Contenido

[Introducción proyecto 2](#_Toc41505690)

[Motivación Introducción Problema 2](#_Toc41505691)

[Objetivo 2](#_Toc41505692)

[Alcance 2](#_Toc41505693)

[Proyecto 2](#_Toc41505694)

[Elementos 2](#_Toc41505695)

[Funcionalidades 2](#_Toc41505696)

[Archivos TODO 2](#_Toc41505697)

[Benchmark 2](#_Toc41505698)

[Métodos de la simulación y rutinas de GPU 2](#_Toc41505699)

[Código TODO 2](#_Toc41505700)

[Optimizaciones 2](#_Toc41505701)

[Ejecución 2](#_Toc41505702)

[Pocos elementos 3](#_Toc41505703)

[Muchos elementos 3](#_Toc41505704)

[Conclusiones 3](#_Toc41505705)

[Pocos elementos 3](#_Toc41505706)

[Muchos elementos 3](#_Toc41505707)

[Conclusiones generales 3](#_Toc41505708)

[Posibles mejoras 3](#_Toc41505709)

# Introducción

## Motivación Introducción Problema

Se busca acelerar los cálculos mediante GPU, del problema de N-Body para realizar la simulación con el mismo nombre. Un problema aplicable a distintos campos de estudio, como el físico, electrónico, químico …

El problema de los N-Cuerpos se basa en realizar una predicción individual de un conjunto de objetos que se encuentran en un determinado espacio. Para ello se calcula la interacción de un cuerpo con todos los demás, para todos los cuerpos del conjunto, acumulándose una estimación del futuro movimiento aplicándose un “paso de tiempo” que reproduzca tal interacción.

La simulación de los N-Cuerpos es la representación gráfica de las posiciones de los cuerpos a cada “paso de tiempo”.

## Objetivo

Realizar una simulación con “muchos” objetos y ver como interaccionan entre si.

Hay varios algoritmos, el “ingenuo” que recorre de forma cuadrática el array tendrá un coste O(n2). Existen además algoritmos más “avanzados” que realizan cómputos de O(nlogn) o aproximan el cálculo a O(n), sin embargo, el computo principal es altamente paralelizadle, puesto que para cada elemento es obvio que se realizara un bucle independiente a otro elemento.

## Alcance

Realizar la representación de círculos de forma gráfica y realizar los cálculos necesarios para su simulación física en 2D y frente a una fuerza gravitatoria

El problema puede contemplarse desde distintas premisas, en este caso: supondremos un espacio bidimensional de 800x800px para representar gráficamente el problema, solo los objetos que sean visibles en este rango se mantendrán en la simulación (lo que no entren se eliminaran) y los objetos no se atraerán entre sí, tan solo “colisionaran”.

Para la representación gráfica usaremos la forma más sencilla posible, un círculo. De igual forma consideraremos que no hay atracción entre cuerpos, realizando solo el cálculo de las colisiones.

Con respecto a los cálculos, tendremos dos opciones, tendremos la posibilidad de realizar estrictamente los cálculos para intentar obtener la mayor potencia de simulación posible o la posibilidad de mostrar gráficamente la simulación del problema.

En general se aplica una fuerza de gravedad a todos los objetos de la pantalla que no son estáticos.

# Proyecto

## Elementos

Las optimizaciones se han llevado a cabo para estas especificaciones y mediante las herramientas:

* Sistema operativo: Linux-Arch con xorg como servidor de pantalla.
* GPU: GeForce940MX
* Paquetes utilizados:
  + OpenGL, específicamente la librearía freeglut, utilizado para realizar la representación gráfica.
  + CUDA: utilizando las aplicaciones incluidas de nvcc, para compilar CUDA, y NVProof para ver la carga de trabajo presente en la ejecución de los kernels.
* Compilador: g++

## Funcionalidades

Señalar que la funcionalidad es aplicable en el modo gráfico, puesto que el modo de consola se realiza el benchmark sin considerar el pintar los objetos

Como función principal tendremos la posibilidad de crear nuevos objetos, círculos, los cuales pueden ser estáticos o móviles. Se introducen mediante el teclado numérico definiendo el tamaño por ejemplo ‘***5***’, al cual se le aplica un multiplicado (MUL). De igual forma se pueden crear “spawners” ‘***alt+num***’ de círculos: por cada frame crean un círculo nuevo en el punto original del primer círculo creado en el modo “spawner”. Señalar que al salir de la pantalla se eliminan.

Para realizar una simulación que sature la capacidad de cómputo, se pueden poner “paredes” ‘***w***’ simuladas como bordes de la pantalla, las cuales se implementan con círculos muy grandes

Así mismo se puede cambiar el entorno de ejecución, entre CPU y GPU ‘***z***’, demostrando visualmente el cambio en el procesamiento. De igual forma se puede realizar la simulación paso a paso ‘***space-bar***’ para poder ver de forma más detallada como interactúan los cuerpos.

