Programación Paralela

OPENMP

Facultad de Ciencias Exactas, Físicas y Naturales.

Universidad Nacional de Córdoba

Colazo, Agustín

Wolfmann, Gustavo

3/11/2017

Indice

Introducción

Openmp es una librería de C, C++ y Fortran que permite implementar programas paralelos en arquitecturas de memoria compartida. Así se puede hacer uso de los distintos núcleos físicos y de los hilos lógicos que ofrece cada procesador, a veces pudiendo acelerar considerablemente los programas.

En este trabajo se implementara un programa en C para la multiplicación de matrices cuadradas haciendo uso de esta librería.

Objetivo

El objetivo es implementar varias configuraciones distintas para analizar el impacto de distintos métodos y parámetros en un programa destinado a la multiplicación de matrices.

Marco Teórico

Actualmente, los procesadores son rápidos pero la memoria no alcanza las mismas velocidades que los procesadores. Es por eso que en los algoritmos no es solo importante el orden de los pasos del algoritmo, sino que cobra mucha importancia el maneja de la memoria.

El problema con la memoria es que para que sea económicamente viable. Las memorias rápidas son pequeñas y las memorias grandes son lentas.

Es por esta diferencia de velocidades que se han diseñado una arquitectura memoria en varios niveles.

- Disco duro
- Memoria de acceso aleatorio
- Memoria cache
  - Cache de nivel 3
  - Cache de nivel 2
  - Cache de nivel 1
- Registros del procesador

Donde el disco duro es la memoria más grande y más lenta.

Hoy en día, el manejo de memoria en los algoritmos es sumamente relevante. Se busca disminuir las fallas de cache para acelerar los algoritmos. Es por esto que se utilizan diversas técnicas como el alineamiento de cache, la multiplicación de matrices en bloques o la vectorización.

La necesidad de alinear cache ocurre de que la memoria se maneja por bloques, y no por direcciones individuales. Por lo tanto, cuando hay varios hilos si uno modifica un valor, y otro hilo necesita acceder al mismo bloque este se marcará como usado. Por lo tanto el bloque modificado tendrá que ser escrito en un nivel inferior de memoria para que el segundo hilo pueda acceder a la información, resultando en fallas de cache que se traduce en peor rendimiento.

La multiplicación de matrices en bloques se relaciona con un uso mas eficiente de la cache. De modo que un bloque de memoria sea puesto en la cache, se realicen todos los cómputos necesarios sobre ese bloque y luego no deba accederse mas a ese bloque. Así se reducen las fallas de cache. Una regla general es $3*(\text{Tamaño de bloque})^2*\text{Tamaño\_de\_variable} = \text{Tamaño\_de\_Cache}$

La vectorización es una opción que se encuentra en algunos procesadores que permite acelerar los cálculos. Por ejemplo los registros mavx de 256 bits que permiten computar varias variables al mismo tiempo.

También es importante la afinidad de los hilos (en un programa multi-hilos), esto es en que núcleo del procesador se coloca cada hilo y si permanece en ese núcleo o no durante su existencia. La razón de esto es la arquitectura de los procesadores y la localidad espacial de los datos.
Por ejemplo, en la computadora Pulqui el procesador tiene dos pastillas, cada pastilla tiene una cache L3 y seis núcleos. Cada dos núcleos se comparte una cache L2 y cada núcleo tiene una cache L1. Si hay hilos que procesan datos cercanos, es conveniente que estos hilos se encuentren en núcleos que compartan la misma cache L2 o cache L3. En cambio, si los datos son lejanos es

conveniente que estos hilos se encuentren en núcleos lejanos así no hay competencia por la cache. Ambos hilos querrían traer sus datos a la cache, compitiendo por esta.
También es importante que los hilos sean bien distribuidos entre ambas pastillas para distribuir la carga de las cache, si llegase a ser necesario.

Diseño e Implementación

Se codifico la multiplicación de matrices en C. Este programa se ejecutara varias veces modificando parámetros y banderas.

Cada configuración se ejecutara varias veces y se medirá el tiempo de ejecución usando las funciones de la librería openmp, estos tiempos se guardaran por configuración.

Para esto se hizo varios scripts para el shell de Ubuntu. Estos se encargan del cambio de configuraciones, las ejecuciones y de guardar los tiempos en un archivo.

Para facilitar la implementación de los scripts, se pasa al programa principal el tamaño de la matriz y el tamaño de bloques como parámetros.

