1. Introducción

En este trabajo tuvimos que optimizar dos diferentes procesos, uno consistente en distorsionar una imagen dada y otro que generaba sombras dada una imagen 3D y diferentes fuentes de luz. Para lograr esto, se nos pedía implementar algunas funciones codeadas en C en assembler, con el objetivo de utilizar los registros SSE para hacer más rápido el procesamiento de los datos.

Dada esta consigna, tratamos siempre desde el principio de hacer el código con la idea de que éste fuera lo más veloz posible. Es por esto que tratamos reducir los saltos al mínimo posible, reducir la cantidad de accesos a memoria, y de procesar la mayor cantidad de elementos al mismo tiempo (o sea, en paralelo) posible.

Primero tratamos de implementar el distort, lo cual nos dio una mejor idea de cómo encarar este TP en sí, es por esto que la primera función nos facilitó mucho la implementación del shadow en assembler, ya que pudimos efectuar un análisis previo mejor sobre cómo diseñar la función (y por ende, esta requirió mucha menos optimización ya que una vez andando, su desempeño ya era el esperado)

Nos encontramos con varias dificultades a lo largo de este TP, no tanto con la optimización de las funciones ya que rápidamente notamos que haciéndolo en lenguaje ensamblador ya significaba cierta optimización por sobre el código en C; sino más con el hecho mismo de poder codear estas funciones en assembler. Haciendo esto nos topamos con varios errores de código, por ejemplo los enumerados a continuacuón

- Problemas surgidos al levantar en un registro SSE un valor en int sin antes haberlo convertido a float y operar con éste de esa manera
- De la misma manera, bajar un número en float a un registro común y tratar de operar normalmente con éste
- No tener en cuenta que instrucciones como movd o movss nos borraban el contenido del registro, cuando nosotros pensábamos que simplemente movía al primer float y dejaba intacto el resto del registro
- Desconocimiento de funciones que provocan mayor optimización en el código que eviten procesos que insumirían más tiempo (cmpss en lugar de jumps, por ejemplo)

2. Distort

2.1. Descripción

Esta función consistía en aplicar un efecto “rubber” sobre una imagen provista por la cátedra. Para optimizar esta función se nos pidió que programáramos en assembler dos funciones, rubberdynamics y updateStates, las cuales se aplican en sucesión para general el efecto de distorsión de la imagen.

2.2. Decisiones tomadas al codear

Al ver el código, rápidamente notamos que la estructura del mass que contenía la información de la imagen en una matriz no era la óptima para el trabajo con SSE, ya que tanto los vectores x como los v eran de 3 floats; sabiendo que un registro SSE tiene capacidad para 4 floats, nos dimos cuenta que si lo dejábamos así, cada vez que levantáramos una posición de mass para ver el vector x levantaríamos la primer componente del vector v. Dado que esto podría causar una sobreescritura de dicha componente cuando no fuera necesaria, y aún teniendo el cuidado de no sobreescribirla podríamos acabar en operaciones redundantes para evitar que dicho valor se pierda. Es por esto que simplemente modificamos la estructura del mass para agregarle un float más a los vectores x y v, este float no cuenta con otra función que no sea la de ocupar espacio, en ningún momento lo usamos para el cálculo durante las funciones.

Al empezar a programar las funciones, notamos que algunos parámetros que estaban definidos en c como globales no podían ser accedidos en assembler. Por esto, decidimos que era mejor cambiar la definición de las funciones, para que éstas tomen como parámetro más valores, como ser, el puntero a mass y el grab en el caso del updateStates y estos dos más el puntero al spring y el tamaño del spring mismo para rubberdynamics; aún cuando esto ralentizara un poco la ejecución, nos pareció más importante que el programa se pudiera ejecutar correctamente.