La gravedad se puede desactivar ‘***g***’, dejando solamente la interacción de los choques entre los cuerpos y también, se puede realizar una captura de pantalla ‘***s***’ del frame que se está visualizando.

Por último, se puede reiniciar la ejecución de la simulación ‘***r***’.

## Archivos TODO

‘***circle***’ contiene, hacer estatico un circulo, applyimpulse se ejecuta cuando se simula el movimiento y render que llama a opengl para realizar la visualización

‘***list\_circles***’ clase auxiliar que permite eliminar círculos que no se visualizan

‘***main***’ se encarga de realiza estancias de opengl y scena asi como de la gestión de la entrada de teclado y ratón mediante opengl

‘scene’ encargada de la gestión de los elementos en pantalla y de la realización de todas las llamadas intermedias entre el acceso del usuario por main con los métodos que implementa la funcionalidad. Se encarga de realizar el benchmark sin visualización. Realizar la primera llamada que se encarga de “avanzar” un paso de simulación y ejecuta la renderización de los círculos

‘physics’ step realiza el paso de simulación calculando el siguiente frame y discrimina si se ejecuta en cpu o gpu.

## Benchmark

Se realiza la simulación en un entorno predefinido, donde hay una “malla” de círculos estáticos en medio del espacio de simulación y se rellena la parte superior de círculos según unos parámetros. Se define el tamaño de los círculos y el espacio que hay entre estos. La ejecución consiste en “dejar” caer todos los círculos hasta atravesar la “malla”.

La ejecución del benchmark se puede realizar de dos formas, gráficamente se visualiza el proceso de “caída” y se puede comprobar si la GPU soporta la carga de procesamiento o por consola, que realiza los mismos cálculos, pero ignora el “pintado” de los objetos e intenta obtener el máximo número de frames por segundo.

## Métodos de la simulación y rutinas de GPU

Métodos y kernels importantes en la ejecución del programa.

‘calculateContacs’ este método calcula en un doble bucle anidado las colisiones entre objetos, método que posee un coste O(n2) que, por mucho, tiene la mayor carga computacional de la simulación. Debido a esto, es el método que más margen de mejora nos puede aportar su paralelización. De igual forma es el más tiempo se ha dedicado a optimizar: memoria compartida, optimizaciones empíricas…

‘integrateForcesObject’ aplica la fuerza de la gravedad a los objetos.

‘solveCollisions’ para cada colisión calcula la nueva velocidad que se produce al chocar dos objetos, resuelto en a su vez el método ‘calculateImpulse’, este método ha sido imposible paralelizarse, debido específicamente a las líneas:

* ‘float rvx = B->vx - (B->angularVelocity \* rby) - A->vx + (A->angularVelocity \* ray);’
* ‘float rvy = B->vy + (B->angularVelocity \* rbx) - A->vy - (A->angularVelocity \* rax)’

Cada colisión depende de cada velocidad del objeto, pero la velocidad depende de las colisiones anteriores, no realizar esta dependencia provoca que el impulso generado sobre los objetos los haga “rebotar” excesivamente y un comportamiento errático.

‘integrateVelocitiesObject’ se aplica el cambio de velocidad del choque calculado para cada objeto.

‘contactCorrection’Corrección de posiciones, grabar video explicativo, aplica un cierto margen de error que permite realizar una simulación, en principio, más realista.

## Código TODO

Eliminar círculos que no se ven

Acceso bloque a bloque en memoria compartida en el cálculo de colisiones

Reservar 30 veces el numero de elementos O(n^2) espacio de colisiones

Implementación de arrays dinámicos CUDA

Optimizaciones de transferencia de memoria, no reservarla y liberarla por cada frame, dejarla instanciada

## Optimizaciones

Optimizaciones realizadas de forma empírica.

```

Circle ri;

Collision c;

float ripx, ripy, riinv\_mass, riradius;

float rjpx, rjpy, rjinv\_mass, rjradius;

```

´´´

float inv\_dist = \_\_frcp\_rd(dist);`

…

…

c.penetration = \_\_fsub\_rd(suma\_radius, dist);

´´´

```

//c.penetration = suma\_radius - dist;

c.penetration = \_\_fsub\_rd(suma\_radius, dist);

//c.normal\_x /= dist;

//c.normal\_y /= dist;

c.normal\_x = \_\_fmul\_rd(c.normal\_x, inv\_dist);

c.normal\_y = \_\_fmul\_rd(c.normal\_y, inv\_dist);

//c.contact\_x = c.normal\_x \* ri.radius + ri.px;

//c.contact\_y = c.normal\_y \* ri.radius + ri.py;