Las varias configuraciones que se utilizaran son:

- Programa sin paralelismo
  - Con o sin anidar.
    - Con distintos tamaños de bloque.
      - 128
      - 256
      - 512
      - 1024
  - Con o sin instrucciones simd.
  - Matrices de tamaño 8000 no se ejecutaron en secuencial.
  - Estas configuraciones se cruzan.
- Programa con paralelismo.
  - Distintas afinidades (places, bind).
    - Cores, close.
    - Cores, spread.
    - Bind=false.
    - Dos afinidades manuales.
  - Distintas banderas.
    - Con o sin mavx.
    - Con o sin O3.
  - Anidado (tamaños)
    - 128
    - 256

        - 512
        - Para tamaño 8000: 16, 32, 64, 128 y 256.
    - Para 2, 4, 6, 8, 10 y 12 hilos. Excepto en el caso de matrices de tamaño 8000 se usaron 4, 8 y 12 hilos.
    - Todas estas configuraciones se cruzan.

También se usa el comando *“perf stat -d ./programa”* para analizar las cargas y fallas de la cache.

Resultados y Análisis en Pulqui

A continuación se mostraran algunos de los resultados obtenidos y se los analizara. La totalidad de los resultados se encuentran en las planillas de datos. Aquí solo se muestra algunos de los resultados de interés.

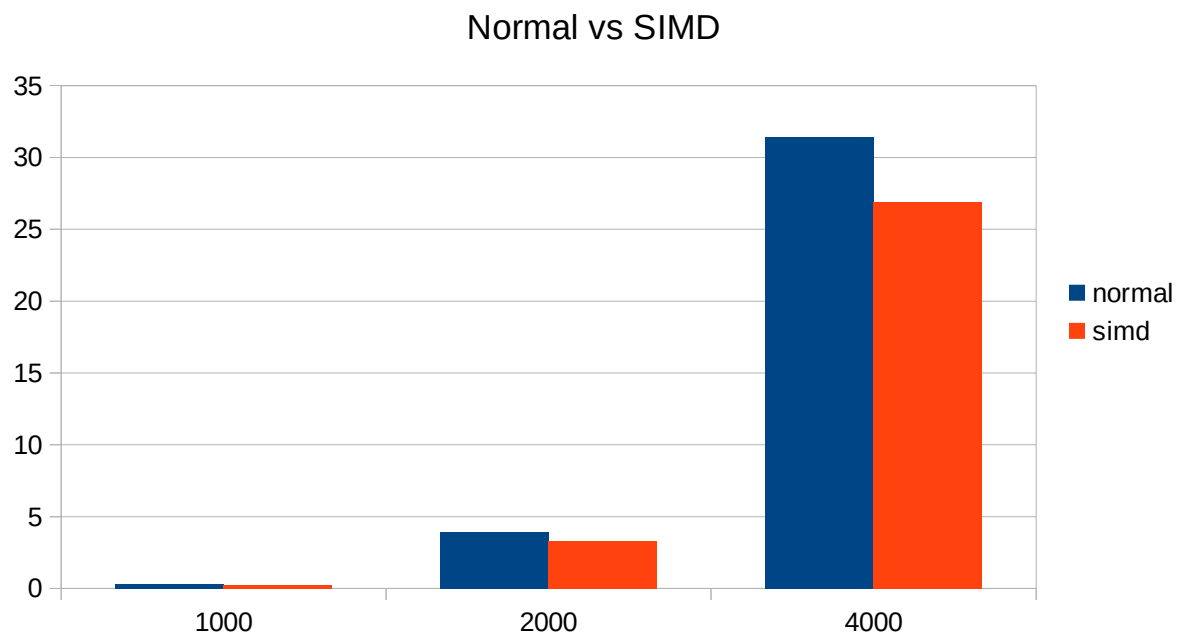
Todos los datos recolectados y sus promedios se encuentran en la carpeta *"informe/planillas de datos"*

Para tomar el tiempo de ejecución de cada configuración, se los ejecuto cinco veces y se tomo el promedio.

Algunas especificaciones de la maquina que se utilizo (pulqui):

- 12 Procesadores físicos, en dos pastillas.
- 24 Procesadores virtuales.
- 256KB de memoria cache de nivel 2.
- Alineamiento de cache: 64
- Cuenta con registros mavx.

