Trabajo Práctico Nº 1
Optimización de algoritmos secuenciales

Grupo 8:

20996/9 Hernández Corsi, Sebastián Alejandro

21583/5 Martinez Osti, María Josefina

Sistemas paralelos- 2025

Características del hardware y del software usados:
Equipo hogareño:

- OS: fedora 41 (workstation edition)
- Procesador: AMD Ryzen 5 5500U
- RAM: 8 GB DDR4
- M.2 256 GB
- version GCC 14.2.1
- Usado mientras carga

Cluster Multicore (partición Blade): Es un cluster conformado por 16 nodos. Cada nodo posee 8GB de RAM y 2 procesadores Intel Xeon E5405 de cuatro 4 cores cada uno que operan a 2.0GHz

Consideraciones:
Compilamos los algoritmos con -O3 ya que fue el que mejores resultados nos dio al realizar la práctica 1.

1. Resuelva el ejercicio 4 de la Práctica Nº 1 usando dos equipos diferentes: (1) cluster remoto y (2) equipo hogareño al cual tenga acceso con Linux nativo (puede ser una PC de escritorio o una notebook).

Ejercicio 4: Dada la ecuación cuadrática: $x^2 - 4.0000000\ x + 3.9999999 = 0$, sus raíces son $r_1 = 2.000316228$ y $r_2 = 1.999683772$ (empleando 10 dígitos para la parte decimal).

a. El algoritmo quadratic1.c computa las raíces de esta ecuación empleando los tipos de datos float y double. Compile y ejecute el código. ¿Qué diferencia nota en el resultado?

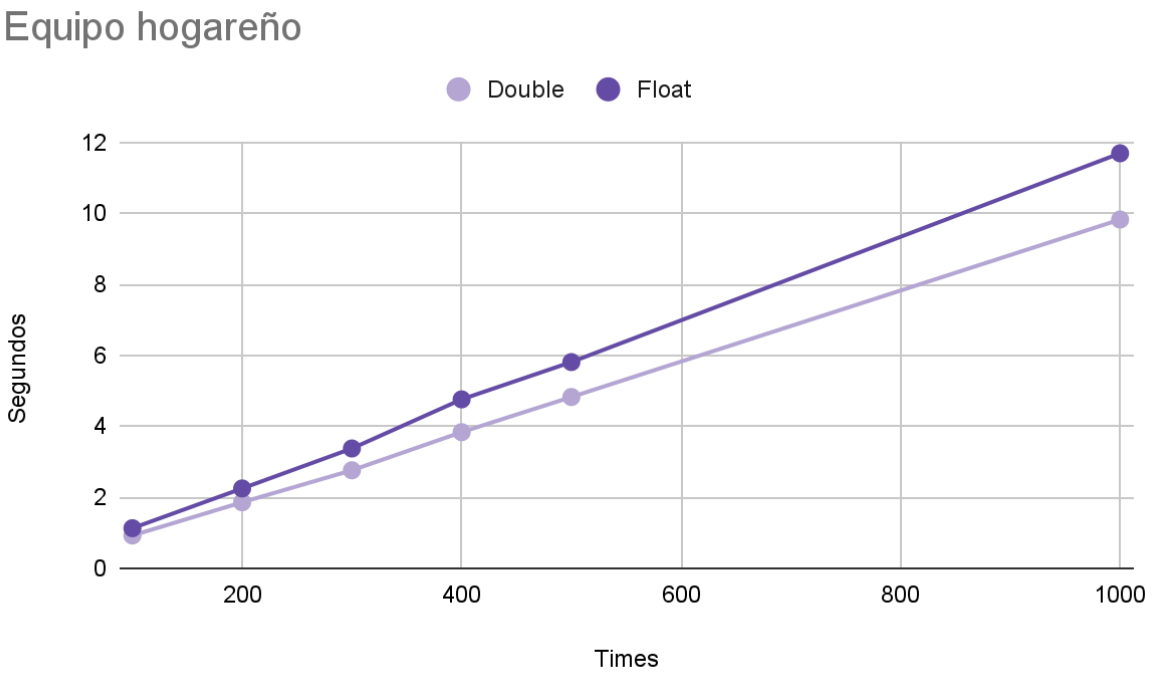
Equipos	Float		Double	
Equipo hogareño	2.00000	2.00000	2.00032	1.99968
Cluster blade	2.00000	2.00000	2.00032	1.99968

