

Proyecto final

arquitectura de hardware

Alejandro Barrera Lozano
Javier Andrés Torres Reyes
Diciembre 11, 2020

Problema

Este trabajo tiene como objetivo emitir un juicio sobre cómo diferentes factores (que definiremos más adelante) afectan el rendimiento de un sistema de cómputo al invertir una imagen, recorriéndola píxel por píxel. Esto haciendo uso de la metodología de conducción de experimentos y la teoría de arquitectura de computadores.

Específicamente, deseamos cuantificar y explicar el impacto en el desempeño que tienen 5 versiones equivalentes de algoritmos que varían en la localidad espacial al recorrer los píxeles de una imagen. También se desea estudiar el impacto del tamaño de las imágenes, el lenguaje de programación y el sistema de cómputo en el tiempo de respuesta normalizado de estos algoritmos.

De este modo, nuestra hipótesis nula es: “variar los diferentes factores no afecta el tiempo promedio de respuesta para invertir una imagen”, y la hipótesis alterna es: “variar los diferentes factores afecta el tiempo promedio de respuesta para invertir una imagen”.

Factores

Factores Primarios:

1. **Equipos de cómputo.** Manejamos los siguientes 2 niveles:
 - a. **PC 1.** Equipo de escritorio con Windows 10 pro, ejecutando Ubuntu 20.04 en el Windows Subsystem for Linux (WSL).

CPU-Z

CPU

Caches

Mainboard

Memory

SPD

Graphics

Bench

About

Processor

Name

Intel Core i7 8700

Code Name

Coffee Lake

Max TDP

65.0 W

Package


Socket 1151 LGA

Technology

14 nm

Core Voltage

0.904 V



Specification

Intel® Core™ i7-8700 CPU @ 3.20GHz

Family

6

Model

E

Stepping

A

Ext. Family

6

Ext. Model

9E

Revision

U0

Instructions

MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, AES, AVX, AVX2, FMA3, TSX

Clocks (Core #0)

Core Speed

4291.62 MHz

Multiplier

x 43.0 (8 - 46)

Bus Speed

99.81 MHz

Rated FSB

Cache

L1 Data

6 x 32 KBytes

8-way

L1 Inst.