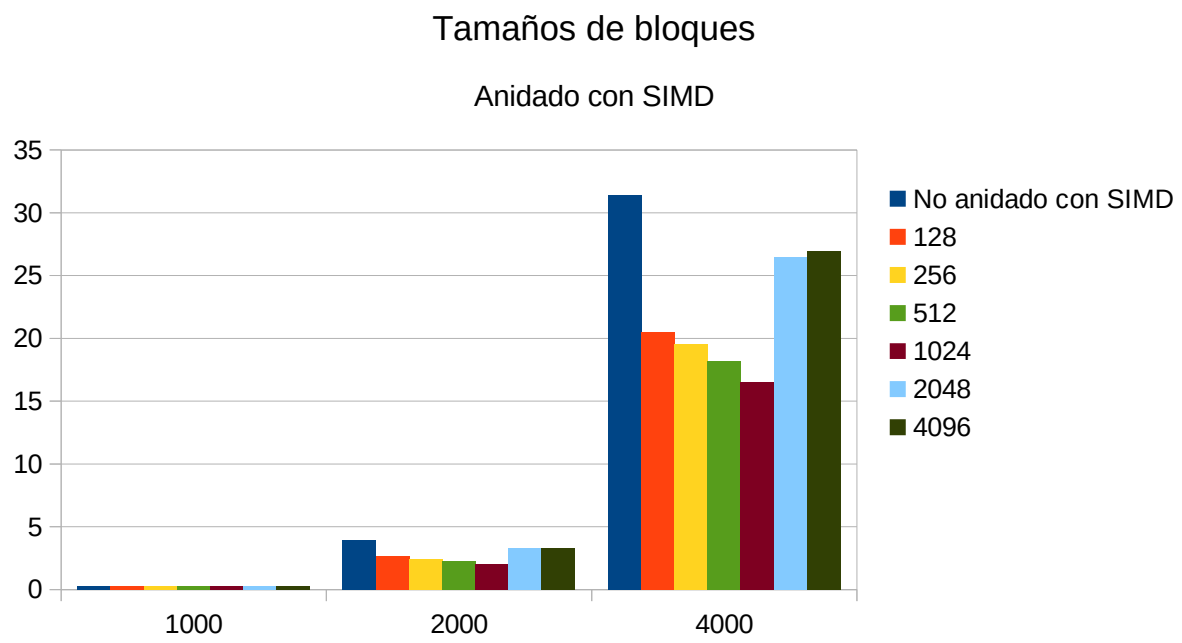
SIMD



Estos datos fueron tomados en la multiplicación de matrices con tres bucles, usando y sin usar registros mavx. Podemos notar que la mejora en los tiempos es significativa.

Multiplicación de matrices con seis bucles

Secuencial



En el primer caso, se tomaron datos de la multiplicación de matrices con 6 bucles en un programa secuencial.

En este caso el mejor tamaño de bloque es 1024.

La mejora que presenta usar seis bucles y simd es significativa. El programa puede llegar a ser dos veces mas rápido.

Paralelo

En el segundo caso se tomaron los tiempos para una matriz de 1000x1000 y se calcula el speedup con distintas cantidades de hilos y con distintos tamaños de bloque. Se utilizo la bandera de optimización -O3 pero no se utilizo la bandera -mavx.

