uc3m | Universidad Carlos III de Madrid

Proyecto 1

ARQUITECTURA DE COMPUTADORES

Ingeniería informática - Grupo 81

Fecha	**22/10/2021**
Autores	Adrián Mancera González - 100429049 100429049@alumnos.uc3m.es
	Gonzalo Juarez Tello - 100467578 100467578@alumnos.uc3m.es
	Hodei Urigoitia Merodio 100374256@alumnos.uc3m.es
	Gonzalo Martínez Martín - 100428963 100428963@alumnos.uc3m.es

1.ÍNDICE

2.DISEÑO ORIGINAL

- ## 2.1.AOS (Array of Structures):

    - Estructura de datos (archivo obj.hpp)

Aquí la estructura de datos se basa en un array de structs.

Inicialmente trabajamos con un **struct obj** donde los miembros eran punteros a **struct vec**, que a su vez tenía los miembros x, y, z. De manera que 3 punteros a distintos struct vec en un struct obj eran suficientes para la fuerza, la velocidad, y la posición. No se guarda la aceleración en el struct porque es calculable y no se accede frecuentemente como para que valga la pena almacenarlo. Tampoco almacenamos un booleano para indicar la existencia de struct obj porque esto se hace comparando su masa con 0 (ver macros *obj_exists* y *obj_delete)*. Esta primera estructura surgió con la idea de mantener **struct obj** por debajo del tamaño de línea de caché (64 bytes).

    - Archivo obj.cpp

En este archivo se pide memoria al sistema para guardar los struct y se asignan valores a todos los objetos en la función **init_obj_list()**. Esto consiste en generar la posición inicial de cada objeto usando una distribución uniforme, y la masa por distribución normal, como lo indica el enunciado. A su vez se setean velocidades y fuerzas a 0 para evitar que los cálculos posteriores se encuentren con lo que sea que haya estado en la memoria previamente (aunque posiblemente alguna implementación de C++ se encargue de esto, mejor prevenir que lamentar).

Otra función importante es **merge_obj()**, la cual se encarga de la fusión de objetos a y b (usado en colisiones). Al objeto a se le suman las componentes del objeto b. El objeto b es “eliminado” al setear su masa a 0 con *obj_delete*. Esto luego puede ser chequeado con el macro *obj_exists.*

Por último, la función **destroy_obj_list()**, básicamente es el destructor de struct obj_list. El manejo de errores en las funciones del archivo obj.cpp se encarga de fallos en las solicitudes por memoria al sistema (mallocs y frees). Valgrind memcheck asegura que no hay leaks de memoria.

    - Archivo sim.cpp

En sim.cpp está la función **main()**, la cual llama a funciones para la toma y parseo de argumentos, luego inicializa un **struct obj_list** llamado **o_list** con el tamaño especificado en la línea de comandos. o_list luego se pasa como argumento a **init_obj_list()** para rellenar los struct con valores iniciales. Al retornar a main, se llama a **simulate()**.

En simulate(), primero se comprueba si hay choques en **collision_check().** Luego, por el número especificado de iteraciones, se itera por todos los objetos, chequeando su existencia. En cada iteración se compara un objeto i con todos los objetos desde i+1 hasta o_list->size (salteando aquellos que no existen). Para cada objeto comparado con el objeto de la posición i, se calcula una fuerza con **calc_fgv()**, función que aplica la fuerza a ambos objetos en sentidos opuestos para no repetir operaciones. Una vez calculadas todas las fuerzas, se calcula la velocidad (que de por medio calcula la aceleración), la posición, y se setean las fuerzas del objeto i a 0 para que no afecten a la siguiente iteración. Al calcular las posiciones de todos los objetos, se verifican colisiones.

- 2.2.SOA (Structure of Arrays):

No hay primera versión de SOA. Nos centramos en hacer un algoritmo para AOS. Una vez obtenida la versión optimizada de AOS, el código se modificó para utilizar un struct de arrays. En el procesador utilizado, con 12 vías en la caché L1 de datos (número de arrays en struct de arrays < número de vías), funcionó bastante bien en especial en hit rate a caché L1d al comparar con las distintas versiones de AOS (original y optimizada). Más sobre esto en las secciones 3 y 4.

3.OPTIMIZACIÓN

- 3.1.AOS:

  - Estructura de datos (archivo obj.hpp):

Ante la excesiva cantidad de instrucciones y accesos a memoria que requiere el primer enfoque para seguir punteros, nos decidimos en dejar todos los campos necesarios dentro de **struct obj**. El struct supera el tamaño de línea de caché (64 bytes), ocupando un total de 80 bytes. **struct vec** fue retirado del código.

También se reemplaza struct obj_list por un vector de la librería estándar de C++. Esto simplifica bastante las funciones declaradas en este archivo .hpp e implementadas en obj.cpp al no tener que hacerse cargo directamente del pedido y liberación de recursos.

  - Archivo sim.cpp:

La optimización más importante que se puede ver en el código fuente y en el binario resultante es el inline de funciones de cálculo usadas exhaustivamente en los loops del programa. Reduciendo así la cantidad de llamadas a funciones. Es decir, se quitaron instrucciones para el paso de argumentos, instrucciones del epílogo y prólogo de funciones, y obviamente las instrucciones de llamada y retorno en sí. Específicamente se hizo esto para **calc_norm(), calc_fgv(), calc_vel(), y calc_pos()**.

Otra optimización fue el reemplazo de la función pow() (algo como pow(x, 3) quedó como x*x*x).

  - Archivo obj.cpp:

Chequear la existencia de un objeto se hace con los macros *obj_exists,* y la “eliminación” de un objeto con *obj_delete.* Esto no ha cambiado con respecto a la versión anterior.

