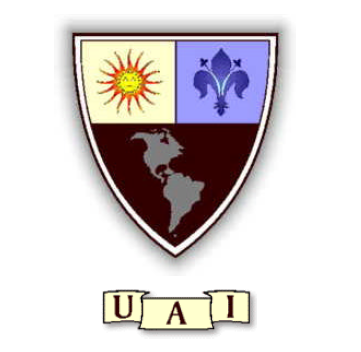
UNIVERSIDAD ABIERTA INTERAMERICANA



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 3](#_Toc78053192)

[Desarrollo 5](#_Toc78053193)

[Conclusión 10](#_Toc78053194)

[Referencias 11](#_Toc78053195)

# Introducción

El microcódigo es una capa de instrucciones a nivel de hardware que implementa instrucciones de código de máquina de nivel superior. El microcódigo normalmente reside en una memoria especial de alta velocidad. Los alumnos de esta cátedra tuvimos contacto con él en nuestra ejercitación sobre un modelo de procesador. Por ejemplo, la instrucción "sumar dos registros" se implementa mediante la activación y desactivación de un conjunto reducido de señales eléctricas en el banco de registros y la unidad aritmético-lógica (ALU).

Desde los micro códigos del procesador de 4 bits Intel 4004 de 1971 se ha recorrido un largo camino hasta las hoy modernas instrucciones intrínsecas, y una de las pocas formas de abordar su descubrimiento, es el estudio del rendimiento. ¿Qué otro tópico lo justificaría?

Debido a que uno de los parámetros de este informe final es la brevedad, no podremos hacer un estudio exhaustivo de la temática (la cual, por otro lado, podría llenar tomos). Y por eso mismo, las láminas necesarias para el presente desarrollo, han sido puestas en un documento anexo a fin de no restar aún más este restringido espacio de conceptualizaciones.

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 estudiar el rendimiento? Creo que la respuesta es que el rendimiento es la moneda de la informática. Se utiliza el rendimiento para comprar otras propiedades. Se paga por algo como "quiero que esto sea fácil de 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 de rendimiento. Y claramente, si el rendimiento se degrada demasiado, esas cosas se vuelven inutilizables. Cuando uno habla con los programadores, a menudo 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 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 necesita. Ésa es la analogía del desempeño. No tiene valor intrínseco, pero contribuye. Puede 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, 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 encajaban sin una intensa ingeniería de rendimiento. Y, sin embargo, en esa época hubo muchas opiniones en contrario porque, cuando se intenta acelerar el rendimiento, el código se vuelve ilegible. Y el arte radica en hacer código que sea legible y rápido. Es por ello, porque otras eran las prioridades, es que 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 se duplicaban cada dos años. Y a medida que se reducían las dimensiones de los chips, la velocidad del reloj también aumentaba en consecuencia (no abordaremos ese tema aquí, el de 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 que, a partir del 2005, 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 de 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-sumar (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, tomaremos 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 recordamos, 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.

C es desde donde nos moveremos de aquí en más porque es el más rápido (**lámina 28**). Ahora bien, podemos cambiar el orden de los bucles sin afectar la corrección. Así que aquí actualizaremos este código. Ésta es otra manera que podemos plantear i, k y j (**lámina 29**) a fin de hacer una actualización 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? Veamos (**lámina 30**). Podemos observar que, cuando enrocamos el orden, 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.

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. Cuando el procesador accede a algo, si está en caché, obtiene algo llamado "un acierto". Es una operación rápida. Pero si falla, debe ir a un caché de nivel más profundo o incluso hasta la memoria principal. Y esa operación es mucho, mucho más lenta. Lo que sucede con este problema de matrices (**lámina 32**) es que las matrices se colocan en la memoria en orden de fila mayor. Recuerde: usted tiene una matriz bidimensional. Y se las presenta, en el procesador, en el orden lineal de las direcciones de memoria. Básicamente, tomando la fila 1, y luego se añade la fila 2, y luego se añade la fila 3, y así sucesivamente, desdoblándola.

Echemos un vistazo al patrón de acceso para el orden i, j, k (**lámina 33**). Para k, obtenemos una excelente localidad espacial porque simplemente estamos accediendo a la misma ubicación. En cada ciclo, estará en caché, será rápido acceder a C. Para A, lo que sucede es que recorremos en un orden lineal y obtenemos una buena localidad espacial. Pero para B, que atraviesa columnas, esos puntos se distribuyen lejos unos de otro 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 al 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.

Si evaluamos otro orden (i, k, j → **lámina 34**) vemos una buena localidad espacial tanto para C como para B y excelente para A. Finalmente, si evaluamos j, k, i (**lámina 35**) vemos que lo hace muy mal. Se pueden medir los diferentes órdenes con una herramienta como Valgrind (**lámina 36**) para resolver esto.

