Multiplicación Matricial - Computación Paralela

Juan Sebastián Pachón Carvajal
Departamento de Sistemas e Industrial
Universidad Nacional de Colombia
Bogotá D.C., Colombia
jupachonc@unal.edu.co

Abstract*— This paper describes the process of parallelization of the classical matrix multiplication algorithm using different techniques to distribute the computational workload, since this is an iterative task with high complexity, parallelism helps to perform it in a shorter time making a better use of the processing components. For the parallelization process we will use the processor threads using OpenMP, the cores of the graphics cards with CUDA and a cloud cluster with OpenMPI.

Keywords*— Multiplicación, Matriz, Paralelización, Rostros, OpenMP, CUDA, MPI.

I. Introducción

La computación paralela es una técnica que consiste en la realización de varias tareas en paralelo, utilizando varios recursos de procesamiento, para aumentar la velocidad de ejecución. La técnica se basa en el principio según el cual, algunas tareas se pueden dividir en partes más pequeñas que pueden ser resueltas simultáneamente. Por ello es interesante emplear la computación paralela en diversos ámbitos que requieren una gran cantidad de cómputo y que resultan altamente demandante para su ejecución secuencial, como por ejemplo, en algoritmos interactivos de alta complejidad como la multiplicación matricial en la que hay que recorrer varias veces la misma matriz para determinar los resultados.

En el presente informe se implementan distintas formas de paralelismo bajo las cuales se busca mejorar el rendimiento de esta operación bajo distintas condiciones de prueba con matrices de distintos tamaños y utilizando distinta cantidad de nodos de procesamiento, pasando por implementaciones en procesador con memoria compartida, usando GPU a través de la plataforma CUDA y finalmente en procesador con memoria distribuida haciendo uso del paso de mensajes a través de red.

Para lograr este objetivo se buscará hacer una división de responsabilidad para cada nodo, repartiendo las celdas de la matriz resultado que debe calcular, ajustando a necesidad de cada tipo de paralelismo.

En principio se generan dos matrices aleatorias de *NxN* bajo las mismas condiciones con números tan grandes como *N* las cuales se almacenan en memoria y son las que posteriormente se operaran con el algoritmo de multiplicación matricial. El algoritmo de multiplicación maneja las matrices como arreglos unidimensionales donde se encuentran contiguas las filas de dimensión *N,* por lo que dado el índice de los elementos dentro de este arreglo se puede obtener las coordenadas *xy* correspondientes a la representación en dos dimensiones para aplicar correctamente la operación.

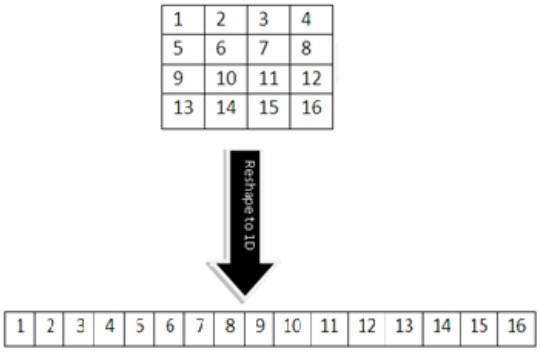

Figura 1. Representación de la matriz como arreglo

II. Desarrollo

El programa está escrito en C++ y hace uso de las librerías estándar para su funcionamiento de manera secuencial donde el algoritmo recorre uno a uno los índices unidimensionales de la matriz representada como arreglo donde una vez calculado sus correspondientes índices xy se recupera la columna *x* y la fila *y*, de manera iterativa con cada uno de los *N* elementos los cuales se multiplican y se suman a la variable acumuladora, finalmente una vez calculado el resultado de la celda dada se asigna en la matriz resultado.

A. Generación de matrices

La generación de matrices se hace de manera unidimensional para que sean compatibles con el algoritmo de multiplicación, para esto se reserva el espacio de enteros de tamaño *NxN* y en cada espacio se asignará un entero aleatorio determinado con la función rand() y acotado con *N,* donde se

toma como semilla el tiempo actual del sistema para la generación.

