RENDIMIENTO

UAI / SC2 / Comisión 1-O-N / Gerardo Tordoya / 2021-07(JUL)-11

LÁMINA 3

El rendimiento no suele ser lo más importante para los desarrolladores. ¿Cuáles son algunas de las cosas que son 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? Así que esto es una paradoja y un enigma. ¿Por qué estudias algo que claramente no es lo primero en la lista de los desarrolladores?

Creo que la respuesta es que el rendimiento es la moneda de la informática. Utiliza el rendimiento para comprar estas otras propiedades. Entonces dirás (pagarán por) algo como "quiero que sea fácil programar y, por lo tanto, estoy dispuesto a sacrificar algo de rendimiento para hacer algo fácil de programar", 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 hablo con los programadores a menudo escucho "¿Hacer performance? El rendimiento no importa, nunca pienso en eso". Pero luego hablo con personas que usan computadoras y les pregunto "¿Cuál es su principal queja sobre los sistemas informáticos que usa?" Respuesta: "Demasiado lentos". Así que es interesante. La verdadera respuesta es que el desempeño es como la moneda. Es algo que gastas.

¿Prefiero tener U$D 100 o una botella de agua? Bueno, el agua es indispensable para la vida. Ciertamente, hay circunstancias en las que preferiría tener el agua en lugar de U$D 100. Pero en nuestra sociedad moderna puedo 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 (ya que el dinero es una solo una moneda) prefiero tener el dinero porque puedo comprar las cosas que necesito.

LÁMINA 4

Y ésa es la analogía del desempeño. No tiene valor intrínseco, pero contribuye a las cosas. Puedo usarlo para comprar cosas que me interesan, como la usabilidad o la capacidad de prueba. O 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 mira estas computadoras de 1964 a 1977, ¿cuántos bytes tienen? En el 64, hay una computadora con 524 kilobytes. Esa era una gran máquina en ese entonces. Eso es kilobytes. No megabytes, no gigabytes. Kilobytes. Muchos programas agotarían los recursos de esa computadora. Su frecuencia de reloj era de 33 kilohercios. ¿Cuál es la frecuencia de reloj típica de hoy? 4 gigahercios.

LÁMINA 5

En esta era se operaban con kilohercios. Varios programas no encajarían sin una intensa ingeniería de rendimiento. Sin embargo, hubo muchas opiniones que surgieron en esa época. Muchas advierten en contra porque, cuando comienzas a intentar acelerar las cosas, tu código se vuelve ilegible.

Hacer un código que sea legible y rápido: ahí está el arte. Es más: durante muchos años, no hubo ninguna intención en trabajar demasiado en ingeniería de rendimiento.

LÁMINA 6

Si observa el escalado de la tecnología y observa cuántos transistores hay en varios diseños de procesadores, hasta aproximadamente 2004, teníamos la ley de Moore a pleno. Con densidades de chips que se duplicaban cada dos años. Es realmente asombroso. Y junto con eso, a medida que se reducían las dimensiones de los chips (miniaturización), la velocidad del reloj también aumentaba en consecuencia.

LÁMINA 7

En ese entonces, algo iba demasiado lento, tan solo había que esperar un par de años y la cosa se volvería más rápida. Si hacías un software feo, no había mejor recompensa en comparación con simplemente esperar. Y en esa era había algo llamado escala de Dennard que establecía que, a medida que las cosas se miniaturizaban, se posibilitaba que las velocidades del reloj aumentaran, principalmente, al reducir la potencia. Se podía reducir la energía y aun así se podía mantener todo rápido.

LÁMINA 8

Miremos lo que sucedió entre 1977 y 2004: aquí hay computadoras Apple con precios similares. Y también se puede ver que la frecuencia del reloj se disparó notablemente. 1 megahertz, 400 megahertz, 1.8 gigahertz. Y los buses de datos pasaron de 8 bits a 30, y luego a 64. La memoria, en consecuencia, crece. Y sin embargo, el costo permaneció en aproximadamente lo mismo.

