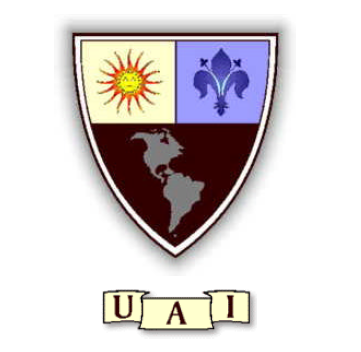
UNIVERSIDAD ABIERTA INTERAMERICANA



FACULTAD DE TECNOLOGÍA INFORMÁTICA

CARRERA: **Analista Programador**

ALUMNO: **Tordoya, Gerardo Rodolfo**

MATERIA: **Sistemas de Computación 2**

INFORME FINAL: **Ingeniería de Rendimiento**

AÑO: **2021**

Tabla de Contenido

[Introducción 2](#_Toc77980981)

[Desarrollo 2](#_Toc77980982)

[Conclusión 2](#_Toc77980983)

[Referencias 2](#_Toc77980984)

# Introducción

Desde los micro códigos del procesador de 4 bits Intel 4004 de 1971 se ha recorrido un largo camino hasta las modernas instrucciones intrínsecas, y una de las pocas formas de abordarlo, es el estudio de rendimiento (¿o qué otro tópico lo justificaría?).

Debido a que uno de los parámetros de este informe final es la brevedad, no podremos (ni por cerca) hacer un estudio exhaustivo de la temática (la cual podría llenar tomos). Y por eso mismo, este trabajo incluye láminas que son puestas en un documento anexo pues restarían aún más el poco espacio con el que se cuenta.

El rendimiento no suele ser lo más importante para los programadores. ¿Cuáles son algunas de las cosas que se consideran más importantes que el rendimiento? Plazos. Costo. Exactitud. Extensibilidad. Entonces, si los programadores están tan dispuestos a sacrificar el rendimiento por estas propiedades, ¿por qué estudiamos el rendimiento? Esto es una paradoja y un enigma. ¿Por qué estudiar algo que claramente no es lo primero en la lista de los programadores?

Creo que la respuesta es que el rendimiento es la moneda de la informática. Utiliza el rendimiento para comprar otras propiedades. Se paga por algo como "quiero algo que sea fácil programar y, por lo tanto, estoy dispuesto a sacrificar algo de rendimiento", o "estoy dispuesto a sacrificar algo de rendimiento para asegurarme de que mi sistema sea seguro".

Todas esas cosas salen de ese presupuesto del rendimiento. Y claramente, si el rendimiento se degrada demasiado, esas cosas se vuelven inutilizables. Cuando uno habla con los programadores, a menudo se oye: "¿Hacer performance? El rendimiento no importa, nunca pienso en eso". Pero luego uno habla con personas que usan computadoras y ante la pregunta "¿Cuál es su principal queja sobre los sistemas informáticos que usa?", su respuesta invariablemente es "Demasiado lenta". La verdadera respuesta es que el desempeño es como la moneda: Es algo que se gasta.

¿Prefiero tener U$D 100 o una botella de agua? El agua es indispensable para la vida. Existen circunstancias en las que uno preferiría tener el agua a U$D 100. Pero en nuestra sociedad se puede comprar agua por mucho menos que U$D 100. Entonces, aunque el agua es esencial para la vida y mucho más importante que el dinero, uno prefiere el dinero porque puede comprar lo que se necesita. Ésa es la analogía del desempeño. No tiene valor intrínseco, pero contribuye. Puedo usarse para comprar aquello que interesa: usabilidad, capacidad de prueba, lo que sea.

En los primeros días de la informática, la ingeniería del rendimiento del software era común porque los recursos de las máquinas eran limitados. Si examinamos computadoras de 1964 a 1977 (**lámina 4**), ¿cuántos bytes tienen? En el 64, hay una computadora con 524 kilobytes. En ese entonces, esa era una gran máquina. Kilobytes, no megabytes, no gigabytes. Kilobytes. Su frecuencia de reloj era de 33 kilohercios. La frecuencia de reloj típica de hoy es 4 gigahercios.