B. *Paralelización en OMP*

El proceso de paralelización usando OpenMP se basa en la distribución de la ejecución del ciclo for que se encarga de calcular cada elemento de la matriz, para ello se calcula una partición del total de elementos de la matriz entre el número de hilos que se tienen disponibles para la ejecución, se busca siempre el entero mayor más cerca al valor de esta partición para garantizar que todos los elementos son incluídos en la carga de los hilos, por lo que en caso de que el número de elementos no sea divisible por el número de hilos los datos se cubrirán completamente, esto garantiza que todos los hilos tengan una carga igual en el caso promedio y en el caso anormal el último hilo tendría una carga ligeramente menor al resto, sin embargo, al ser una diferencia pequeña no repercute en gran medida sobre el tiempo de ejecución por lo que no representa un retraso para la ejecución total del algoritmo.

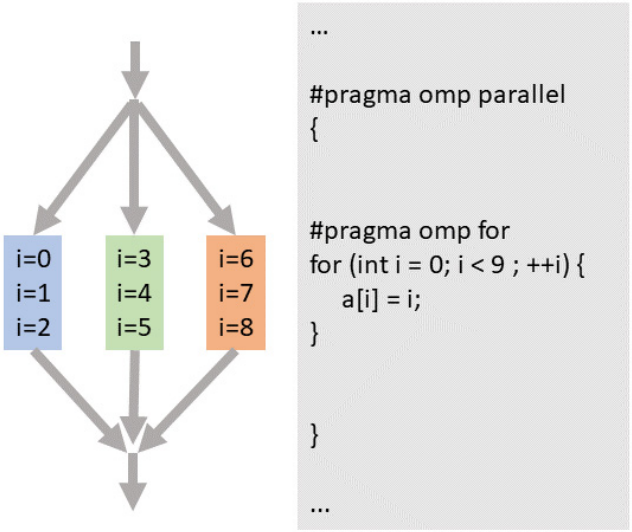

Figura 2. Paralelización OMP

C. *Paralelización en GPU*

Para la paralelización en GPU se sigue el mismo principio de particionamiento utilizado para OMP que es distribuir el ciclo que recorre el arreglo de datos que representan la matriz, sin embargo, para aprovechar de mejor manera las características propias de las GPU, el ciclo original se transforma en dos que recorren en dos dimensiones el arreglo.

Dado este particionamiento en dos ciclos se define una división tanto en bloques como en hilos, por lo que, entre los bloques se reparten las filas de la matriz y dentro de estos bloques se reparte la carga de cada elemento entre los hilos logrando así un particionamiento en dos dimensiones lo que aprovecha las cualidades de las GPU y mejora en gran medida el tiempo de procesamiento.

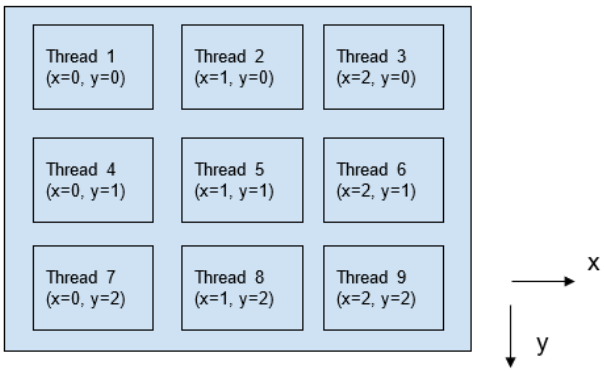

Figura 3. Paralelización CUDA

D. *Paralelización en MPI*

Para la paralelización usando el paso de mensajes se usó el mismo principio de particionamiento usado para OMP que es dividir el ciclo principal que calcula cada elemento de la matriz en el número de procesos disponibles para ejecución, por lo que cada proceso se encarga de una parte igual de elementos.

