Trabajo Práctico Final

Organización del Computador II

Grupo : Los Para Lelos

Integrante	LU	Correo electrónico
Höss, Emiliano	664/04	emihoss@gmail.com
Schinca, Herman	527/06	enfractado@yahoo.com.ar
Sebrie, Lisandro	358/03	lsebrie@gmail.com

DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar/

Índice

1. Introducción

Desde que el hombre se constituyó como tal y adquirió el uso de la razón que ha tratado de comprender e imitar a la naturaleza tanto para la construcción de herramientas y máquinas que automatizaran las tareas como para poder describir el mundo físico que lo rodeaba. Con el advenimiento de las computadoras y la explosión tecnológica y científica en el reciente Siglo XX, muchos de los aspectos tanto matemáticos como físicos que se habían estudiado de un modo abstracto pudieron ser interpretados por las máquinas y así obtener nuevos y sorprendentes resultados que, de otro modo, hubieran llevado una gran cantidad de horas hombre para su cálculo.

Surge entonces la posibilidad de interpretar modelos matemáticos y físicos en simulaciones computacionales. No resulta difícil ver cuánto cálculo subyacente hay en este tipo de modelados puesto que la cantidad de variables que suelen afectar a los sistemas suele ser muy grande, sin mencionar las relaciones que se pueden dar entre ellas. En general, cuando a uno le comentan que se ha simulado cierto evento físico, uno descree de la real simulación del mismo puesto que pone en duda su alcance. Es decir, ¿Realmente lo modelaste por completo? ¿Tuviste en cuenta la geometría de las moléculas que componen a los materiales físicos que simulaste? ¿Y las fuerzas intermoleculares? ¿Suponiendo que sí lo hayas considerado, también tuviste en cuenta las fuerzas que se suceden dentro de cada átomo de la molécula? ¿Y los quarks?

Como se ve, la lista de consideraciones puede crecer más y más a medida que querramos ser más y más consistentes respecto de modelar cabalmente cada aspecto y detalle del mundo físico. Como la intuición sugiere, cuantas más variables querramos tener en cuenta para nuestro modelo, más tiempo y poder de cómputo se requerirá puesto que habrá que resolver más ecuaciones para cada instante. En este punto hay que detenerse un momento y pensar: ¿Cómo modelamos el tiempo? En principio, el tiempo físico es algo constante o, al menos, así lo percibimos. También sabemos que las computadoras trabajan de manera discreta, procesan cada cierto intervalo de tiempo cierta cantidad de información. Por lo tanto, pretender modelar el tiempo de manera continua se vuelve una tarea literalmente imposible. Es por esto que nos vemos obligados a tener que discretizarlo. Nos topamos pues con un problema más: ¿Cuál es una buena discretización? ¿Intervalos de 1 segundo? ¿1 milisegundo? ¿1 nanosegundo? Depende. Sí, depende única y exclusivamente de nuestro objetivo y de lo que estemos modelando. Si se tratan de reacciones químicas dentro de un acelerador de partículas pues entonces la discretización de 1 segundo puede que no sea muy buena puesto que dichos eventos inducen cambios de estado interesantes en el sistema en intervalos menores. Y si queremos modelar la evolución de las placas tectónicas en procesos geológicos, ¿será una buena medida de tiempo entonces 1 segundo? Probablemente tampoco ya que, contrariamente, en este caso los cambios que afectan al sistema suceden en intervalos más grandes de tiempo. Como se deduce, la discretización del tiempo ya es un problema en si y está relacionado intrínsicamente con el tipo de sistema que se quiera modelar y de cuánto poder de cómputo estamos dispuestos a utilizar (convengamos que casi sin importar el sistema si usamos como medida los nanosegundos vamos a tener un modelo que implementa bastante bien el transcurso del tiempo pero, en su defecto vamos a necesitar demasiado poder de cálculo).

Como vemos, hay una infinidad de cuestiones a tener en cuenta y que considerar u optar dejar de lado en nuestro modelo en función de nuestros recursos y objetivos. Pero lo mencionado anteriormente no es lo único que influye a la hora de realizar una simulación computacional. Un hecho que no tuvimos en cuenta hasta ahora es el modo en el cual se iba a presentar, es decir, si en tiempo real o luego de haber hecho todos los cálculos. Este factor es determinante. Muchas simulaciones se realizan en grandes clusters durante meses y se almacenan todos los datos de dichas "corridas" para luego interpretarlos y representarlos graficamente en un video. En las simulaciones en tiempo real, en cambio, a medida que el sistema es puesto en acción, todos los cálculos se van realizando en tiempo real y la representación gráfica se obtiene "on the fly". Nótese la diferencia entre ambos esquemas. En el primero podemos estar computando durante meses para obtener tan sólo un segundo. En el otro esto no es posible ya que lo que se busca es que el tiempo de la simulación suceda conjuntamente con el tiempo físico y, por ende, los cálculos tienen deadlines muy precisas ya que si las exceden se puede distender el tiempo simulado con el real y dar la sensación de estar "andando lento". Esta diferencia entre ambos modelos es fundamental a la hora de dedicir la discretización del tiempo y las variables que tendremos en consideración que fueron los problemas que estuvimos analizando primeramente. En tiempo real, debemos obligadamente acotar el alcance de nuestro sistema o, en su defecto, conseguir la capacidad de cómputo necesaria para poder visualizar debidamente nuestro modelo.

2. Objetivos

Habiendo introducido algunos de los conceptos y problemáticas relativas a las simulaciones, trataremos ahora de hacer un pequeño esbozo de cuáles son los objetivos y cualidades del proyecto que comenzamos hace ya casi 1 año. En principio, cómo es de suponer, nuestro trabajo se trata de la simulación de ciertas propiedades físicas y la evolución de un sistema con dichas características.

Desde sus orígenes hasta el momento, el proyecto ha ido pasando por diversas etapas y sufriendo muchos cambios, pero ha mantenido siempre un mismo objetivo central el cual consiste en modelar en tiempo real un sistema físico en el cual hubiera una cantidad importante de esferas interaccionando todas contra todas. Si bien el producto final no consiste sólo en esto, la más complicado y lo que más tiempo nos llevó fue ir puliendo cada uno de los aspectos y detalles de la física que queríamos retratar. En las secciones que siguen, profundizaremos más respecto de cuál es nuestro modelo matemático, qué decidimos tener en cuenta y qué descartar, y cuáles fueron los obstáculos con los cuales nos fuimos topando a medida que el proyecto iba tomando forma.

Vale aclarar que, actualmente, nuestra pieza de software se asemeja más a un FPS (First Person Shooter) que a lo descrito en el párrafo anterior. Sin embargo, como mencionamos, lo realmente desafiante del trabajo fue lograr una física sólida y que no atentara contra lo que uno está acostumbrado a percibir en el mundo real. Y he aquí también la posibilidad de, independientemente de si se trata de un FPS, una simulación de la evolución de un sistema o un Pong 3D, poder reutilizar la implementación de nuestro modelo físico en innumerables nuevos contextos.

3. Modelo Físico

Sin ahondar mucho en la realidad del asunto, a priori, uno podría pensar que modelar cómo se mueve una pelotita y cómo interaccionan un conjunto de ellas, chocando entre si y contra algunas paredes no pareciera tarea difícil ni que involucre mucha teoría. Contrariamente a esta un tanto errado pensamiento (como dirían los Les Luthiers: “Estás razonando fuera del recipiente”), este engañoso simple sistema involucra muchas leyes físicas. Si nos ponemos a profundizar más en el asunto, podemos ver que no sólo se trata de determinar la posición y velocidad de las pelotitas en cada instante (supongamos por un momento que el problema del tiempo ya lo ttenemos resuelto) sino también la fuerza de choque, la nueva trayectoria luego del mismo, el tamaño de las esferas, el material, los coeficientes de rozamiento de cada material, la gravedad, la viscosidad del ambiente, el punto de impacto, el torque, etc. Como se ve, hay muchas variables en juego y, dado que nuestro trabajo se desarrolla en tiempo real, algunas de ellas tuvieron que ser descartadas para que los cálculos no fueran “tan pesados”. En los párrafos siguientes serán explicitadas todas las características físicas de nuestro modelo.

Dado que uno de los mayores desafío que tuvimos fue modelar adecuadamente las colisiones entre las esferas procedamos primero a describir detalladamente este evento. Empezemos suponiendo primero que tenemos 2 esferas de igual masa que están colisionando. La Figura 1 detalla este suceso. Para hacer más simple el sistema no consideramos rozamiento por lo que su efecto no estará contemplado en las ecuaciones ni tampoco la rotación de las esferas.

v	velocidad de la esfera
m	Masa de la esfera
ϵ	Coeficiente de restitución
n	Línea de colisión
t	Línea tangente a la colisión
j	Impulso
N	Fuerza normal a la superficie de colisión
g	Fuerza de gravedad
R	Radio de las esferas

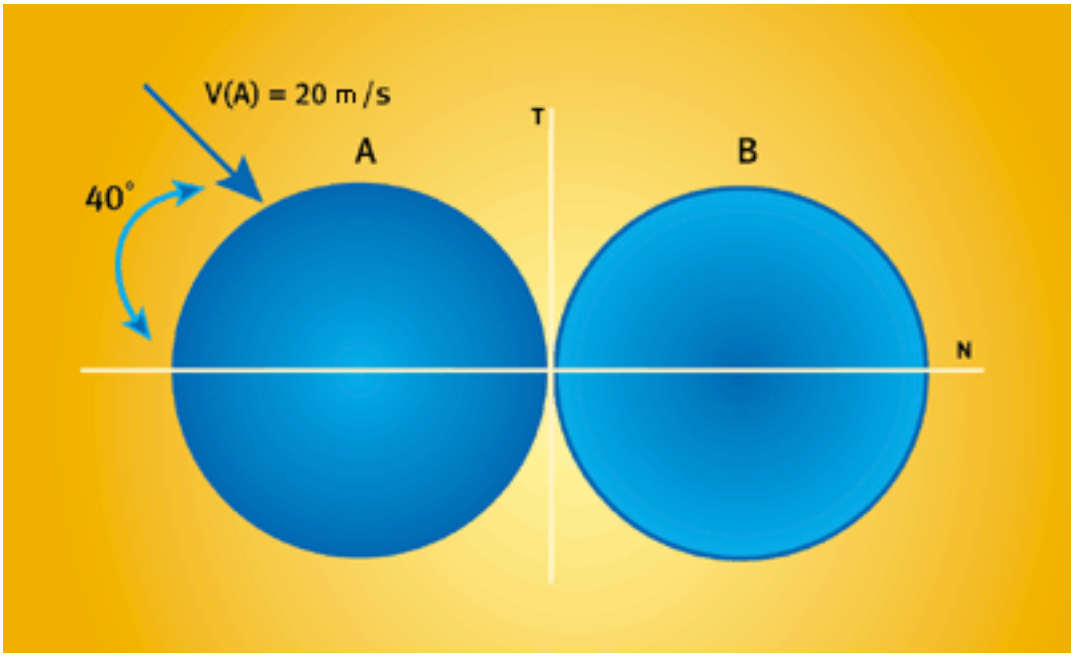

Figura 1: Dos bolas de igual masa colisionando.