En esta era se operaba con kilohercios. Varios programas concurrentes no encajarían sin una intensa ingeniería de rendimiento. Y sin embargo, en esa época hubo muchas opiniones en contra porque, cuando se intenta acelerar el rendimiento, el código se vuelve ilegible. Y en hacer código que sea legible y rápido radica el arte. Durante años no hubo un esfuerzo real en ingeniería de rendimiento.

En la **lámina 6** se observa el escalado de la tecnología y cuántos transistores hay en varios diseños de procesadores. Hasta el 2004, la ley de Moore se cumplía a rajatabla: las densidades de chips que se duplicaban cada dos años. Y a medida que se reducían las dimensiones de los chips, la velocidad del reloj también aumentaba como consecuencia (no abordaremos ese tema aquí), cuestión que fue abordada en la escala de Dennard (**lámina 7**).

Hasta 2004, la ley de Moore y la escala de frecuencia de reloj (la llamada escala de Dennard) fueron esencialmente la moneda de rendimiento. Vemos en la **lámina 10** que todo eso llegó a su fin en 2004 cuando la velocidad del reloj se estancó. Se puede observar, a partir de 2005, que las velocidades logradas (de 2 a 4 gigahercios) llegaron a una meseta y que a partir de entonces no se consiguió chips más rápidos. La razón por la que la velocidad del reloj se aplanó fue por la densidad de potencia.

La **lámina 11** es una diapositiva de Intel de esa época que analiza el crecimiento de la densidad de potencia. Ahí se puede ver una proyección de las temperaturas de unión de los transistores en el chip: si seguían escalando de la forma que lo habían estado haciendo, comenzarían a acercarse, en primer lugar, a la temperatura de un reactor nuclear, luego a la temperatura de una boquilla de cohete, y luego a la de la superficie del sol. Concluyeron que no había manera de construir una micro tecnología que enfríe eso, dado el caso.

Incluso resolviendo esto, por otro lado, ya no se podían escalar las frecuencias de reloj. ¿Por qué? Porque, originalmente, la frecuencia del reloj se escaló asumiendo que la mayor parte de la potencia era potencia dinámica funcionando cuando se hacía SWITCH en un circuito. Y lo que sucedió es que, mientras se seguía miniaturizando, algo que solía estar en el ruido (las corrientes de fuga) comenzaron a volverse significativas hasta el punto que hoy la potencia dinámica es mucho menos preocupante que la energía estática del circuito que está allí, con fugas. Cuando se miniaturiza, no puede evitarse este efecto.

Entonces, ¿qué hicieron los proveedores hacia el 2005? Ya no podían conseguir más velocidad de los transistores. Lo que hicieron fue introducir el paralelismo en forma de procesadores multinúcleo. Y a partir de entonces, para escalar el rendimiento, usaron múltiples núcleos. Así, cada generación de la ley de Moore, ahora duplicaba el número de núcleos.

El impacto de esto fue que el rendimiento dejó de ser gratuito. No se puede simplemente acelerar el hardware. Por otro lado, si se quiere usar este nuevo potencial, hay que hacer programación paralela (algo que, en realidad, nadie en la industria ha hecho de verdad). No solo eso: en ese tiempo intermedio se agregaron unidades vectoriales, GPU, jerarquías de caché más pronunciadas; incluso en algunas computadoras: lógica configurable, etcétera.

Ahora es el turno del software de adaptarse y ponerse al día. Y es por eso que, aunque no queremos lidiar con el rendimiento, tenemos que lidiar con el rendimiento. Y a lo largo de su vida, usted tendrá que lidiar con el rendimiento del software si quiere producir software eficaz. Tan simple como eso.

# Desarrollo

Los procesadores modernos se han vuelto realmente muy complicados, y la gran pregunta es: ¿Cómo escribimos software para que use ese hardware moderno de manera eficiente? Veamos un ejemplo de ingeniería del rendimiento de un problema muy bien estudiado, a saber, la multiplicación de matrices.