Los datos se pasan desde el proceso root, el cuál transfiere los datos de la matrices a cada uno de los procesos garantizando que cada proceso trabaja sobre los mismos datos, luego cada proceso guarda los resultados en un arreglo del tamaño de la partición que se le asignó y finalmente una vez terminado el cálculo en cada uno de los procesos se envían los datos calculados por cada proceso al proceso raíz el cuál los almacena en la matriz resultado.

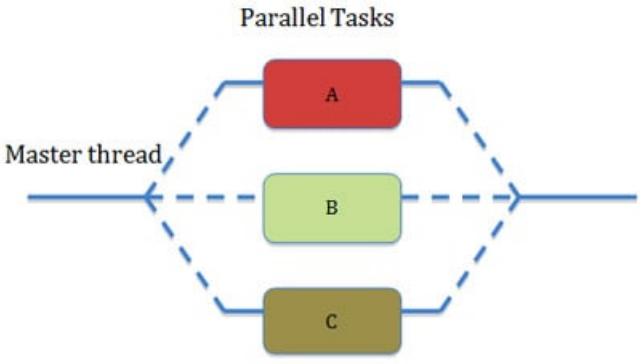

Figura 1. Paralelización MPI

III. Pruebas

Para las pruebas se hicieron varias ejecuciones para los distintos tipos de paralizaciones variando el tamaño de las matrices usando los siguientes valores.

- 8
- 16
- 32
- 64
- 128
- 256
- 512
- 1024

IV. Resultados OMP

Se obtuvieron los siguientes resultados al procesar las matrices

A. N = 8

Size	Threads	Time	SpeedUp
8	1	0,000011	1,000
8	2	0,000083	0,133
8	4	0,000141	0,078
8	6	0,000181	0,061
8	8	0,023168	0,000
8	16	0,003159	0,003

B. N = 16

Size	Threads	Time	SpeedUp
16	1	0,000033	1,000
16	2	0,000106	0,311
16	4	0,000122	0,270
16	6	0,018190	0,002
16	8	0,025805	0,001
16	16	0,000565	0,058

C. N = 32

Size	Threads	Time	SpeedUp
32	1	0,000106	1,000
32	2	0,000111	0,955
32	4	0,000157	0,675
32	6	0,000197	0,538
32	8	0,025671	0,004
32	16	0,000779	0,136

D. N = 64

Size	Threads	Time	SpeedUp
64	1	0,000827	1,000
64	2	0,000508	1,628
64	4	0,000547	1,512
64	6	0,015798	0,052
64	8	0,031702	0,026
64	16	0,002338	0,354

E. N = 128

Size	Threads	Time	SpeedUp
128	1	0,010241	1,000
128	2	0,006090	1,682
128	4	0,006696	1,529
128	6	0,017804	0,575
128	8	0,028742	0,356
128	16	0,004588	2,232

F. N = 256

Size	Threads	Time	SpeedUp
256	1	0,084313	1,000
256	2	0,039133	2,155
256	4	0,039851	2,116
256	6	0,048549	1,737
256	8	0,045822	1,840
256	16	0,035844	2,352

G. N = 512

Size	Threads	Time	SpeedUp
512	1	0,628136	1,000
512	2	0,343207	1,830
512	4	0,217993	2,881
512	6	0,207128	3,033
512	8	0,199002	3,156
512	16	0,172781	3,635

H. N = 1024

Size	Threads	Time	SpeedUp
1024	1	9,980951	1,000
1024	2	5,668151	1,761
1024	4	3,328831	2,998
1024	6	4,022914	2,481
1024	8	3,974278	2,511
1024	16	3,724093	2,680

V. Resultados GPU

Se obtuvieron los siguientes resultados en GPU al procesar las matrices en el entorno de ejecución proveído por Google Colab.

