Informe práctico 1

Maximiliano Bove - Mauricio Vanzulli
Curso: Computación de Propósito General en Unidades de Procesamiento Gráfico.
20 de abril de 2021

Plataforma computacional:

Todas las librerías de este trabajo se ejecutaron en un sistema operativo Linux con arquitectura de 64 bits en un procesador i5-8250U a una frecuencia del reloj de 1.6 GHz y una memoria RAM de 8 Gb.

Ejercicio 1

Para este apartado se implementaron las funciones de suma estática por fila y columna, etiquetadas con los nombres *suma_est_fil* y *suma_est_col*. Estas computan las operaciones correspondientes a las sumas de todas las filas de la matriz y de todas sus columnas, sin retornar el resultado. Además, se extendieron estas funciones de forma dinámica, permitiendo la variación de recursos utilizados en memoria, haciendo uso de punteros simples. Estas funciones llevan el nombre de *suma_din_fil* y *suma_din_col* para la suma por filas y columnas de la matriz respectivamente. Complementario a esto se implementó la función *suma_din_rand* que accede de forma aleatoria a las entradas de la matriz verificando que no se repitan las celdas que se sumaron anteriormente. Esta forma caótica de acceso a memoria no permite al procesador predecir el comportamiento y ejecutar un *cache-prefetch* de optimización anticipada.

Resultados experimentales

La cantidad de operaciones y tiempos de ejecución en milisegundos para diferentes tamaños de una matriz cuadrada N x N considerando sumas estáticas se presenta en la siguiente tabla:

N	*t_suma_est_fil*	*t_suma_est_col*	*t_fil_est* / N^2	*t_col_est* / N^2
128 (2^7)	0.09	0.09	549	549
256 (2^8)	0.38	0.39	579	595
512 (2^9)	1.52	1.53	579	583

Cuadro 1: Tiempos de ejecución en ms y GFlops para las funciones suma dinámicas.

Análogamente al considerar sumas dinámicas se obtuvieron los siguientes resultados:

N	*t_suma_din_fil*	*t_suma_din_col*	*t_suma_din_rand*
128 (2^7)	0.09	0.09	03.78
256 (2^8)	0.37	0.38	19.00
512 (2^9)	1.49	1.51	87.99
1024 (2^{10})	6.00	7.95	675.29
2048 (2^{11})	24.11	47.02	4403.93
4096 (2^{12})	95.88	230.86	17342.75

Cuadro 2: Tiempos de ejecución en ms y GFlops para las funciones suma dinámicas.

Los tiempos de ejecución de las anteriores rutinas se obtuvieron a partir del *template* disponible en la plataforma EVA. Debido a las incertidumbres inherentes al procesamiento de datos realizado por el procesador que son del orden de milisegundos las medidas de esta magnitud poseen un excesivo error relativo. Para disminuir el error porcentual el código se ejecuto sucesivas veces promediando el tiempo de ejecución.

Análisis de resultados

A continuación se describen las diferencias en términos de tiempos de ejecución para las matrices de suma estática, por filas y columnas, presentados en la Tabla 1. En primer lugar se observa como el tiempo de ejecución aumenta con el tamaño de las matrices, ya que se ve incrementada la cantidad de accesos a memoria. Este costo de sustraer las variables de memoria es de mayor orden que el de realizar una operación algebraica.

Existe una relación de orden N^2 al realizar la suma por filas para los tamaños 256 y 512. No obstante, esta relación no se cumple para un tamaño de 128 utilizando la misma función y tampoco al realizar la suma por columnas. Además, para los mismos valores de N el tiempo de ejecución aumenta levemente al sumar por columnas respecto a sumar por filas. Este comportamiento se observa a partir de tamaños de 256 con diferencias de 1 ms. Una explicación para este resultado se puede atribuir a la forma consecutiva en que el procesador accede a memoria con los punteros simples. Al realizarse por filas, el acceso a memoria es de manera secuencial mientras que al sumarse por columnas el procesador debe intercalar los indicies de acceso, lo que conlleva a un mayor tiempo de ejecución.