En una multiplicación de matrices, si se suma el número total de operaciones, obtenemos un aproximado de 2n al cubo (porque hay una multiplicación y una suma por cada par de términos que deben acumularse). Entonces, es básicamente 2n al cubo. Lo veremos asumiendo (por simplicidad) que nuestra n es una potencia exacta de 2.

La máquina sobre la que conduciremos nuestras pruebas la podemos encontrar en AWS (Amazon Web Services, **lámina 20**). Es una máquina optimizada para computación, que tiene una micro arquitectura Haswell que se ejecuta a 2,9 gigahercios, con 2 chips de procesador y 9 núcleos de procesamiento por chip (es decir, un total de 18 núcleos). Realiza hyperthreading bidireccional (pero no tendremos en cuenta este aspecto porque el hyperthreading, aunque brinda un poco más de rendimiento, también hace difíciles las mediciones, por lo que lo desactivaremos). La unidad de punto flotante es capaz de realizar 8 operaciones de precisión de doble precisión. Eso es: operaciones de punto flotante de 64 bits, incluida una combinación de multiplicar y agregar (por núcleo, por ciclo). Es una unidad vectorial. Cada uno de estos 18 núcleos pueden realizar 8 operaciones de doble precisión, incluida fusión-multiplicación-suma (que, en la realidad, son 2 operaciones). Tiene un tamaño de línea de caché de 64 bytes. El iCache es de 32 kilobytes, que es asociativo de 8 vías. Tiene una dCache del mismo tamaño. Tiene una caché L2 de 256 kilobytes. Tiene una caché L3 (o lo que a veces se llama LLC, caché de último nivel) de 25 megabytes. Y 60 gigabytes de DRAM. Es decir, es una computadora muy potente.

El cálculo de rendimiento máximo es: velocidad del reloj multiplicada por 2 chips de procesador por 9 núcleos de procesamiento por chip, cada uno capaz de 16 operaciones de punto flotante. Total: 836 gigaflops. Mucha, pero mucha potencia.

En la **lámina 21** vemos el código en Python para realizar la multiplicación de matrices. Cuando ejecutemos este código, antes y después del bucle triplemente anidado tomamos una medición de tiempo para ver la diferencia. Se necesitaron unos 21.000 segundos, es decir, unas 6 horas. ¿Es esto rápido? ¿No? ¿Cómo podemos saber si esto es rápido o no? ¿Qué debemos esperar de nuestra máquina? Hagamos un cálculo inicial de cuántas operaciones hay que hacer y qué tan rápido deberíamos poder hacerlo.

Hay 2n operaciones al cubo que deben realizarse. Eso es 2n a la 37 operaciones de punto flotante. El tiempo de ejecución es de 21.000 segundos, por lo que estamos obteniendo aproximadamente 6.25 megaflops de nuestra máquina cuando ejecutamos ese código. El pico, como recordará, es de unos 836 gigaflops. Entonces estamos obteniendo aproximadamente un 0,00075% del pico. Esto no es rápido. Codifiquémoslo en Java en lugar de Python.

Ejecutamos ese bucle triplemente anidado en Java (**lámina 23**) y obtenemos que el tiempo de ejecución ahora es de poco menos de 3.000 segundos, lo que equivale a unos 46 minutos total. El mismo código. Python, Java. Conseguimos una aceleración de casi 9 veces simplemente codificándolo en un lenguaje diferente. Bien, probemos C (**lámina 24**). ¿Qué sucede cuando se codifica en C? Mismo código. Ahora son 1.100 segundos base, que son aproximadamente 19 minutos. 2 veces más rápido que Java y aproximadamente 18 veces más rápido que Python.

Así es como vamos a evaluar nuestro progreso (**lámina 25**). Aquí es donde nos encontramos ahora. Tenemos el tiempo de ejecución. La aceleración relativa es *cuánto más rápido es* que la fila anterior. La aceleración absoluta es cómo se compara con la primera fila. Hasta ahora, hemos logrado obtener un 0.014% del pico. Aún somos lentos, debemos seguir optimizando.

LÁMINA 26

Es decir, tenemos todas estas etapas generales versus simplemente hacer operaciones.