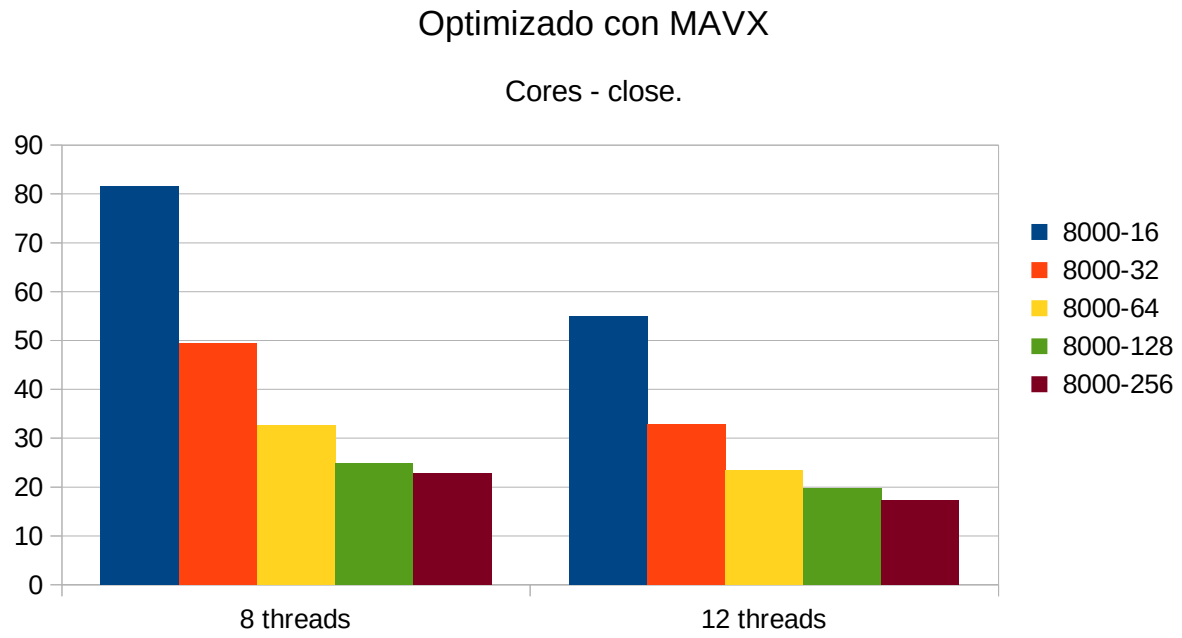
	128	256	512	Speedup 128	Speedup 256	Speedup 512
Secuencial	0.329565	0.3092962	0.2842876	1	1	1
2 threads	0.241086	0.2258234	0.209601	1.367001817	1.369637513	1.356327498
4 threads	0.167009	0.1408996	0.2356256	1.973336766	2.195153145	1.206522551
6 threads	0.1641686	0.152616	0.2483826	2.007478897	2.026630235	1.144555214
8 threads	0.1118442	0.1656744	0.255045	2.946643635	1.866891928	1.114656629
10 threads	0.1132782	0.1654396	0.25607	2.90934178	1.869541512	1.110194869
12 threads	0.1126558	0.1636972	0.2583236	2.925415292	1.889440992	1.100509593

El mejor tamaño de bloque a utilizar dependerá principalmente del tamaño de la cache local . En menor medida de la cantidad de hilos que se esta utilizando y el tamaño del problema. En este caso usar un tamaño de bloque de 512 presenta los peores resultados en todos los casos. Mientras que el tamaño de bloque de 128 presenta los mejores resultados al usar varios hilos. El mejor rendimiento

se obtiene al usar 8 hilos con un tamaño de bloque de 128.

El tercer caso que se analiza es con matrices de tamaño 8000x8000. En este caso se utilizan tamaños de bloque de 16, 32, 64, 128 y 256. No se llevo a cabo el computo del programa secuencial con matrices 8000x8000.

PLACES:cores|BIND:close					
	8000-16	8000-32	8000-64	8000-128	8000-256
8 threads	81.6084884	49.3755872	32.6216328	24.8031282	22.7495942
12 threads	54.9750632	32.7814406	23.3936902	19.6468332	17.3578688



Aquí los mejores resultados se obtienen con tamaños de bloque de 256, y no con tamaños menores, al usar una matriz de dimensiones mayores. También notamos que aumentar la cantidad de hilos mejora los tiempos de ejecución.

El mejor tamaño de bloques esta fuertemente ligado a la arquitectura de un sistema y el tamaño optimo variara entre distintas computadoras. Sobre todo se asocia con el tamaño de la cache. En general, los mejores resultados se han obtenido con un tamaño de 128 (Hay otro caso en la sección a continuación).

Cantidad de hilos

Aquí se multiplica matrices 4000x4000 con tamaño de bloque 128. El código se compilo con la bandera de optimización y se utiliza los registros mavx.

Threads	Cores, close	Secuencial	Speedup
2 threads	11.4186892	20.5158344	1.79668909808
4 threads	6.036821		3.398450012018
6 threads	4.7883886		4.284496542323
8 threads	3.2846972		6.245882999504
10 threads	3.3014698		6.214151769615
12 threads	2.5364866		8.088288106864

A medida que se utilizan mas hilos en este tamaño de matriz hay mejoras en el tiempo de ejecución.