Una desventaja del uso de *obj_exists* y *obj_delete* es que el objeto b tras una colisión no es realmente eliminado y el código se plaga de *obj_exists* en loops críticos. Una alternativa que surgió fue un cambio en **merge_obj()** donde se usaba una operación std::move() para llevar el último objeto del array hacia la posición del objeto b, y luego vector.pop_back() para acortar el tamaño del vector. Esto resultaba en un código que nunca debía chequear por la existencia de un objeto. A pesar de la ausencia de condicionales, no presentó una diferencia significativa entre el código con chequeo de existencia de struct obj. Esto sería porque dada la rareza de una colisión, la predicción de saltos funciona bastante bien. Esta “optimización”, nos pareció lo suficientemente interesante como para anotar en la memoria aunque no haya quedado plasmada en el código final. Al alterar el orden de los objetos, el output de las simulaciones ejemplo y esta implementación presentaban diffs en simulaciones con colisiones.

  - Resto de los archivos:

El resto de los archivos siguen casi igual a sus equivalentes en la versión original.

  - Optimizaciones del compilador:

Se añadió el flag **-Ofast** que activa:

- ffast-math: optimización para mejorar la velocidad de operaciones matemáticas. Principalmente nos interesa -funsafe-math-optimizations, que permite reordenamientos en operaciones de punto flotante que no conforman especificaciones IEEE. Si bien las operaciones en punto flotante no son asociativas, no hemos notado diffs al probar contra el output de las simulaciones ejemplo.
  Otro punto interesante es que --fast-math permite el uso de aproximaciones por métodos numéricos (ej.: Newton-Raphson) para divisiones y raíces cuadradas. También evita el chequeo de 0 o NaN en lugares que causarían daños (división por 0). La función collision_check() y el chequeo con el macro *obj_exists* se encargan de eliminar tales casos dañinos (donde por ejemplo el divisor sea una masa igual a 0). En resumen, no afecta a nuestro programa, por lo cual decidimos dejar la opción --fast-math activada.

- fallow-store-data-races: optimizaciones como hoisting de variables o conversiones de if-statements que pueden causar que un valor que ya esté en memoria sea reescrito con el mismo valor. Puede ser peligroso para programas con varios hilos de ejecución. No afecta al programa ya que no posee más que un único thread. Posiblemente deba desactivarse en la versión paralela de este programa.

También se usó el flag **-march=native**, que optimiza el código para la máquina donde sea compilada. En el caso de la computadora donde se desarrolló el programa, con un i7-1065G7, esto resultó en el uso de instrucciones AVX. **Valgrind** presentó problemas para simular estas instrucciones AVX. Por lo que al correr con -march=native en i7 10ma gen valgrind no reconocía las instrucciones AVX y finalizaba con “Illegal opcode”. El chequeo de hits a caché se hizo sin -march=native, y para la verificación de leaks de memoria se tuvo que usar un sanitizer (LeakSanitizer: -fsanitizer=address). El sanitizer no está incluído en las opciones de compilación de release pero la línea para hacerlo está comentada en CMakeLists.txt.

● 3.2.SOA:

○ Estructura (archivo obj.hpp):

En este caso se usa un struct de arrays para almacenar la información. Dentro de un **struct obj_soa** se tiene la misma cantidad de miembros que en **struct obj** de AOS, pero cada miembro es un vector del tamaño especificado (un array en el heap). En esta versión escribimos un constructor **obj_soa()** donde tiene lugar la inicialización que en AOS se da en init_obj_list().

○ Archivo obj.cpp:

En este archivo se implementa la función **merge_obj()**, en la cual se lleva a cabo la fusión de los objetos; asignamos al primer objeto la suma de las masas y la suma de las velocidades de cada componente. Luego se setea la masa en el índice del objeto “eliminado” a 0 usando el macro *obj_delete*.

○ Archivo sim.cpp:

En **main()** se crea un **struct args** en el que quedan almacenados los argumentos de la línea de comando una vez parseados. El resto de main() es similar a lo descrito para AOS, con la diferencia que el objeto inicializado es obj_soa y la inicialización se da en el mismo constructor de obj_soa en lugar de una función aparte (como lo es init_obj_list() en AOS).

El funcionamiento y la implementación del resto del programa es prácticamente idéntico a lo mencionado anteriormente en AOS, con pequeñas modificaciones para funcionar con el cambio en la estructura de datos.

Para el desarrollo y las pruebas se usó un procesador i7-1065G7 (specs anotadas en la sección 4). A nivel de caché de datos L1d esta versión presenta una tasa de fallos muchísimo menor al ejercicio en AOS. Por ejemplo: para input 1000 50 666 1000000.0 0.1 la tasa de fallos gira en torno a 0.7% para D1 y tan pequeña para las caché de nivel 2 y 3 que valgrind coloca la tasa de fallos a LL en 0%. Mientras que para AOS, el mismo input gira en torno a 6% para D1 y 0% para LL.

- Optimizaciones del compilador:

Ídem optimizaciones en AOS.

4.EVALUACIÓN DE RENDIMIENTO.

El CPU donde se desarrolló el código y se ejecutaron las pruebas es, como ya fue mencionado, un intel i7 de 10ma generación (i7-1065G7):

- 4 núcleos (8 hilos).
- caché L1 de datos:
  - 192 KiB (48 KiB para c/núcleo).
  - 12 vías (64 conjuntos por núcleo).
  - 64 bytes por línea de caché.
- caché L1 de instrucciones:
  - 128 KiB (32 KiB para c/núcleo).
  - 8 vías (64 conjuntos por núcleo).
  - 64 bytes por línea de caché.
- caché L2:
  - 2 MiB (512 KiB para c/núcleo).
  - 8 vías (1024 conjuntos por núcleo).
  - 64 bytes por línea de caché.
- caché L3 (compartida):
  - 8 MiB
  - 16 vías (8192 conjuntos).
  - 64 bytes por línea de caché.