Otro punto a tener en cuenta antes de programar las funciones fue cómo manejaríamos las múltiples constantes que son usadas a lo largo de las funciones. Una de las cosas que notamos de entrada fue que si bien se definían las constantes CLIPNEAR y CLIPFAR, éstas nunca se usaban con ese valor sino que eran modificadas por otras constantes. Por ello, nos pareció mejor mirar bien el código en C y en vez de definir estas constantes tal como lo hace la función de C, definir las constantes ya operadas (o sea, $-clipnear + 0{,}01$, $-(clipnear - clipfar)/4$, etc) de antemano y simplemente usar esas mismas cada vez que debemos operar con ellas. Esto nos evita cálculos innecesarios, ya que estos datos son constantes, y contribuye a la optimización

2.3. Optimización

Una vez que conseguimos el efecto deseado (o sea, un efecto idéntico al de la función en C), nos dedicamos a optimizar lo más posible la función, recalcando que la optimización más que nada fue hecha al codear ya que nos aseguramos que el código mismo estuviera optimizado desde un principio. Una de las cosas que notamos fue que las posiciones i y j del spring no eran aleatorias, es más, seguían un parámetro muy claro. Luego de cierto debuggeo, concluimos que durante las primeras 209 posiciones, dada cualquier posicion x (x $<$ 209) del spring, $spring[x].j = spring[x].i + 1$, y más aún, dado $y = x + 1$, $spring[x].j = spring[y].i$

Con esto en mente, podemos evitar el acceso a memoria para verificar el valor de las constantes del spring para esa posicion del ciclo durante 209 iteraciones del mismo, simplemente nos alcanzaba un acceso al principio para obtener el primer i.

Otro ítem que tuvimos en cuenta para optimizar fue las comparaciones que se llevan a cabo en la función updatestates, en la versión en C de la función se ve que hay dos if que actualizan el valor $x[2]$ del mass si este pasa de dos límites. Al pasar esto a assembler, nos dimos cuenta que si aplicáramos estos if de manera similar con jumps, perderíamos demasiado tiempo tirando por la borda la optimización. Es por esto que descartamos cualquier posible jump en esa parte del código y lo hicimos por intermedio de máscaras, en las cuales comparamos dos veces el dato con la instrucción cmpss, una vez por menor y la otra por no menor (en la segunda compación, menor igual y no menor igual), con lo cual nos aseguramos que una de los dos resultados sea todo 0 y el otro todo 1; hecho esto, sólo basta hacer ands con los resultados que se deberían colocar según la comparación y sumarlos, sabiendo que uno de los dos es necesariamente 0. Esto nos permite hacer esas dos comparaciones sin pasar por ningún salto que ralentice el código.

2.4. Resultados

Para calcular la cantidad de ciclos que demora una función, utilizamos las líneas de código provistas por la cátedra, con lo cual leemos el registro tsc tanto antes de entrar a la función como al salir, y restamos esos valores para obtener cuánto tardó la función. Para asegurarnos una representación adecuada del verdadero tiempo que insume la función, decidimos tomar 30 muestras de la cantidad de ciclos, acumulándolos en una variable, y con otra tomar la cantidad de medidas. Una vez que ésta llega a 30, dividimos el acumulador por este número, e imprimimos por pantalla la cantidad de ciclos promedio insumidos.

Para confeccionar el gráfico tratamos de igualar los movimientos que hacíamos con el mouse en C con los que hacíamos en asm, de manera de que el cálculo que el procesador fuera a hacer fuera lo más similar posible en cuanto a las diferentes variables involucradas. Sin embargo, se puede observar que salvo algunas mediciones puntuales, los ciclos se mantienen más o menos constantes para un lenguaje, por lo tanto se puede concluir que el tipo de distorsionamento aplicado sobre la imagen no tiene una influencia primordial sobre la cantidad de ciclos que se emplean para aplicar el efecto