LÁMINA 9

Y ese es el legado de la ley de Moore y los tremendos avances en tecnología de semiconductores. Hasta 2004, la ley de Moore y la escala de frecuencia de reloj (la llamada escala de Dennard) fueron esencialmente la imprenta de la moneda de rendimiento. No tenías que hacer nada y al mismo tiempo, acabas de hacer que el hardware sea más rápido. Muy fácil.

LÁMINA 10

Pero todo eso llegó a su fin en 2004 cuando la velocidad del reloj se estancó. Se puede observar, alrededor de 2005, todas las velocidades logradas: de 2 a 4 gigahercios, y no hemos podido hacer que los chips vayan más rápido que eso desde entonces. Pero las densidades se han mantenido excelentes. Ahora bien, la razón por la que la velocidad del reloj se aplanó fue por la densidad de potencia.

LÁMINA 11

Ésta 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: que, si seguían escalando de la forma en que habían estado escalando, 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. "No hay forma de construir una micro tecnología que enfríe eso bien", concluían.

E incluso si resolver algo de esto, las cartas ya estaban sobre la mesa. Porque, por otro lado, ya no se podían escalar las frecuencias de reloj. La razón de esto es que, originalmente, la frecuencia del reloj se escaló asumiendo que la mayor parte de la potencia era potencia dinámica, que estaba funcionando cuando se hacía SWITCH en un circuito. Y lo que sucedió es que, mientras seguíamos reduciendo (miniaturizando), algo que solía estar en el ruido (es decir, las corrientes de fuga) comenzaron a volverse significativas hasta el punto en 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.

LÁMINA 12

Entonces, ¿qué hicieron los proveedores en 2004 y 2005? Dijeron: "Tenemos transistores, pero ya no podemos usarlos para conseguir más velocidad". Entonces, lo que hicieron, fue introducir el paralelismo en forma de procesadores multinúcleo. Ponen más de un núcleo de procesamiento en un chip. Y a partir de entonces, para escalar el rendimiento, se usarían múltiples núcleos. Así, cada generación de la ley de Moore, ahora duplicaba el número de núcleos.

LÁMINA 13

Si se observa lo que sucedió con los núcleos de procesador, se puede ver que, alrededor de 2004-2005, se comenzaron a producir múltiples núcleos de procesamiento por chip. Hasta tal punto que, hoy en día, es casi imposible encontrar un chip de un solo núcleo para una computadora portátil o una estación de trabajo o lo que sea. Todo es multinúcleo. No se puede comprar tan solo uno. Hay que comprar un procesador paralelo.

LÁMINA 14

El impacto de esto fue que el rendimiento dejó de ser gratuito. No se puede simplemente acelerar el hardware. Y, 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: hoy en día existen muchas otras cosas que sucedieron en este tiempo intermedio (de inacción): se agregaron unidades vectoriales como algo común, GPU, jerarquías de caché más pronunciadas; incluso en algunas computadoras: lógica configurable, etcétera.

Ahora es el turno del software 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.

LÁMINA 17

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

LÁMINA 19

Ésta es una multiplicación de matrices en la que, esencialmente, si se suma el número total de operaciones, es aproximadamente 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.

LÁMINA 20

Ahora, la máquina que veremos será una de las que usted tendría acceso en AWS (Amazon Web Services).

Es una máquina optimizada para computación, que tiene una micro arquitectura Haswell que se ejecuta a 2,9 gigahercios.

Hay 2 chips de procesador para cada una de estas máquinas 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 brinda un poco más de rendimiento, pero también hace que sea muy difícil medir las cosas, por lo que desactivaremos el hyperthreading).

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). Esa es una unidad vectorial. Básicamente, cada uno de estos 18 núcleos puede realizar 8 operaciones de doble precisión, incluida una 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 luego tiene 60 gigabytes de DRAM.

Es decir, es una computadora muy potente.

Si observa el cálculo de rendimiento máximo, es la 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 (si puede usar tanto la multiplicación como la suma). Esto significa un total de 836 gigaflops. Mucha, pero mucha potencia.