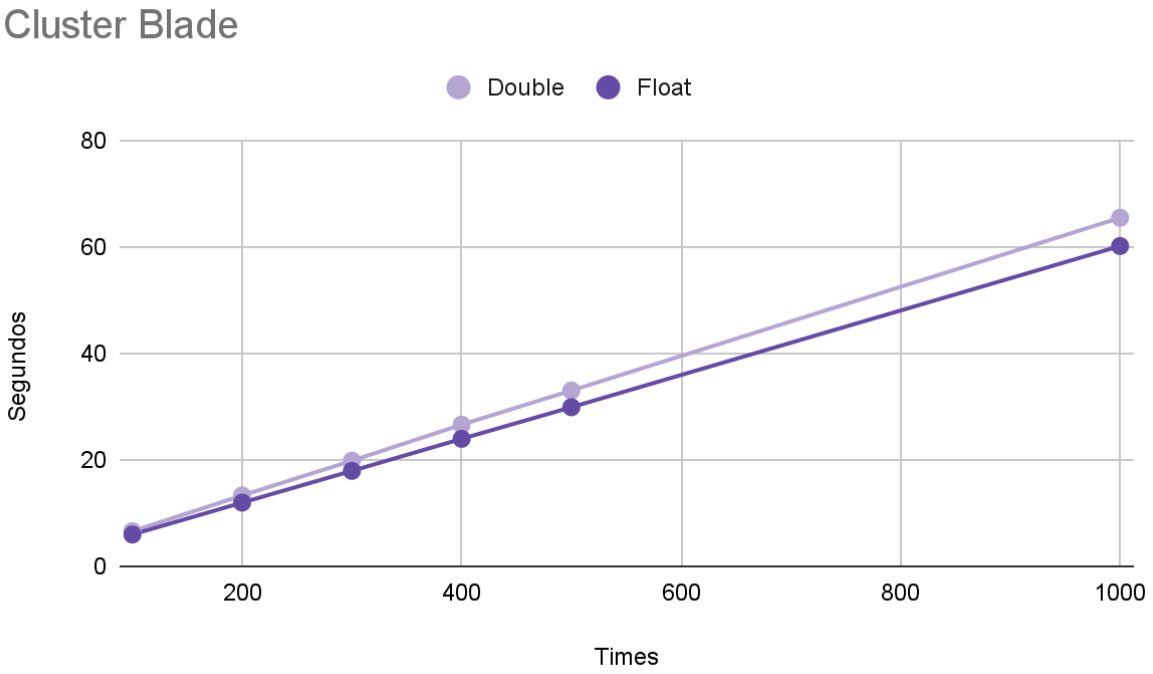
Los resultados obtenidos fueron idénticos tanto en el clúster remoto como en el equipo local. La solución que usa el tipo de variable double (64 bits) es más precisa en ambos casos, ya que usa más bits en memoria comparado con float (32 bits), y por lo tanto puede representar el resultado con mayor exactitud.