Los resultados obtenidos se pueden ver en la tabla provista correspondiente a la función distort, se puede observar que la disminución de ciclos cuando se utiliza assembler es evidente, ya que esta cantidad consistentemente menos de la mitad de la insumida por C. Es indudable que podemos atribuir esta mejora en ciclos a las optimizaciones que hicimos sobre el código, que evidentemente reducen los accesos a memoria y las operaciones hechas; lo que nos lleva a bajar la cantidad de ciclos de alrededor de 130.000 a la mucha menor cantidad que ronda los 55.000

3. Shadow

3.1. Descripción

La función shadow se encarga de generar la sombra de un objeto 3D, dados diferentes haces de luz. Durante la ejecución del programa se pueden cambiar la intensidad de los haces, la cantidad de los mismos, se puede alterar el ángulo de visión, etc; todos esos cambios deben ser reflejados adecuadamente en la pantalla por la función samplelights

3.2. Decisiones tomadas al codear

El mayor desafío de esta función fue descubrir cómo implementar los diferentes cálculos que se le efectúan a los diferentes vectores de una forma que significara cierta optimización por sobre el código en c. Esto mayormente puede notarse en la implementación del shadowmatrix y también en las funciones auxiliares cross y normalize. Para el caso del shadowmatrix, fue evidente desde un principio que no podíamos hacer las cuentas de la manera que lo hace la función en C, ya que esta altera las posiciones de la matriz por columnas, y nosotros procesamos de a filas. Por lo tanto, fue necesario determinar bien de entrada cómo nos aseguraríamos de que los registros SSE tuvieran los datos que necesitábamos con la menor cantidad de accesos a memoria posible. Esto se puede notar en los shifts que le hacemos a xmm3 que nos aseguran que el dot cargado pase a la posición que nos conviene sin necesidad de leerlo nuevamente y con la posibilidad de hacer un procesamiento en paralelo ya que calculamos los valores de la fila todos al mismo tiempo

Para el samplelights nos encontramos no tanto con la dificultad de procesar en paralelo, lo cual no era demasiado difícil, sino más con el hecho de que la función misma nos exigía usar todos o casi todos los registros disponibles para contener la información. Esto se ve claramente durante el ciclo, donde prácticamente usamos todos los registros SSE con tal de evitar la mayor cantidad de accesos a memoria posible.

En esta función surgieron problemas que fue complicado resolver más que nada por el hecho de que no era fácil localizarlos. El más claro de todos fue un problema que tuvimos con el redondeo de la cantidad de samples pasados por parámetro, nosotros pedíamos un malloc para el tamaño de lights igual a la cantidad de samples (multiplicado por 16), pero el ciclo se ejecutaba una cantidad de veces igual al cuadrado del redondeo de la raíz cuadrada de los samples, esto no causa problemas salvo cuando la raíz se redondea para arriba al convertir de float a int; para solucionar esto tuvimos que cambiar el tipo de conversión efectuada y elegir una instruccion (cvttss2si) que truncara el valor para asegurarnos que siempre éste resultara menor o igual a la cantidad de memoria pedida

3.3. Optimización

Una de las optimizaciones llevadas a cabo fue cambiar la manera de organizar los tres vectores locales de la función samplelights (u,v,n), al principio la función llamaba tres veces a malloc, cosa que la función en C no hacía, nos dimos cuenta que esto era innecesario y procedimos a cambiar esta implementación por una más eficiente; finalmente, decidimos utilizar la pila al comenzar la función para pedir 48 bytes como variables locales, así nos evitábamos las llamadas a malloc que no eran indispensables y a su vez nos liberábamos de la necesidad de liberar esta memoria pedida una vez terminada la función.

Para evaluar el rendimiento de esta función, nos pareció necesario, dada la diferente naturaleza de la misma, encarar el cálculo de los ciclos insumidos de diferente manera a como encaramos el distort. Si bien mantuvimos la idea de hacer 30 muestras y obtener el promedio, nos pareció que para la función samplelight, dado que sólo se llama una vez ante cada cambio en el sistema a moldear (incrementar los haces, el tamaño de la luz, etc), no correspondía simplemente esperar a que esta se ejecutara 30 veces. Es por eso que, aprovechando el diseño de la función, utilizamos el case para agregar un nuevo parámetro ('e'), que efectúa el testeo del rendimiento de samplelights por sí solo, con un for al cual se entra 30 veces. Dado que la función shadowmatrix se ejecuta varias veces a lo largo de cada entrada a un caso, no nos pareció tan necesario poner un caso especial de testeo para ésta, y simplemente mantuvimos un contador que devuelve el promedio una vez que esta se ejecutó 30 veces.