Con relación a la Tabla 2 se aprecia un aumento del tiempo de ejecución al aumentar el tamaño de la matriz para todas las funciones implementadas. Contrastando la suma dinámica por filas y por columnas se evidencia, al igual que para el caso estático, el mayor costo computacional que implica acceder a la matriz por columnas, nuevamente debido al salto de casillas en las lineas del cache. Además, esta diferencia acentúa notoriamente al aumentar el tamaño de la matriz.

Al cotejar los resultados de *performance* arrojados por la función *t_suma_din_rand* en comparación con el resto de las rutinas, se observa como el costo computacional aumenta de manera considerable. Esto se debe a que el procesador no es capaz de anticipar los índices de acceso a memoria, y por lo tanto ejecutar el *cache prefetch* que le permita optimizar el acceso a partir de un cierto patrón. En este caso se accede de manera aleatoria y no constante, lo que conlleva a un aumento significativo en el tiempo de ejecución.

Ejercicio 2

En este ejercicio se implementaron diferentes funciones para multiplicar dos matrices ($C = A * B$) cuadradas de tamaño N x N. En primer lugar la función *mult_simple* realiza el producto de matrices usual accediendo a B por columnas y a A por filas. Otra lógica de para consumar el producto de matrices consiste en acceder a ambas por filas e ir realizando sumas parciales; esto se implementó en la función *mult_fila*. Cuando las matrices son excesivamente grandes podrían exceder el espacio en memoria requerido para su producto, una forma de solucionar el problema es subdividirla por bloques y realizar el producto de estos. Esta técnica, accediendo de la forma tradicional y por filas, se implementó en las funciones *t_mult_bl_simple* y *t_mult_bl_filas* respectivamente.

Resultados

En primer lugar se calcularon los tamaños máximos que soporta es capaz de almacenar cada *cache*. Este umbral se encuentra determinado por el tamaño máximo en bytes de cada *cache* indicado con el nombre $T(c_i)$. A su vez, cada matriz posee una entrada tipo *double* que ocupa un tamaño de 8 bytes y se deberán almacenar las tres matrices A, B y C. Dadas estas variables el tamaño máximo de matriz N_{max} para el $cache_i$ esta dado por la siguiente expresión:

$$N^i_{max} = \sqrt{\frac{\sum_1^i T(c_i)}{3 \text{ sizeof(double)}}}. \tag{1}$$

Aplicando la Ecuación (1) para los tres caché se obtuvieron los siguientes resultados:

	L_1 I	L_1 D	L_2	L_3
Tb_c	32768	32768	262144	6291456
N_{max}	36	36	116	525

Cuadro 3: Tamaño de máximo de matrices para cada *cache*.

En función de los valores máximos explicitados en la Tabla 5 se aplicaron estos criterios a los tamaños de matrices descritas en la letra. Los resultados obtenidos se expresan en la tabla a continuación:

N	3 N^2 sizeof(double)	Encaje *cache*
2^5 32	24576	Entra en L1
2^6 64	98304	Entra en L2+L1
2^7 128	393216	Entra en L3+L2+L1
2^5 1024	25165824	Excede *cache*
2^5 1152	31850	Excede *cache*
2^5 2048	100663	Excede *cache*

Cuadro 4: Encaje de las matrices en los diferentes *cache*.

A continuación se presentan los resultados obtenidos para el análisis de las diferentes funciones de multiplicación implementadas variando el tamaño de las matrices y sus bloques. Para esto se utilizó la herramienta *perf* ejecutando el siguiente comando:

```
sudo perf stat -g -e instructions,cycles,L1-dcache-loads-L1-dcache-loads-misses
```

A partir de esta ejecución se relevaron el tiempo de ejecución en milisegundos t_ms, la cantidad de operaciones por segundo (GFlops), Las instrucciones por Ciclo del procesador y el porcentaje de *cache-misses* y *cache-misses-L1*. Los datos obtenidos para todas las funciones se encuentra en la Sección .