4000	256	OPT-MAVX
Threads	Cores, close	Speedup
Secuencial	19.5237794	1
2 threads	10.524237	1.855125402
4 threads	5.5449178	3.521022331
6 threads	4.3159568	4.52362716
8 threads	3.0093166	6.487778454
10 threads	3.0265484	6.450839973
12 threads	2.9975128	6.513326449

Como se menciono anteriormente, al usar mayor cantidad de hilos se obtiene mejores resultados con bloques de 128. En ambos casos, hasta ocho hilos el speedup es considerable.

En el primer caso al usar ocho o diez hilos no hay cambios, pero al usar doce si.

En el segundo caso, no hay cambios considerables al usar mas que ocho hilos.

La cantidad de hilos óptimos a utilizar depende del tamaño del problema. Para problemas pequeños puede ser mejor hacer un programa secuencial. Y mientras mayor es un problema, mayor cantidad de hilos se puede usar. Sin exceder la cantidad de núcleos físicos.

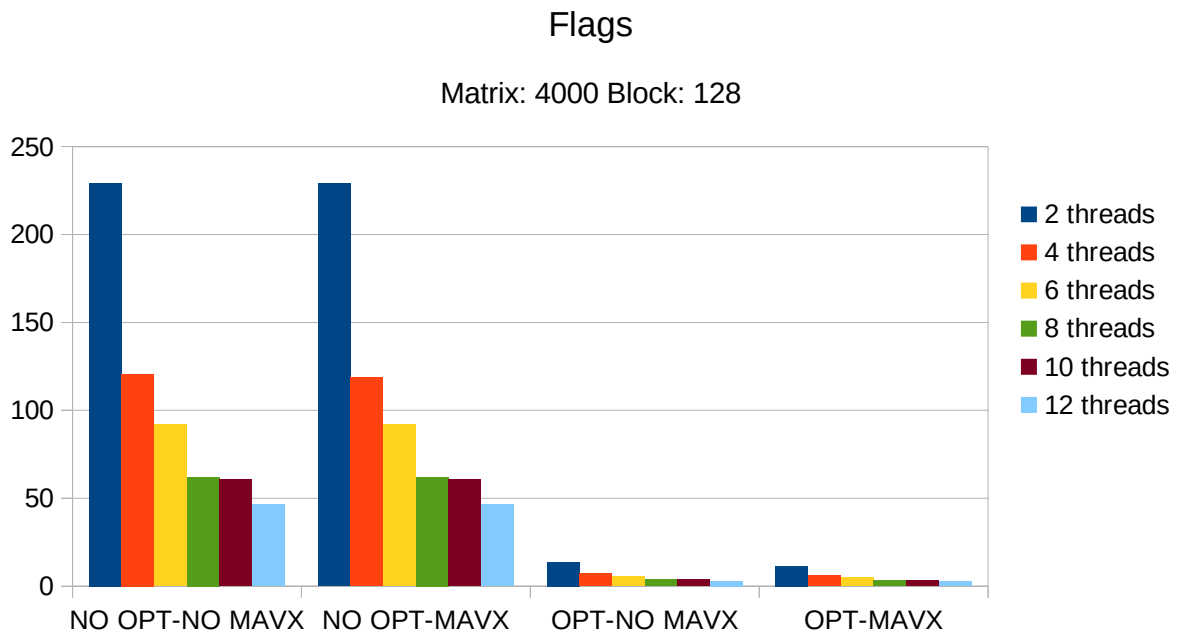
Banderas de compilación

Matriz 4000x4000 con tamaño de bloque de 128.

	NO OPT-NO	NO OPT-MAV	OPT-NO MAVX	OPT-MAVX
2 threads	229.1872478	228.982343	13.6812526	11.4186892
4 threads	120.5169276	118.7724724	7.183581	6.036821
6 threads	91.9488732	91.8259088	5.6827774	4.7883886
8 threads	61.9870918	62.1922272	3.8888228	3.2846972
10 threads	60.7767304	60.7233484	3.8783206	3.3014698
12 threads	46.4618122	46.4490032	2.9720958	2.5364866

De todos los casos analizados, aquello que tiene mayor peso es la utilización de la bandera de optimización. Mejorando el rendimiento del programa en mas de un 1000%.

Por otro lado el uso de registros mavx no es tan significativo al usar varios hilos como lo era en el programa secuencial. Aun así, el mejor rendimiento se da al combinar la bandera de optimización y los registros mavx.