LÁMINA 21

Ahora aquí está nuestro código básico. Este es el código completo de Python para realizar la multiplicación de matrices. Ahora, en general, en Python, uno no usaría este código porque simplemente llamaría a una subrutina de biblioteca que hace la multiplicación de matrices. Pero vamos a ilustrar el concepto para cuando haya que escribir un código como el de la multiplicación de matrices. Quiero darle una idea del tipo de rendimiento que se obtiene de Python. Cuando ejecuta este código, para que pueda ver que la hora de inicio, ya sabe, antes del ciclo triplemente anidado, tomamos una medición de tiempo y luego tomamos otra medición de tiempo al final, y luego imprimimos la diferencia.

Resulta que se necesitan unos 21.000 segundos, que son unas 6 horas. Increíble. ¿Es esto rápido? No. ¿Sabe cómo podemos saber si esto es rápido o no? ¿Qué debemos esperar de nuestra máquina? Así que hagamos un cálculo inicial de cuántas operaciones hay que hacer y qué tan rápido deberíamos poder hacerlo. Simplemente pasamos y dijimos cuáles son todos los parámetros de la máquina. Entonces hay 2n operaciones al cubo que deben realizarse. No estamos haciendo el algoritmo de Strassen ni nada de eso. Solo estamos haciendo un ciclo recto triplemente anidado.

LÁMINA 22

Entonces eso es 2 a las 37 operaciones de punto flotante. El tiempo de ejecución es de 21,000 segundos, por lo que dice que estamos obteniendo aproximadamente 6.25 megaflops de nuestra máquina cuando ejecutamos ese código. Con solo dividirlo, ¿cuántas operaciones de punto flotante por segundo obtenemos? Tomamos el número de operaciones dividido por el tiempo.

El pico, como recordará, es de unos 836 gigaflops. Y estamos obteniendo 6.25 megaflops. Así que estamos obteniendo aproximadamente un 0,00075% del pico. Esto no es rápido. Entonces hagamos algo realmente simple. Codifiquémoslo en Java en lugar de Python. Entonces tomamos solo ese bucle. El código es casi el mismo.

LÁMINA 23

Solo el bucle triplemente anidado. Lo ejecutamos en Java. Y resulta que el tiempo de ejecución ahora es de poco menos de 3.000 segundos, lo que equivale a unos 46 minutos. El mismo código. Python, Java. Conseguimos una aceleración de casi 9 veces simplemente codificándolo en un idioma diferente. Bien, probemos C.

LÁMINA 24

¿Qué sucede cuando se codifica en C? Es exactamente el mismo código. Ahora son básicamente 1.100 segundos, que son aproximadamente 19 minutos. Así que obtuvimos aproximadamente el doble de rápido que Java y aproximadamente 18 veces más rápido que Python.

LÁMINA 25

Así que aquí es donde nos encontramos hasta ahora. Tenemos el tiempo de ejecución. Y la aceleración relativa es cuánto más rápido es que la fila anterior, y la aceleración absoluta es cómo se compara con la primera fila. Y hasta ahora estamos logrando obtener 0.014% del pico. Así que todavía somos lentos, pero antes de intentar optimizarlo aún más, por ejemplo, ¿por qué Python es tan lento y C tan rápido?

La gran razón por la que Python es lento y C es tan rápido es que Python es interpretado y C se compila directamente en código de máquina. Y Java está en algún lugar en el medio porque Java se compila en código de bytes, que luego se interpreta y luego se compila justo a tiempo en códigos de máquina.

LÁMINA 26

Entonces, hay como todos estos gastos generales en comparación con simplemente hacer sus operaciones.

LÁMINA 27

Por lo tanto, 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

Digamos que tenemos la versión C, que es desde donde nos moveremos desde este punto porque es lo más rápido que obtuvimos. Hasta ahora, resulta que puede cambiar el orden de los bucles en este programa 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, ¿crees que el orden de los bucles es importante para el rendimiento? Ésta es como la pregunta clave.

LÁMINA 30

Entonces, 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 ahí? Se debe a la localidad de caché.