La bola A tiene una velocidad de 20 m/s y colisiona, formando un ángulo de 40°, con la bola B que tiene velocidad nula. Para determinar cuál será la velocidad (cada vez que hablemos de velocidad nos estaremos refiriendo al vector velocidad y no tan sólo a su módulo, es decir que también hay que tener presente a la dirección y el sentido) que tendrán ambos cuerpos luego de la colisión aplicar las leyes de la dinámica. Antes de ello, familiarizemonos con el vocabulario y la notación que vamos a empezar a utilizar de ahora en más. Cuando dos cuerpos rígidos impactan la fuerza que se genera en dicho evento se denomina *fuerza impulsiva* o simplemente *impulso*, la cual denotaremos con *j*. La normal a la superficie de las 2 bolas en contacto se la conoce como *línea de colisión* y es representada por la letra *N*.

Primero procedamos a descomponer la velocidad inicial de la bola A en sus componentes, tanto respecto a la línea de colisión como a la tangente a dicha línea. Nos queda:

$$v_A = 20m/s$$

$$v_A \cdot n = v_A \cdot \cos(40) = 15{,}32m/s$$

$$v_A \cdot t = v_A \cdot \text{sen}(40) = -12{,}86m/s$$

Luego de la colisión, la velocidad de ambas bolas en la componente tangencial no se verá afectada puesto que el impulso sólo actúa en la línea de la colisión. Por ende, dichas componentes nos quedarán como sigue (las velocidades luego del choque las notaremos con un apóstrofe):

$$v'_A \cdot t = -12{,}86m/s$$

$$v'_B \cdot t = 0m/s$$

Nos queda ahora determinar cómo quedará la otra componente. Recordemos la Tercera Ley de Newton (Acción-Reacción) que nos dice que la fuerza actuante entre dos partículas en en choque tiene igual magnitud en ambos cuerpos pero dirección opuesta. Por lo tanto, el momento lineal del sistema se conserva antes y después de la colisión. La definición de momento lineal para un cuerpo es el producto entre su masa y el módulo de su velocidad. Tenemos entonces la siguiente ecuación de conservación del momento:

$$m_A \cdot (v_A \times n) + m_B \cdot (v_B \times n) = m_A \cdot (v'_A \times n) + m_B \cdot (v'_B \times n) \tag{1}$$

Reemplazando con nuestros valores y recordando que las masas eran iguales obtenemos:

$$m \cdot (15{,}32) + m \cdot (0) = m \cdot (v'_A \times n) + m \cdot (v'_B \times n)$$

$$v'_A \times n + v'_B \times n = 15{,}32 \tag{2}$$

Como se puede apreciar, tenemos una única ecuación con 2 variales. Necesitamos pues un poco más de información para poder resolverla. Introduzcamos entonces el concepto de "Coeficiente de Restitución". El mismo se utiliza como una medida de la conservación de la energía cinética en el choque entre 2 partículas y se obtiene mediante la siguiente ecuación:

$$v'_B \times n - v'_A \times n = \epsilon \cdot (v_A \times n - v_B \times n) \tag{3}$$

Este valor depende preponderantemente del material del que están compuestas las partículas y cuando vale 1 entonces estamos en presencia de un choque perfectamente elástico, es decir, la energía cinética del sistema se conserva, y cuando vale 0 se dice que se trata de un choque perfectamente inelástico, perdiéndose parte de la energía cinética, transformándose ésta en sonido y deformaciones de los cuerpos. Supongamos, para continuar con nuestro ejemplo, que ϵ=0,8. Nos quedaría entonces lo siguiente y podríamos junto con la Ecuación 1 resolver el sistema:

$$v'_B \times n - v'_A \times n = 0{,}8 \cdot (15{,}32 - 0)$$

$$v'_B \times n - v'_A \times n = 12{,}26 \tag{4}$$

Resolviendo el sistema de ecuaciones nos quedan las siguientes velocidades:

$$v'_A = (1{,}53, -12{,}86) m/s$$

$$v'_B = (13{,}79, 0) m/s$$

En nuestro modelo, más allá de este ejemplo particular, utilizamos un coeficiente de restitución igual a 1. A pesar de todas las cuentas realizadas hasta ahora aún no hemos hallado el impulso. Éste genera un cambio en el momento de cada cuerpo (no en el sistema, como mencionamos con antelación, puesto que dicho momento sí se conserva) siguiendo las siguientes relaciones:

$$m_A \cdot v_A + j \cdot n = m_A \cdot v'_A \tag{5}$$

$$v'_A = v_A + \frac{j}{m_A} \cdot n$$

$$m_B \cdot v_B - j \cdot n = m_B \cdot v'_B \tag{6}$$

$$v'_B = v_B - \frac{j}{m_B} \cdot n$$

Puede verse que sumando las 5 y 6 obtenemos la ecuación 1, que mostraba que el momento del sistema se conserva. Combinando estas fórmulas con la ecuación 3 podemos determinar j sabiendo el coeficiente de restitución:

$$j = \frac{-(1+\epsilon) \cdot v_{AB} \times n}{n \times n \cdot (\frac{1}{m_A} + \frac{1}{m_B})}$$

Cabe aclarar que nuestro modelo es en 3D aunque el ejemplo haya sido desarrollado en 2D. Las cuentas son análogas pero teniendo en cuenta ahora una coordenada más y utilizando la normal en vez del módulo. Tampoco se detalló cómo sería el choque entre una bola y una pared puesto que la idea es la más intuitiva: imaginar a la pared como un plano y reflejar el vector velocidad de la bola respecto de la pared.

Respecto de la viscosidad y la gravedad, el primero es un valor constante que en cada frame decrementa mínimamente la energía cinética de las bolas mientras que para el segundo tomamos el valor usual de 9,81 m/s^2 y análogamente, en cada frame aplicamos dicha fuerza a cada una de las bolas.

4. Implementación

En las secciones anteriores ya hemos estado describiendo cuáles eran nuestros objetivos principales y cuál era el modelo físico en el cual nos íbamos a basar. Como todo trabajo, primero se parte de ideas o deseos abstractos (como dirían los ingenieros del Software "Objetivos Blandos" :-P) para luego, continuando el léxico ingenieril, ir refinando más y más el Mundo de las Ideas (citando a Platón :-D). Pues bien, ya que tenemos claro qué es lo que queremos modelar necesitamos ahora saber cómo. En principio, dado que nuestro gran objetivo era hacer una simulación en tiempo real, necesariamente su objetivo hermano es realizar una implementación eficiente. Y aquí es donde aparece otra de las grandes motivaciones de este trabajo: optimizar la implementación utilizando las funciones de procesamiento vectorial de SSE. A priori, pareciera ser un planteo razonable puesto que vamos a tener muchas esferas a las cuales vamos a tener que aplicarles por igual la misma física por lo que trabajar en paralelo reduciría el tiempo de procesamiento.

Pero no todo en la vida sucede a tan bajo nivel como en Assembler. No resulta difícil intuir que renderizar gráficos 3D no es la tarea más amena para hacerla de lleno en ASM. Fue por ello que volvimos al mundo de la abstracción y utilizamos una API para este tipo de desarrollos. Dicha API es DirectX 10 la cual permite trabajar con código C++, lenguaje que utilizamos para la implementación.

Uno de los datos mencionados a tener en cuenta anteriormente era el de la discretización del tiempo. Basándonos en el hecho de que el ojo humano no puede diferenciar más de unos cuántos cuadros por segundo, teniendo en cuenta estudios realizados, alrededor de 30-40, nosotros optamos por un valor de 60 dado que con esto será más que suficiente para nuestro objetivo.

Lo que detallaremos será lo que se realiza en cada uno de los frames. Cabe aclarar que casi todas las estructuras de datos que utilizamos tienen la particularidad de ser vectores o punteros. Esta estructuración nos permitirá luego la optimización en SSE (profundizaremos este aspecto más adelante).

Uno de los grandes desafíos fue el de cómo actuar ante las colisiones entre las bolas. El resto del modelo, como ya mencionamos, resulta de una sencilla implementación dado que sólo se trata de aplicar la fuerza de gravedad y la viscosidad. Por lo tanto, nos centraremos básicamente en la detección y manejo de las colisiones. Dadas 2 bolas pueden ocurrir los siguientes escenarios:

1. Las bolas no están colisionando.
2. Las bolas están colisionando.
3. Luego de obtener las nuevas posiciones de las bolas, éstas colisionan.
4. Luego de obtener las nuevas posiciones de las bolas, éstas no colisionan.

El primer y tercer caso son los que menos importan puesto que ahí no hay nada por hacer. Los escenarios complejos son el segundo y el cuarto. Nótese que, a pesar de que parezcan idénticos, no lo son. Claramente en ambos hay que solucionar el problema de la colisión pero el cuarto escenario acarrea un problema más que el segundo. Supongamos que el problema de la colisión ya lo tenemos resuelto. En el segundo caso ya estaríamos, sin embargo, en el cuarto caso no sólo tenemos que pensar en la colisión sino también en el solapamiento puesto que al desplazar las bolas estas pueden haber quedado "metidas" dentro de otras. Habría que separarlas, y al hacer esto corremos el riesgo de que por arreglar dos bolas solapadas y desaparejarlas, estemos generando nuevos solapamientos y colisiones con otras bolas que ya habían sido analizadas. Esto nos podría llevar a un livelock del cual nos sería difícil salir. Para ello tenemos que tomar ciertas precauciones y medidas Ad Hoc para que esto no ocurra. De todos modos, nótese que siempre puede haber un caso tan malo como se quiera: si llenamos el espacio con más bolas de las que éste puede contener, entonces siempre ocurrirá que haya solapamientos y, por ende, livelock. Sin embargo, muy astutamente, la cantidad de bolas está restringida para que esto no ocurra y, en el caso de ocurrir, se necesitaría tanta capacidad de cómputo para que la simulación ande que perdería sentido ya que nuesto espacio consta de un cubo de 200x200x200 y el radio de nuestras bolas es suficientemente pequeño en comparación, es decir, se necesitarían tantas bolas que la simulación se relentizaría demasiado.

Antes de seguir con este problema que estuvimos analizando solucionemos los problemas más sencilos para así luego abocarnos de lleno al conflicto planteado en el cuarto item. Al principio de cada frame, tenemos la certeza de que todas nuestras bolas se encuentran en un estado correcto puesto que en el frame anterior tuvimos la precaución de separar todas las bolas y aplicar correctamente las leyes de nuestro modelo. Por lo tanto, lo primero que realizamos es el desplazamiento de las bolas. Esta tarea resulta sencilla puesto que solo debemos modificar la posición de cada bola teniendo en cuenta cuál es su velocidad. Notar que el efecto de la gravedad y viscosidad ya es realizado por otra función por lo que no debemos tenerla en cuenta al desplazar las bolas dadas sus velocidades. También es interesante ver que dada nuestra decisión de, a priori, saber que al inicio de cada frame no hay bolas colisionando, el segundo item de nuestra enumeración queda entonces descartado. Luego de obtener su nueva posición debemos chequear si hay bolas que se encuentran colisionando. Para ello lo que hacemos básicamente es determinar si la distancia entre los radios de ambas es menor o igual que la suma de dichos radios. En dicho caso, las bolas están colisionando. Para calcular la distancia utilizamos la definición de norma:

$$Distancia = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2 + (A_z - B_z)^2}$$