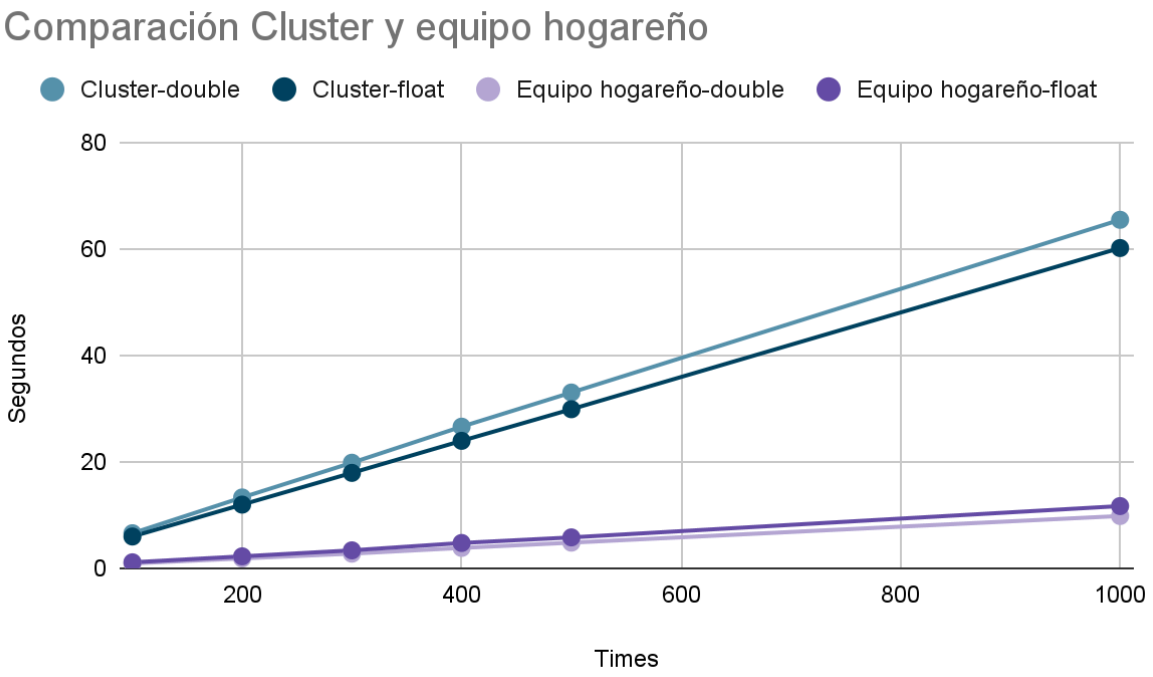
En las soluciones usando variables de tipo float, como ambas raíces dan el mismo resultado, no se puede apreciar la existencia de la segunda solución.

b. El algoritmo quadratic2.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución?

Equipos	Times	Double	Float
Equipo Hogareño	100	0.9251	1.1300
	200	1.8603	2.2499
	300	2.7615	3.3757
	400	3.835477	4.758758
	500	4.825102	5.809571
	1000	9.823823	11.689536
Cluster Blade	100	6.591222	5.973710
	200	13.302818	11.963128
	300	19.824670	17.919868
	400	26.600182	23.938124
	500	33.010659	29.875260
	1000	65.444170	60.141822