Afinidad

Matriz 2000x2000 con tamaño de bloque de 256.

	cores, close	cores, spread	threads, false
2 threads	28.6093084	28.663629	28.6793584
4 threads	14.8966492	14.541887	14.6413516
6 threads	14.95824	14.7043848	14.7155926
8 threads	7.8108586	7.7906348	7.5908494
10 threads	7.8090434	7.7389872	7.5613104
12 threads	7.8037238	7.8148992	7.671415

Matriz 4000x4000 con tamaño de bloque de 128.

OPT, MAVX			
	cores, close	cores, spread	threads, false
2 threads	11.4186892	11.815281	12.06729
4 threads	6.036821	5.9824018	5.9874282
6 threads	4.7883886	4.6701586	4.675512
8 threads	3.2846972	3.3322692	3.2053252
10 threads	3.3014698	3.2041644	3.3201132
12 threads	2.5364866	2.5330012	2.5333224

Matriz 4000x4000 con tamaño de bloque de 256.

NO OPT-NO MAVX			
	cores, close	cores, spread	threads, false
2 threads	229.127389	229.2365186	229.9066336
4 threads	119.716658	117.0170438	116.2899706
6 threads	91.9220734	89.2516886	88.3689532
8 threads	61.776725	61.6043754	59.7776412
10 threads	61.8370652	60.1248154	60.6293822
12 threads	60.8054252	60.694425	60.238109

En ultimo lugar, se analiza la afinidad. En este caso se muestran varias afinidades distintas.

No ha habido cambios significativos con distintas afinidades. Esto ocurre ya que el pragma for se encuentra al principio de los bucles, la división de los datos entre hilos se hace en bandas. Por lo tanto, hay poca localidad espacial.

También se hicieron dos afinidades manuales para aprovechar mejor la arquitectura del procesador de la computadora “Pulqui”. Las llamaremos P1 y P2.

P1="{0}, {3}, {1}, {4}, {2}, {5}, {6}, {9}, {7}, {10}, {8}, {11}"

P2="{0}, {3}, {6}, {9}, {1}, {4}, {7}, {10}, {2}, {5}, {8}, {11}"

Bind=close

Estas afinidades tienen la ventaja que dos core consecutivos comparten cache de nivel 2.

P1 llena una pastilla primero, mientras que P2 va intercalando de pastilla cada dos hilos.

Se probo con matrices 4000x4000 con tamaño de bloque 128 y usando bandera de optimización y registros mavx. Ademas, esta vez se utilizo un scheduling de 4.

4000	128	O3	MAVX
	P1	P2	FALSE
4 threads	10.0014912	11.9795538	11.046751
8 threads	7.5007608	7.3964708	8.6344896
12 threads	8.4565004	8.3521934	8.2236404

Nuevamente, no hubo diferencias significativas entre usar distintas afinidades. Hay que tener en consideración que en el momento de realizar estas últimas prueba alguien más estaba utilizando el cluster.

Se probo usar distintas afinidades en el cluster Rocky usando un scheduling de 10 y tampoco hubo resultados significativos.

4000	256	O3	NO MAVX
	sockets, true	cores, true	FALSE
2 threads	30.1161698	30.2820252	29.931211
4 threads	31.4362844	30.6393628	31.5715942
8 threads	32.208604	31.1329742	31.3227634

Recién se comienzan a ver diferencias al cambiar el orden los indices del bucle. Se pone primero las columnas, por lo tanto al hacer la división de los datos en hilos no se hace por filas, sino que por columnas.

Aquí se uso un scheduling de 4.

4000	128	O3	MAVX
	P1	P2	FALSE
4 threads	9.2715724	9.4243638	10.7237654
8 threads	5.3945012	5.3588954	6.5679974
12 threads	5.4159368	5.5109816	6.1588108

Por ultimo se utiliza matrices de 8000x8000, con bloques de 128, con bandera de optimización y registros mavx. Se usa un scheduling de 4 y se cambia los indices de lugar para poner las columnas primero. Estos son los resultados de la afinidad.

8000	128	O3	MAVX
	P1	P2	FALSE
4 threads	73.852846	74.595436	79.5559252
6 threads	55.333225	55.1841294	60.387535
8 threads	40.6696418	40.840699	44.9702402
12 threads	38.4464636	38.355881	43.838204

Los peores tiempos son cuando hay enlazamiento falso. Las dos afinidades manuales tienen tiempos de ejecución similares.