De encontrarse colisionando, habría que desaparejarlas ya que, de lo contrario, al correr la simulación veríamos que las bolas se meten unas dentro de otras y luego cambian su dirección ya que le aplicamos el choque. De hecho, en alguna primera versión, cuando aún no teníamos implementada la función que separaba las bolas, sucedía exactamente esto: uno podía ver cómo las bolas se traspasaban unas a otras y de pronto cambiaban su rumbo inesperadamente. Como ya hemos demostrado anteriormente, generalmente lo que parece sencillo de simular del mundo físico, en general, suele tener alguna sutileza que lo vuelve completamente complejo e intrincado. Esta aparente simple cuestión no es la excepción. Desaparejarlas, despegarlas, separarlas... Pero, ¿Cómo? En el mundo real todo sucedería al mismo tiempo. ¿Nosotros deberíamos procesar cada bola en paralelo entonces? Pero, la idea era trabajar con un solo procesador y optimizar los cálculos usando Assembler, es decir, que, a lo sumo, sólo podemos trabajar con un subconjunto del total de bolas por vez. Hasta aquí, no surge ningún problema. Notemos, sin embargo, que según este análisis, el orden en el que se irían desacoplando las bolas sería secuencial, primero algunas, luego otras y así siguiendo. Pero ¿Cómo nos aseguramos de no estar volviendo a generar colisiones con bolas que ya habíamos chequeado al realizar los sucesivos reacomodamientos? Desde los comienzos hasta ahora hemos pensado cantidad de algoritmos para poder solucionar este escollo, la mayoría de ellos muy completos pero por ende muy costosos respecto del tiempo de procesamiento. Decidimos finalmente establecer un orden para las bolas y para cada una de ellas, analizar secuencialmente si colisionaba con otras bolas o no y luego separarla siguiendo la misma dirección que su velocidad pero en sentido contrario. De esta manera, reacomodamos todas las bolas pudiendo generar, quizás, nuevas colisiones. Sin embargo, esto lo solucionamos moviendo cada bola que sigue colisionando con gravedad negativa, es decir, hacia arriba. Si bien pareciera un recurso "tirado de los pelos" tiene bastante sentido pensar que los cuerpos se comportan así ya que si un objeto lo presiono en todas direcciones y no lo dejo seguir ocupando el espacio en el que se encuentra, entonces va a tratar de escapar hacia donde haya algún espacio, es decir, hacia arriba. Puede realizarse el experimento con cualquier pelota de tenis que se halle en el suelo y se podrá ver que si la vamos presionando de todos lados, la pelotita finalmente tenderá a subir y apoyarse en los objetos que la empujaban. Esto es básicamente lo que quisimos retratar con este recurso que además es liviano de procesar.

5. Optimizaciones

Para analizar las optimizaciones que se llevaron a cabo en el TP, lo haremos en torno a la función sobre la cual se realizaron, siguiendo una a una el avance o retroceso en cuanto a la mejora de performance de la misma que fuimos obteniendo a lo largo de nuestro trabajo.

5.1. ApplyCollision

Pseudocódigo de la función:

```
Para cada bola i:

Si i es visible:

Para cada bola j > i :

Si j es visible:

velocidadIniI = obtenerVelocidad(i)
velocidadIniJ = obtenerVelocidad(j)

masaI = masaEsfera(i)
masaj = masaEsfera(j)

normal = obtenerPosicion(i) - obtenerPosicion(j)

//normalizar es dividir un vector por su norma
normalizar(normal);

//dotProduct es el producto interno entre 2 vectores, devuelve un escalar
impulso = (2.0f * (dotProduct(velocidadIniI, normal) - dotProduct(velocidadIniJ, normal))) / masaI +masaj);

choque = dotProduct(normal, velocidadIniJ) - dotProduct(normal, velocidadIniI);

Si choque > 0.0f :

setearVelocidad(i,x,velocidadIniI.x - impulso * masaJ * normal.x)
setearVelocidad(i,y,velocidadIniI.y - impulso * masaJ * normal.y)
setearVelocidad(i,z,velocidadIniI.z - impulso * masaJ * normal.z)

setearVelocidad(j,x,velocidadIniJ.x + impulso * masaI * normal.x)
setearVelocidad(j,y,velocidadIniJ.y + impulso * masaI * normal.y)
setearVelocidad(j,z,velocidadIniJ.z + impulso * masaI * normal.z)

return Verdadero
```

Esta función es la más complicada de todas, ya que la misma involucra más de la mitad del tiempo que tarda cada frame del juego en renderizarse, es la función que chequea que bolas colisionaron, y tal como se explicó anteriormente aplica la física de colisiones a cada choque.

Para esta función hay que destacar que la paralelización es un tanto difícil, ya que las velocidades de una bola dependen de las velocidades anteriores, en efecto, si una bola colisiona al mismo tiempo contra otras 5 bolas, no puedo calcular la colisión de las cinco al mismo tiempo ya que la velocidad de la segunda depende de la primera, la tercera de la segunda y asi sucesivamente.

Sin embargo, y como se verá más adelante, se logró optimizar muy bien esta función aprovechando otras ventajas de la arquitectura de SSE.

Para comenzar, lo que se hizo sobre esta función fue pasar el código del ciclo solamente a SSE, considerando que ya estaba chequeado si la bola i estaba colisionando contra la j, y descartando las que no fueran así.

Esto significa que el codigo de SSE estaría corriendo para algunos casos, y no se estaría ejecutando para otros muchos que son de hecho la mayoría.

La primera implementación que realizamos resultó satisfactoria notando una mejora que se refleja claramente en el siguiente gráfico:

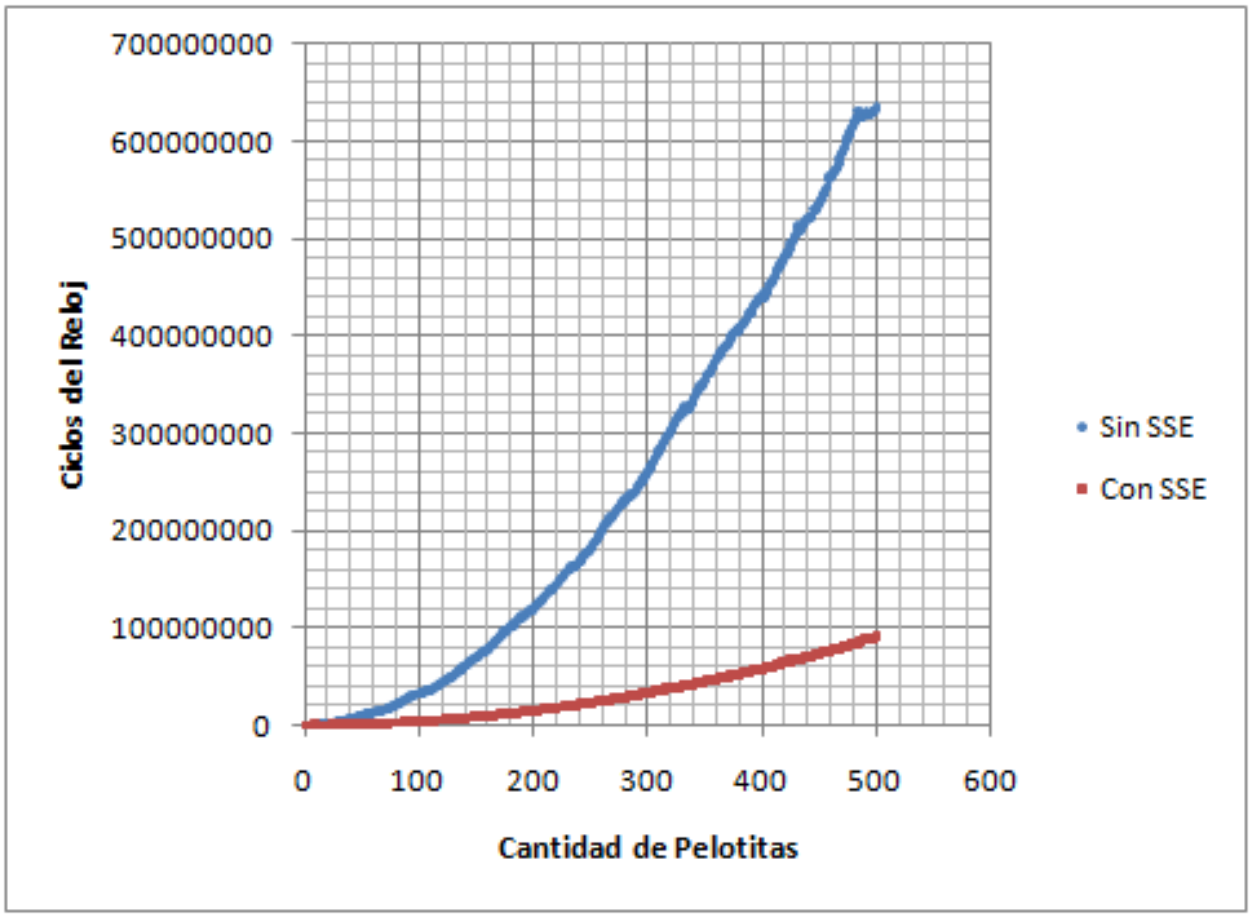

Figura 2: Applycollision con y sin SSE

Sin embargo, el código de SSE aún quedaba muy simple, y eran varias las pruebas que podiamos hacer para ver si llegábamos a optimizarlo aún más.

Lo primero que pensamos es qué pasaría si eliminaramos el salto condicional que hace que el código de SSE no se ejecute todo el tiempo. A simple vista uno pensaría que si ese salto, en el hipotético caso de 500 bolas colisionando, descarta que el código de SSE se ejecute 490 veces, por el hecho de que solo 10 pelotas aproximadamente podrian llegar a estar en los límites de otra, entonces parecería bueno dejarlo. Pero el codigo de SSE propiamente dicho es bastante extenso y, como tal, el salto condicional podría resultar demasiado abrupto al punto de desacomodar la cache y, no sólo generar un acceso a memoria por el salto en si, sino también perjudicar la tasa de aciertos en memoria.

No fue fácil agregar el salto condicional al código, ya que la cantidad de registros utilizada estaba al límite, y debimos reorganizar lo que previamente habíamos programado, para poder aprovechar el uso de un registro más, que era lo que necesitábamos para obtener la comparación por la cual antes se lanzaba un salto.

Cabe destacar tambien que C++ para realizar la comparación del salto condicional está usando la FPU, lo cual agrega un overhead innecesario muy importante al ciclo.

Finalmente, hicimos la prueba para ver cómo mejoraría la performance de la función eliminando este salto condicional y, ya que estábamos, también eliminando los ciclos de C++, dejando todo implementado en assembler.

Como puede verse en el gráfico a continuación, no sólo mejoró, sino que fue muy significativa la mejora, lo que hace pensar que un salto condicional puede ser mucho peor que decenas de instrucciones de SSE.