6 x 32 KBytes

8-way

Level 2

6 x 256 KBytes

4-way

Level 3

12 MBytes

16-way

Selection

Socket #1

Cores

6

Threads

12

CPU-Z

Ver. 1.94.8.x64

Tools

Validate

Close

CPU-Z

CPU

Caches

Mainboard

Memory

SPD

Graphics

Bench

About

L1D-Cache

Size

32 KBytes

x 6

Descriptor

8-way set associative, 64-byte line size

L1I-Cache

Size

32 KBytes

x 6

Descriptor

8-way set associative, 64-byte line size

L2 Cache

Size

256 KBytes

x 6

Descriptor

4-way set associative, 64-byte line size

L3 Cache

Size

12 MBytes

Descriptor

16-way set associative, 64-byte line size

Size

Descriptor

Speed

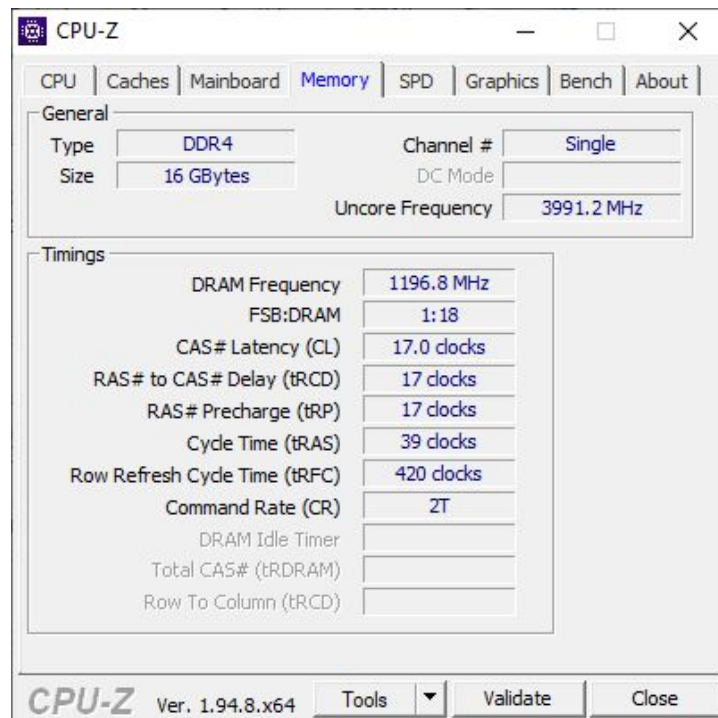
CPU-Z

Ver. 1.94.8.x64

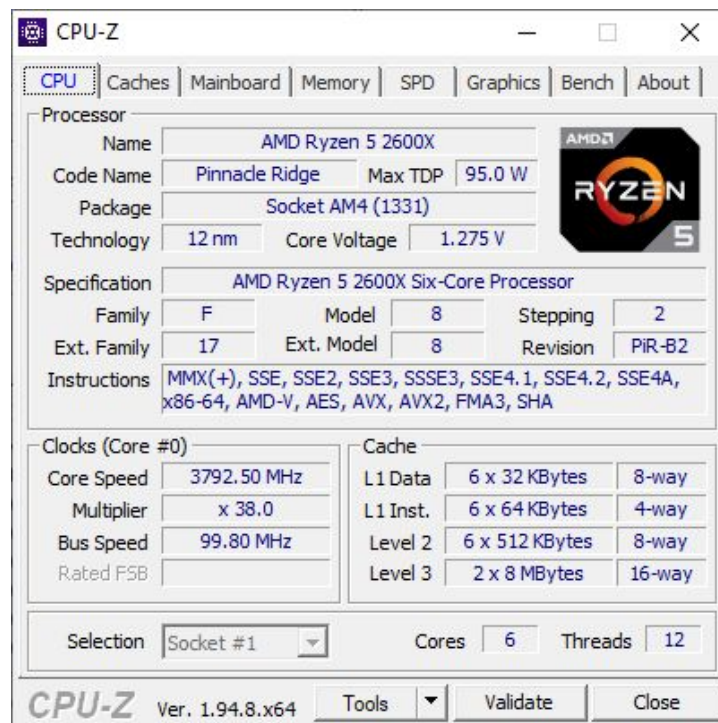
Tools

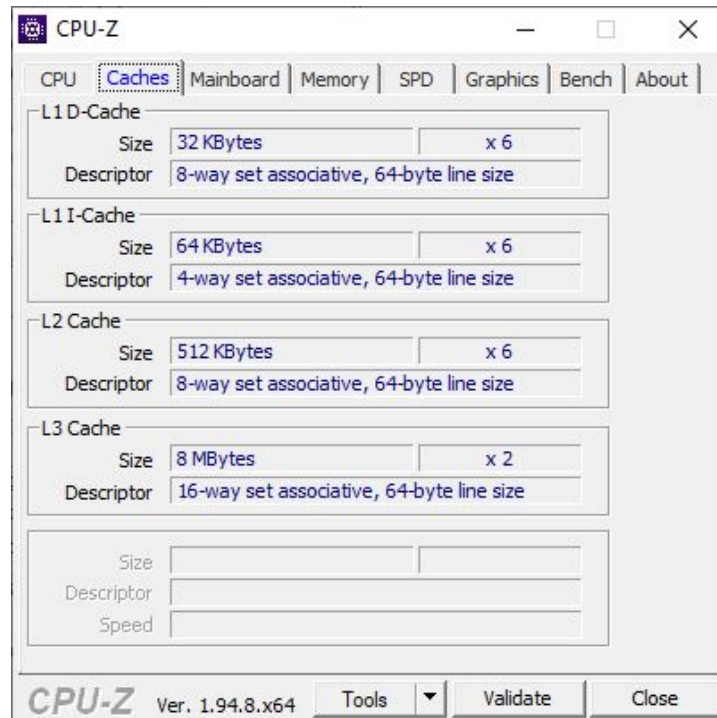
Validate

Close



b. **PC 2.** Equipo de escritorio con Pop os (distribución basada en ubuntu)





Nota: Debido a la alta variación que puede haber debido a otros factores entre ambos equipos, en lugar de tratarlo como un factor más, realizamos en análisis de cada equipo por separado.

2. **Tamaño de imagen.** Escogimos imágenes cuadradas de formato BMP, teniendo en cuenta dos cosas para sus tamaños: debían ser lo suficientemente grandes para que el tiempo que tardaban en ser procesadas fuera medible y lo suficientemente pequeñas para que las librerías encargadas de cargarlas pudieran hacerlo sin problema. De este modo, manejamos como niveles los siguientes 8 tamaños de la misma imagen: A continuación la imagen usada para el experimento.



- a. 400x400
- b. 500x500
- c. 600x600

- d. 700x700
- e. 800x800
- f. 900x900
- g. 1000x1000
- h. 1100x1100

3. **Algoritmo.** Usamos 5 algoritmos equivalentes (misma complejidad espacial y temporal) para invertir la imagen. Sin embargo difieren en la forma en que recorren la matriz, de modo que aprovechan (o no) la localidad espacial de forma diferente.

a.

```
for R=1: Rows
    for C=1:Columns
        ImRGBO(R,C,1)=255-ImRGB(R,C,1);
        ImRGBO(R,C,2)=255-ImRGB(R,C,2);
        ImRGBO(R,C,3)=255-ImRGB(R,C,3);
        count = count + 3;
    end
end
```

b.

```
for R=1: Rows
    for C=1:Columns
        ImRGBO(R,C,1)=255-ImRGB(R,C,1);
        count = count + 1;
    end
end
for R=1: Rows
    for C=1:Columns
        ImRGBO(R,C,2)=255-ImRGB(R,C,2);
        count = count + 1;
    end
end
for R=1: Rows
    for C=1:Columns
        ImRGBO(R,C,3)=255-ImRGB(R,C,3);
        count = count + 1;
    end
end
```

c.

```
for C=1:Columns
    for R=1: Rows
        ImRGBO(R,C,1)=255-ImRGB(R,C,1);
        ImRGBO(R,C,2)=255-ImRGB(R,C,2);
        ImRGBO(R,C,3)=255-ImRGB(R,C,3);
        count = count + 3;
    end
end
```

d.


```

for R=1: Rows
    for C=1:Columns
        ImRGBO(R,C,1)=255-ImRGB(R,C,1);
        count = count + 1;
    end
end
for R=Rows :-1:1
    for C=Columns:-1:1
        ImRGBO(R,C,2)=255-ImRGB(R,C,2);
        ImRGBO(R,C,3)=255-ImRGB(R,C,3);
        count = count + 2;
    end
end