El sistema operativo es OpenSUSE Tumbleweed, usando el kernel 5.14.11 y la versión de gcc 11.2.1 (es una distribución rolling-release, así que estas versiones pueden haber cambiado a lo largo del desarrollo).

Para las pruebas individuales se usaron las **configuraciones fijas random_seed**=666, **size_enclosure**=1000.0, **time_step**=0.1. Luego a **num_objects** se le asignaron los valores 1000, 2000, y 4000. Mientras que **num_iterations** tomó los valores 50, 100, 200. Se probaron combinaciones de los valores de num_objects y num_iterations, y se obtuvieron un total de 9 pruebas para c/u de los casos estudiados (AOS original, AOS optimizado, y SOA: 27 pruebas en total).

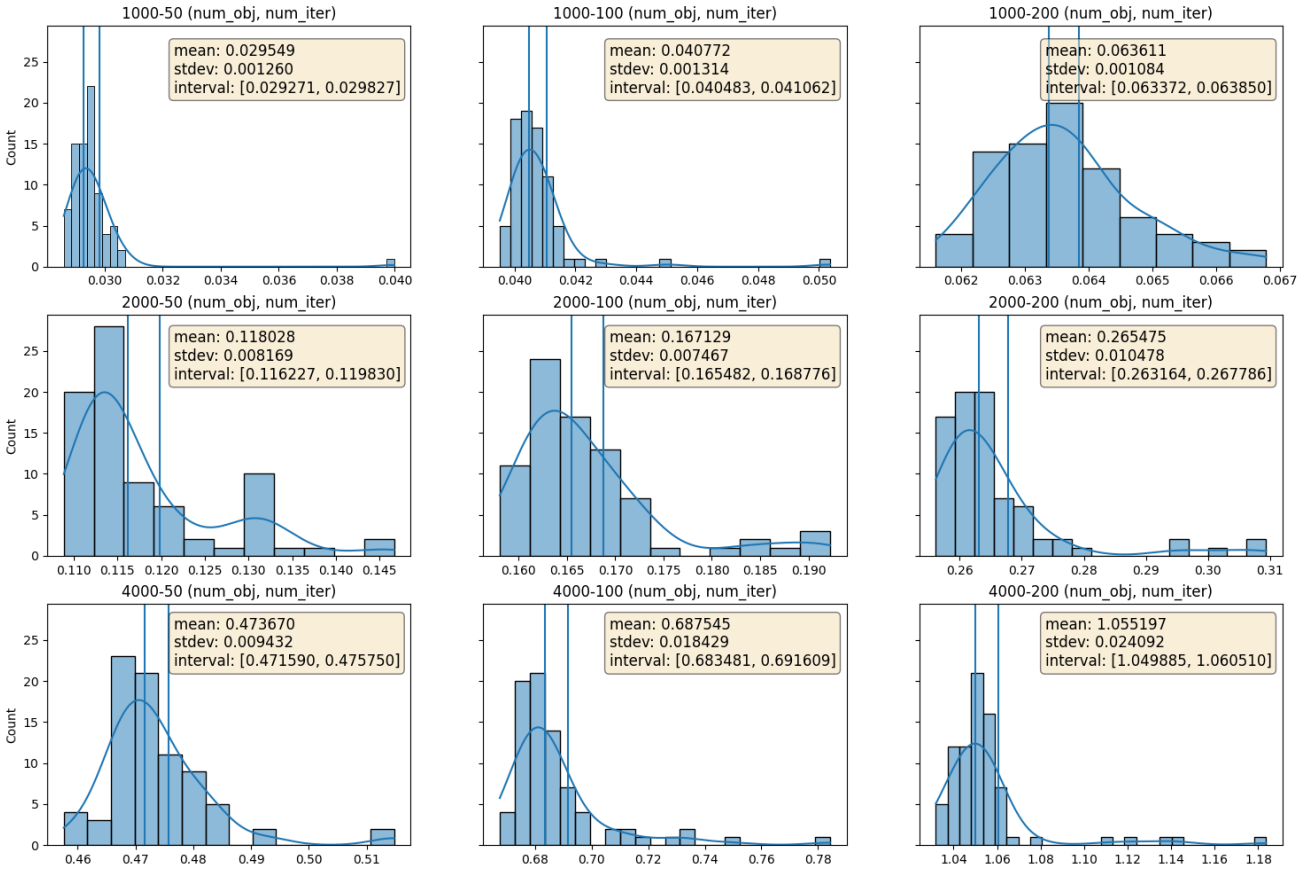
Cada prueba tiene una muestra de tamaño n=80 y un nivel de significancia α=0.05. Se eligió este valor para n porque es mayor a 30, por lo que podemos usar una distribución normal para la obtención de estadísticos, y porque con semejante tamaño la desviación estándar es poca.

Los gráficos se corresponden a las 9 pruebas en cuestión. Son histogramas donde el eje de las abscisas se a **tiempos de ejecución en segundos**, y el eje de las ordenadas a la **frecuencia de**

clase del histograma (cantidad de ejecuciones que entran en el rango de una única barra del histograma). Las líneas verticales indican comienzo y fin del intervalo de confianza.

Los gráficos de izquierda a derecha aumentan en número de iteraciones, mientras que de arriba hacia abajo aumentan en número de objetos. Desde ya, una observación común a todos será que hay una mayor diferencia con el gráfico vecino vertical, que con el gráfico vecino horizontal. Una conclusión temprana nos daría a pensar que aumentar el número de objetos afecta más que aumentar el número de iteraciones. Pero hay que tener en cuenta que num_objects aumenta de 1000, a 2000, a 4000, mientras que num_iter lo hace de 50, a 100, a 200, o sea, aumento_num_objects > aumento_num_iter. Lo que sí es cierto es que ambos factores incrementan la cantidad de instrucciones que se ejecutan en la parte computacionalmente intensiva del programa porque ambos aumentan la cantidad de veces que se ejecuta el loop anidado de la función simulate().