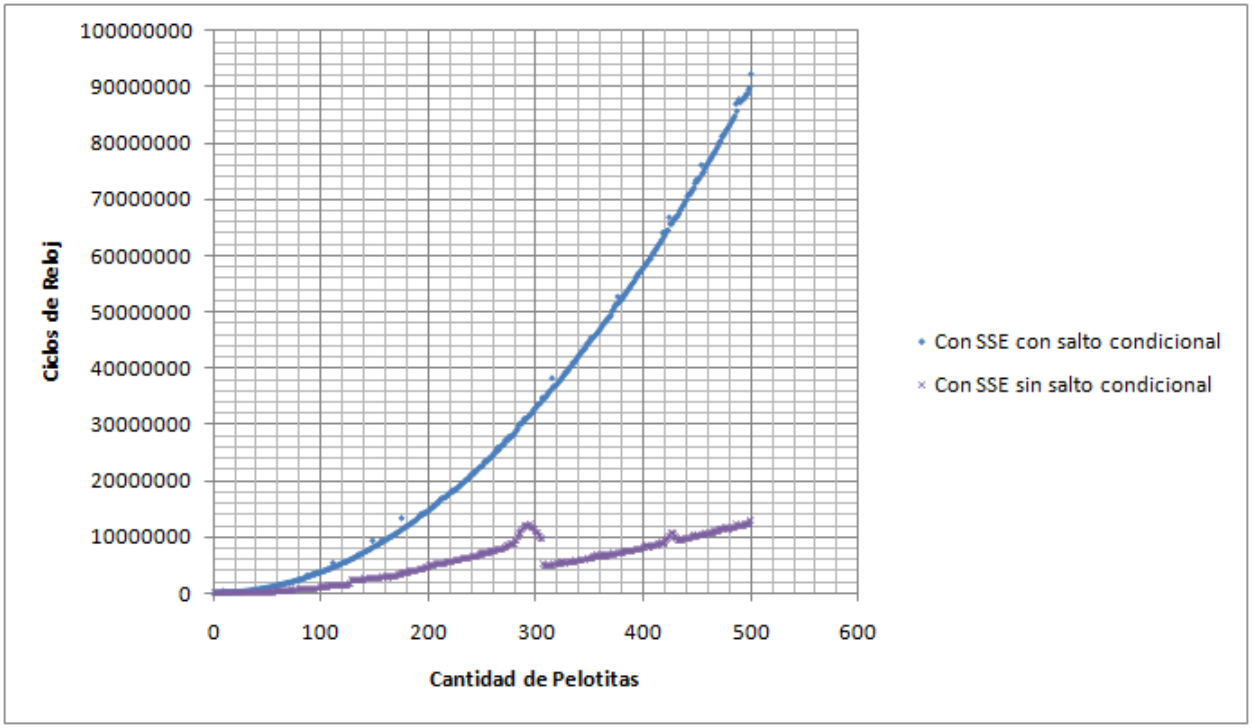

Figura 3: Applycollision con y sin salto condicional en SSE

Siguiendo con las pruebas de más ideas sobre qué optimizar, lo que buscamos ahora es aprovecharnos de la instrucción MOVAPS de SSE, la cual es más eficiente que MOVUPS, pero requiere que la memoria esté alineada.

Para esto cambiamos la forma que teníamos para crear nuestra estructura, reemplazando el operador *new* de C++ por el *mallocaligned* que, dada una cantidad de memoria y un cantidad de bytes, pide memoria alineada a esa cantidad.

A su vez, alineamos tambien a 16 bytes todas las estructuras para que, al acceder de SSE, pudiéramos movernos de a 16 posiciones de memoria por vez para no perder la alineación.

Finalmente, haciendo los reemplazos propiamente mencionados, obtuvimos nuevamente resultados positivos, como se ve en el siguiente gráfico, la cantidad de ciclos disminuyó notablemente.

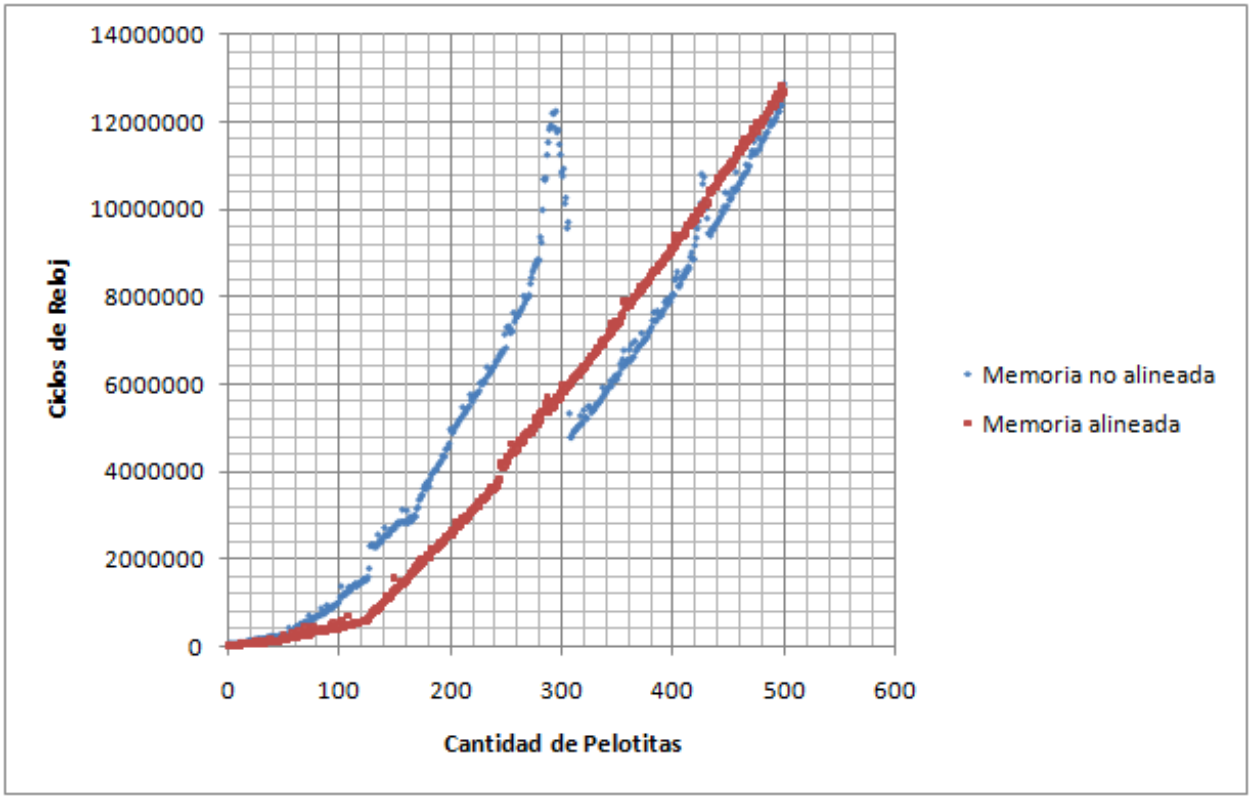

Figura 4: Applycollision con y sin memoria alineada

Ya terminando con las opciones para optimizar, comenzamos a analizar qué instrucciones eran las que más influían sobre la performance del ciclo, y llegamos a la conclusión de que una de las intrucciones que mas ciclo de CPU consumia era PSHUFD y además se repetía 5 veces en el ciclo.

Esta instrucción la utilizamos para replicar el valor del float de la posición menos significativa de un registro de SSE, a sus otras tres posiciones más significativas.

Buscamos una manera de realizar esto de otra manera, y llegamos a la conclusión de que la única alternativa viable consistía en utilizar dos instrucciones relativamente nuevas de SSE, las cuales son:

MOVSLDUP–Move Packed Single-FP Low and Duplicate

MOVDDUP

Sin embargo, al reemplazar las llamadas a las instrucciones de PSHUFD, por estas otras dos llamadas a estas instrucciones, notamos que el código no sólo no mejoró en cuanto a su performance, sino que además empeoró levemente, por lo que decidimos descartar este cambio.

En el siguiente gráfico se ve cómo es que empeoró levemente la cantidad de ciclos por función.

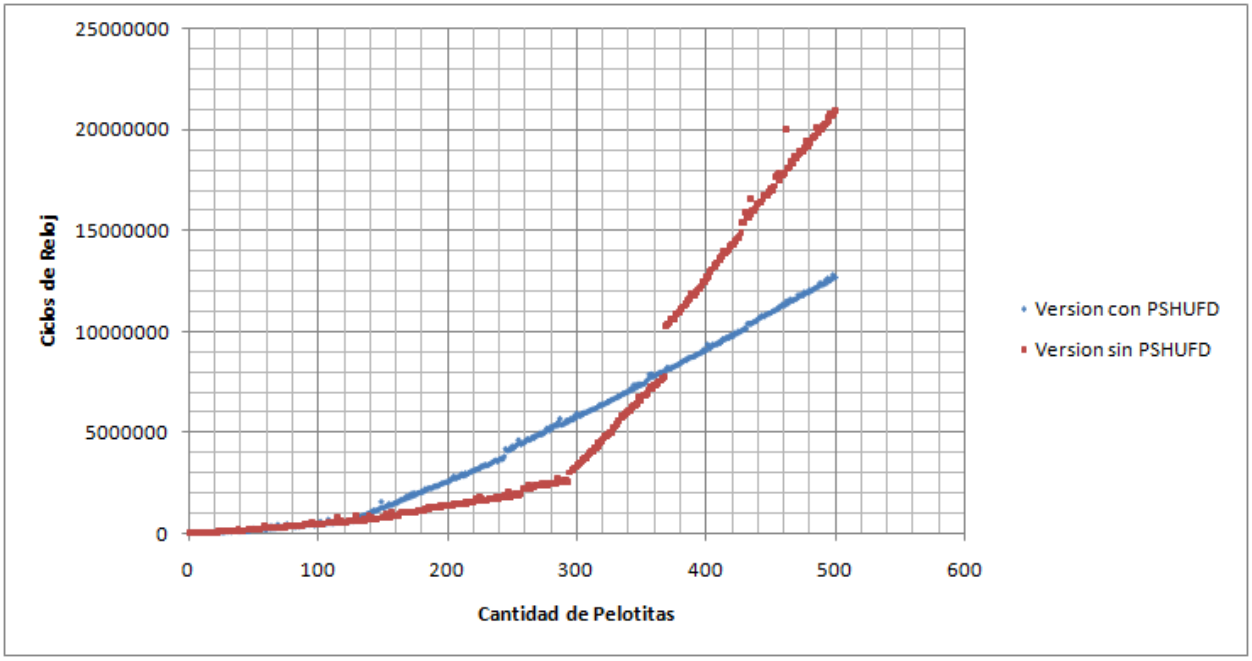

Figura 5: Con y sin PSHUFD

Por último, decidimos ver si era posible mejorar aún más nuestra función, aprovechándonos de que podemos manipular la cache de la computadora con ciertas instrucciones de prefetch de SSE.

Agregamos dichas instrucciones inmediatamente después de cargar las bolas *i-ésimas*, pero justo antes de cargar las *j-ésimas*, que serían contra las cuales verificaría si la bola *i-ésima* colisiona, con el propósito de tenerlas ya cargadas en cache.