```

```

c.contact\_x = \_\_fmaf\_rd(c.normal\_x, riradius, ripx);

c.contact\_y = \_\_fmaf\_rd(c.normal\_y, riradius, ripy);

```

```

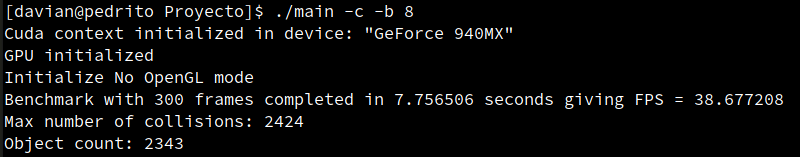
int idx = atomicInc(n\_cols, 1e9); //Faster than atomicAdd(n\_cols, 1)

```

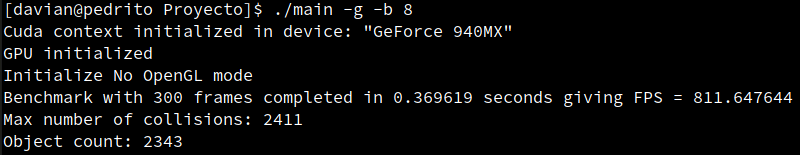
# Ejecución

## Pocos elementos

* CPU

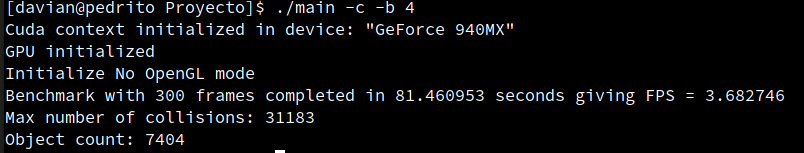


* GPU

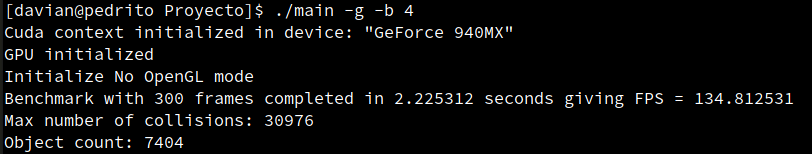


## Muchos elementos

* CPU



* GPU



# Conclusiones de las ejecuciones

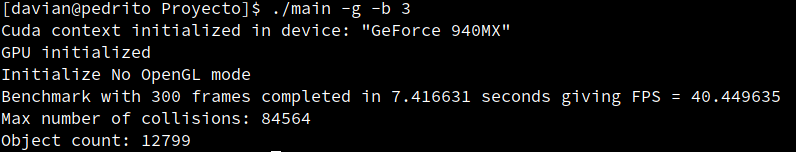
Para las conclusiones se tienen en cuenta los frames por segundo, los cuales incluyen la medida del tiempo y la “capacidad” computacional.

## Pocos elementos

En los pocos casos ya se ve una clara diferencia en el número de frames, donde la ejecución de la CPU aun puede ser visible, considerando aceptable 30fps para visualizar un video, sin embargo, en GPU consigue obtener una cantidad “absurda” de frames, lo cual demuestra la capacidad computacional de la GPU para un caso “pequeño” de 2.000 objetos.

## Muchos elementos

Las conclusiones en la ejecución de 7404 objetos, se muestra la capacidad y lo fundamental de la ejecución en GPU, pues para la ejecución de CPU obtenemos 3.68fps, lo se considera bastante lento pues la ejecución dura más de un minuto, frente a la ejecución por GPU que consigue realizar la ejecución con una tasa de fps muy aceptable, superando los 120fps, y en un tiempo muy ajustado pudiendo llegar a soportar de forma óptima la ejecución de aun mas casos, por ejemplo una ejecución de 12799 objetos mejorando el tiempo de CPU con pocos elementos.



# Posibles mejoras

Implementar diferentes tipos de objetos o incluso poder realizar un objeto general ‘polígono’ que abstraiga la representación de las figuras y permita la representación de cualquier elemento.

Añadir las interacciones gravitatorias del problema N-Body, haciendo que la atracción entre los objetos se aplique.

Realizar los cálculos de forma que las colisiones sean independientes de las anteriores, por ejemplo, implementando otro algoritmo. Permitiendo la ejecución completa del programa en GPU.

Mejorar el hardware añadiendo más unidades de cómputo y añadiendo más memoria.

# Conclusiones

De forma general, la diferencia entre ejecutar en CPU y GPU es contundente. El algoritmo cuadrático de la ejecución en CPU no consigue soportar un aumento en el número de objetos, frente al algoritmo “lineal” de la GPU que consigue afrontar un mayor número de objetos y permitiendo aún más margen aumentando la capacidad computacional del dispositivo.