A. N = 8

Size	Blocks	Threads	Time	SpeedUp
8	1	1	0,000060	1,000
8	1	4	0,000040	1,500
8	1	16	0,000043	1,395
8	1	64	0,000040	1,500
8	1	256	0,000041	1,463
8	1	1024	0,000056	1,071
8	4	1	0,000044	1,364
8	4	4	0,000032	1,875
8	4	16	0,000038	1,579
8	4	64	0,000035	1,714
8	4	256	0,000040	1,500
8	4	1024	0,000049	1,224
8	16	1	0,000039	1,538
8	16	4	0,000031	1,935
8	16	16	0,000033	1,818
8	16	64	0,000031	1,935
8	16	256	0,000031	1,935
8	16	1024	0,000068	0,882
8	64	1	0,000032	1,875
8	64	4	0,000033	1,818
8	64	16	0,000033	1,818
8	64	64	0,000032	1,875
8	64	256	0,000038	1,579
8	64	1024	0,000064	0,938
8	256	1	0,000041	1,463
8	256	4	0,000032	1,875
8	256	16	0,000048	1,250

8	256	64	0,000044	1,364
8	256	256	0,000057	1,053
8	256	1024	0,000139	0,432
8	1024	1	0,000042	1,429
8	1024	4	0,000041	1,463
8	1024	16	0,000042	1,429
8	1024	64	0,000058	1,034
8	1024	256	0,000129	0,465
8	1024	1024	0,000434	0,138

B. N = 16

Size	Blocks	Threads	Time	SpeedUp
16	1	1	0,000205	1,000
16	1	4	0,000078	2,628
16	1	16	0,000049	4,184
16	1	64	0,000048	4,271
16	1	256	0,000050	4,100
16	1	1024	0,000065	3,154
16	4	1	0,000077	2,662
16	4	4	0,000043	4,767
16	4	16	0,000032	6,406
16	4	64	0,000035	5,857
16	4	256	0,000055	3,727
16	4	1024	0,000049	4,184
16	16	1	0,000041	5,000
16	16	4	0,000045	4,556
16	16	16	0,000032	6,406
16	16	64	0,000032	6,406
16	16	256	0,000039	5,256
16	16	1024	0,000048	4,271
16	64	1	0,000061	3,361
16	64	4	0,000036	5,694
16	64	16	0,000033	6,212
16	64	64	0,000031	6,613
16	64	256	0,000040	5,125
16	64	1024	0,000066	3,106
16	256	1	0,000048	4,271
16	256	4	0,000043	4,767
16	256	16	0,000033	6,212
16	256	64	0,000038	5,395
16	256	256	0,000059	3,475
16	256	1024	0,000139	1,475
16	1024	1	0,000048	4,271
16	1024	4	0,000050	4,100
16	1024	16	0,000068	3,015
16	1024	64	0,000056	3,661
16	1024	256	0,000132	1,553
16	1024	1024	0,000436	0,470

C. N = 32

32	1	4	0,000323	3,533
32	1	16	0,000115	9,922
32	1	64	0,000084	13,583
32	1	256	0,000083	13,747
32	1	1024	0,000097	11,763
32	4	1	0,000312	3,657
32	4	4	0,000109	10,468
32	4	16	0,000050	22,820
32	4	64	0,000041	27,829
32	4	256	0,000052	21,942
32	4	1024	0,000059	19,339
32	16	1	0,000108	10,565
32	16	4	0,000075	15,213
32	16	16	0,000042	27,167
32	16	64	0,000034	33,559
32	16	256	0,000042	27,167
32	16	1024	0,000051	22,373
32	64	1	0,000075	15,213

32	64	4	0,000041	27,829
32	64	16	0,000041	27,829
32	64	64	0,000034	33,559
32	64	256	0,000042	27,167
32	64	1024	0,000066	17,288
32	256	1	0,000075	15,213
32	256	4	0,000045	25,356
32	256	16	0,000045	25,356
32	256	64	0,000044	25,932
32	256	256	0,000060	19,017
32	256	1024	0,000144	7,924
32	1024	1	0,000074	15,419
32	1024	4	0,000049	23,286
32	1024	16	0,000041	27,829
32	1024	64	0,000061	18,705
32	1024	256	0,000132	8,644
32	1024	1024	0,000444	2,570