Ya habiendo seleccionado el mejor orden, obtenemos una aceleración relativa de aproximadamente 6 .50 con respecto a la medición anterior (**lámina 37**). ¿Qué otros cambios podemos probar? Podemos cambiar las banderas del compilador (**lámina 38**). Clang es un compilador que usaremos para esto, que proporciona una colección de conmutadores de optimización, y puede especificar un conmutador al compilador para pedirle que optimice. (Consultando su documentación, se verá que se pueden 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 la mejor configuración. ¿Por qué? Porque las optimizaciones son hasta cierto punto heurísticas. Con esta simple tecnología (**lámina 39**), eligiendo una buena bandera de optimización (que en nuestro caso fue O2) obtuvimos un factor de mejora del 3.25 sin hacer mucho. Pero no 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?

No estamos usando todos los núcleos (**lámina 40**). Hasta ahora estamos usando solo 1, ¿y cuántos núcleos tenemos? 18 núcleos, 17 inactivos, mientras intentamos optimizar tan solo uno. Tenemos que usar todos, (**lámina 41**) y para eso, usaremos la infraestructura de otra herramienta: Cilk. 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.

Ahora bien, con Cilk, podemos hacer esa retransmisión bucle a bucle. Entonces la pregunta es: ¿cuál versión paralela funciona mejor? (**lámina 42**). Podemos poner en paralelo el ciclo i, podemos hacer el ciclo j en paralelo, y podemos hacer paralelos i y j juntos. Paralelizar k no resultará en una mejoría (no tocaremos este tema ahora). Entonces, si se mira, ¡qué variedad de tiempos de ejecución! Si paralelizamos solo el ciclo i queda en 3,18 segundos. Si paralelizamos el ciclo j se ralentiza. Si paralelizamos tanto i como j nos sigue dando malos resultados. Así que nos quedaremos con el bucle exterior paralelizado. Esto tiene que ver con algo llamado sobrecarga de programación (tema que tampoco tocaremos ahora). Como regla general es mejor paralelizar los bucles externos en lugar de los bucles internos (**lámina 43**). Cuando paralelizamos bucles, obtenemos una aceleración de casi 18 veces (obvio, por los 18 núcleos).

Ahora estamos aproximadamente a un poco más del 5% del pico. Hemos logrado algún avance, pero ¿en dónde estamos perdiendo aquí? ¿Por qué obtenemos solo el 5%? Podemos administrar mejor las fallas de caché (**lámina 44**). Así que volvamos a las cachés de hardware y reestructuremos el cálculo para reutilizar los datos en la caché tanto como sea posible. Veamos (**lámina 45**). Suponga que vamos a calcular solo una fila de C. Puesto que hay un vector de 4096 de largo allí, son 4096 escrituras que vamos a hacer. Vamos a conseguir algo de localidad espacial allí, lo cual es bueno, pero básicamente lo que el procesador está haciendo son 4096 escrituras. Ahora, para calcular esa fila, necesito acceder a 4096 lecturas de A. Y necesito todo B (en términos de localidad, véase lámina 32). Los detalles los podemos discutir luego, pero lo principal que hay que entender es que se está accediendo a todo B. Luego, para calcular otra fila de C, tendremos que hacer lo mismo: pasar por una fila de A y de nuevo por todo B. De modo que, cuando terminemos, habremos hecho alrededor de 17 millones de accesos a la memoria. Es mucho.

¿Qué pasa si, en lugar de hacer eso, hago las cosas en bloques? (**lámina 46**). ¿Qué pasa si calculamos un bloque de C de 64 por 64 en lugar de una fila de C? Echemos un vistazo a un bloque de 64 por 64 (que también implica 4096 escrituras en C). El mismo número. Pero ahora tengo que hacer unas 200.000 lecturas de A y acceder a 64 columnas de B (262.000 lecturas más). ¿Resultado? Medio millón de accesos a la memoria en total. Y 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. Ese es un esquema llamado mosaico. Por lo que, si se hace una multiplicación de matrices en mosaicos, lo que se hace es descomponer sus matrices en, digamos, 64 por 64 sub matrices, y luego hacemos dos niveles de multiplicación de matrices (**lámina 47**). Para esto, 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. Dividiendo el problema de esta manera, terminamos con 6 bucles anidados. Hay un parámetro de ajuste, que es s, el tamaño del mosaico. Ahora bien, ¿qué tan grande hago el tamaño del mosaico? ¿Debería ser 64? ¿Debería ser 128? ¿Qué número debo usar allí? ¿Cómo encontramos el valor correcto de s (el parámetro de ajuste)? Probemos algunos de ellos. Veamos cuál arroja buenos números. Una vez que lo hemos probado, resulta que 32 nos brinda el mejor rendimiento para este problema en particular. Nos dio un tiempo de ejecución de aproximadamente 1.7 segundos (**lámina 48**).

Ahora estamos casi al 10% del pico. Y si se usa Cachegrind o una herramienta similar, puede verse cuántas referencias de caché hay (tabla inferior) y puede observarse que, de hecho, se reducen considerablemente cuando se trabaja en mosaico en comparación con solo bucles paralelos rectos.