LÁMINA 27

Es decir, los intérpretes pueden admitir fácilmente funciones de programación de alto nivel y pueden hacer cosas como la alteración de código dinámico, etcétera, a costa del rendimiento.

LÁMINA 28

C es desde donde nos moveremos desde este punto porque es lo más rápido que obtenemos. Ahora bien, podemos cambiar el orden de los bucles sin afectar la corrección. Así que aquí actualizaremos este código.

LÁMINA 29

Ésta es otra manera que podríamos plantear para i, para k, y para j, a fin de hacer nuestra actualización para que calcule exactamente lo mismo. O podríamos rotar k por j y por i. Podemos cambiar el orden sin afectar la corrección. Entonces, ¿el orden de los bucles es importante para el rendimiento? Ésta es la pregunta clave.

LÁMINA 30

Nos encontramos con que, cuando hacemos eso, obtenemos que el orden del ciclo afecta el tiempo de ejecución por un factor de 18. Simplemente cambiando el orden. ¿Qué está pasando? Se debe a la localidad de caché.

LÁMINA 31

En el hardware, cada procesador lee y escribe la memoria principal en bloques contiguos llamados líneas de caché. Las líneas de caché a las que se accedió anteriormente se almacenan en una pequeña memoria llamada caché que se encuentra cerca del procesador.

LÁMINA 32

Cuando el procesador accede a algo, si está en la caché, obtiene un acierto. Eso es rápido. Si falla, debe ir a un caché de nivel más profundo o hasta la memoria principal. Eso es mucho, mucho más lento. Entonces, lo que sucede con este problema de matrices es que las matrices se colocan en la memoria en orden de fila mayor. Recuerde: usted tiene una matriz bidimensional. Se presenta en el orden lineal de las direcciones de la memoria básicamente tomando la fila 1, y luego, después de la fila 1, se añade la fila 2, y luego se añade la fila 3, y así sucesivamente, desdoblándola.

LÁMINA 33

Hay otro orden en el que se podrían haber dispuesto las cosas (de hecho, así está en Fortran) que se llama orden de columna mayor. Resulta que C y Fortran operan en diferentes órdenes. Y resulta que de la manera que lo hacen, afecta al rendimiento. Así que echemos un vistazo al patrón de acceso para el orden i, j, k.

Para k, obtenemos una excelente localidad espacial porque simplemente estamos accediendo a la misma ubicación. En cada ciclo, estará en caché. Siempre estará ahí. Será rápido acceder a C. Para A, lo que sucede es que atravesamos en un orden lineal y obtenemos una buena localidad espacial. Pero para B, que atraviesa columnas, esos puntos se distribuyen muy lejos en la memoria, por lo que el procesador necesitará 64 bytes para operar en un dato en particular. Y luego ignora 7 de los 8 elementos de punto flotante en esa línea de caché y pasa a la siguiente. Se desperdicia muchísimo. A tiene una buena ubicación espacial en el sentido de que todo es adyacente y usa las líneas de caché de manera efectiva. B tiene 4096 elementos de distancia, es decir, tiene una localidad espacial pobre.

LÁMINA 34

Si evaluamos otro orden (i, k, j) resulta que obtenemos una buena localidad espacial tanto para C como para B y excelente para A.

LÁMINA 35

Y evaluando el orden restante, vemos que este lo está haciendo muy mal.

LÁMINA 36

Se pueden medir los diferentes órdenes con una herramienta como Valgrind para resolver esto.

LÁMINA 37

Ya habiendo seleccionado el mejor orden, obtenemos una aceleración relativa de aproximadamente 6 .50 con respecto a la medición anterior. ¿Qué otros cambios podemos probar? A partir de aquí vamos a ver una serie de optimizaciones que podríamos hacer sin siquiera tocar el código.

LÁMINA 38

Podemos cambiar las banderas del compilador. Clang, que es un compilador que usaremos para esto, proporciona una colección de conmutadores de optimización, y puede especificar un conmutador al compilador para pedirle que optimice. Consultando su documentación vemos que podemos hacer evaluaciones tipo O0 "No optimizar". O1 "Optimizar". O2 "Optimice más". O3 "Optimizar aún más".