Se graficaron los resultados con el objetivo de ilustrar las relaciones entre las diferentes variables de forma pictórica. En primer lugar se estudió el vínculo que existe entre el tamaño de las matrices para la función *mult_simple* encontrándose una relación cúbica (verificada con el ratio $\frac{t}{N^3}$) tal y como se muestra en la figura que prosigue:

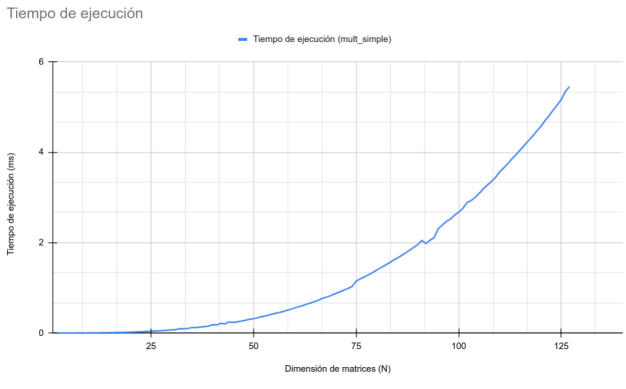

Figura 1: Relación tamaño de matriz y tiempo de ejecución.

Los resultados de la Figura 1 se vincularon a la cantidad de operaciones en memoria que requiere la multiplicación tradicional de matrices es N^3. Llevando acabo N veces lectura de datos por columnas de B y N veces columnas de A para luego recorrer N veces las celdas de C.

Posteriormente se investigaron las respuestas en términos de desempeño para las funciones de multiplicación variando los tamaños N de la matriz, hasta 1152. Se analizaron las funciones *mult_simple* *mult_fila* según se muestra a continuación:

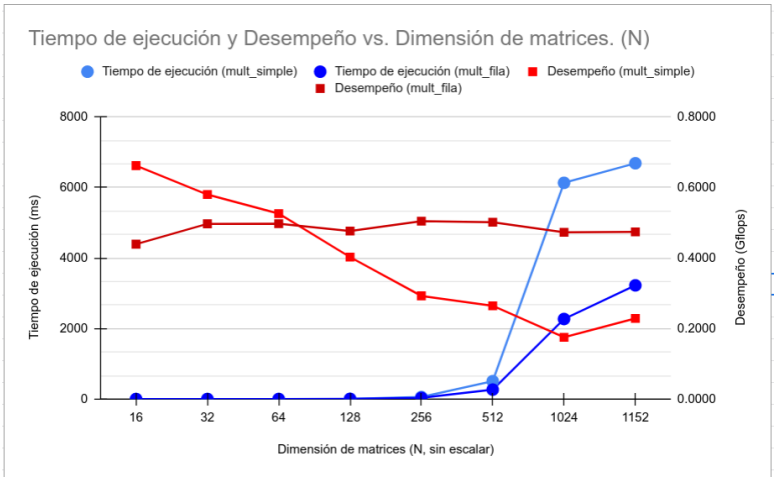

Figura 2: Estudios de desempeño en *mult_simple* *mult_fila*.

En la Figura 2 se puede confirmar como el tiempo de ejecución se incrementa al aumentar el tamaño de las matrices, de igual modo, la función de multiplicación por filas resuelve el problema en un menor tiempo de ejecución por los argumentos de secuencia de acceso postulados anteriormente. Este resultado también se evidencia al comparar los desempeños en GFlops de la multiplicación simple y por fila. Por otra parte, la multiplicación por filas presenta un desempeño constante mientras que la multiplicación simple decrece con el tamaño de la matriz. Otro comportamiento interesante se observa en el aumento de tiempos de ejecución y la caída de desempeño que se produce al pasar de $N = 512$ a $N = 1024$. Esto puede deberse al exceso del cache L3 según lo calculado en la Tabla 5.

Una vez analizadas las funciones simples se prosiguió a estudiar como afectan las distribución de bloques a las funciones de multiplicación por bloques en columnas y por filas. Esto se realizó para un tamaño de $N = 1152$ variando el tamaño del bloque en el vector $N_b = [16\ 18\ 32\ 64\ 128]$. Para esto se realizó un profiling de la función *mult_bl_simple* registrando la cantidad de GFlops y el tiempo de ejecución según plasmandose en la siguiente figura:

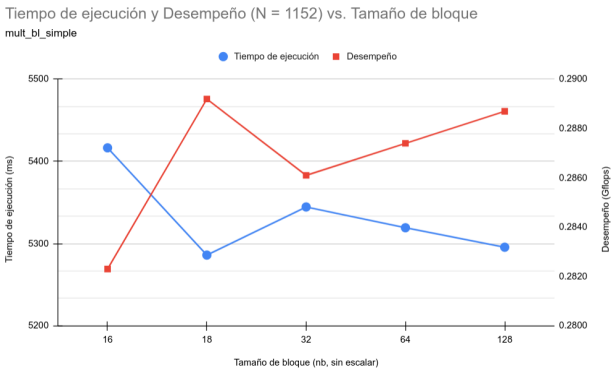

Figura 3: Estudios de desempeño en *mult_simple mult_fila.*

Análogamente se graficaron las mismas variables para la función *mult_bl_fil* obteniéndose la siguiente figura:

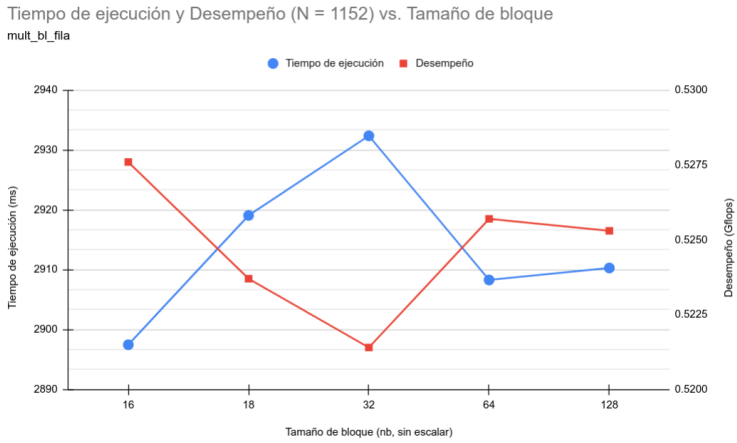

Figura 4: Estudios de desempeño en *mult_simple mult_fila.*

En las Figuras 3 y 4 se observa un leve incremento del desempeño para un tamaño de bloques de 18 en comparación con el resto de los casos. Esto se puede atribuirse al comando *aligned_alloc* que reserva un determinada cantidad de celdas en memoria para una variable aun no computada. Para el caso del código implementado se guardaron en unidades de 64 bytes.

Al considerar una matriz de N x N se tienen N^2 entradas, estas serán divididas en (N_b^2) elementos por bloques resultando un total de subdivisiones igual a $\frac{N_b^2}{N^2}$. A su vez, como cada casilla en memoria posee 64 bytes según lo reservado por el comando *aligned_alloc* se utilizarán un total de $\frac{N_b^2}{64N^2}$ casillas. Para el caso de una matriz de 1152 y los diferentes tamaños de bloques se tienen los siguientes valores:

N_b	N^2	N_b^2	N_b^2/N^2	N_b^2/N^264
16	1327104	256	496.5	7.75
18	1327104	324	4096	64
32	1327104	1024	1296	20.25

Cuadro 5: Número de accesos por linea de bloques en memoria.

Como el encaje no es entero el procesador dejará espacios vacíos (*padding*) en línea de memoria reservada disminuyendo así el desempeño computacional.

Anexo de resultados

Función *mult_simple*

mult_simple				
N	t_ms	GFlops	instructions / cycle	% L1 cache-misses
16	0.0062	0.6619	3.53	0.010%
32	0.0565	0.5799	3.04	0.070%
64	0.4985	0.5258	2.69	10.430%
128	5.2106	0.4025	2.04	52.140%
256	57.3429	0.2926	1.48	50.250%
512	507.4383	0.2645	1.34	49.700%
1024	6,131.0359	0.1751	0.89	51.040%
1152	6,685.8208	0.2287	1.16	49.840%
2048	100,077.0029	0.0858	0.44	80.570%