Nuevamente logramos resultados positivos, ya que este agregado al código de la función redujo notablemente la cantidad de ciclos de CPU por cada llamado a la función. A continuación, un gráfico que lo refleja:

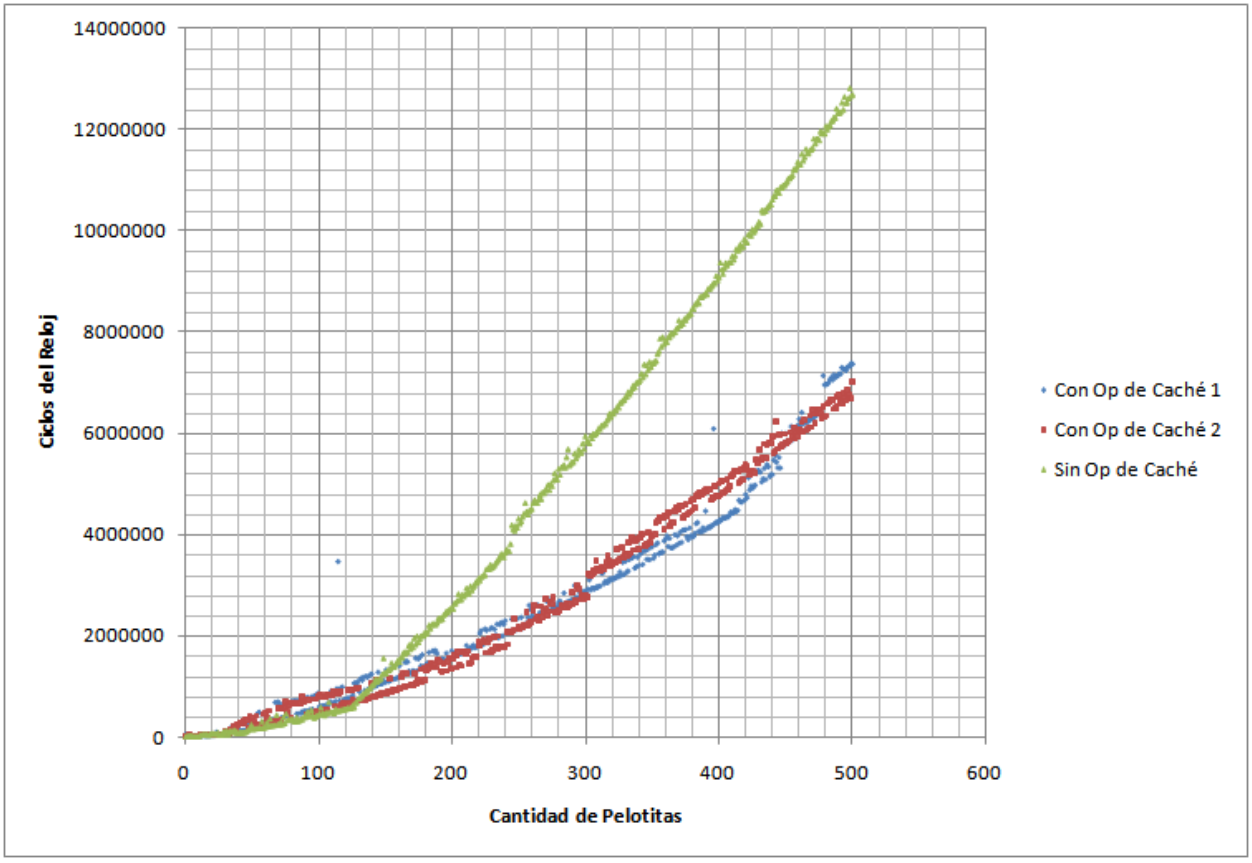

Figura 6: Con y sin optimización de cache

En conclusión, para esta función podemos decir que logramos reducir una tasa de 600 millones de ciclos de CPU por cada llamado a la función, a tan sólo 6 millones de ciclos, lo que representa una reduccion de 100 veces el tiempo que se obtenía con la implementacion de C++.

Además, notamos cómo un salto condicional puede perjudicar drásticamente el tiempo de ejecución de un programa, reflejando en esto el por qué aun hoy en día se siguen buscando soluciones para predecir y mejorar los saltos condicionales.

También observamos cómo el uso correcto de la cache puede favorecernos ampliamente, tanto en los accesos alineados a memoria como en la carga de datos de manera sistemática y ordenada en la misma.

5.2. MoveBalls

Para esta función las optimizaciones no fueron tantas, pero debido a la simpleza del ciclo, se logró realizar una optimización del pipeline llamada Loop Unrolling, que no habia sido posible en otras funciones debido a que la complejidad de las mismas requería del uso de varios registros de SSE que son necesarios para realizar esta optimización.

Para comenzar analizamos cómo se comporta el primer código de SSE que programamos, en comparación con el escrito en C++, y vemos cómo ya en un comienzo la optimización en SSE llega a ser hasta 5 veces mejor que el codigo de C++ en términos de performance, aún con un código assembly bastante elemental, lo que nos da la idea de lo mal que debe estar trabajando el compilador con un ciclo tan sencillo como este e incluso a pesar de realizar optimizaciones con SSE.

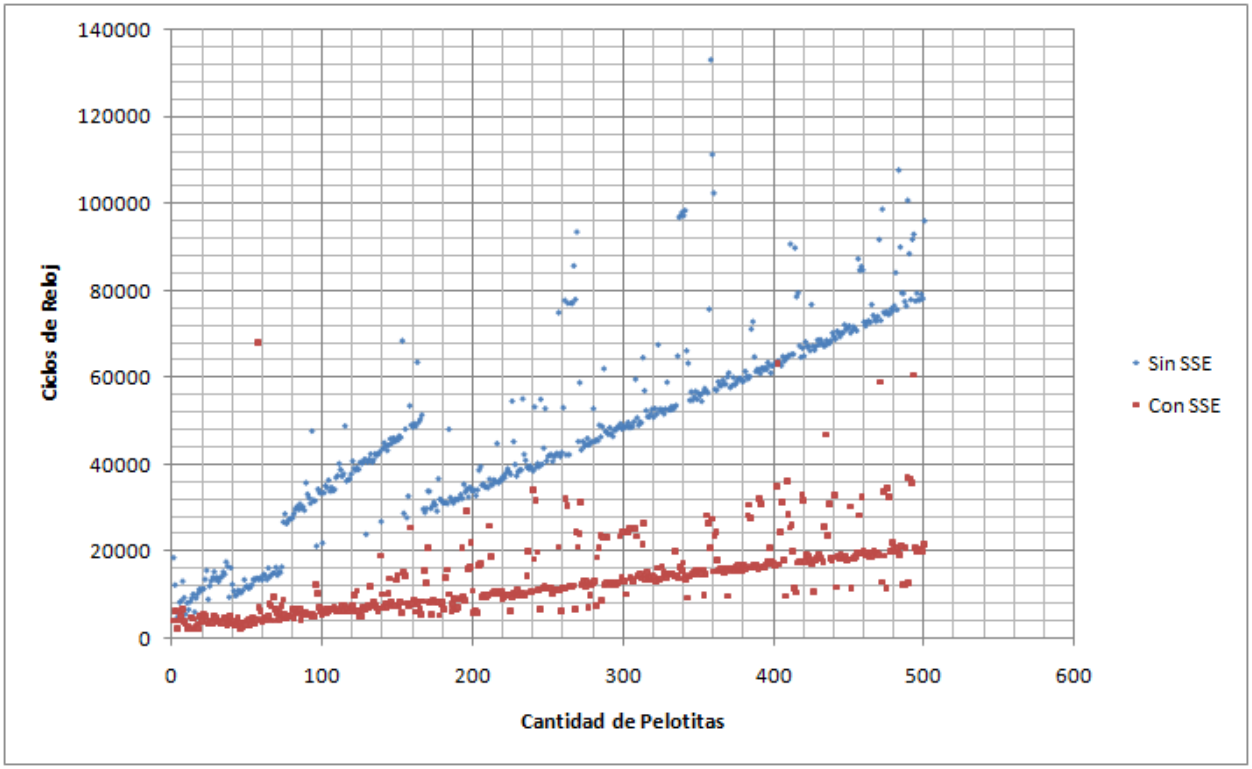

Figura 7: Con y sin SSE

Como primera optimización, volvimos a la idea de acceder de manera alineada a memoria, y nuevamente la performance mejoró notablemente, en el siguiente gráfico puede verse cómo para valores cada vez más grandes, los tiempos de ejecución de un algoritmo y otro van siendo cada vez más distantes llegando incluso a obtenerse una diferencia de casi 4 veces por sobre el algoritmo que no accede alineadamente a memoria.

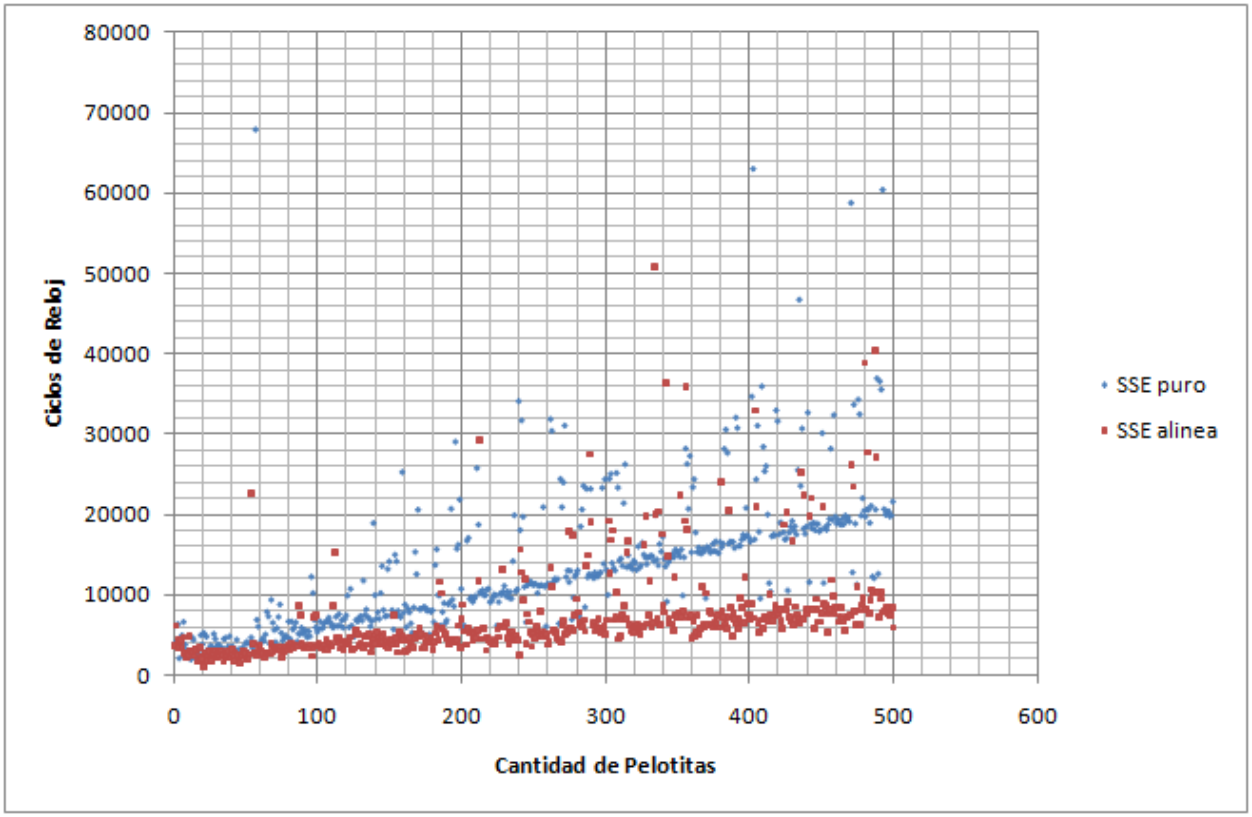

Figura 8: Memoria alineada y sin alinear

Luego, quicimos ver si podíamos optimizar el uso de la cache del sistema. Realizamos dos pruebas las cuales fueron muy exitosas. Esto se debe en gran parte a que el aspecto principal de la función es copiar y pegar por lo que la caché cobra gran importancia dado que cuanto más rapido se acceda a memoria, más rapido se podrán realizar las operaciones.

En los siguientes gráficos vemos dos optimizaciones que intentamos, en la primera cargamos menos en cache que en la segunda.