Fallas de Cache

Considerar que al hacer estas pruebas alguien más estaba utilizando el cluster “pulqui”. Pero como el análisis se pidió sobre esta computadora, y las configuraciones optimas varían entre distintas computadoras (como la mejor afinidad o el tamaño de bloque óptimo). También, para mantener uniformidad entre los datos recolectados. Se opto por seguir tomando los datos en la computadora “pulqui”, a pesar que estos no podrán ser completamente fieles ya que había otros procesos que consumían muchos recursos ejecutándose al mismo tiempo.

Aclaro que estos procesos que estoy mencionando acá se han estado ejecutando durante la última semana y media, no he tenido oportunidad de encontrar el cluster libre mientras ejecutaba las últimas pruebas.

Algunas aclaraciones: Cuando hay una falla de cache significa que se busca el dato en el nivel superior de memoria, y cada nivel lo busca en el superior. Por ejemplo, si hay una falla en la cache de nivel 1, esta busca el dato en la cache de nivel 2, si esta no lo tiene, busca el dato en la cache de nivel 3. También ocurre del mismo modo con la memoria RAM y el disco duro.

Además, las cargas y fallas de cache en niveles superior tienen mayor impacto por unidad, ya que a mayor nivel de memoria más lentas son.

Por último, acá analizaremos las cargas y fallas de cache pero en el rendimiento de un programa hay otros factores de importancia como la cantidad de instrucciones, cambios de contexto, cargas de ramas o fallas de ramas (*branches*), entre otros.

Se recolectaron datos con el comando:

perf stat -d ./programa

Optimización

Datos recolectados para multiplicación de matrices de 4000x4000 con bloques de 128.

Con banderas -O3 y -mavx

O3	MAVX	
4.78E+10	L1-dcache-loads	
5.26E+09	L1-dcache-load-misses	11.01% of all L1-dcache hits
6.03E+08	LLC-loads	
4.66E+07	LLC-load-misses	7.73% of all LL-cache hits
4.83E+00	Seconds time elapsed	
	MAVX	
1.67E+12	L1-dcache-loads	
4.98E+09	L1-dcache-load-misses	0.30% of all L1-dcache hits
9.40E+08	LLC-loads	
8.20E+07	LLC-load-misses	8.72% of all LL-cache hits
9.27E+01	Seconds time elapsed	

Se observa que al usar la bandera de optimización hay dos ordenes menos de cargas de cache de nivel 1.

Ademas hay dos tercios menos de cargas de cache de nivel 3, y la mitad de fallas de cache de nivel 3.

El tiempo de ejecución es considerablemente mejor usando la bandera de optimización. Del mismo modo, esto se relaciona en gran parte con el mejor manejo de la cache.

Tamaños de bloque

En esta ocasión los mejores resultados se obtienen con un tamaño de bloque de 128.

Si bien el tamaño 128 tiene mas fallas de cache de L1 que el tamaño 64, tiene menos cargas de cache.

También, tiene un orden menos de cargas de cache de nivel 3 que los bloques de 256. A pesar, de que los bloques de 256 este tenga menos fallas de cache de L1.

Hay que tener en cuenta que mientras de mayor nivel sea la cache o la memoria en general, más impacto tiene la cantidad de cargas que se hace en esta. Ya que las cache de mayor nivel (L3 es de mayor nivel que L1) son más lentas.

4000	O3-MAVX
64	
6.33E+10	L1-dcache-loads
2.30E+09	L1-dcache-load-misses
1.36E+08	LLC-loads
6.05E+07	LLC-load-misses
5.53E+00	Seconds time elapsed
128	
4.78E+10	L1-dcache-loads
5.16E+09	L1-dcache-load-misses
1.92E+08	LLC-loads
2.59E+07	LLC-load-misses
4.74E+00	Seconds time elapsed
256	
3.97E+10	L1-dcache-loads
4.70E+09	L1-dcache-load-misses
2.67E+09	LLC-loads
2.63E+07	LLC-load-misses
5.38E+00	Seconds time elapsed

Al descubrir que el tamaño de bloque de 128 es más eficiente en términos de cache, se probo con tamaños de bloque cercanos a 128 para analizar los resultados de estos. El problema es que como hay otros procesos ejecutándose en el cluster, los resultados no son muy fiables. Lo optimo es hacer estas pruebas sin que haya otros procesos demandantes ejecutándose.

También es importante analizar la cache de nivel 2.