D. N = 64

Size	Blocks	Threads	Time	SpeedUp
64	1	1	0,007849	1,000
64	1	4	0,002041	3,846
64	1	16	0,000570	13,770
64	1	64	0,000207	37,918
64	1	256	0,000206	38,102
64	1	1024	0,000222	35,356
64	4	1	0,001998	3,928
64	4	4	0,000535	14,671
64	4	16	0,000173	45,370
64	4	64	0,000084	93,440
64	4	256	0,000085	92,341
64	4	1024	0,000097	80,918
64	16	1	0,000536	14,644
64	16	4	0,000166	47,283
64	16	16	0,000080	98,113
64	16	64	0,000051	153,902
64	16	256	0,000059	133,034
64	16	1024	0,000068	115,426
64	64	1	0,000175	44,851
64	64	4	0,000075	104,653
64	64	16	0,000052	150,942
64	64	64	0,000051	153,902
64	64	256	0,000054	145,352
64	64	1024	0,000084	93,440
64	256	1	0,000173	45,370
64	256	4	0,000092	85,315
64	256	16	0,000063	124,587
64	256	64	0,000058	135,328
64	256	256	0,000067	117,149
64	256	1024	0,000158	49,677
64	1024	1	0,000175	44,851
64	1024	4	0,000103	76,204
64	1024	16	0,000060	130,817
64	1024	64	0,000066	118,924
64	1024	256	0,000146	53,760
64	1024	1024	0,000453	17,327

E. N = 128

Size	Blocks	Threads	Time	SpeedUp
128	1	1	0,061508	1,000
128	1	4	0,022093	2,784
128	1	16	0,006457	9,526
128	1	64	0,001679	36,634
128	1	256	0,000836	73,574
128	1	1024	0,000865	71,108
128	4	1	0,015444	3,983
128	4	4	0,005614	10,956

128	4	16	0,001551	39,657
128	4	64	0,000451	136,381
128	4	256	0,000279	220,459
128	4	1024	0,000292	210,644
128	16	1	0,003925	15,671
128	16	4	0,001452	42,361
128	16	16	0,000447	137,602
128	16	64	0,000189	325,439
128	16	256	0,000138	445,710
128	16	1024	0,000147	418,422
128	64	1	0,001042	59,029
128	64	4	0,000459	134,004
128	64	16	0,000210	292,895
128	64	64	0,000114	539,544
128	64	256	0,000112	549,179
128	64	1024	0,000154	399,403
128	256	1	0,000685	89,793
128	256	4	0,000378	162,720
128	256	16	0,000154	399,403
128	256	64	0,000106	580,264
128	256	256	0,000112	549,179
128	256	1024	0,000209	294,297
128	1024	1	0,000682	90,188
128	1024	4	0,000413	148,930
128	1024	16	0,000165	372,776
128	1024	64	0,000113	544,319
128	1024	256	0,000180	341,711
128	1024	1024	0,000506	121,557

F. N = 256

Size	Blocks	Threads	Time	SpeedUp
256	1	1	0,355394	1,000
256	1	4	0,070872	5,015
256	1	16	0,021945	16,195
256	1	64	0,005712	62,219
256	1	256	0,001675	212,176
256	1	1024	0,001744	203,781
256	4	1	0,048618	7,310
256	4	4	0,017775	19,994
256	4	16	0,005665	62,735
256	4	64	0,001673	212,429
256	4	256	0,000582	610,643
256	4	1024	0,000845	420,585
256	16	1	0,021631	16,430
256	16	4	0,008256	43,047
256	16	16	0,002928	121,378
256	16	64	0,000972	365,632
256	16	256	0,000444	800,437
256	16	1024	0,000458	775,969
256	64	1	0,007641	46,511
256	64	4	0,003076	115,538
256	64	16	0,001392	255,312
256	64	64	0,000611	581,660
256	64	256	0,000371	957,935
256	64	1024	0,000415	856,371
256	256	1	0,003866	91,928
256	256	4	0,002163	164,306
256	256	16	0,002040	174,213
256	256	64	0,000621	572,293
256	256	256	0,000363	979,047
256	256	1024	0,000530	670,555
256	1024	1	0,003912	90,847
256	1024	4	0,002158	164,687
256	1024	16	0,001117	318,168
256	1024	64	0,000630	564,117
256	1024	256	0,000447	795,065
256	1024	1024	0,000783	453,888

