *Computer Graphics International* (May 2007), 63–70. 2

[Hwu11] HWU W.-M. W.: *GPU Computing Gems Jade Edition*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011. 3, 4

[IABT11] IHMSEN M., AKINCI N., BECKER M., TESCHNER M.: A parallel SPH implementation on multi-core cpus. *Comput. Graph. Forum 30*, 1 (2011), 99–112. 1, 2, 3, 4

[Kno09] KNOWLES P.: Gpgpu based particle system simulation. *School of Computer Science and Information Technology RMIT University Melbourne, Australia 12*, 04 (2009), 55–58. 2

[KZN08] KUMAR N., ZHANG L., NAYAR S.: What is a good nearest neighbors algorithm for finding similar patches in images? In *Computer Vision–ECCV 2008*. Springer, 2008, pp. 364–378. 2

[LCT14] LI X., CAI W., TURNER S. J.: Efficient neighbor searching for agent-based simulation on gpu. In *Distributed Simulation and Real Time Applications (DS-RT), 2014 IEEE/ACM 18th International Symposium on* (2014), IEEE, pp. 87–96. 5, 6

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. In *ACM Transactions on Graphics (TOG)* (2006), vol. 25, ACM, pp. 579–588. 2

[LOS10] LEISCHNER N., OSIPOV V., SANDERS P.: Gpu sample sort. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (2010), IEEE, pp. 1–10. 3

[MCPC15] MORILLO D., CARMONA R., PEREA J. J., CORDERO J. M.: A more efficient parallel method for neighbour search using cuda. In *Workshop on Virtual Reality Interaction and Physical Simulation* (2015), Jaillet F., Zara F., Zachmann G., (Eds.). doi:10.2312/vriphys.20151339. 5

[MG10] MERRILL D. G., GRIMSHAW A. S.: Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th international*

conference on Parallel architectures and compilation techniques (2010), ACM, pp. 545–546. 4

[Mic12] MICIKEVICIUS P.: Gpu performance analysis and optimization. In *GPU Technology Conference* (2012). 3

[NVI15] NVIDIA: Cuda c programming guide. docs.nvidia.com/cuda/cuda-c-programming-guide, 2015. 3

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2003), Eurographics Association, pp. 41–50. 2

[PSHL10] PETERS H., SCHULZ-HILDEBRANDT O., LUTTENBERGER N.: Fast in-place sorting with cuda based on bitonic sort. In *Parallel Processing and Applied Mathematics*. Springer, 2010, pp. 403–410. 3

[RBA08] ROŹEN T., BORYCZKO K., ALDA W.: Gpu bucket sort algorithm with applications to nearest-neighbour search. *Journal of WSCG 16* (2008). 2

[RBG*12] RUSTICO E., BILOTTA G., GALLO G., HERAULT A., DEL NEGRO C.: Smoothed particle hydrodynamics simulations on multi-gpu systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on* (2012), IEEE, pp. 384–391. 2

[Ros13] ROSEN P.: A visual approach to investigating shared and global memory behavior of CUDA kernels. *Comput. Graph. Forum 32*, 3 (2013), 161–170. 4

[SA08] SEAL S., ALURU S.: Spatial domain decomposition methods in parallel scientific computing. *Handbook of Parallel Computing-Models, Algorithms and Applications* (2008), 44. 6

[SGS10] STONE J. E., GOHARA D., SHI G.: Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering 12*, 1-3 (2010), 66–73. 3

[SHG09] SATISH N., HARRIS M., GARLAND M.: Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), IEEE, pp. 1–10. 3, 4

[THM*03] TESCHNER M., HEIDELBERGER B., MUELLER M., POMERANETS D., GROSS B.: Optimized spatial hashing for collision detection of deformable objects. In *Proceeding Vision, Modeling, Visualization VMV'03* (Nov 2003), pp. 47–54. 2

[VBC08] VICCIONE G., BOVOLIN V., CARRATELLI E. P.: Defining and optimizing algorithms for neighbouring particle identification in sph fluid simulations. *International Journal for Numerical Methods in Fluids 58*, 6 (2008), 625–638. 3

[WBK07] WRÓBLEWSKI P., BORYCZKO K., KOPEÆ M.: Sph-a comparison of neighbor search methods based on constant number of neighbors and constant cut-off radius. *TASK Quart 11* (2007), 275–285. 2, 3, 4, 5

[XQ16] XILIN X., QIUHUA L.: A gpu–accelerated smoothed particle hydrodynamics (sph) model for the shallow water equations. *Environmental Modelling & Software 75* (Jan 2016), 28–43. doi:10.1016/j.envsoft.2015.10.002. 2, 6