4000	O3-MAVX
120	
4.87E+10	L1-dcache-loads
5.24E+09	L1-dcache-load-misses
1.41E+08	LLC-loads
2.73E+07	LLC-load-misses
128	
4.78E+10	L1-dcache-loads
5.16E+09	L1-dcache-load-misses
1.92E+08	LLC-loads
2.59E+07	LLC-load-misses
150	
4.77E+10	L1-dcache-loads
5.08E+09	L1-dcache-load-misses
3.54E+08	LLC-loads
2.58E+07	LLC-load-misses

168	
4.37E+10	L1-dcache-loads
4.98E+09	L1-dcache-load-misses
4.47E+08	LLC-loads
2.76E+07	LLC-load-misses
200	
4.17E+10	L1-dcache-loads
4.83E+09	L1-dcache-load-misses
1.60E+09	LLC-loads
2.18E+07	LLC-load-misses

Al llegar a bloques de tamaño 200 el orden de cargas de cache L3 aumenta en un orden. Es probable que el optimo se encuentre entre 100 y 200. Para esto habría que hacer un análisis fino, con varias iteraciones y tomando promedios; sin factores externos que afecten el uso de las cache y de los procesadores.

Afinidades

En el primer caso, se usaron matrices 4000x4000. Aquí los ordenes de cargas y fallas son los mismos. Pero la afinidad ***cores-close*** tiene menos cargas y fallas de cache L3 que las otras afinidades. Menos de un tercio de la cantidad de cargas de cache de nivel 3.

cores, close	cores, spread	FALSE	4000-128 O3-MAVX
4.78E+10	4.78E+10	4.77E+10	L1-dcache-loads
5.17E+09	5.17E+09	5.18E+09	L1-dcache-load-misses
1.88E+08	6.60E+08	7.07E+08	LLC-loads
2.53E+07	3.18E+07	3.53E+07	LLC-load-misses

En el segundo caso, con matrices 8000x8000 el orden es el mismo. Pero los números de cargas y fallas en cache son más elevados en la afinidad ***falsa***. Mientras que repartir entre cores o sockets es similar. Aunque, nuevamente, el uso de ***cores-close*** tiene los mejores resultados (menor cargas y fallas de cache L3).

cores, close	sockets, close	FALSE	8000-128 O3-MAVX
3.80E+11	3.80E+11	3.79E+11	L1-dcache-loads
4.24E+10	4.27E+10	4.28E+10	L1-dcache-load-misses
1.63E+09	2.95E+09	3.66E+09	LLC-loads
2.03E+08	2.74E+08	3.33E+08	LLC-load-misses

Conclusión

Usar la librería openmp para usar multi-hilos en un programa puede acelerar el tiempo de ejecución del mismo. Si el tamaño del problema es chico, usar varios hilos puede empeorar el rendimiento ya que el uso de estos tiene un overhead. Además se puede aumentar las fallas de cache y esto tiene un impacto importante en el rendimiento. A mayor tamaño de problema y si no hay dependencia entre los datos, mejor rendimiento se tendrá al usar varios hilos.

Las bandera de optimización mejorará considerablemente el rendimiento del programa. Debe ser una bandera que siempre se utilice. Mientras que la bandera mavx también puede traer mejoras, que serán mas significativas en programas secuenciales. Aunque hay que ser precavido ya que los registros mavx no están disponibles en todas las computadoras.

La cantidad de hilos que se utilicen esta directamente ligado al tamaño del problema. Mientras que el tamaño de los bloques en una multiplicación de matrices de seis bucles esta relacionado al tamaño de la cache local. El mejor tamaño de bloque se puede estimar a partir del tamaño de la cache, pero para obtener una mejor aproximación en la práctica hay que hacer pruebas con distintos tamaños.

En el problema que se dio a resolver, donde el mayor tamaño de matriz es de 8000x8000, la afinidad no tiene un rol importante en el rendimiento, siempre y cuando los indices del bucle estén ordenados de manera eficiente.

Bibliografía

http://www.netlib.org/utk/papers/autoblock/node2.html

http://www.cs.ucsb.edu/~gilbert/cs140/notes/ComputationalIntensityOfMatMul.pdf

http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-affinity.html

http://cml06.engr.ucdavis.edu/~sumeet/ecs231/ECS231_2016_Assignment2.pdf

http://dc-vis.irb.hr/repository/2014/Optimal%20Block%20Size%20for%20Matrix%20Multiplication%20Using%20Blocking.pdf