```

e.

```

for R=1:2: Rows -1
    for C=1:2:Columns-1

        ImRGBO(R,C,1)=255-ImRGB(R,C,1);
        ImRGBO(R,C,2)=255-ImRGB(R,C,2);
        ImRGBO(R,C,3)=255-ImRGB(R,C,3);

        ImRGBO(R,C+1,1)=255-ImRGB(R,C+1,1);
        ImRGBO(R,C+1,2)=255-ImRGB(R,C+1,2);
        ImRGBO(R,C+1,3)=255-ImRGB(R,C+1,3);

        ImRGBO(R+1,C,1)=255-ImRGB(R+1,C,1);
        ImRGBO(R+1,C,2)=255-ImRGB(R+1,C,2);
        ImRGBO(R+1,C,3)=255-ImRGB(R+1,C,2);

        ImRGBO(R+1,C+1,1)=255-ImRGB(R+1,C+1,1);
        ImRGBO(R+1,C+1,2)=255-ImRGB(R+1,C+1,2);
        ImRGBO(R+1,C+1,3)=255-ImRGB(R+1,C+1,3);
        count = count + 12;
    end
end

```

4. **Lenguaje de programación.** Elegimos 2 lenguajes de programación, teniendo en cuenta que ambos debían ser lenguajes compilados para asegurar que en ejecución las matrices estuvieran contiguas en memoria:

- a. **Go.**
- b. **C++.** Usando el compilador g++ con la opción de compilación O2 para que el código corriera más eficientemente.

Factores Secundarios:

1. **Entorno de ejecución.** Para evitar la variación que pueden tener diferentes IDE, ejecutamos los programas por consola (específicamente, con un script bash)

2. **Cantidad de tareas en segundo plano:** Se prescindió de todas las tareas en segundo plano que no eran necesarias para ejecutar el script que ejecutaba los algoritmos de manera aleatoria (en secciones posteriores hablaremos más sobre este script).
3. **Temperatura del ambiente.** Pues los procesadores ajustan su velocidad de reloj de acuerdo a la temperatura a la que estén. Para mitigarlo, corrimos todos los experimentos en un lapso corto de tiempo (con ayuda del script).
4. **Fuente de poder.** Debido a variaciones en la corriente que la alimenta y a la forma en que está construida, la fuente de poder de los equipos no entrega todo el tiempo exactamente la misma potencia, afectando la capacidad de cómputo de los procesadores. De forma similar al factor anterior, correr todos los experimentos en un lapso corto de tiempo ayuda a mitigar este factor.
5. **Interrupciones por hardware.** Mientras se ejecutaba el script, no se movió ni el ratón ni el teclado, de modo que las interrupciones por hardware no afectaran el desempeño de los algoritmos.

Variables de respuesta:

Como variable de respuesta tenemos el tiempo en nanosegundos que se demoró el programa en ejecutar la versión del algoritmo pedida normalizada con respecto al tamaño de la imagen, es decir, el tiempo dividido por el producto del ancho y el alto de la imagen.

Diseño experimental:

Teniendo en cuenta que buscamos ver cómo los diferentes factores afectan la variable de respuesta, el tipo de experimento que usamos fue factorial completo. Además, para cada combinación de factores hicimos 3 repeticiones. De este modo, el número total de corridas es de $2 \times 8 \times 5 \times 2 \times 3 = 480$ (240 por equipo de cómputo).

Para facilitar la recolección de la información, construimos los programas de modo que para ejecutarlos solo tuviéramos que correr por consola un comando que recibiera como argumentos información sobre qué tratamiento se iba a ejecutar. Por ejemplo, para ejecutar el programa de c++ con el equipo 1, usando el algoritmo 5, en la repetición 3 de la imagen de 600 x 600, bastaba con escribir por consola el comando `./invertir_cpp 1 5 3 600`.

Aun así, ejecutar todos los tratamientos implicaría que en cada equipo se tendrían que introducir uno por uno 240 comandos, lo que resultaría extremadamente tedioso y tomaría mucho tiempo. De este modo, construimos un script de bash que recibiera como parámetro desde qué equipo se estaba ejecutando (para poder guardar correctamente la información), y se encargara de ejecutar todos los comandos (el archivo adjunto `./script.sh`).

Sin embargo, escribir los 240 comandos dentro del script seguía siendo un trabajo muy tedioso, de modo que escribimos un programa que se encargara de generar los 240 comandos, ordenarlos aleatoriamente y añadirlos al script (el archivo adjunto `./generarscript`).

Experimentación preliminar:

Aprovechando que teníamos scripts que nos ayudaban a llevar a cabo los experimentos fácilmente, ejecutamos los tratamientos con 100 datos cada uno. Además, modificamos los programas para que ellos mismos se encargaran de calcular información como la media, la varianza y la desviación estándar del tratamiento que estaban ejecutando y los añadieran a un archivo CSV (el archivo adjunto `metricas.csv`).

Con esta información, pudimos calcular cuál era el tamaño de muestra necesario para que cada tratamiento tuviera un nivel de confianza del 95% y un error de 0,25 ns al predecir la media del tiempo de ejecución. En efecto, en el peor de los casos eran necesarios 80 datos, de modo que los 100 datos que habíamos escogido eran estadísticamente relevantes.

Además, observando estos datos de forma preliminar, nos dimos cuenta de que Go era más rápido al ejecutar los algoritmos que C++, lo que contradecía la idea que teníamos de que C++ debía ser más rápido. Sin embargo, nos dimos cuenta de que esto se debía a optimizaciones que hacía el compilador de Go, de modo que volvimos a correr los tratamientos, esta vez usando el flag de compilación `O2`, con lo que ahora C++ fue más rápido.

Efectuar el experimento

Sabiendo ya que con 100 datos por cada tratamiento sería suficiente, solo quedaba ejecutar los scripts. Para facilitar aún más el trabajo de pasar la información a minitab, modificamos los programas para que se encargaran de apilar los datos en un archivo CSV (el archivo adjunto `apilados.csv`).

Ahora sí, cerramos los programas en ejecución (tanto en primer como en segundo plano) y corrimos el script. Mientras se terminaba de ejecutar, dejamos los equipos completamente quietos.