3.4. Resultados

Una de las primeras conclusiones que se pueden tomar con sólo observar los resultados es que en shadowmatrix no se observa demasiada optimización. Esto es fácilmente explicable por el hecho de que lo único que la función hace es crear una matriz, no posee ciclos ni depende del tamaño de la entrada. Esto explica tanto que la mejora en assembler no sea tan notable como que la cantidad de ciclos que la misma insume no dependa de los samples que se tengan.

Todo lo contrario se puede ver en la función samplelights, donde la diferencia entre los lenguajes es notable y crece aún más mientras más samples se tengan. Para observar mejor los resultados tomamos cinco sets de muestras, cada una con una cantidad de samples diferentes. Se observa que la cantidad de ciclos en C aumenta exponencialmente,

mientras que en asm el crecimiento es mucho menos aparente (se puede asumir que el crecimiento es bastante cercano a una función lineal). Dado un programa profesional que debiera calcular samples para muchos más samples que los pocos que calculamos en este caso, la diferencia en ciclos insumidos es abismal y rápidamente deja en claro cuál es el lenguaje que conviene usar.

La diferencia en esta función es inmediatamente atribuible al doble ciclo, que debe ser recorrido n^2 veces, y sobre el cual en asm usamos el procesamiento en paralelo de los datos para asegurar la menor cantidad de accesos a memoria posible, toda esta optimización que no se lleva a cabo en C permite una mayor velocidad por entrada en el ciclo de la función, cuando la cantidad de samples aumenta, la cantidad de veces que se entra al ciclo aumenta exponencialmente, de ahí la diferencia en tiempo insumido entre los dos lenguajes.

4. Recursos utilizados

Durante este TP utilizamos diversos procesadores, teniendo en cuenta la imposibilidad de compilar en el laboratorio dado que no estaban las librerías instaladas, lo cual dificultó nuestro trabajo. En primer lugar, contamos con un procesador Intel Atom que nos demostró que con procesadores menos poderosos las diferencias de ejecución no son tan notorias, probablemente debido al hecho de que la implementación de SSE en un Atom no es la mejor (este procesador cuenta con un sistema operativo Ubuntu 10.04 de 32 bits). También pudimos verificar los datos en un procesador Core 2 Duo (con Debian en el laboratorio, Ubuntu 10.04 en nuestra computadora personal) donde vimos diferencias más sustanciales en los resultados obtenidos, que fue de donde sacamos los resultados definitivos que detallamos más arriba

5. Conclusiones

Después de este TP, podemos concluir que, dado un procesador adecuado, una implementación adecuada en lenguaje ensamblador es mucho más eficiente e insume menos ciclos de clock que un desarrollo simple en C, es innegable que si programáramos todo en assembler sin tener en cuenta tanto las ventajas del mismo (procesamiento en paralelo en SSE, acceso a registros por sobre memoria, evitar el uso de saltos innecesarios, usar las operaciones más simples que hacen lo que necesitamos (shift en lugar de shuffle cuando sólo necesitamos mover un valor a otra posición dentro del mismo registro); no obtendríamos mejores resultados, es más, es muy probable que estos fueran peores que en C. Pero con un cuidado en el desarrollo y atención a las bondades del lenguaje, se puede obtener una optimización del código que es bastante notoria, cuyo valor depende completamente de la función pero que en nuestros casos estuvo siempre demostró una mayor velocidad en el lenguaje ensamblador, llegando incluso hasta ser seis veces más rápido en casos extremos.