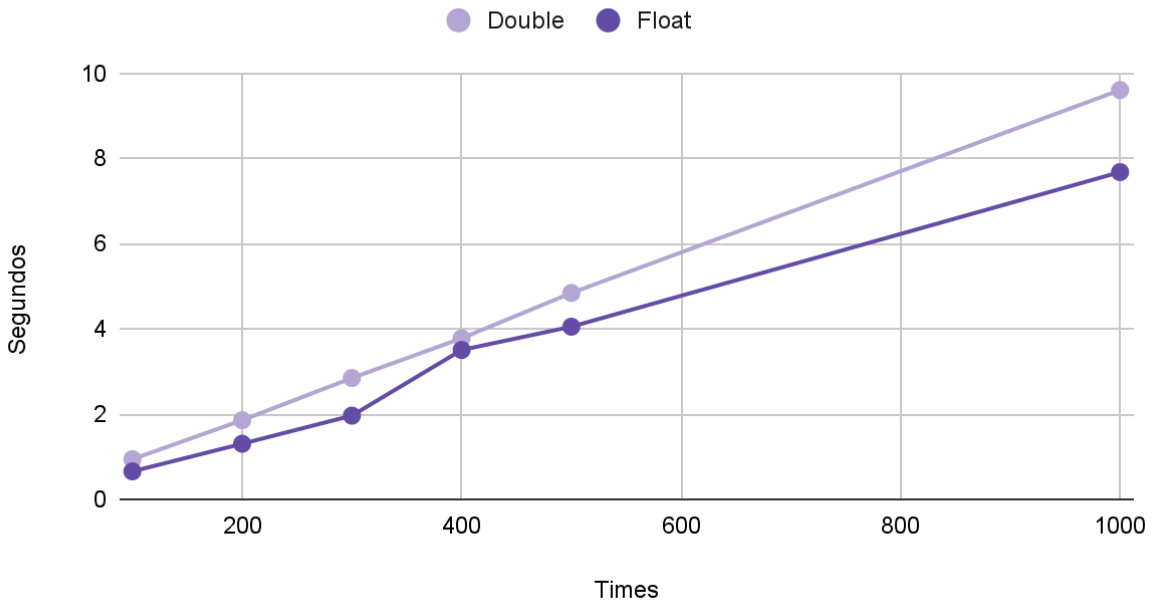
Observando los resultados, podemos ver que en el equipo hogareño la ejecución con floats tarda más que la que se realiza con doubles, mientras que en el cluster sucede lo contrario. Es curioso porque la intuición diría que operar con floats debería ser más rápido debido a que los datos son más pequeños (64 bits los double vs 32 bits los floats), pero en el equipo hogareño no se da el caso que estamos planteado.

Puede que una de las cosas que haga que la ejecución con float tarde más tenga que ver con el uso de funciones matemáticas tales como **pow()** y **sqrt()** que esperan valores de tipo **double** y por tanto deben realizar una conversión implícita antes de operar. La ejecución en todos nuestros casos de prueba fue mucho más rápida en el equipo hogareño que en el cluster.

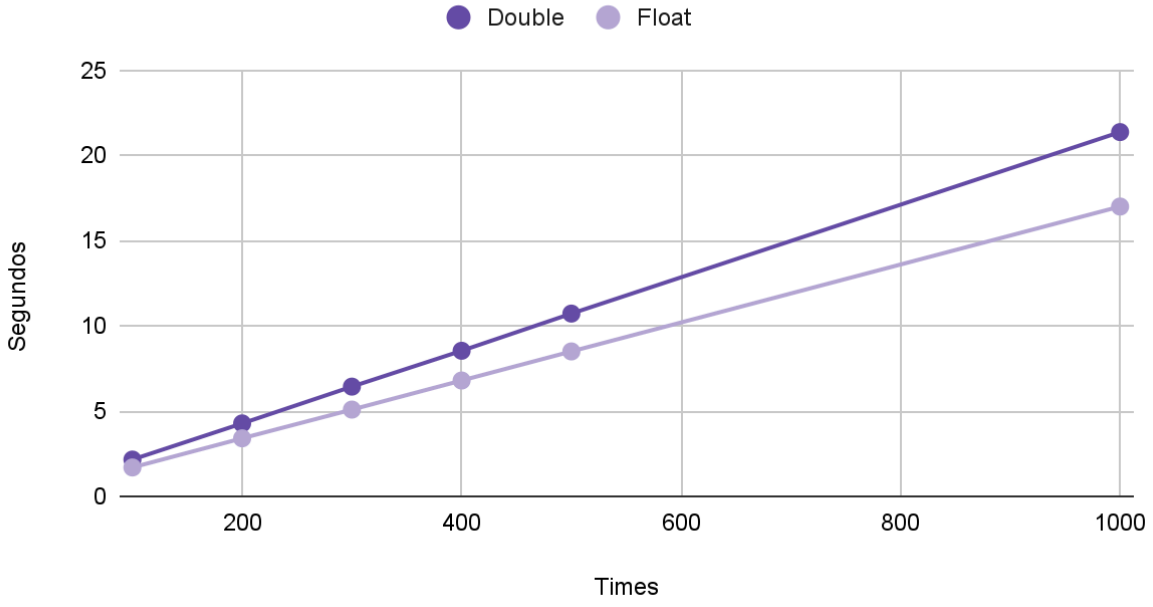
c. El algoritmo quadratic3.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución? ¿Qué diferencias puede observar en el código con respecto a quadratic2.c? Nota: agregue el flag -lm al momento de compilar. Pruebe con el nivel de optimización que mejor resultado le haya dado en el ejercicio anterior.