Finalmente, obtuvimos como salida los tiempos de cada tratamiento, tanto de forma individual (en la carpeta `data`) como apilados (en el archivo `apilados.csv`); las métricas que indicamos en la sección anterior (media, varianza, etc.); y las imágenes invertidas.

Análisis y Conclusiones

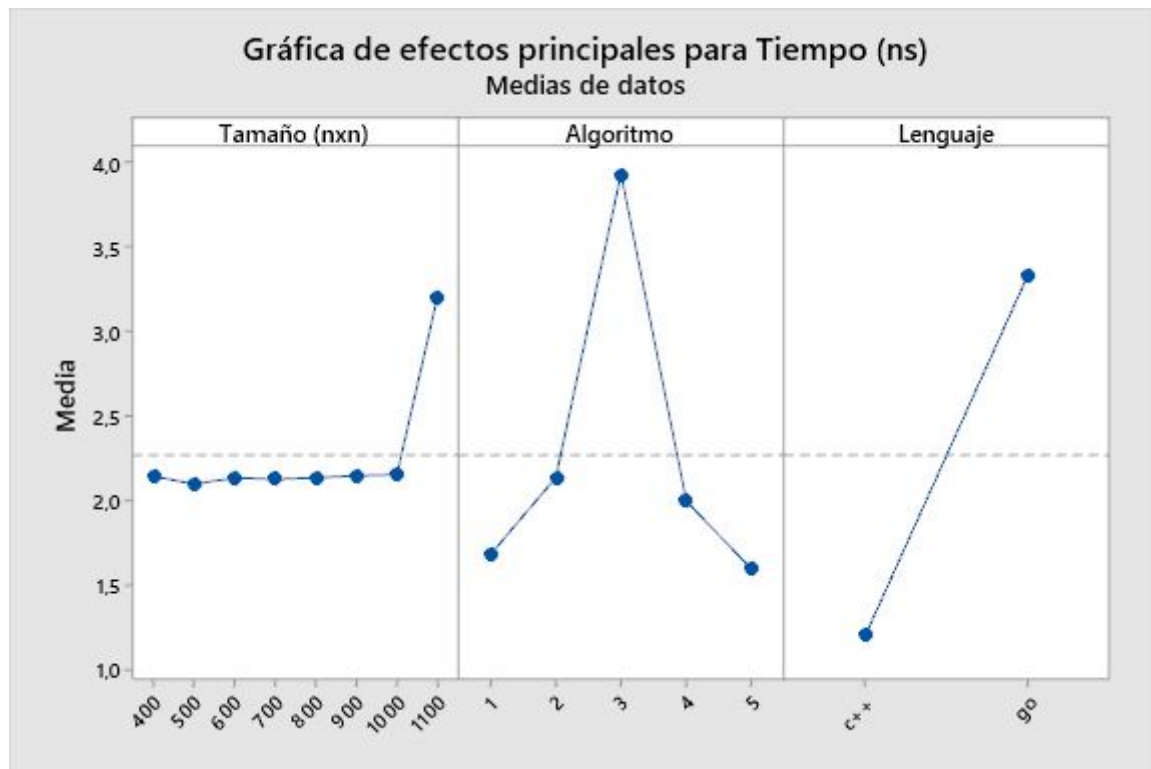
Los análisis los haremos con un nivel de significancia (alfa) del 5% para cada uno de los equipos de cómputo.

PC 1:

Análisis de Varianza

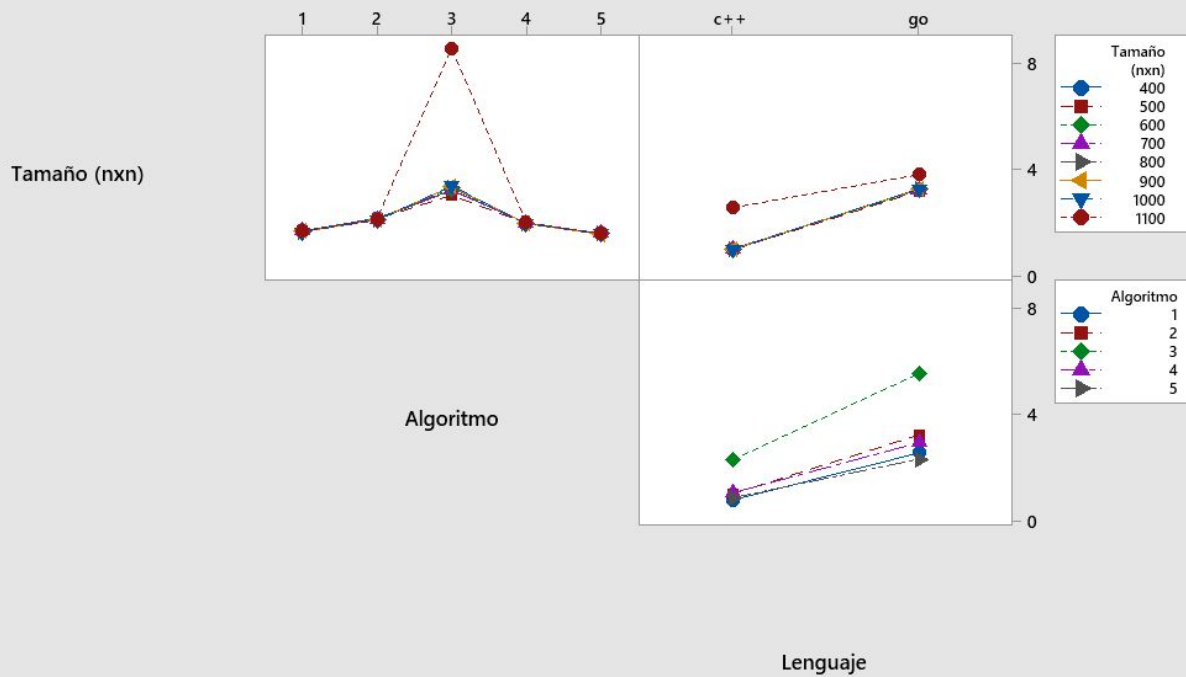
Fuente	GL	SC Ajust.	MC Ajust.	Valor F	Valor p
Tamaño (nxn)	7	3005,5	429,4	4776,22	0,000
Algoritmo	4	17490,0	4372,5	48640,33	0,000
Lenguaje	1	27062,1	27062,1	301043,61	0,000
Tamaño (nxn)*Algoritmo	28	11855,7	423,4	4710,18	0,000
Tamaño (nxn)*Lenguaje	7	665,4	95,1	1057,47	0,000
Algoritmo*Lenguaje	4	2288,2	572,0	6363,45	0,000
Tamaño (nxn)*Algoritmo*Lenguaje	28	2618,6	93,5	1040,37	0,000
Error	23920	2150,3	0,1		
Total	23999	67135,8			