G. N = 512

Size	Blocks	Threads	Time	SpeedUp
512	1	1	2,343970	1,000
512	1	4	0,549999	4,262
512	1	16	0,179826	13,035
512	1	64	0,085365	27,458
512	1	256	0,021999	106,549
512	1	1024	0,012318	190,288
512	4	1	0,542099	4,324
512	4	4	0,138604	16,911
512	4	16	0,045635	51,363
512	4	64	0,022008	106,505
512	4	256	0,006166	380,144
512	4	1024	0,003768	622,073
512	16	1	0,136217	17,208
512	16	4	0,035228	66,537
512	16	16	0,012157	192,808
512	16	64	0,008013	292,521
512	16	256	0,002579	908,868
512	16	1024	0,001608	1457,693
512	64	1	0,034695	67,559
512	64	4	0,009822	238,645
512	64	16	0,006701	349,794
512	64	64	0,003461	677,252
512	64	256	0,001654	1417,152
512	64	1024	0,001676	1398,550
512	256	1	0,024508	95,641
512	256	4	0,013068	179,367
512	256	16	0,011846	197,870
512	256	64	0,007789	300,933
512	256	256	0,002717	862,705
512	256	1024	0,001914	1224,645
512	1024	1	0,033274	70,444
512	1024	4	0,016763	139,830
512	1024	16	0,013844	169,313
512	1024	64	0,007892	297,006
512	1024	256	0,002573	910,987
512	1024	1024	0,002136	1097,364

H. N = 1024

Size	Blocks	Threads	Time	SpeedUp
1024	1	1	17,698921	1,000
1024	1	4	4,485062	3,946
1024	1	16	1,456573	12,151
1024	1	64	0,707003	25,034
1024	1	256	0,345132	51,282
1024	1	1024	0,097245	182,003
1024	4	1	4,317943	4,099
1024	4	4	1,102092	16,059
1024	4	16	0,381100	46,442
1024	4	64	0,185336	95,496
1024	4	256	0,087808	201,564
1024	4	1024	0,027225	650,098
1024	16	1	1,084022	16,327
1024	16	4	0,278430	63,567
1024	16	16	0,102180	173,213
1024	16	64	0,052759	335,467
1024	16	256	0,026159	676,590
1024	16	1024	0,008892	1990,432
1024	64	1	0,272551	64,938
1024	64	4	0,075905	233,172
1024	64	16	0,052362	338,011
1024	64	64	0,030553	579,286
1024	64	256	0,016178	1094,012
1024	64	1024	0,006357	2784,162
1024	256	1	0,107066	165,309
1024	256	4	0,046737	378,692

1024	256	16	0,041049	431,166
1024	256	64	0,027758	637,615
1024	256	256	0,015752	1123,598
1024	256	1024	0,006048	2926,409
1024	1024	1	0,095844	184,664
1024	1024	4	0,050443	350,870
1024	1024	16	0,040744	434,393
1024	1024	64	0,026429	669,678
1024	1024	256	0,015099	1172,192
1024	1024	1024	0,005947	2976,109

VI. Resultados MPI

Se obtuvieron los siguientes resultados al procesar las matrices con el programa paralelizado MPI, usando OpenMPI sobre el clúster de 4 nodos.