- 4.1.AOS:
  - *4.1.1.Versión original*



Para controlar la cantidad de comparaciones, vamos a hablar de la línea de histogramas del medio, con 2000 objetos e iteraciones que van desde 50 hasta 200. ¿Por qué no probar con más objetos, de manera que el array no entre en caché LL? Con una suma de caché L2+L3 = 512 KiB + 8

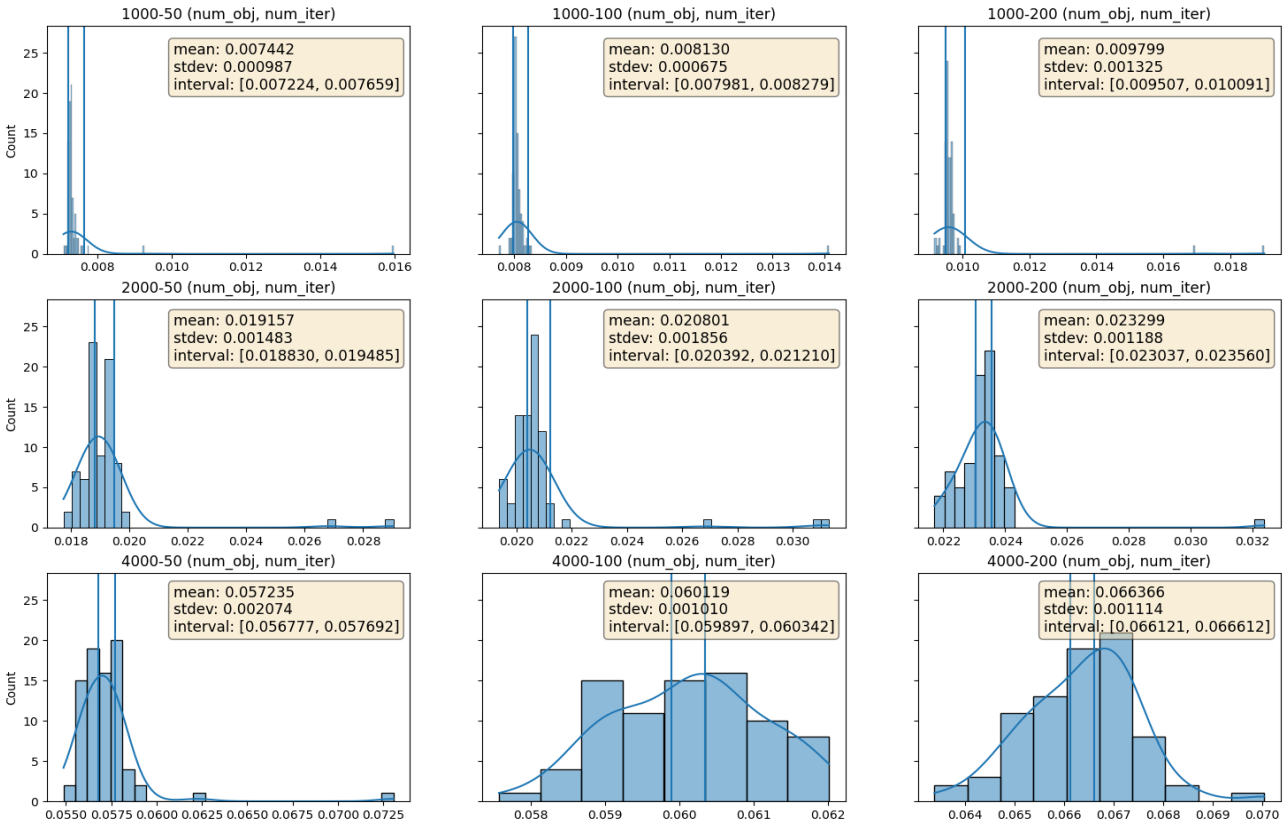
MiB. Siendo que struct obj es de 32 bytes, y referencia 3 struct vec de 24 bytes c/u. Eso nos deja con poco más de 35mil objetos (size_LL/[32+24*3]). Manteniendo el número de objetos en un punto medio y fijo podemos ver como aumenta la cantidad de instrucciones, tasa de fallos a caché L1 (solo tiene capacidad para 472 objetos), y tiempo de ejecución según qué tanto se incremente la parte computacionalmente intensiva del programa (y la duración de las simulaciones en un tiempo mentalmente saludable).

Acá se presentan resultados de fallos a caché y referencias a instrucciones . El resto del análisis sigue una vez presentados los resultados de la versión optimizada.

- 50 iteraciones-2 mil objetos:
  - 1.114.379.127 referencias a instrucciones, 0.1% fallos a L1i, 0.0% fallos a LL por instrucciones.
  - 288,289,177 referencias a datos (245,896,145 lecturas, 42,393,032 escrituras), con 9.1% fallos a L1d (26,348,799 lecturas, 7,433 escrituras), 0.0% fallos a LL por datos.
- 100 iteraciones-2 mil objetos:
  - 1.720.612.512 referencias a instrucciones, 0.1% fallos a L1i, 0.0% fallos a LL por instrucciones.
  - 389,374,890 referencias a datos (346,853,364 lecturas, 42,521,526 escrituras), con 11.5% fallos a L1d (44,915,351 lecturas. 8,233 escrituras), 0.0% fallos a LL por datos.
- 200 iteraciones-2 mil objetos:
  - 2.933.079.165 referencias a instrucciones, 0.0% fallos a L1i, 0.0% fallos a LL por instrucciones.
  - 591.546.103 referencias a datos (548.767.581 lecturas, 42.778.522 escrituras), con 13.9% fallos a L1d (82.051.138 lecturas, 8.651 escrituras), 0.0% fallos a LL por datos.