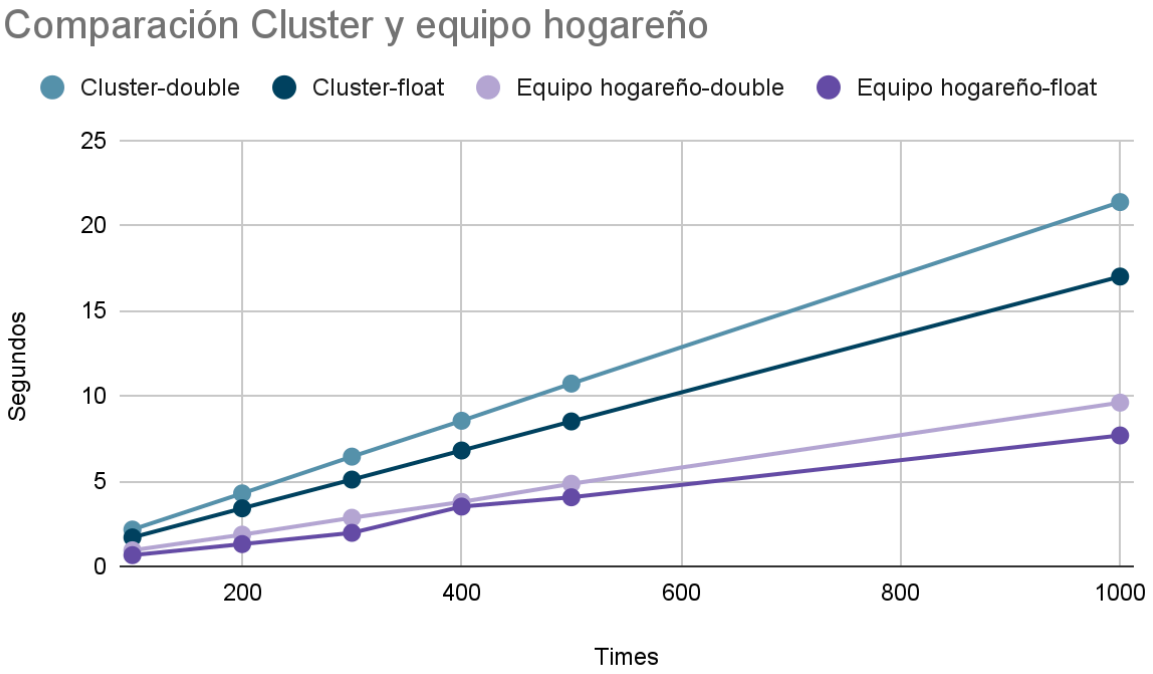
Equipos	Times	Double	Float
Equipo hogareño	100	0.9419	0.6561
	200	1.8565	1.3046
	300	2.8471	1.9643
	400	3.779482	3.505360
	500	4.843120	4.051939
	1000	9.606049	7.680262
Cluster Blade	100	2.157311	1.695164
	200	4.278431	3.404177
	300	6.437047	5.089048
	400	8.540751	6.795744
	500	10.722676	8.502475
	1000	21.373850	17.004047

Equipo hogareño



Cluster Blade





Observando los resultados, podemos ver que el cluster tuvo un tiempo de ejecución muchísimo menor que en el algoritmo del inciso anterior mientras que la ejecución en el equipo hogareño se mantuvo muy similar (incluso empeorando en algunos casos).