Figura 5: Métricas de la función *mult_simple*

Función *mult_fila*

mult_fila				
N	t_ms	GFlops	instructions / cycle	% L1 cache-misses
16	0.0093	0.4393	2.9	0.030%
32	0.0659	0.4969	3.19	0.100%
64	0.5273	0.4972	3.15	1.770%
128	4.4018	0.4764	3.01	6.220%
256	33.2682	0.5043	3.17	6.240%
512	267.6334	0.5015	3.14	6.250%
1024	2,270.2961	0.4730	2.95	6.200%
1152	3,224.2148	0.4742	2.97	6.220%
2048	18,170.9292	0.4727	2.96	6.510%

Figura 6: Métricas de la función *mult_fila*

Función *mult_ bl_ simple*

2^	N	Nb	mult_bl_simple				
			t (ms)	desempeño (GFlops)	instructions / cycle	L1 cache-misses (%)	cache-misses (%)
5	32	16	0.0909	0.3603	1.93	0.140%	19.904%
		32	0.0891	0.3677	1.96	0.140%	14.550%
		64					
		128					
6	64	16	0.6906	0.3796	1.96	11.610%	17.353%
		32	0.7076	0.3705	1.94	11,7%	10.959%
		64	0.6953	0.3770	1.96	11.490%	16.095%
		128					
7	128	16	5.8772	0.3568	1.81	50.960%	7.104%
		32	5.7711	0.3634	1.86	50.920%	0.370%
		64	5.7460	0.3650	1.85	50.910%	3.153%
		128	5.8119	0.3608	1.85	50.850%	1.489%
10	1024	16	9358.3188	0.1147	0.59	50.780%	64.709%
		32	7475.7729	0.1436	0.73	50.870%	42.094%
		64	6780.5420	0.1584	0.81	50.680%	30.052%
		128	5643.8201	0.1903	0.98	51.180%	9.308%
	1152	16	5416.5139	0.2823	1,14	50.000%	1.816%
		18	5285.9202	0.2892	1.47	50.080%	1.345%
		32	5344.5251	0.2861	1.46	49.870%	0.967%
		64	5319.2251	0.2874	1.47	49.870%	0.727%
		128	5295.5378	0.2887	1.48	49.820%	0.685%
11	2048	16	96795.7151	0.0887	0.45	80.520%	48.078%
		32	97657.8479	0.0880	0.45	80.480%	45.203%
		64	97764.6389	0.0879	0.45	80.600%	49.593%
		128	96698.7000	0.0888	0.45	80.390%	55.507%

Figura 7: Métricas de la función *mult_ bl_ simple*

Función *mult_ bl_ bl*

2^	N	Nb	mult_bl_fila				
			t (ms)	desempeño (GFlops)	instructions / cycle	L1 cache-misses (%)	cache-misses (%)
5	32	16	0.0599	0.5473	3.53	0.070%	15.753%
		32	0.0589	0.5561	3.60	0.100%	21.362%
		64					
		128					
6	64	16	0.4980	0.5264	3.42	0.610%	16.367%
		32	0.4767	0.5499	3.53	0,51%	15.965%
		64	0.4720	0.5554	3.52	1.890%	8.752%
		128					
7	128	16	4.3188	0.4856	3.06	0.950%	21.095%
		32	4.2655	0.4917	3.12	3.100%	21.698%
		64	4.2433	0.4942	3.14	6.130%	15.642%
		128	4.2598	0.4923	3.11	6.230%	15.710%
10	1024	16	2101.3870	0.511	3.18	6.500%	24.139%
		32	2093.2798	0.5129	3.19	6.340%	9.743%
		64	2144.8940	0.5006	3.12	6.270%	2.721%
		128	2171.0510	0.4946	3.09	6.220%	3.442%
	1152	16	2897.4631	0.5276	3.20	6.580%	17.126%
		18	2919.0708	0.5237	3.18	6.460%	13.679%
		32	2932.3879	0.5214	3.16	6.380%	4.656%
		64	2908.2891	0.5257	3,16	6.310%	2.671%
		128	2910.2908	0.5253	3.19	6.250%	2.247%
11	2048	16	16933.3220	0.5073	3.17	7.110%	7.618%
		32	16660.1570	0.5156	3.22	6.700%	3.523%
		64	17324.5649	0.4958	3.17	6.640%	4.018%
		128	16879.8049	0.5089	3.18	6.600%	5.279%

Figura 8: Métricas de la función *mult_ bl_ fila*