LÁMINA 31

Entonces, 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 muy barato y 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 afecta el rendimiento de qué manera lo hace. Así que echemos un vistazo al patrón de acceso para el orden i, j, k. Entonces, lo que estamos haciendo es que, una vez que averigüemos qué es i y qué es j, vamos hacer el ciclo de 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 traerá 64 bytes para operar en un dato en particular. Y luego ignora 7 de las 8 palabras de punto flotante en esa línea de caché y pasa a la siguiente. Así que está desperdiciando muchísimo. Entonces A tiene una buena ubicación espacial en el sentido de que todo es adyacente y usaría las líneas de caché de manera efectiva. B tiene 4096 elementos de distancia, es decir, tiene una localidad espacial pobre.

LÁMINA 34

Entonces, si miramos otro orden (i, k, j) resulta que obtienes una buena localidad espacial tanto para C como para B y excelente para A.

LÁMINA 35

Y si miras incluso otro, no se vuelve tan bueno como los anteriores. Este lo está haciendo muy mal.

LÁMINA 36

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

LÁMINA 37

Entonces, cuando eliges el mejor de esos, obtenemos una aceleración relativa de aproximadamente 6 y 1/2. Entonces, ¿qué otros cambios simples podemos probar? De hecho, hay una colección de cosas que podríamos hacer que ni siquiera nos hacen tocar el código.

LÁMINA 38

Puede cambiar las banderas del compilador. Clang, que es un compilador que estamos usando para esto, proporciona una colección de conmutadores de optimización, y puede especificar un conmutador al compilador para pedirle que optimice. O0 (si mira la documentación) dice "No optimizar". O1 dice "Optimizar". O2 dice: "Optimice más". O3 dice "Optimizar aún más".

En este caso, resulta que a pesar de que se optimizó más en O3, resulta que O2 era una configuración mejor. Este es uno de estos casos. 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. Y también hay otros tipos de optimización. Puede hacer que realice una 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

Y hay una variedad de otras cosas. Entonces, con esta tecnología simple, eligiendo una buena bandera de optimización, en este caso, O2, obtuvimos gratis, básicamente, un factor de 3.25, ¿de acuerdo? Sin tener que trabajar mucho en absoluto. Y ahora estamos empezando a acercarnos al 1% del rendimiento máximo. Tenemos un 0,3% de rendimiento máximo. Y, sin embargo, ¿qué está aún causando el bajo rendimiento? ¿Por qué no estamos obteniendo un mayor rendimiento de esta computadora?

LÁMINA 40

No estamos usando todos los núcleos. Hasta ahora estamos usando un solo núcleo, ¿y cuántos núcleos tenemos? 18 núcleos allí, 17 inactivos, mientras intentamos optimizar 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, y entonces se retransmitirán todas esas iteraciones en paralelo.

LÁMINA 42

El compilador y el sistema de ejecución son libres de programarlos, etc. Y también podríamos hacerlo para el bucle interno. 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 i y j juntos. No puede hacer solo k en paralelo y esperar obtener una mejoría. 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 no tocaremos ahora). Entonces, la regla general aquí es paralelizar los bucles externos en lugar de los bucles internos. Entonces, cuando paralelizamos bucles, obtenemos una aceleración de casi 18 veces en 18 núcleos. Sin embargo, déjeme asegurarle que no todo código es tan fácil de paralelizar.

LÁMINA 43

Así que ahora estamos hasta, aproximadamente, un poco más del 5% del pico. Entonces, ¿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. Trate 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. Y entonces se termina dividiendo así. Hay un parámetro de ajuste, por supuesto, que es, ¿qué tan grande hago el tamaño de mi mosaico? Es s.

¿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 en ... ¿qué? Estamos casi al 10% del pico. Y si usa Cachegrind o una herramienta similar, puede averiguar cuántas referencias de caché hay y así sucesivamente, y puede ver que, de hecho, se reduce considerablemente cuando hace el 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

Bueno, resulta que nuestros chips no tienen un solo caché. Tienen tres niveles de cachés. Hay caché L1. Y hay datos e instrucciones, por lo que estamos pensando en los datos aquí, para los datos de la matriz. Y tiene una caché L2, que también es privada para el procesador, y luego una caché L3 compartida, y luego sale a la DRAM. También puede ir a sus procesadores vecinos y más.