Como todos los valores p son menores que alfa, vemos que todos los factores y sus interacciones afectan el tiempo promedio de respuesta para invertir una imagen, es decir, rechazamos la hipótesis nula. Esto hace que sea válido ingresar todos los factores en el análisis.



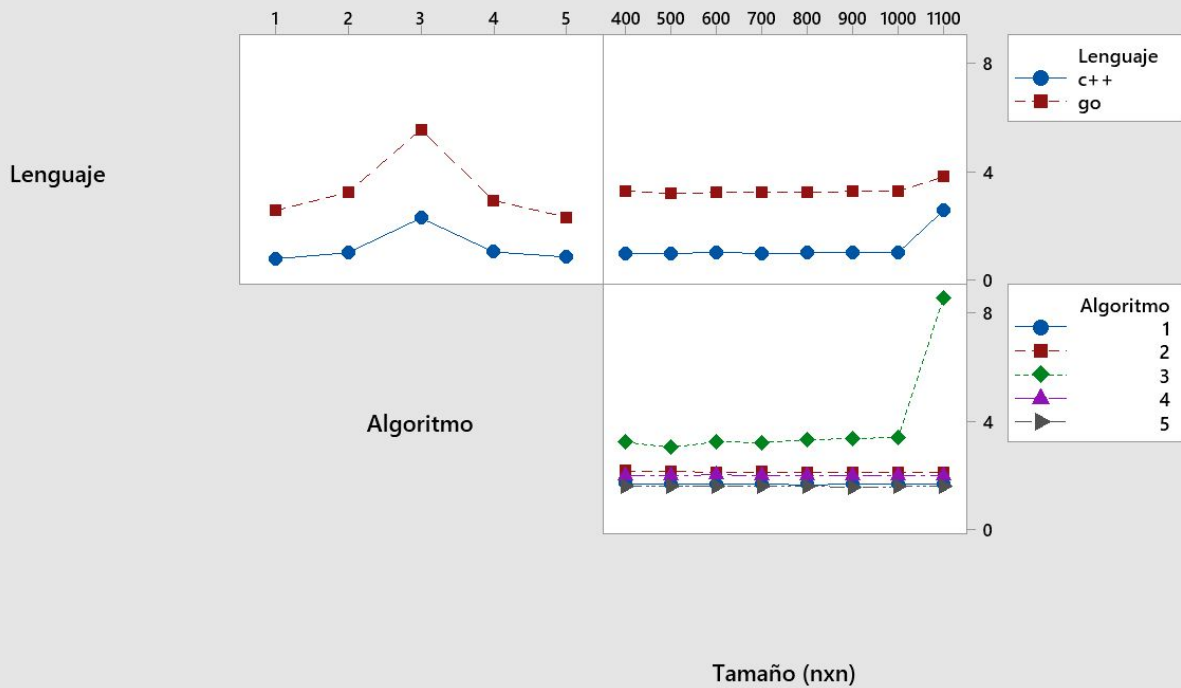
Gráfica de interacción para Tiempo (ns)

Medias de datos



Gráfica de interacción para Tiempo (ns)

Medias de datos



Agrupar información utilizando el método de Tukey y una confianza de 95%

Tamaño (nxn)	N	Media	Agrupación
1100	3000	3,20571	A
1000	3000	2,15630	B
900	3000	2,14999	B C
400	3000	2,14708	B C
800	3000	2,13786	B C
600	3000	2,13775	B C
700	3000	2,12880	C
500	3000	2,09984	D

Las medias que no comparten una letra son significativamente diferentes.

Comparaciones por parejas de Tukey: Algoritmo

Agrupar información utilizando el método de Tukey y una confianza de 95%

Algoritmo	N	Media	Agrupación
3	4800	3,93108	A
2	4800	2,13828	B
4	4800	2,00023	C
1	4800	1,68456	D
5	4800	1,59793	E

Las medias que no comparten una letra son significativamente diferentes.

Agrupar información utilizando el método de Tukey y una confianza de 95%

Lenguaje	N	Media	Agrupación
go	12000	3,33230	A
c++	12000	1,20854	B

Las medias que no comparten una letra son significativamente diferentes.