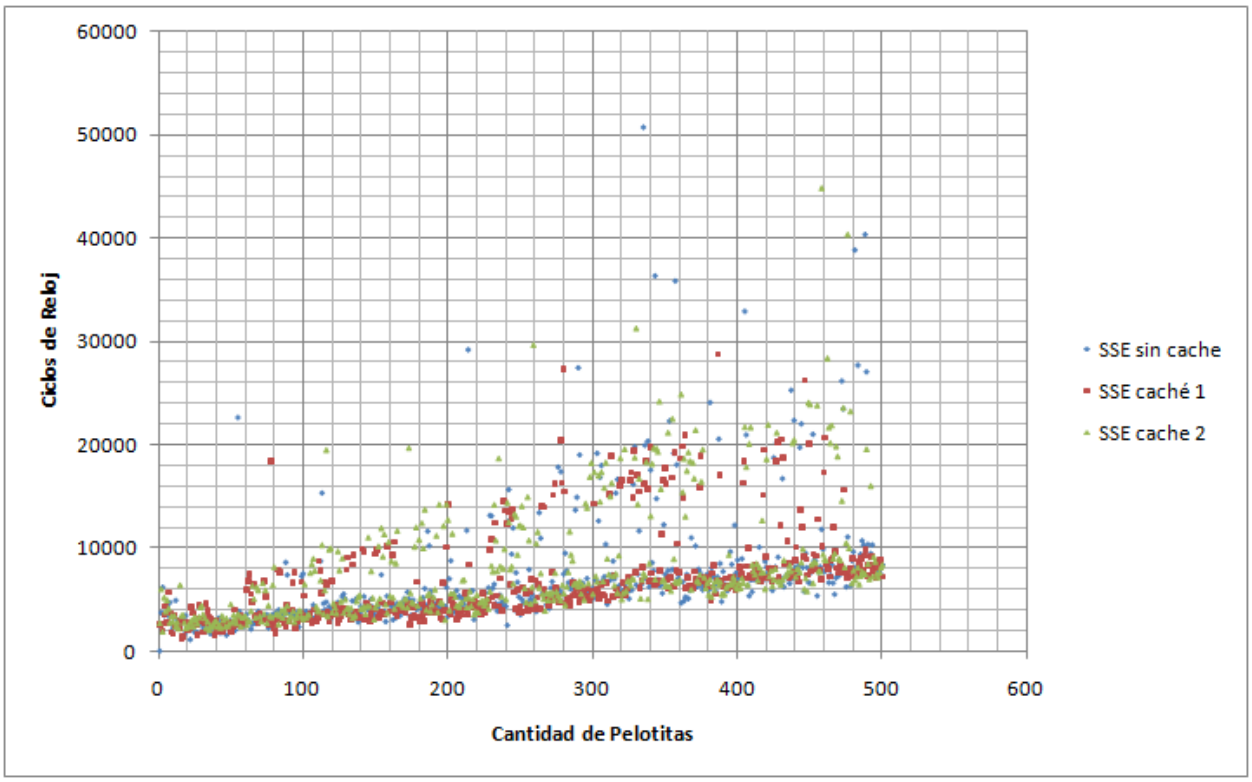

Figura 9: Optimizaciones de cache

Por último, aplicamos la técnica de Loop Unrolling, la cual consiste en aprovechar los registros de la arquitectura (si es que estos están libres) para que en un sólo ciclo del algoritmo se procesen mayor cantidad de elementos, con el fin de paralelizar mejor el prefetch y la ejecución de las instrucciones, evitando que una instrucción tenga que esperar a que la anterior termine para poder ejecutar en los casos en donde una instrucción escribe en un registro y la siguiente lo lee, estos problemas en el pipeline de una arquitectura son denominados RAW Read After Write.

Los mismos se solucionan intercalando entre estas, instrucciones que realicen lo mismo pero con otros registros, así la CPU podrá cargar en paralelo varias posiciones de memoria, y ejecutar las instrucciones siguientes sin que ocurra el problema anterior.

En conclusión, para esta función, notamos que a pesar de ser un ciclo simple y que el compilador utiliza optimizaciones de SSE para el mismo, el código compilado de C++ no fue tan óptimo como el que programamos nosotros. Algo que diferencia esta función del resto, es que por el hecho de ser tan simple logramos aplicar la tecnica de Loop Unrolling para mejorar la performance de la función.

5.3. GetInBounds

Pseudocódigo de la función:

```
Para cada bola i :

posicionActual = obtenerPosicion(i)
radioActual = obtenerRadio(i)

Si posicionActual.x > bordeSuperiorX - radioActual :
posicionActual.x = bordeSuperiorX - radioActual
Si obtenerVelocidad(i).x > 0 :
setearVelocidad(i,x,-obtenerVelocidad(i).x)

Si posicionActual.x < bordeInferiorX - radioActual :
posicionActual.x = bordeInferiorX + radioActual
Si obtenerVelocidad(i).x < 0 :
setearVelocidad(i,x,-obtenerVelocidad(i).x)

Si posicionActual.z > bordeSuperiorZ - radioActual :
posicionActual.z = bordeSuperiorZ - radioActual
Si obtenerVelocidad(i).z > 0 :
setearVelocidad(i,z,-obtenerVelocidad(i).z)

Si posicionActual.z < bordeInferiorZ - radioActual :
posicionActual.z = bordeInferiorZ + radioActual
Si obtenerVelocidad(i).z < 0 :
setearVelocidad(i,z,-obtenerVelocidad(i).z)

Si posicionActual.y > bordeSuperiorY - radioActual :
posicionActual.y = bordeSuperiorY - radioActual
Si obtenerVelocidad(i).y > 0 :
setearVelocidad(i,y,-obtenerVelocidad(i).y)

Si posicionActual.y < bordeInferiorY - radioActual :
posicionActual.y = bordeInferiorY + radioActual
setearVelocidad(i,y,-obtenerVelocidad(i).y)
//abs --> valor absoluto
Si abs(obtenerVelocidad(i).y) < 0 :
setearStandingStatus(true)
setearVelocidad(i,y,0)

setearPosicion(i,posicionActual)
```

Cuando comenzamos a realizar esta función estábamos convencidos de que la optimización iba a ser notablemente mejor que la generada por el compilador desde el codigo de C++ y, sin embargo, al realizar las primeras mediciones nos llevamos la sorpresa de que la performance de nuestro código de assembly era peor que la versión de C++.

Analizando el código y los datos que habíamos obtenido de las mediciones llegamos a la conclusión de que quizás, como los saltos condicionales no eran tan largos e incluso eran relativamente cortos ya que no llegaban a saltar más de una o dos líneas de código, esto podría no ser necesariamente mejor en SSE. Ya que como la posición de memoria a la que se accede con los saltos no es tan distante, la misma tenía grandes posibilidades de ya estar cargada en cache, lo cual implicaba que el overhead por usar SSE era peor que el causado por los saltos condicionales.

Sin embargo, seguimos midiendo tiempos para instancias más grandes, llegando a 500 pelotitas, y notamos que para estos valores, la diferencia en tiempos de ejecucion comenzaba a reducirse cada vez más respecto de la versión de C++:

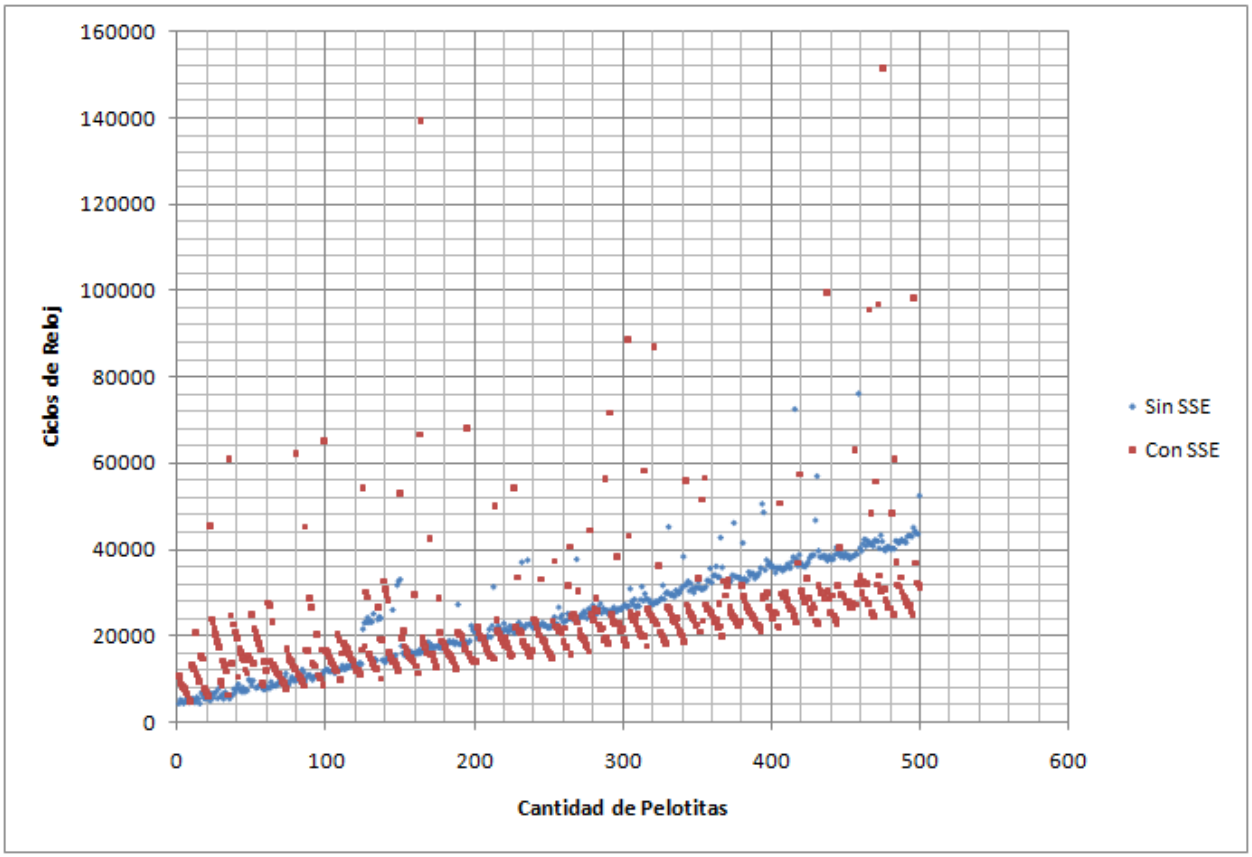

Figura 10: Con y sin SSE

Por ende concluimos que para valores muy chicos de pelotitas la constante de tiempo agregada por causa de usar SSE causa que la función demore más pero, sin embargo, para valores de 180 pelotitas en adelante, comenzaba a notarse una mejora leve pero que iba incrementándose junto con la cantidad de pelotitas usadas.

Luego, seguimos optimizando el código de SSE con el fin de aumentar esta diferencia, y volvimos a realizar, como antes, una otpimización en cuanto a accesos a memoria alineados.

En este punto se nos presentó un problema con el vector de Radios de las pelotitas, ya que el mismo es de floats y, por ende, cada elemento ocupa 4 bytes; como nuestro ciclo ejecuta únicamente una pelotita a la vez, ya no se iba a poder acceder más a la posición del segundo radio de manera alineada, ya que el mismo no estaría alineado a 16 bytes, y así para tres cuartos de los elementos del vector.

Para solucionar esto probamos dos ideas diferentes:

La primera prueba que hicimos fue generar una estructura en la cual almacenar dicho float, con el fin de alinearla a 16 bytes, así dentro del ciclo podríamos movernos de manera alineada a 16 bytes para recorrer el vector de Radios. Por más que en un principio nos pareció poco confiable esta optimización, ya que estaríamos levantando cuatro veces más memoria que si simplemente accediéramos de manera no alineada a cada float, al realizar las mediciones nos llevamos la sorpresa de que los tiempos de ejecución se habían reducido!

Este hecho nos llevó a la conclusión de que acceder de manera alineada a memoria es algo que debería realizarse siempre que se pueda en cualquier código que procesa vectorialmente, ya sea assembly o C++.