En nuestro caso, a pesar de que se optimizó hasta O3, O2 termina siendo una configuración mejor. Esto no sucede todo el tiempo. Por lo general, O3 funciona mejor que O2, pero en este caso O2 en realidad ha optimizado mejor que O3 porque las optimizaciones son hasta cierto punto heurísticas. En Clang, también hay otros tipos de optimización. Por ejemplo, optimización guiada por perfiles, en la que se observa cuál fue el rendimiento y lo retroalimenta en el código (para que luego el compilador pueda ser más inteligente sobre cómo se optimiza).

LÁMINA 39

Con esta tecnología simple, eligiendo una buena bandera de optimización, que en este caso fue O2, obtuvimos un factor de mejora del 3.25, sin tener que hacer mucho. Pero nos estamos acercando ni al 1% del rendimiento máximo. Tenemos, hasta el momento, un 0,3% de rendimiento máximo. ¿Qué está aún causando el bajo rendimiento? ¿Por qué no estamos obteniendo un mayor rendimiento?

LÁMINA 40

No estamos usando todos los núcleos. Hasta ahora estamos usando un solo núcleo, ¿y cuántos núcleos tenemos? 18 núcleos, 17 inactivos, mientras intentamos optimizar tan solo uno. Tenemos 9 núcleos por chip, y hay 2 de estos chips en nuestra máquina de prueba. Estamos ejecutando solo uno de ellos, así que usémoslos todos.

LÁMINA 41

Para hacer eso, usaremos la infraestructura de Cilk y, en particular, podemos usar lo que se llama un bucle paralelo, que en Cilk, se llama cilk\_for, que retransmitirá todas esas iteraciones en paralelo.

LÁMINA 42

El compilador y el sistema de ejecución son libres de programarlos (agendarlos, en la jerga). También podríamos hacerlo para el bucle interno. Entonces, a partir de este momento, la pregunta es, ¿qué versión paralela funciona mejor?

Podemos poner en paralelo el ciclo i, podemos hacer el ciclo j en paralelo, y podemos hacer paralelos i y j juntos. Paralelizar solo k no resultará en una mejoría (no tocaremos este tema ahora). Entonces, si se mira, ¡qué variedad de tiempos de ejecución! Si paralelizo solo el ciclo i, son 3,18 segundos, y si paralelizo el ciclo j, se ralentiza. Y luego, si hago tanto i como j, sigue siendo malo. Así que nos quedaremos con el bucle exterior paralelizado. Resulta que esto tiene que ver con la sobrecarga de programación (tema que tampoco tocaremos ahora). La regla general aquí es paralelizar los bucles externos en lugar de los bucles internos.

LÁMINA 43

Cuando paralelizamos bucles, obtenemos una aceleración de casi 18 veces (por los 18 núcleos). Sin embargo, déjeme asegurarle que no todo código es tan fácil de paralelizar.

Ahora estamos aproximadamente a un poco más del 5% del pico.¿Dónde estamos perdiendo aquí? ¿Por qué obtenemos solo el 5%?

LÁMINA 44

Podemos administrar mejor las fallas de caché. Así que volvamos a las cachés de hardware y reestructuremos el cálculo para reutilizar los datos en la caché tanto como sea posible. Porque los errores de caché son lentos y los aciertos son rápidos. Tratemos de aprovechar al máximo la caché reutilizando los datos que ya están allí. Así que echemos un vistazo.

LÁMINA 45

Suponga que vamos a calcular solo una fila de C. Así que veamos una fila de C. Eso nos llevará, ya que hay un vector de 4096 de largo allí, a que básicamente serán 4096 escrituras que vamos a hacer. Y vamos a conseguir algo de localidad espacial allí, lo cual es bueno, pero básicamente lo que el procesador está haciendo son 4.096 escrituras. Ahora, para calcular esa fila, necesito acceder a 4096 lecturas de A. Y necesito todo B.