A lo largo de las pruebas, la media de instrucciones por ciclo es de ~2.93 (o sea 0.34 CPI). Ya se dispone de CPI, cantidad de instrucciones, y tiempo medio de ejecución.

- *4.1.2.Versión optimizada.*



Por si no es obvio viendo los valores en los histogramas, la versión optimizada de AOS es significativamente más rápida que la versión original. Se hizo un estudio usando las muestras obtenidas y una comparación por análisis de varianza para respaldar numéricamente esta conclusión. Pero queda bastante claro viendo la diferencia numérica. Para una comparación más interesante, al final de la sección 4 hay una comparación entre SOA y AOS optimizados.

Volviendo a la comparación AOS original vs. AOS optimizado. Los resultados de la estructura optimizada se mostrarán sin el flag *-march=native* porque *valgrind* no soporta las instrucciones AVX generadas para el procesador usado. Podría haberse optado por las herramientas de caché que proporciona *perf*, pero hay varias opciones de interés que perf no soporta en el sistema en el que se hicieron las pruebas. Además, introduciría las inconsistencias propias de comparar resultados de distintas herramientas. Los resultados son:

- 50 iteraciones-2 mil objetos:
  - 222,432,948 referencias a instrucciones, 0.4% fallos a L1i, 0.0% fallos a LL por instrucciones.
  - 72,262,968 referencias a datos (52,266,752 lecturas, 19,996,216 escrituras), con 8.7% fallos a L1d (6,299,741 lecturas, 8,653 escrituras), 0.0% fallos a LL por datos.
- 100 iteraciones-2 mil objetos:
  - 238,784,674 referencias a instrucciones, 0.4% fallos a L1i, 0.0% fallos a LL por instrucciones.
  - 74,099,558 referencias a datos (54,090,185 lecturas, 20,009,373 escrituras), con 10.9% fallos a L1d (8,096,693 lecturas, 8,953 escrituras), 0.0% fallos a LL por datos.
- 200 iteraciones-2 mil objetos:
  - 271,487,937 referencias a instrucciones, 0.0% fallos a L1i, 0.0% fallos a LL por instrucciones.
  - 77,772,642 referencias a datos (57,736,974 lecturas, 20,035,668 escrituras), con 15.0% fallos a L1d (11,690,593 escrituras, 9,553 lecturas), 0.0% fallos a LL por datos.

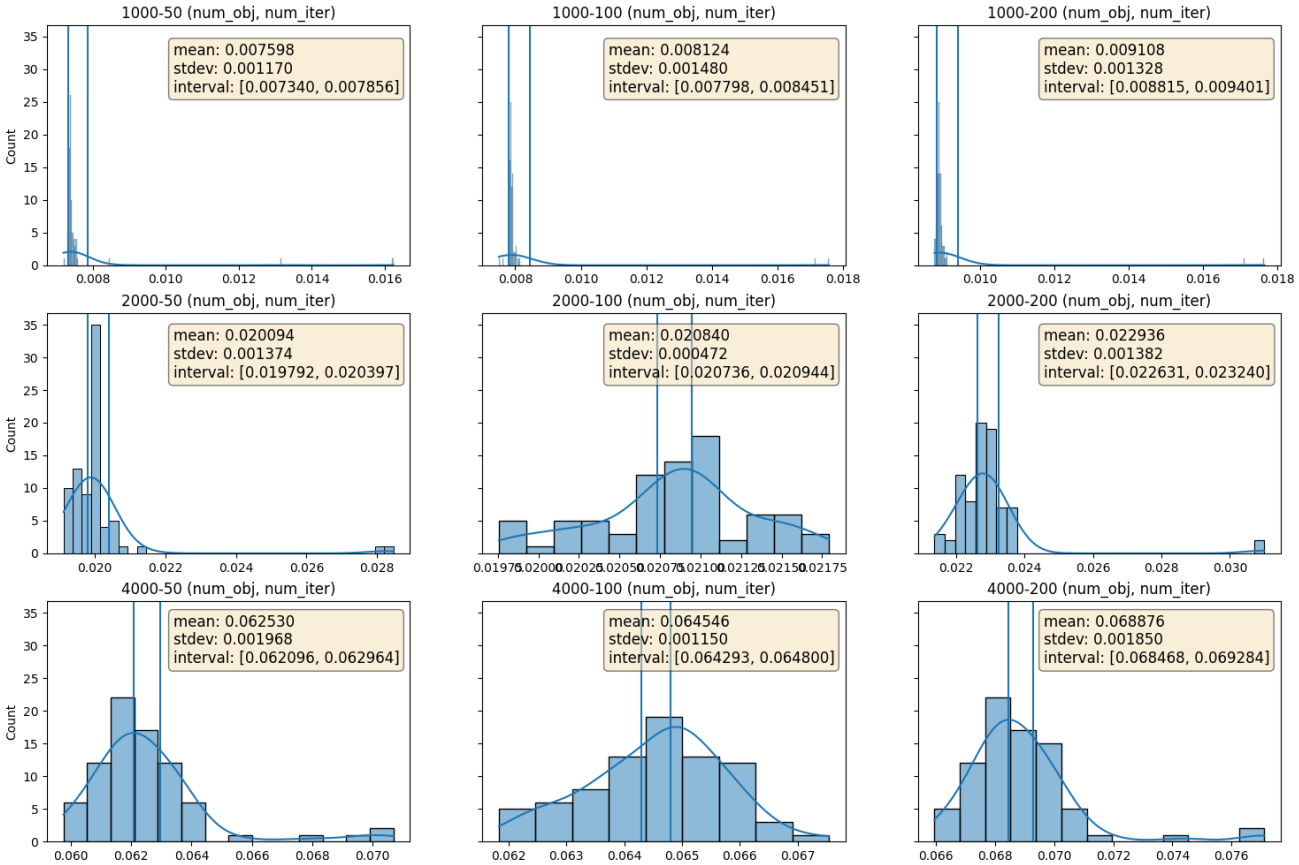
A lo largo de las pruebas, la media de instrucciones por ciclo está un poco por encima de aquella de la versión original, con un valor de ~3.03 (o sea 0.33 CPI).