A. N = 8

Size	Processes	Time	SpeedUp
8	1	0,000011	1,000000
8	2	0,010735	0,001025
8	4	0,022533	0,000488

B. N = 16

Size	Processes	Time	SpeedUp
16	1	0,000024	1,000000
16	2	0,011035	0,002175
16	4	0,022065	0,001088

C. N = 32

Size	Processes	Time	SpeedUp
32	1	0,000118	1,000000
32	2	0,011034	0,010694
32	4	0,024305	0,004855

D. N = 64

Size	Processes	Time	SpeedUp
64	1	0,001023	1,000000
64	2	0,011629	0,087970
64	4	0,023954	0,042707

E. N = 128

Size	Processes	Time	SpeedUp
128	1	0,008564	1,000000
128	2	0,016021	0,534548
128	4	0,028265	0,302990

F. N = 256

Size	Processes	Time	SpeedUp
256	1	0,105024	1,000000
256	2	0,065060	1,614264
256	4	0,057770	1,817968

G. N = 512

Size	Processes	Time	SpeedUp
512	1	0,909463	1,000000
512	2	0,487727	1,864697
512	4	0,288604	3,151249

H. N = 1024

Size	Processes	Time	SpeedUp
1024	1	7,081757	1,000000
1024	2	3,673603	1,927742
1024	4	2,104877	3,364452

VII. Conclusiones

1. La paralelización de la operación matricial se puede ver evidenciada principalmente en matrices de gran tamaño dado que en matrices pequeñas los procesos de gestión de las tareas resultan en una ejecución más lenta que si se hace de manera secuencial.
2. La paralelización que da mejor resultado para la operación de multiplicación matricial es la que usa tarjetas gráficas CUDA, dado su gran cantidad de núcleos y que los datos y tareas pueden ser particionados en matrices equivalentes lo que nos garantiza un mejor tratamiento y rendimiento.
3. En general, los resultados con MPI son malos dado que el paso de mensajes ralentiza en gran medida el algoritmo, por lo que este tipo de implementación sólo es de gran utilidad cuando se haga en matrices con tamaños grandes.
4. A excepción de la implementación en CUDA se puede observar que el salto de velocidad o SpeedUp se da desde las matrices de tamaño 256.
5. El limitante de núcleos de procesamiento de las CPU se puede ver evidenciado en la ejecución dado que cuando se pasa el límite de núcleos físicos vemos una caída importante del rendimiento.
6. La diferencia de rendimiento entre CUDA y el resto de implementaciones es altísima y se evidencia como se hace un uso mucho más eficiente de los núcleos para realizar estas tareas aritméticas.
7. Para lograr una buena paralelización es necesario pensar en un algoritmo fácilmente divisible tanto a nivel de responsabilidad como a nivel de datos, por ello, la estructura de datos donde se almacenan los datos juega un papel fundamental, en el diseño secuencial del algoritmo para su posterior paralelización

References

[1] «La computación paralela: alta capacidad de procesamiento», Teldat Blog - Connectando el mundo, 14 de julio de 2020. https://www.teldat.com/blog/es/computacion-paralela-capacidad-procesamiento/ (accedido 29 de septiembre de 2022).

[2] P. Rodó, «Multiplicación de matrices», Economipedia. https://economipedia.com/definiciones/multiplicacion-de-matrices.html (accedido 28 de noviembre de 2022).

[3] «mxm_openmp». https://people.sc.fsu.edu/~jburkardt/c_src/mxm_openmp/mxm_openmp.html (accedido 28 de noviembre de 2022).

[4] «Matrix-Matrix Multiplication on the GPU with Nvidia CUDA | QuantStart». https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA/ (accedido 28 de noviembre de 2022).

[5] 262588213843476, «MPI Matrix Multiplication», Gist. https://gist.github.com/AshanthaLahiru/bfa1a631f6af05af93e98538eeca3018 (accedido 28 de noviembre de 2022).

[6] «Inicio - Universidad Nacional de Colombia». https://dis.unal.edu.co/~capedrazab/material_cursos/20222/pc/pc.html (accedido 28 de noviembre de 2022).

Anexos

- Enlace vídeo. https://drive.google.com/file/d/1B5xqNIW9l6XM3R2qQr_3WXbz17MR2zJI/view?usp=sharing