Pero nuestros chips no tienen una sola caché (**lámina 49**). Tienen tres niveles de cachés. Hay caché L1 con datos e instrucciones, por lo que estaremos enfocándonos aquí como albergue de 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. Las cachés son de diferentes tamaños y puede verse que aumentan: 32 kilobytes, 256 kilobytes, 25 megabytes, y la memoria principal, 60 gigabytes.

Y desde aquí, ¿hacia dónde continuar? ¿Cómo aprovechar esas cachés? ¿Qué de los mosaicos de 2 niveles? (**lámina 50**). Para ello, debemos tener dos parámetros de ajuste, s y t. Desafortunadamente, no podemos usarlos para hacer una búsqueda binaria porque es multidimensional. Hay que hacerlo de forma exhaustiva. Y cuando se hace eso, se termina con 9 bucles anidados (**lámina 51**). Y por supuesto, no queremos eso. Es justamente el punto que tratamos al principio. El código se pone feo cuando uno quiere que las cosas vayan rápido. Pero resulta que hay un truco (**lámina 52**). 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. Se divide cada una de las matrices en 4 sub matrices, y luego (si se observan los cálculos que se necesitan hacer) resolver 8 subproblemas de la mitad del tamaño del original (y luego, por supuesto, hacer la suma). Quedarían entonces 8 multiplicaciones de tamaño n sobre 2 por n sobre 2 y 1 suma de n por n matrices.

¿Qué estamos diciendo? Que vamos a dividir para conquistar (**lámina 53**). Lo que se hace es resolver cada uno de ellas de forma recursiva. En el código hemos escrito esto usando paralelismo (ya que 4 se pueden hacer 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 "no comiences la siguiente fase hasta que termines la primera".

Ahora bien, una vez hecho esto (**lámina 54**), obtenemos un tiempo de ejecución de aproximadamente 93 segundos, que es aproximadamente 50 veces más lento que la última versión. Pero estamos usando la caché mucho mejor, ¿qué pasó aquí? Si se observan los números de almacenamiento en caché (con las herramientas antes mencionadas) se pueden ver excelentes resultados: muy pocas pérdidas de caché, muchos aciertos de caché. Pero ahora 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. Escojamos un umbral (**lámina 55**), y por debajo de ese umbral, usemos un algoritmo estándar, y por encima de él, dividamos y conquistemos (**lámina 56**). Una vez hecho eso, se puede mirar una vez más para ver cuál es el mejor valor para el caso base (**lámina 57**), y resulta que, en nuestro caso, es 32. Bajamos a 1,30 segundos. Hemos resuelto esta parte (**lámina 58**).

Ahora estamos al 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.

Ahora bien, no estamos usando hardware vectorial (**lámina 59**). Tenemos vectores sobre los que podemos operar. Tenemos 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-multiplicación fusionada. 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 se puede tener un carril de la unidad vectorial haciendo una cosa y a otro carril haciendo otra. Todos tienen que estar haciendo esencialmente lo mismo, la única diferencia es la indexación de la memoria (**lámina 60**). Clang puede generar un informe de vectorización 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. Entonces, si está compilando para una máquina en particular (**lámina 61**), 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-sumar. Puede usar la cadena que le indique la arquitectura en la que se está ejecutando.

Cuando usamos hardware vectorizado (**lámina 62**), y particularmente usando arquitectura nativa y matemáticas rápidas, estaremos obteniendo aproximadamente el doble de rendimiento de dicha vectorización simplemente haciendo que el compilador vectorice. Lo último que podemos hacer es que usted mismo use las instrucciones vectoriales (las llamadas instrucciones intrínsecas) en lugar de confiar en la forma en que la que lo hace el compilador (**lámina 63**). Hay un manual completo de instrucciones intrínsecas a las que puede llamar desde C que le permiten ejecutar instrucciones vectoriales tan específicas como desee. Y así, libera al compilador de tener que realizar esa tarea.

# Conclusión

Se puede más hacer más: preprocesamiento, transposición de matrices, alienación de datos, algoritmos inteligentes para el caso base, etcétera. Solo es cuestión de indagar 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 nuestro caso hemos llegado con nuestras pruebas hasta los intrínsecos de AVX, y 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 solo hasta aquí llegaremos. ¿Y por qué? 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 sería: ¿por qué no hemos alcanzado todo el pico? Este es un tema que, por las limitaciones del presente trabajo, ya no podemos abordar (en las referencias se puede encontrar el documento para explayarse sobre el asunto). 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, lo cual, en la práctica, no es posible.

Aún a pesar de las mejoras 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 en 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 y comparándolas, 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

Valgrind is an instrumentation framework for building dynamic analysis tools. https://valgrind.org/

The Clang project provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM project. https://clang.llvm.org/

Cilk aims to make parallel programming a simple extension of ordinary serial programming. https://cilk.mit.edu/