¿Por qué la diferencia en tiempos de ejecución? Más allá de las optimizaciones del compilador explicadas anteriormente, el **cambio en la estructura de datos** cambió a montones la cantidad de **instrucciones ejecutadas y referencias a datos**. Por ejemplo, en 50 iteraciones, la versión original ejecuta 1.1 mil millones de instrucciones, y tiene 288 millones de referencias a datos. Entre que la versión optimizada solo ejecuta 220 millones de instrucciones, y tiene referencia a 72 millones de datos (y la mayor parte de esa diferencia son lecturas). No solo eso, si no que la diferencia de referencias a instrucciones y datos al cambiar de configuración es altísima para la versión original, mientras que la versión optimizada tiene cambios relativamente pequeños. De 50 a 100 iteraciones hay una diferencia de 101 millones de referencias a datos en el código original, mientras que en el optimizado esta diferencia es de poco menos de 2 millones (recomiendo ver el mismo caso pero para referencias a instrucciones).

Si bien a medida que aumenta el número de iteraciones, vemos como la cantidad a fallos de caché de la versión optimizada supera la cantidad de fallos de caché de la versión original, al ver los valores absolutos de estos fallos queda claro que son muchísimos menos (poca diferencia en el porcentaje que se aplica sobre una gran diferencia en referencias totales). Esta diferencia en fallos a caché puede explicarse porque de hecho la versión original sí tiene estructuras que entran en una línea de caché, no como la optimizada.

Tanto la versión optimizada como la original sufren de escalabilidad. Mientras más iteraciones o más número de objetos hay en el sistema, mayor es el porcentaje de fallos a caché L1. Esto es porque **la caché L1d NO tiene capacidad suficiente para contener al array de structs.** En la siguiente subsección analizamos el caso de SOA y cómo afecta a la tasa de fallos a L1.

- 4.2.SOA:



Viniendo de analizar AOS optimizado, podemos notar que los histogramas, estimadores (media y varianza), e intervalos de confianza son bastante similares. Para proceder de la misma manera que los anteriores, ahora presentamos los resultados de valgrind-cachegrind. La misma situación con *-march=native* explicada anteriormente aplica aquí, por lo que estos resultados tampoco contemplan ese flag. Los resultados son:

- 50 iteraciones-2 mil objetos:
  - 241,714,668 referencias a instrucciones, 0.4% fallos a L1i, 0.0% fallos a LL por instrucciones.
  - 105,505,748 referencias a datos (85,248,738 lecturas, 20,257,010 escrituras), con 1.7% fallos a L1d (1,763,803 lecturas, 6,159 escrituras), 0.0% fallos a LL por datos.
- 100 iteraciones-2 mil objetos:
  - 255,113,844 referencias a instrucciones, 0.3% fallos a L1i, 0.0% fallos a LL por instrucciones.

        - 108,372,038 referencias a datos (87,998,621 lecturas, 20,373,417 escrituras), con 1.6% fallos a L1d (1,763,805 lecturas, 6,159 escrituras), 0.0% fallos a LL por datos.
    - 200 iteraciones-2 mil objetos:
        - 281,912,007 referencias a instrucciones, 0.3% fallos a L1i, 0.0% fallos a LL por instrucciones.
        - 114,104,522 referencias a datos (93,498,310 lecturas, 20,606,212 escrituras), con 1.6% fallos a L1d (1,763,805 escrituras, 6,159 lecturas), 0.0% fallos a LL por datos.

A lo largo de las pruebas, la media de instrucciones por ciclo está un poco por encima de aquella de la versión original, con un valor de ~3.25 (o sea 0.307 CPI).

Acá hay un buen ejemplo de cómo el número de instrucciones ejecutadas incluso en un mismo procesador es una medida engañosa de performance. En valores absolutos SOA ejecuta más de 10 millones de instrucciones en todos los casos, sin embargo SOA tiene un CPI menor al de AOS optimizado (0.307 < 0.33).

De manera similar, SOA accede a muchos más datos que AOS optimizado. AOS fluctúa en las 72-78 millones de referencias a datos mientras que SOA entre los 105-114 millones (así todo, ninguno varía de prueba a prueba tanto como la versión original de AOS). Pero la cantidad de referencias a datos no sirve sin tener en cuenta la tasa de fallos a caché. Cuando se trata de caché L2 y L3 (juntas LL) la tasa es la misma y es 0%. Pero en SOA el problema de escalabilidad desaparece, sin importar el número de iteraciones, cuando el número de objetos es constante, la tasa de fallos es relativamente constante entre las pruebas y es de 1.6%. Esto es porque con 2 mil objetos, cada array de double es de 16 mil bytes (más muy pocos bytes extra mantenidos por std::vector). **16 mil bytes entran perfectamente en la caché L1 de datos de 48 KiB**. Es más, entran 3 de éstos arrays (la asociatividad de 12 no se interpone en el camino). A diferencia de AOS donde quedó claro que un array de 80 bytes * 2 mil = 160 mil bytes no entraría jamás en la caché L1d de nuestro sistema de pruebas.