Y son de diferentes tamaños. Puede ver que aumentan de tamaño: 32 kilobytes, 256 kilobytes, a 25 megabytes, la memoria principal que es de 60 gigabytes. Entonces, lo que puede hacer es, si desea hacer un mosaico de dos niveles, puede tener dos parámetros de ajuste, s y t.

LÁMINA 50

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

LÁMINA 51

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

Pero luego, por supuesto, lo que se hace es resolver cada uno de ellos de forma recursiva. Y eso resultará, esencialmente, el mismo tipo de desempeño. Aquí está el código. Hemos escrito esto usando en paralelo, porque resulta que se puede hacer 4 de ellos en paralelo. Y el spawn de Cilk aquí dice "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 la ingeniería de rendimiento.

¿Qué pasó aquí? ¿Por qué empeoró esto? Si observa los números de almacenamiento en caché, obtendrá grandes resultados en el caché. Es decir, muy pocas pérdidas de caché, muchas visitas a la caché, pero aún somos más lentos. ¿Por qué es esto? La sobrecarga al inicio de la función y, en particular, el lugar que importa es 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 cuenta cuántas fallas de caché tenemos, puede ver que aquí está la caché de datos para L1, y con dividir y conquistar en paralelo es la más baja, pero ahora también lo es el almacenamiento en caché de último nivel. Y luego, el número total de referencias también ha bajado. Así que dividir y conquistar resultó una gran victoria aquí.

LÁMINA 59

Ahora, la otra cosa que mencionamos, que es que no estamos usando 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 da una instrucción y 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 palabras. En la computadora que estamos usando son 4 palabras. Pero es importante que cuando los usa, no puede usarlos de cualquier manera. Tiene que operar con los datos como un fragmento de datos vectoriales. No puede tener un carril de la unidad vectorial haciendo una cosa y un carril diferente 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 solicitándolo, y el sistema le dirá qué tipo de cosas se están vectorizando y cuáles no. Y, en particular, la mayoría de las computadoras no admiten los conjuntos de instrucciones vectoriales más recientes, por lo que el compilador usa instrucciones vectoriales de forma conservadora de manera predeterminada.

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 una 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 usa eso, 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

Entonces, lo último que vamos a hacer es que usted mismo pueda usar las instrucciones vectoriales en lugar de confiar en el compilador para hacerlo. 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.

LÁMINA 64

Y también puede usar más información para hacer cosas como: puede hacer un preprocesamiento y puede transponer las matrices, lo que resulta ser de ayuda, y alinear los datos. Y hay muchas otras cosas que utilizan algoritmos inteligentes para el caso base. Y haces más ingeniería de rendimiento.

LÁMINA 65

Piense en lo que está haciendo, codifique y luego ejecute (y ejecute y ejecute) para probar, y esa es una buena razón para optar por la nube: porque puede hacer pruebas en paralelo. Le toma menos tiempo hacer sus pruebas en términos de estar sentado el tiempo que la computadora está haciendo algo que cuando dice "quiero hacer 10 pruebas, pongamos en marcha 10 máquinas y hagamos todas las pruebas al mismo tiempo".

LÁMINA 66

Una vez hecho todo eso, (y en este caso estamos ya probando los intrínsecos de AVX), obtenemos hasta 0.41 del pico, es decir, un 41% del pico, y obtuvimos un factor arriba de 50,000 de aceleración.

LÁMINA 67

Y resulta que hasta aquí llegaremos. Y la razón es que en este momento vencimos a la biblioteca del kernel matemático (diseñada profesionalmente) de Intel. Una buena pregunta es, ¿por qué no estamos alcanzando todo el pico? Resulta que Intel MKL es mejor que lo que hicimos porque (recuerde) asumimos que era una potencia de 2. Intel no asume que es una potencia de 2, y por eso es más robusta, y aunque ganamos con matrices de 496 por 496, MKL gana en otros tamaños de matrices.

LÁMINA 68

El final de la historia es, ¿qué hemos hecho? Acabamos de lograr un factor de más de 50.000. Si observara la economía de gasolina de un jumbo jet y obtuviera el tipo de rendimiento que acabamos de obtener en términos de millas por galón, podría cargar un jumbo jet en una pequeña Vespa. Eso es lo que hemos llegado a hacer.

REFERENCIAS

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