En el siguiente gráfico puede verse cómo a medida que se incrementa la cantidad de pelotitas comienza a generarse una curva por debajo de los tiempos del algoritmo de C++:

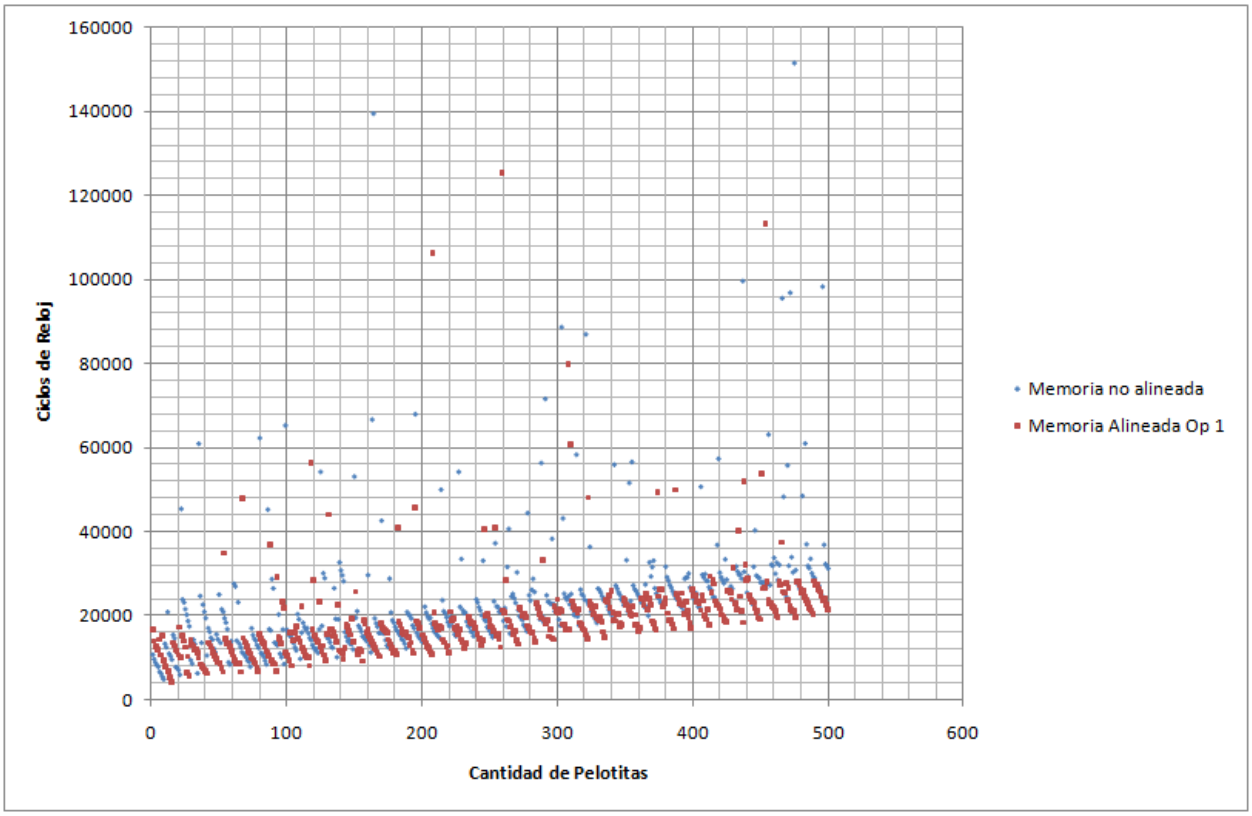

Figura 11: Con y sin memoria alineada, optimización 1

La segunda prueba que hicimos fue no acceder a memoria en dicho arreglo de Radios pero sí acceder en el resto de los vectores de Posiciones y Velocidades, y nuevamente al realizar las mediciones no sólo vimos que era más eficiente que el código de C++, sino que nos quedaron muy parecidas las mediciones a las realizadas para la optimización anterior, salvo por las siguiente diferencias:

Para una cantidad de pelotitas no muy grande, menor a 150, la segunda optimización resultó mejor y más estable, presentaba menos cantidad de picos. Sin embargo, para una cantidad de pelotitas mayor, se notaba cómo la primera optimización comenzaba a estabilizarse por debajo de los tiempos de la segunda optimización.

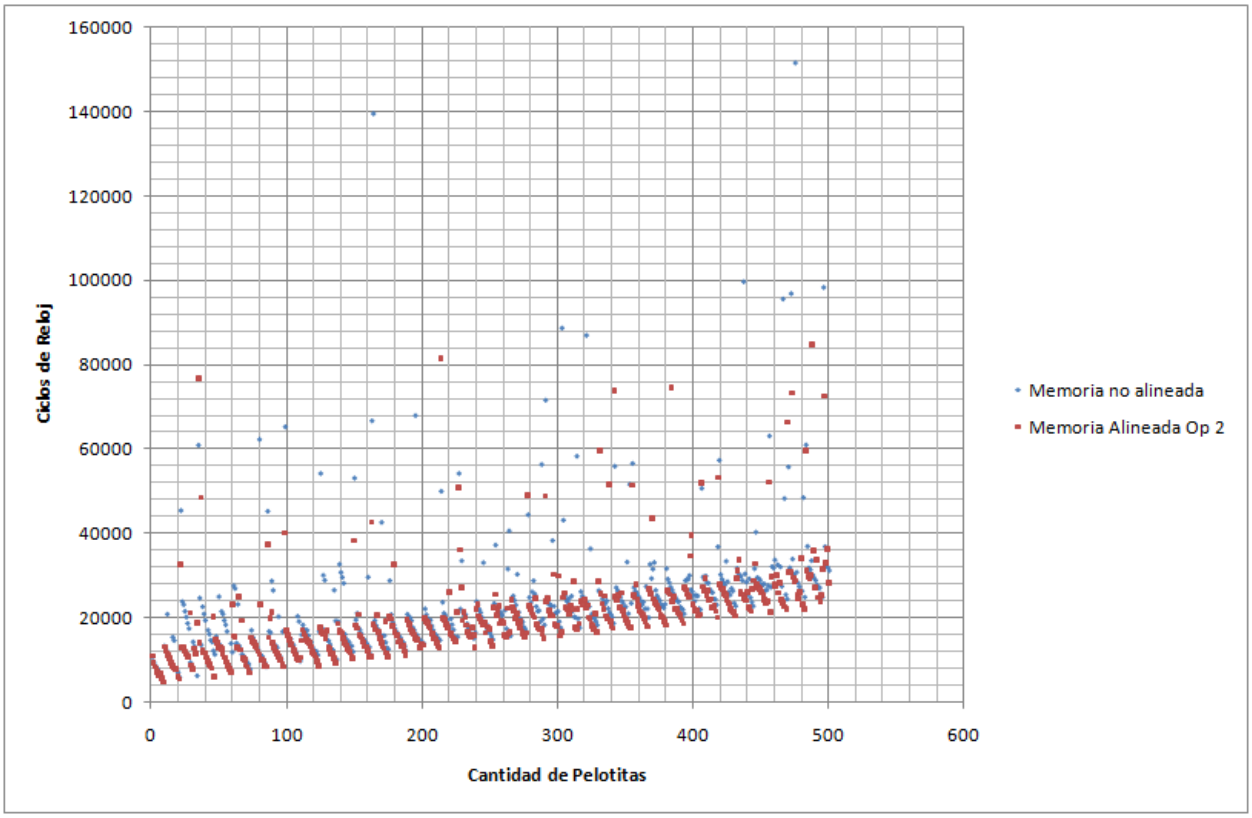

Figura 12: Con y sin memoria alineada, optimización 2

Tomando en cuenta esto y que nuestro código no llegaba a procesar tantas pelotitas decidimos dejar la segunda optimización.

En el siguiente gráfico puede verse lo anteriormente mencionado, midiendo ambas optimizaciones:

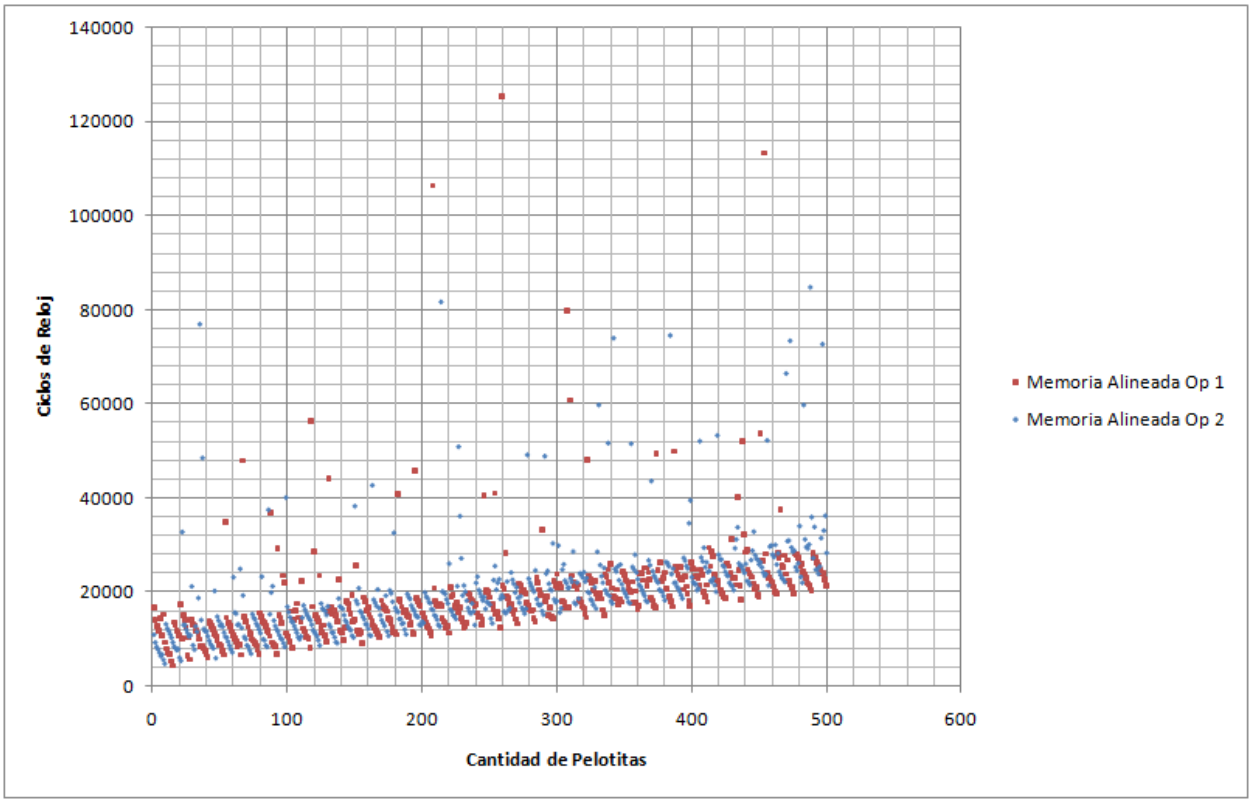

Figura 13: Ambas optimizaciones con memoria alineada

Finalmente, buscamos optimizar utilizando la cache y logramos reducir levemente los tiempos y no sólo eso, sino que para instancias cada vez más grandes de pelotitas, los tiempos comenzaban a estabilizarse mejor que sin esta optimización e incluso se parecían bastante a los realizados en el paso anterior con la primera optimización.