Otra manera en que SOA es más amigable con la caché, es que **solo hace falta traer datos que sean necesarios para el cómputo**. Por ejemplo; no se trae la velocidad a caché cuando lo que se va a calcular es la fuerza que depende únicamente de la posición y la masa.

● 4.3.AOS vs SOA:

Ya se mostraron AOS optimizado, y SOA. Se habló un poco de cómo difieren en cuanto referencia de datos, referencia de instrucciones, y uso de la caché. Pero la palabra final en performance la tiene el tiempo total de ejecución.

Vamos a comparar los tiempos de ejecución y buscar una diferencia en las medias de SOA y AOS optimizado. Para estas pruebas (9 pruebas, 1 por cada configuración) se usaron las muestras obtenidas previamente para realizar un análisis de varianza (ANOVA) bajo un nivel de significancia $\alpha=0.05$. Se establecieron:

- Hipótesis nula: Total_time_SOA == Total_time_AOS
- Hipótesis alternativa: Total_time_SOA != Total_time_AOS

En caso de rechazar la hipótesis nula, se considera como algoritmo más rápido aquél con menor media en la configuración dada.

1. para num_objects-num_iterations = 1000-50:
    p_valor: 0.366524, no hay diferencia significativa.

2. para num_objects-num_iterations = 1000-100:
   p_valor: 0.973767, no hay diferencia significativa.

3. para num_objects-num_iterations = 1000-200:
   p_valor: 0.001303, si hay diferencia y **SOA** es significativamente más rápido
   media_aos: 0.009799, media_soa: 0.009108
   mejora = media_aos / media_soa = 1.075867

4. para num_objects-num_iterations = 2000-50:
   p_valor: 0.000061, si hay diferencia y **AOS** es significativamente más rápido
   media aos: 0.019157, media soa: 0.020094
   mejora = media_soa / media_aos = 1.048912

5. para num_objects-num_iterations = 2000-100:
   p_valor: 0.858010, no hay diferencia significativa.

6. para num_objects-num_iterations = 2000-200:
   p_valor: 0.078649, no hay diferencia significativa.

7. para num_objects-num_iterations = 4000-50:
   p_valor: 0.000000, sí hay diferencia y **AOS** es significativamente más rápido.
   media_aos: 0.057235, media_soa: 0.062530
   mejora = media_soa / media_aos = 1.092513

8. para num_objects-num_iterations = 4000-100:
   p_valor: 0.000000, si hay diferencia y **AOS** es significativamente más rápido
   media_aos: 0.060119, media_soa: 0.064546
   mejora = media_soa / media_aos = 1.073637

9. para num_objects-num_iterations = 4000-200:
   p_valor: 0.000570, sí hay diferencia y **AOS** es significativamente más rápido.
   media_aos: 0.066366, media_soa: 0.068876
   mejora = media_soa / media_aos = 1.0378206

Podemos notar que no hay grandes diferencias, incluso cuando el resultado de ANOVA indica que sí las hay bajo un nivel de significancia de 0.05. De todas maneras, los resultados estadísticos parecen posicionar AOS por encima de SOA.

NOTA: Por el gran tamaño de muestra se asumieron las condiciones para utilizar ANOVA. También se probaron las mismas hipótesis usando Kruskal-Wallis H-test bajo el mismo nivel de significancia. Las conclusiones de Kruskal-Wallis siempre presentaron un p-valor bajo y beneficiaron a AOS en 6 de las 9 pruebas.

5.PRUEBAS FUNCIONALES REALIZADAS

- PRUEBAS PARA ASEGURAR LA CORRECTA INTERPRETACIÓN DE ARGUMENTOS POR LÍNEA DE COMANDOS:

#	Descripción de la prueba	Salida esperada	Input	Output
1	Todos los parámetros son correctos	Creating simulation: ./sim-soa invoked with 5 parameters. Arguments: num_objects: 1000 num_iterations: 50 random_seed: 666 size_enclosure: 1000000.0 time_step: 0.1 salida 0	./sim-soa 1000 50 666 1000000.0 0.1	Creating simulation: ./sim-soa invoked with 5 parameters. Arguments: num_objects: 1000 num_iterations: 50 random_seed: 666 size_enclosure: 1000000.0 time_step: 0.1 salida 0
2	Nº incorrecto de parámetros	./sim-soa invoked with 1 parameters. Arguments: num_objects: 100 num_iterations: ? random_seed: ? size_enclosure: ? time_step: ? salida -1	./sim-soa 100	./sim-soa invoked with 1 parameters. Arguments: num_objects: 100 num_iterations: ? random_seed: ? size_enclosure: ? time_step: ? salida -1
3	Nº invalido de objetos (tiene que ser entero mayor que 0)	Error: Invalid number of objects ./sim-soa invoked with 5 parameters. Arguments: num_objects: -34 num_iterations: 200 random_seed: 1 size_enclosure: 1000000 time_step: 0.1 salida -2	./sim-soa -34 200 1 1000000 0.1	Error: Invalid number of objects ./sim-soa invoked with 5 parameters. Arguments: num_objects: -34 num_iterations: 200 random_seed: 1 size_enclosure: 1000000 time_step: 0.1 salida -2
4	Nº invalido de iteraciones (tiene que ser entero mayor que 0)	Error: Invalid number of iterations ./sim-soa invoked with 5 parameters. Arguments: num_objects: 100 num_iterations: -4 random_seed: 1 size_enclosure: 1000 time_step: 0.1 salida -2	./sim-soa 100 -4 1 1000 0.1	Error: Invalid number of iterations ./sim-soa invoked with 5 parameters. Arguments: num_objects: 100 num_iterations: -4 random_seed: 1 size_enclosure: 1000 time_step: 0.1 salida -2
5	Semilla aleatoria incorrecta (número entero positivo y genera distribuciones aleatorias).	Error: Invalid random seed ./sim-soa invoked with 5 parameters. Arguments: num_objects: 100 num_iterations: 200 random_seed: -78 size_enclosure: 10000000 time_step: 0.1 salida -2	./sim-soa 100 200 -78 10000000 0.1	Error: Invalid random seed ./sim-soa invoked with 5 parameters. Arguments: num_objects: 100 num_iterations: 200 random_seed: -78 size_enclosure: 10000000 time_step: 0.1 salida -2
6	Tamaño de tablero invalido (es un cubo perfecto con vértice en el origen de coordenadas y con lado el valor size_enclosure)	Error: Invalid size enclosure ./sim-soa invoked with 5 parameters. Arguments: num_objects: 100 num_iterations: 20000 random_seed: 1 size_enclosure: 0 time_step: 0.1 salida -2	./sim-soa 100 20000 1 0 0.1	Error: Invalid size enclosure ./sim-soa invoked with 5 parameters. Arguments: num_objects: 100 num_iterations: 20000 random_seed: 1 size_enclosure: 0 time_step: 0.1 salida -2