Debido a que voy a cada columna de B a medida que avanzo para calcular completamente C, necesito, para calcular solo una fila de C, acceder a una fila de A y a todas las de B. Porque el primer elemento de C necesita toda la primera columna de B. Y el segundo elemento de C necesita toda la segunda columna de B.

Pero lo principal que hay que entender es que se está accediendo a todo B. Luego, para calcular otra fila de C, voy a hacer lo mismo. Voy a pasar por una fila de A y de nuevo por todo B, de modo que cuando termine, haremos alrededor de 17 millones de accesos a la memoria en total. Eso es mucho acceso a la memoria.

LÁMINA 46

¿Qué pasa si, en lugar de hacer eso, hago las cosas en bloques? Entonces, ¿qué pasa si quiero calcular un bloque de C de 64 por 64 en lugar de una fila de C? Así que echemos un vistazo a lo que sucede allí. Así que recuerde ese número: 17 millones, porque luego vamos a compararlo.

Entonces, ¿qué hay de calcular un bloque? Si miro un bloque, eso tomará 64 por 64 que también toma 4.096 escrituras en C. El mismo número. Pero ahora tengo que hacer unas 200.000 lecturas de A porque necesito acceder a todas esas filas. Y luego para B, necesito acceder a 64 columnas de B. Y eso es otras 262,000 lecturas de B. Lo que termina siendo medio millón de accesos a la memoria en total.

Así que termino haciendo muchos menos accesos si esos bloques caben en mi caché. Por lo tanto, hago mucho menos para calcular la huella del mismo tamaño si calculo un bloque en lugar de calcular una fila. Mucho más eficiente. Y ese es un esquema llamado mosaico. Por lo que, si hace una multiplicación de matrices en mosaico, lo que se hace es descomponer sus matrices en, digamos, 64 por 64 sub matrices, y luego hace dos niveles de multiplicación de matrices.

LÁMINA 47

Se hace un nivel externo de multiplicación de los bloques usando el mismo algoritmo, y luego, cuando le toca al interno, se hace una multiplicación de matriz de 64 por 64, y luego se hacen tres bucles anidados. Entonces se termina con 6 bucles anidados dividiendo el problema de esta manera. Hay un parámetro de ajuste, por supuesto, que es s. Ahora bien, ¿qué tan grande hago el tamaño de mi mosaico?

¿Cuánto debería ser la variable s? ¿Debería ser 64? ¿Debería ser 128? ¿Qué número debo usar allí? ¿Cómo encontramos el valor correcto de s, este parámetro de ajuste? Pruebe algunos de ellos. Experimente. Vea cuál le da buenos números. Y cuando lo haga, resultará que 32 le brinda el mejor rendimiento para este problema en particular. Nos dio una aceleración de aproximadamente 1.7.

LÁMINA 48

Así que ahora estamos casi al 10% del pico. Y si usa Cachegrind o una herramienta similar, puede averiguar cuántas referencias de caché hay y se puede ver que, de hecho, se reduce considerablemente cuando trabaja en mosaico en comparación con solo bucles paralelos rectos. Una vez más, puede utilizar herramientas que le ayuden a resolver esto y a comprender la causa de lo que está sucediendo.

LÁMINA 49

Pero nuestros chips no tienen una sola caché. Tienen tres niveles de cachés. Hay caché L1 con datos e instrucciones, por lo que estamos enfocándonos aquí para los datos de la matriz. El procesador tiene una caché L2, que también es privada para el procesador, y luego una caché L3, ya compartida, y luego sale a la DRAM. Y luego puede ir a sus procesadores vecinos y más.

Las cachés son de diferentes tamaños. Puede ver que aumentan de tamaño: 32 kilobytes, 256 kilobytes, 25 megabytes, y la memoria principal de 60 gigabytes.

LÁMINA 50

Entonces, lo que se puede hacer, para tener un mosaico de dos niveles, tener dos parámetros de ajuste, s y t.

LÁMINA 51

Pero, desafortunadamente, no se puede hacer una búsqueda binaria porque es multidimensional. Hay que hacerlo de forma exhaustiva. Y cuando se hace eso, se termina con 9 bucles anidados. Y por supuesto, no queremos eso. Tenemos tres niveles de almacenamiento en caché. ¿Cuál es el número inductivo para tres niveles de almacenamiento en caché? 12.