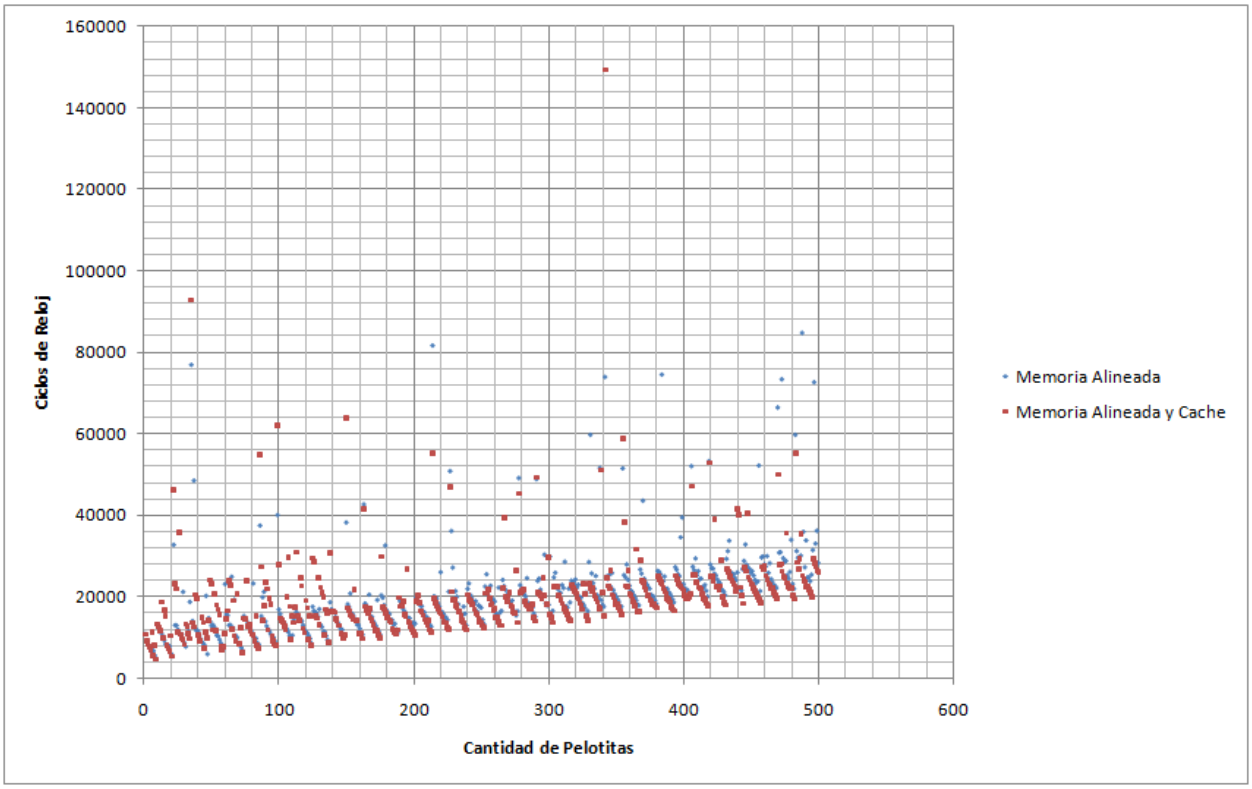

Figura 14: Optimización de cache

En conclusión, para esta función notamos por un lado lo relativo que puede ser mejorar un salto condicional utilizando SSE, ya que si la cantidad de instancias no es lo suficientemente grande, la constante de tiempo que agrega SSE puede empeorar nuestra performance. Por otro lado, notamos también que a pesar de levantar cuatro veces más memoria con una optimización que accede alineadamente, los resultados siguen siendo mejores que si se accede a menos cantidad de memoria pero desalineadamente! Este hecho refleja nuevamente lo importante que es alinear la memoria que se va a usar exaustivamente.

5.4. raySphere

Salta a la vista que esta función es una clara candidata a ser realizada en FPU dado que todas las variables y parámetros que se trabajan son *float*, y se realizan demasiadas cuentas con dichos valores. Uno de los mayores desafíos con los cuales se debe topar un programador de FPU es cómo lograr realizar una carga adecuada de los valores para no quedarse sin registros o tener que realizar demasiados swapeos entre estos ya que confunden el seguimiento de la pila y complican la lectura del código.

Como bien sabemos, contamos con 8 registros en FPU. Lamentablemente necesitamos realizar cuentas con 10 parámetros y almacenar el valor de 3 variables (a, b y c). Si bien a priori pareciera imposible poder cumplir con el requisito óptimo de sólo usar los registros sin la necesidad de tener que acceder a memoria, mirando fijamente las fórmulas e invocando al Dios de la Programación del FPU, se puede esotéricamente llegar a una forma de realizar las cuentas la cual permite prescindir de los accesos recurrentes a memoria más que para cargar inicialmente los parámetros a FPU.

De esta función, no hay grandes optimizaciones (o al menos no se nos ocurrieron) por hacer puesto que sólo se trataron de muchas cuentas. Lo ventajoso de programarlo en FPU fue que nosotros pudimos decidir cómo realizar las cuentas y en qué orden para así sólo utilizar registros. Por ende, pudimos obtener una función más eficiente que la hecha en C++, no siendo, sin embargo, grandes órdenes de magnitud mejor dado que no paralelizamos ni precargamos datos, simplemente no accedimos innecesariamente a memoria.

Cabe destacar por último, que los detalles de la implementación no son muy amistosos para redactar en un informe pero los mismos se pueden encontrar en la implementación de la función, la cual está vastamente comentada.

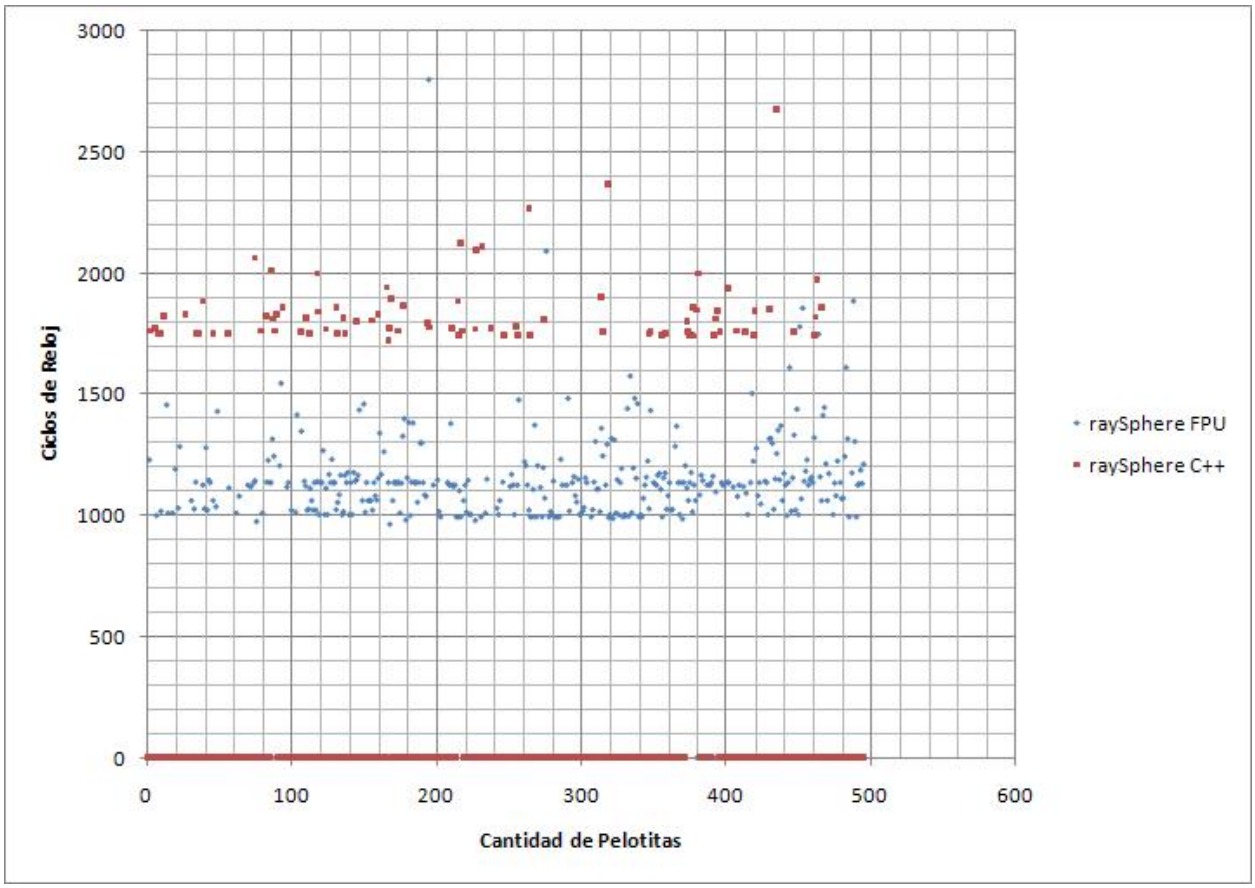

Figura 15: Comparación del rendimiento en C++ y FPU

6. Conclusiones

Como conclusiones de este trabajo práctico podemos destacar que nos sirvió mucho no sólo para aprender a optimizar utilizando una arquitectura SIMD como es SSE, sino que además nos dio pautas para tomar en cuenta a la hora de programar en lenguajes de más alto nivel, para lograr beneficiar el código assembly al cual será traducido nuestro código.

También nos pareció muy llamativa la gran ganancia que estas arquitecturas SIMD representan en cuanto a performance y que, sin embargo, parecería ser un tanto complicada su escalabilidad a futuro, ya que de aparecer una nueva arquitectura SIMD, llamémosla SSEX, la cual procese de a 32 o 64 bytes, todo el código generado hoy en dia deberia ser reescrito para lograr beneficiarse de esta posible nueva arquitectura.

Esto nos abrió los ojos a lenguajes de Pixel Shaders o Vertex Shaders, lo cuales solucionan este problema brindando al usuario una alternativa mas escalable, en donde un programador simplemente se preocupa por programar el codigo que se ejecutaría dentro de un ciclo y para un solo elemento, y luego el compilador es el encargado de traducir esto a un assembly de placa de video con el fin de aprovechar la arquitectura SIMD de la misma y procesando en paralelo. Sería muy interesante ver si realmente estos códigos son óptimos a tal nivel que se vuelva díficil o imposible generar un código en assembly para placa de video mejor que el generado por los Pixel Shaders o Vertex Shaders.

Otro tema que nos llamó la atención a partir de la realización de este TP, gira en torno a la existencia de modelos físicos que puedan ser paralelizables, quizás planteados como sistemas de ecuaciones, o de alguna otra manera. Dado que estos sistemas físicos son muy utilizados en simulaciones o video juegos, sería de gran utilidad poder paralelizarlos para lograr reducir lo más posible su tiempo de ejecución.

En cuanto a la programación en SSE que realizmos podríamos destacar lo siguiente:

Por un lado, el importantisimo papel que juega la memoria alineada a 16 bytes, al punto de determinar la eficacia o no de esta arquitectura para ciertos algoritmos. Tomando en cuenta esto se podría decir que fue un gran avance por parte de Intel el haber agregado al set de instrucciones la instruccion MOVAPS.

También podría hacerse referencia a la optimización con cache, que no siempre resulta efectiva y debe tratarse con cuidado, ya que sino podría interferirse con otras optimizaciones que realiza el procesador sobre los códigos que compila o, incluso, optimizaciones on run time que efectúa, más que nada, sobre memoria.

Por otro lado, el overhead que implica usar SSE, no siempre puede resultarnos admisible, y debería realizarse un estudio de nuestro algoritmo en cuanto a mediciones de performance para saber si efectivamente resulta útil. Esto nos da la pauta de lo importante que es el desarrollo de optimizaciones automáticas por parte de los compiladores, como O3 por ejemplo, para poder eliminar el tiempo que lleva programar en assembly y que además podría resultar contraproducente en muchos casos.