Las operaciones con floats son ahora más rápidas en ambos casos, lo cual probablemente se debe al uso de las funciones **powf()** y **sqrtf()**, que esperan recibir floats.

```
#define FA 1.0f
#define FB -4.0000000f
#define FC 3.9999999f

#define DA 1.0
#define DB -4.0000000
#define DC 3.9999999
```

En quadatric3, a diferencia de cuadatric2, se definen por separado los floats y doubles:

Y luego también se operan de manera acorde a su tipo de datos, usando, como ya mencionamos antes, **powf()** y **sqrtf()** para los floats y **pow()** y **sqrt()** para los doubles, que es como se almacenan por defecto los números al no indicar explícitamente que son floats. En este caso, al igual que en el ejemplo anterior, la ejecución en el cluster fue más lenta que en equipo hogareño, aunque ahora hay mucho menos diferencia.

2. Desarrolle un algoritmo en el lenguaje C que compute la siguiente ecuación:

$$R = \frac{(MaxA \times MaxB - MinA \times MinB)}{PromA \times PromB} \times [A \times B] + [C \times B^T]$$

− Donde A, B, C y R son matrices cuadradas de NxN con elementos de tipo double.
− MaxA, MinA y PromA son los valores máximo, mínimo y promedio de la matriz A, respectivamente.
− MaxB, MinB y PromB son los valores máximo, mínimo y promedio de la matriz B, respectivamente.
− B^T es la matriz transpuesta de B.
Mida el tiempo de ejecución del algoritmo en el clúster remoto. Las pruebas deben considerar la variación del tamaño de problema (N={512, 1024, 2048, 4096}). Por último, recuerde aplicar las técnicas de programación y optimización vistas en clase.

Construcción del algoritmo:

- Definimos e implementamos las funciones necesarias para los cálculos relacionados con las matrices y el tiempo de ejecución.
- Verificamos la correctitud de los parámetros de entrada: optamos por no hacer fijo el tamaño de bloque a modo de poder realizar pruebas y determinar cuál es el que mejor se adapta al equipo hogareño en el que hicimos las pruebas y cuál es el mejor para el cluster.
- Alocamos memoria e iniciamos las matrices necesarias. Dado que la consigna no nos indicaba qué valores debían tener, las generamos con números de tipo double aleatorios en un rango determinado (1 a 100).
- Hacemos cuentas: multiplicamos las matrices y en simultáneo (secuencialmente) operamos para conseguir máximos, mínimos y promedios para evitar recorridos adicionales.
- Utilizamos la función **dwalltime()** para medir el tiempo de ejecución del algoritmo que desarrollamos para saber la eficiencia de este.

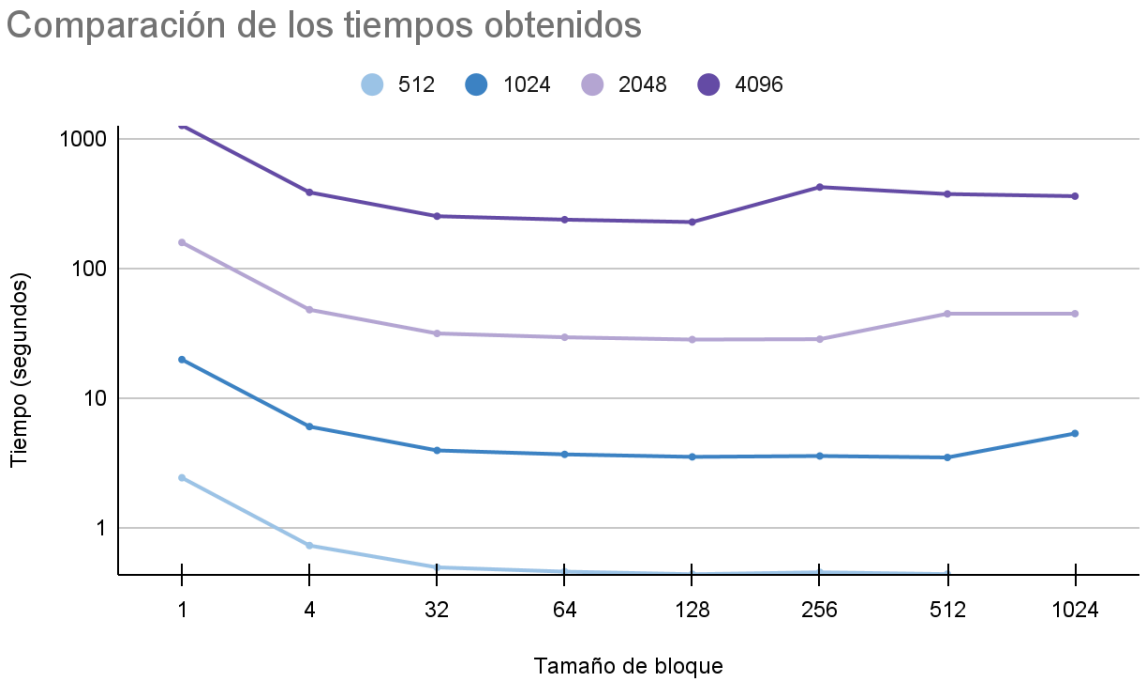
Consideraciones al diseñar:

- Definimos las matrices como arreglos dinámicos como vectores de elementos ya que nos da varias ventajas frente a otras formas de definir matrices: Los datos quedan contiguos en memoria permitiendo agilizar el acceso y podemos elegir cómo organizar los datos (por filas o por columnas) según lo que optimice mejor el rendimiento.
- Optamos por multiplicar las matrices dividiéndolas por bloques para aprovechar la localidad de memoria y así reducir el número de accesos a memoria principal.
- Aprovechamos la flexibilidad que nos da para organizar los datos para guardar unas matrices por filas y otras por columnas, de manera tal de aprovechar la localidad de datos ya que al realizar la multiplicación de matrices se multiplica cada elemento de una fila de la primera matriz por cada elemento correspondiente a una columna de la otra.
- Debido a que no organizamos todas las matrices de la misma manera, tuvimos que tener cuidado de acceder a los datos de la misma forma en que los guardamos al momento de realizar otros cálculos como la búsqueda de mínimos, máximos y promedios.

- Optamos por calcular los mínimos, máximos y promedios en la misma iteración en la que se hace la multiplicación por bloques a modo de evitar recorrer la matriz múltiples veces de manera innecesaria.

Equipo	Tamaño de matriz	Tamaño de bloque	Tiempo
Hogareño	512	1	2.0366
		4	0.7594
		32	0.7570
		64	0.7506
		128	0.7553
	1024	1	16.2414
		4	6.0517
		32	6.0425
		64	5.9692
		128	6.0207
	2048	1	130.4406
		4	48.1787
		32	47.8403
		64	47.8370
		128	48.0704
	4096	1	1039.94
		64	382.8406
		128	597.2767
Cluster blade	512	1	2.440399
		4	0.733305
		32	0.497661
		64	0.461336
		128	0.440730

		256	0.455905
		512	0.441564
	1024	1	19.863421
		4	6.048082
		32	3.962578
		64	3.694006
		128	3.534697
		256	3.589693
		512	3.497011
		1024	5.355588
	2048	1	158.521141
		4	48.099089
		32	31.555043
		64	29.495725
		128	28.321734
		256	28.507926
		512	44.780784
		1024	44.815457
	4096	1	1264.176694
		4	385.632159
		32	252.529764
		64	237.652900
		128	227.741592
		256	423.461718
		512	374.254806
		1024	360.40234



Se puede observar que en nuestras pruebas el tamaño óptimo de bloque parece ser 128 ya que es el que nos dio mejores tiempos de ejecución para todos los tamaños de matrices salvo para el tamaño de matriz 1024, en cuyo caso el tamaño de bloque con mejor rendimiento fue el de 512.

Realizamos por curiosidad las pruebas también en nuestro equipo hogareño y los resultados nos mostraron que el tamaño de bloque más óptimo es 64. Esta diferencia se debe principalmente a las diferencias entre las arquitecturas de ambos dispositivos.