El código se pone feo cuando comienzas a hacer que las cosas vayan rápido.

LÁMINA 52

Pero resulta que hay un truco. Se puede colocar en mosaico cada potencia de 2 simultáneamente simplemente resolviendo el problema de forma recursiva. Así que la idea es dividir y conquistar. Divides cada una de las matrices en 4 sub matrices, y luego (si observa los cálculos que necesita hacer) debe resolver 8 subproblemas de la mitad del tamaño y luego hacer una suma. Y entonces tiene 8 multiplicaciones de tamaño n sobre 2 por n sobre 2 y 1 suma de n por n matrices, y eso le dará su respuesta.

LÁMINA 53

Lo que se hace es resolver cada uno de ellos de forma recursiva. Y eso resultará, esencialmente, en el mismo tipo de desempeño. Aquí está el código. Hemos escrito esto usando paralelismo ya que se pueden hacer 4 de ellos en paralelo. El spawn de Cilk funciona diciendo: "ve y haz esta subrutina (que es básicamente un subproblema) y luego, mientras lo haces, puedes ir y ejecutar la siguiente instrucción" (que hará otro spawn y otro spawn y así). Luego la declaración sync dice que no comience la siguiente fase hasta que termine la primera fase.

LÁMINA 54

Ahora bien, una vez hecho esto, obtenemos un tiempo de ejecución de aproximadamente 93 segundos, que es aproximadamente 50 veces más lento que la última versión. Estamos usando la caché mucho mejor, pero resulta que nada es gratis, nada es fácil, por lo general, en ingeniería de rendimiento.

¿Qué pasó aquí? ¿Por qué empeoró esto? Si se observan los números de almacenamiento en caché, se tienen excelentes resultados: muy pocas pérdidas de caché, muchos aciertos de caché, pero somos más lentos. ¿Qué pasó? Se trata de la sobrecarga al inicio de la función y, en particular, la variable de cálculo. Lo que hicimos fue tener un caso base muy pequeño. Estamos haciendo esta sobrecarga hasta n igual a 1. Así que hay una sobrecarga de llamada de función incluso cuando se está multiplicando 1 por 1.

LÁMINA 55

Escojamos un umbral, y por debajo de ese umbral, usemos un algoritmo estándar para ese umbral. Y por encima de él, dividiremos y conquistaremos.

LÁMINA 56

Entonces, lo que hacemos es: si estamos por debajo del umbral, llamamos un caso base, y el caso base se parece mucho a una multiplicación de matriz ordinaria.

LÁMINA 57

Una vez hecho eso, se puede mirar una vez más para ver cuál es el mejor valor para el caso base, y resulta que, en este caso, es 32. Bajamos a 1,30 segundos.

LÁMINA 58

Ahora estamos obteniendo el 12% del pico. Si se cuentan cuántas fallas de caché tenemos, podemos ver que los errores de caché de datos para L1, con dividir y conquistar en paralelo, es más bajo, y también para el almacenamiento en caché de último nivel. El número total de referencias también ha bajado. Es decir, dividir y conquistar resultó una gran idea.

LÁMINA 59

Ahora, la otra cosa que no estamos usando es el hardware vectorial. Todas estas cosas tienen vectores sobre los que podemos operar. Tienen hardware vectorial que procesa datos en lo que se llama estilo SIMD, que significa flujo de instrucción única, datos múltiples. Eso significa que, ante una instrucción, realiza operaciones en un vector. Y como mencionamos, tenemos 8 unidades de punto flotante por núcleo, de las cuales también podemos hacer una suma fusionada y multiplicada.

Entonces, cada registro de vector contiene varias WORDs. En la computadora que estamos usando, son 4 WORDs. Lo importante es que cuando se las usa, no se puede usarlas de cualquier manera. Hay que operar con los datos como un fragmento de datos vectoriales. No puede tener un carril de la unidad vectorial haciendo una cosa y otro carril haciendo otra cosa. Todos tienen que estar haciendo esencialmente lo mismo, la única diferencia es la indexación de la memoria.