7	Tiempo de paso invalido (es un número real positivo)	Error: Invalid delta time ./sim-soa invoked with 5 parameters. Arguments: num_objects: 100 num_iterations: 20000 random_seed: 1 size_enclosure: 1 time_step: -8 salida -2	./sim-soa 100 20000 1 1 -8	Error: Invalid delta time ./sim-soa invoked with 5 parameters. Arguments: num_objects: 100 num_iterations: 20000 random_seed: 1 size_enclosure: 1 time_step: -8 salida -2

- PRUEBAS DE OUTPUT GENERADO (init_config.txt & final_config.txt):

Se compararon, para distintos argumentos de línea de comandos, los archivos init_config.txt y final_config.txt generados contra aquel de los simuladores ejemplo (facilitados como ayuda para la práctica). Si el comando “diff” no mostraba ninguna diferencia entre los archivos, entonces el algoritmo funcionalmente cumple su trabajo.

6.Conclusiones

En esta parte final comentamos conclusiones del código escrito, las optimizaciones aplicadas, y las pruebas realizadas. Ya se han interpretado resultados especialmente en la sección de evaluación de rendimiento (4). Por lo que un análisis más detallado de las conclusiones aquí presentadas podrían encontrarse allí.

Este proyecto se ha centrado en dos estructuras de datos para un algoritmo dado. Una es el struct de arrays (SOA), y otra el array de structs (AOS). El algoritmo refleja la localidad espacial de las instrucciones, siendo que la caché de instrucciones L1 en las pruebas tuvo una tasa de fallos que osciló entre 0.00% y 0.04%.

Las optimizaciones aplicadas a AOS sirvieron para demostrar cómo influye la indirección en los programas. Donde un algoritmo básicamente igual, tuvo un tiempo de ejecución 10 veces menor en la mayoría de los casos, solo por el cambio en la estructura de datos y los punteros involucrados.

Sea la versión optimizada o no, cuando de AOS se trata, no se necesitan muchos objetos para que la caché de datos L1 sea incapaz de contener el array. Las versiones de AOS tienen un constante incremento en tasa de fallos de caché a medida que la configuración aumenta en número de iteraciones. Al volver a comenzar una iteración, vuelven a ocurrir los fallos, esto pasa tanto en simulate() como en collision_check().

Otra cosa que sucede en AOS y no beneficia a este algoritmo en particular, es que se traen datos a caché que no siempre son necesarios. Por ejemplo, si se quiere calcular la fuerza resultante de un objeto “a” contra todos los objetos restantes, al traer la masa, posición y fuerza a caché de un objeto “b”, también se ingresan 8 bytes de la velocidad en x.

SOA no presenta algunos de estos problemas. Al tratarse de arrays individuales, solamente se traen a caché los datos necesarios. En la sección 4 tomamos el ejemplo con 2 mil objetos, donde cada array sería de 8 bytes * 2 mil = 16 mil bytes, o sea, entra sin problemas en la caché L1 de 48 KiB. A diferencia de 2 mil * 80 bytes = 160 mil bytes en el array de structs de AOS. Estas 2 observaciones se traducen en una tasa de fallos reducida, y constante para un número fijo de objetos. El número de iteraciones parece no afectar a SOA en los tamaños probados.

Al comparar SOA y AOS tuvimos un ejemplo práctico de cómo el número de instrucciones y datos referenciados pueden ser engañosos para medir performance si no se tienen en cuenta otros datos relevantes como CPI, tiempo de ciclo, y tasa de fallos a caché. SOA referencia unas cuantas millones más de instrucciones y datos, sin embargo su CPI es más pequeño (mejor) y su fallo a caché es varias veces menor.

Yendo a la comparación interesante entre SOA y AOS: el tiempo de ejecución. Usamos un análisis de varianza bajo un nivel de significancia de 0.05 para cada una de las 9 pruebas llevadas a cabo. 4 favorecieron a AOS, 1 a SOA, y el resto no mostraron diferencias significativas. En los casos que sí mostraron diferencias, estas no fueron de gran factor. Posiblemente se deban a los tamaños de muestra y número de iteraciones que manejamos. Porque si la tendencia a subir la tasa de fallos de caché de AOS continúa avanzando al subir los valores de esos parámetros, eventualmente SOA podría posicionarse por encima de AOS. Por otro lado, la mayor parte del tiempo de desarrollo la concentración fue en el algoritmo para AOS. SOA fue escrito simplemente modificando el código final de AOS. Por lo que optimizaciones en el código específicas para SOA pueden no haber sido contempladas. Así todo, SOA dio, según consideramos, buenos resultados.