Peores combinaciones

Comparaciones por parejas de Tukey: Tamaño (nxn)*Algoritmo*Lenguaje

Agrupar información utilizando el método de Tukey y una confianza de 95%

Tamaño (nxn)*Algoritmo*Lenguaje	N	Media	Agrupación
1100 3 c++	300	9,13020	A
1100 3 go	300	8,02513	B
900 3 go	300	5,36402	C
1000 3 go	300	5,34044	C
400 3 go	300	5,28167	C
800 3 go	300	5,27682	C
700 3 go	300	5,17146	D
600 3 go	300	5,13292	D
500 3 go	300	4,89471	E
400 2 go	300	3,30291	F
700 2 go	300	3,26749	F

Mejores combinaciones

1100 5 c++	300	0,89770	Q	R
600 5 c++	300	0,88027		R
700 5 c++	300	0,87524		R
800 5 c++	300	0,86638		R
500 5 c++	300	0,86430		R
400 5 c++	300	0,85489		R
1000 5 c++	300	0,84777		R
900 5 c++	300	0,84031		R
900 1 c++	300	0,80587		R
1100 1 c++	300	0,80335		R
600 1 c++	300	0,79998		R
500 1 c++	300	0,79931		R
400 1 c++	300	0,79518		R
700 1 c++	300	0,79195		
1000 1 c++	300	0,79096		
800 1 c++	300	0,78638		

Mitad de la tabla: (de ahí para abajo es todo c++)

900 5 go	300	2,32236	I	
800 5 go	300	2,31728	I	
1000 3 c++	300	1,47353	J	
900 3 c++	300	1,39446	J	K
800 3 c++	300	1,37224	J	K L
600 3 c++	300	1,36543		K L

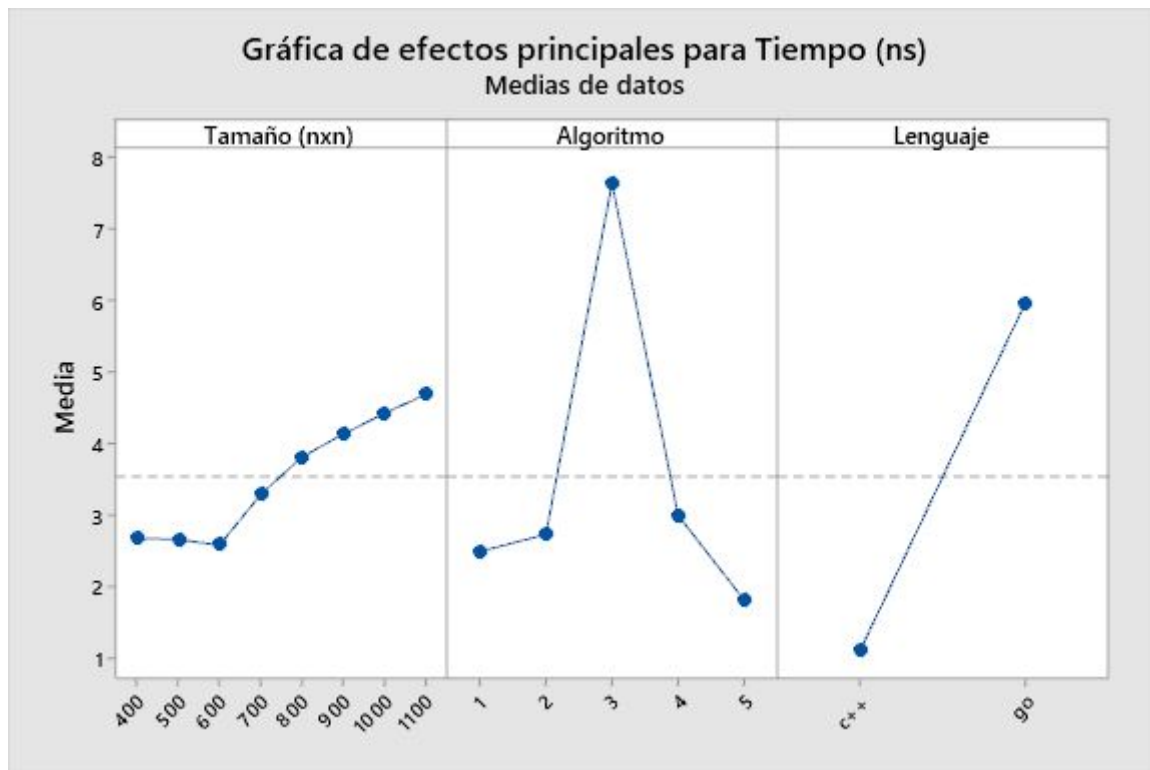
Vemos que para todos los casos c++ es mejor que Go excepto en el caso en el que se usa el algoritmo 3 sobre la imagen de tamaño 1100 x 1100, la más grande.

PC 2

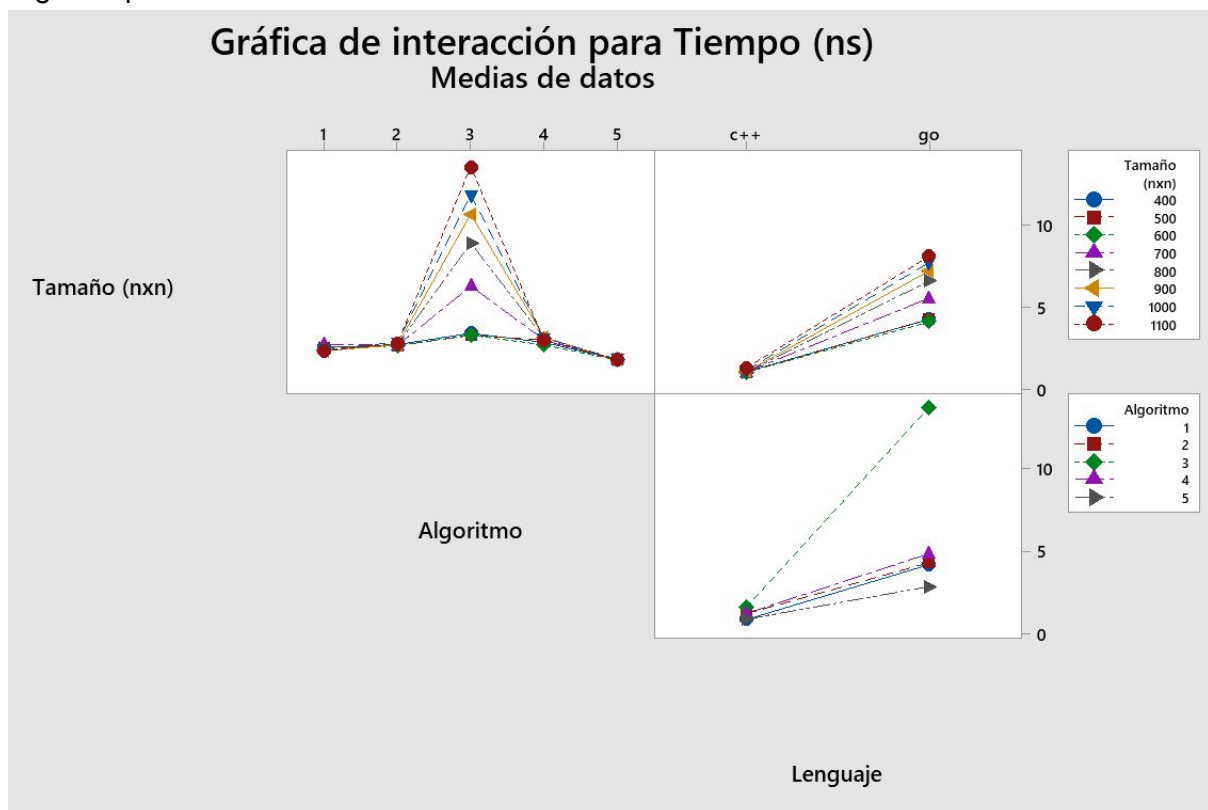
Análisis de Varianza

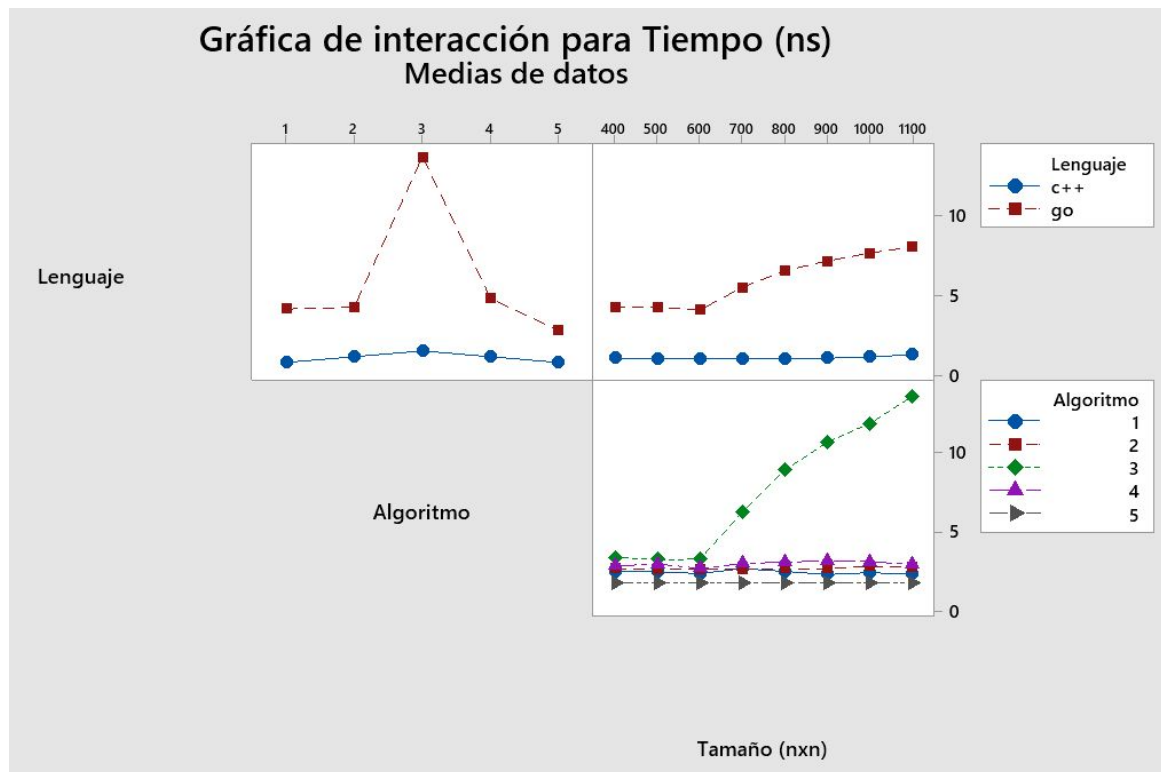
Fuente	GL	SC Ajust.	MC Ajust.	Valor F	Valor p
Tamaño (nxn)	7	15085	2155	8545,69	0,000
Algoritmo	4	104646	26161	103743,89	0,000
Lenguaje	1	141399	141399	560723,71	0,000
Tamaño (nxn)*Algoritmo	28	56729	2026	8034,35	0,000
Tamaño (nxn)*Lenguaje	7	12884	1841	7298,83	0,000
Algoritmo*Lenguaje	4	82463	20616	81752,51	0,000
Tamaño (nxn)*Algoritmo*Lenguaje	28	47455	1695	6720,89	0,000
Error	23920	6032	0		
Total	23999	466693			

Como todos los valores p son menores que alfa, vemos que todos los factores y sus interacciones afectan el tiempo promedio de respuesta para invertir una imagen, es decir, rechazamos la hipótesis nula. Esto hace que sea válido ingresar todos los factores en el análisis.



En comparación al *PC 1* vemos que el tamaño empieza a afectar la media desde mucho menor tamaño, puede ser por la diferencia entre tamaño del caché entre ambos equipos, el *PC 1* teniendo 12 MB de caché nivel 3 y el *PC 2* teniendo 8 MB de caché nivel 3. No obstante, puede haber sido un factor secundario no controlado como un proceso en segundo plano.





Podemos observar que todas las variaciones debido al tamaño de la imagen ocurren solo con el algoritmo 3, y principalmente en el lenguaje Go.

Aun así, al final vemos que los tiempos tienen comportamientos coherentes en ambos equipos.

Agrupar información utilizando el método de Tukey y una confianza de 95%

Tamaño (nxn)	N	Media	Agrupación
1100	3000	4,69559	A
1000	3000	4,43199	B
900	3000	4,14216	C
800	3000	3,81983	D
700	3000	3,29682	E
400	3000	2,68494	F
500	3000	2,66755	F
600	3000	2,58872	G

Las medias que no comparten una letra son significativamente diferentes.

Agrupar información utilizando el método de Tukey y una confianza de 95%

Algoritmo	N	Media	Agrupación
3	4800	7,64249	A
4	4800	3,00470	B
2	4800	2,73751	C
1	4800	2,50025	D
5	4800	1,81980	E

Las medias que no comparten una letra son significativamente diferentes.

Agrupar información utilizando el método de Tukey y una confianza de 95%

Lenguaje	N	Media	Agrupación
go	12000	5,96822	A
c++	12000	1,11368	B

Las medias que no comparten una letra son significativamente diferentes.

Agrupar información utilizando el método de Tukey y una confianza de 95%

Tamaño (nxn)*Algoritmo*Lenguaje	N	Media	Agrupación
1100 3 go	300	24,4571	A
1000 3 go	300	21,7638	B
900 3 go	300	19,7807	C
800 3 go	300	16,4380	D
700 3 go	300	11,2248	E
400 3 go	300	5,5867	F
600 3 go	300	5,3300	G
800 5 go	300	2,8030	Q
900 5 go	300	2,8007	Q
600 5 go	300	2,7756	Q
700 5 go	300	2,7731	Q
1100 3 c++	300	2,5353	R
1000 3 c++	300	1,8582	
900 3 c++	300	1,4853	
800 3 c++	300	1,3410	

700 5 c++	300	0,8192
900 5 c++	300	0,8185
400 5 c++	300	0,8178
900 1 c++	300	0,8175
500 5 c++	300	0,8168
700 1 c++	300	0,8165
800 5 c++	300	0,8149

Comparando los resultados de ambos equipos, podemos observar que las gráficas de interacción son muy similares (excepto en lo mencionado previamente sobre los tamaños de las imágenes). En ambos equipos C++ es más rápido que Go. Además, uniendo ambos resultados, podemos ver que los dos mejores algoritmos son el 1 y el 5, seguidos por el 2 y el 4, y, finalmente, el peor algoritmo es el 3 (sobre todo en imágenes grandes).

Esto se puede justificar desde el punto de vista teórico, partiendo de que C++ y Go son lenguajes que guardan las matrices en memoria en orden row-major. Debido a que el sistema de caché de ambos equipos no usa mapeo directo, sino asociativo por conjuntos, es complicado calcular exactamente el miss rate de cada algoritmo. Sin embargo, sí podemos entender a alto nivel qué está sucediendo:

El algoritmo 1 y el algoritmo 5 aprovechan por completo la localidad espacial, pues dentro del ciclo interno modifican los datos que están contiguos (los tres colores del píxel, y, en el caso del algoritmo 5, también los 3 colores del píxel de al lado).

El algoritmo 2 y el algoritmo 4 también aprovechan la localidad espacial al hacer el recorrido por filas, sin embargo, al saltarse colores, tienen mayor miss rate que los dos algoritmos anteriores.

Finalmente, el algoritmo 3 no aprovecha la localidad espacial, pues siempre salta a un dato que no está cercano a él en la memoria. Más aún, en las imágenes grandes es posible que se llene la caché más rápidamente, lo que explica la subida tan grande de tiempo en estos casos.

Basándonos en estos datos, podemos decir que, cuando se quiere invertir una imagen, es útil tener en cuenta las siguientes pautas:

- El tamaño de la imagen no es realmente relevante en el tiempo normalizado, aunque solo si el algoritmo que se está usando aprovecha correctamente la localidad espacial (por ejemplo, los algoritmos 1 y 5). De lo contrario, usar imágenes más grandes aumenta mucho el tiempo de respuesta (por ejemplo, lo que ocurrió con el algoritmo 3).
- Respecto al punto anterior, es importante aclarar que decimos que los algoritmos 1 y 5 aprovechan la localidad espacial porque los lenguajes que usamos son row-major. En lenguajes column-major como Fortran, MATLAB, R, Julia, entre otros, estos algoritmos hubieran sido menos eficientes, y el 3 hubiera funcionado mejor.
- C++ es un lenguaje sumamente eficiente, pero es importante que para obtener estos resultados se tengan en cuenta los flags de compilación que optimizan el código (en nuestro caso, el flag O2).

- Los factores secundarios se controlaron lo suficientemente bien, como para que la varianza fuera mínima y los resultados fueran consistentes en los dos equipos de cómputo.

Como opción de mejora, para reducir factores secundarios se podría usar una máquina con una instalación mínima de sistema operativo más las dependencias necesarias para ejecutar el código de Go y C++. Esto se puede lograr usando virtualización pues a la máquina virtual se le asignan recursos fijos. Además, sería útil ejecutar el experimento cuando la máquina esté recién iniciada, de modo que haya la menor cantidad posible de procesos en segundo plano.