LÁMINA 60

Puede producir un informe de vectorización con la herramienta Clang que le dirá qué cosas se están vectorizando y cuáles no. Pero como la mayoría de las computadoras no admiten los más recientes conjuntos de instrucciones vectoriales, el compilador usa, de manera predeterminada, instrucciones vectoriales de forma conservadora.

LÁMINA 61

Entonces, si está compilando para una máquina en particular, use esa máquina en particular. Y aquí están algunas de las banderas de vectorización. Puede decir, use las instrucciones de AVX si tiene AVX. Puede utilizar AVX2. Puede usar las instrucciones de vector fusionado-multiplicar-agregar. Puede usar la cadena que le indique la arquitectura en la que se está ejecutando.

Ahora, los números de punto flotante resultan tener algunas propiedades indeseables, como si no fueran asociativos, así que, si hace A por B por C, la forma en use los paréntesis puede darle dos números diferentes. Entonces, si proporciona una especificación en el código, normalmente, el compilador no cambiará el orden de asociatividad. Puede usar una bandera llamada matemática rápida que le permitirá hacer ese tipo de reordenamiento.

LÁMINA 62

Cuando se usa esto, y particularmente usando arquitectura nativa y matemáticas rápidas, estaremos obteniendo aproximadamente el doble de rendimiento con la vectorización simplemente haciendo que el compilador vectorice.

LÁMINA 63

Lo último que podemos hacer es que usted mismo use las instrucciones vectoriales en lugar de confiar en la forma en que lo hace el compilador. Hay un manual completo de instrucciones intrínsecas a las que puede llamar desde C que le permiten ejecutar las instrucciones vectoriales específicas que desee. Y así, libera al compilador de tener que realizar por sí mismo esa tarea.

# Conclusión

Se puede más hacer más: preprocesamiento, transposición de matrices, alienación de datos, algoritmos inteligentes para el caso base, ahí está, en la documentación de Intel.

Planificar lo que se quiere hacer, codificar si es necesario y luego ejecutar (y ejecutar y ejecutar) haciendo pruebas. Esa es una buena razón para optar por la nube: para poder hacer pruebas en paralelo. Toma menos tiempo hacer 10 pruebas todas al mismo tiempo en 10 máquinas diferentes que esperar sentado frente a una sola.

Una vez hecho todo eso (**lámina 66**), en este caso hemos llegado con nuestras pruebas hasta los intrínsecos de AVX, podemos ver que hemos obtenido un factor 0.41, es decir, un 41% del pico con un factor de aceleración arriba de los 50.000.

Pero nosotros solo hasta aquí llegaremos. Y la razón de esto es que (**lámina 67**) hemos vencido a la biblioteca del núcleo matemático (MKL) de Intel. En este punto, una buena pregunta será: ¿por qué no hemos alcanzado todo el pico? Este es un tema que abordaremos posteriormente. Pero baste por el momento saber que, para que un procesador logre su máximo rendimiento de punto flotante, implica que debe proporcionar entradas a todas sus unidades de punto flotante al mismo tiempo y en cada ciclo además de ocultar todas sus fuentes de latencia.

Aún a pesar de las medidas que hemos obtenido, el MKL de Intel sigue siendo mejor que lo que nosotros hemos conseguido porque, recuerde, nosotros asumimos una potencia de 2. Intel no asume una potencia de 2, y por eso, es más robusta ya que, aunque le ganamos con matrices de 496 por 496, MKL gana muchos otros tamaños de matrices.

Resumiendo, ¿qué hemos hecho? Acabamos de lograr un factor de aceleración mayor a 50.000. Tomando mediciones de uso de combustible, hicimos andar un jumbo jet con la carga de un scooter Vespa.

# Referencias

**Charles Leiserson**, *Lecture 1: Introduction and Matrix Multiplication*, Fall 2018.

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/

**Michael Parker**, *Understanding Peak Floating-Point Performance Claims*, 2017.

https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf