Ordenamiento de arreglo de enteros con SIMD

Brandon Diaz Videla, brandon.diaz@alumnos.uv.cl

**Resumen: en ese trabajo se implementará un algoritmo de ordenamiento utilizando SIMD ósea funciones intrínsecas del procesador, que utiliza la memoria vectorial de esta para poder acelerar cierto tipo de algoritmo. Para esto se implemento un algoritmo de ordenamiento para conjuntos de 16 números utilizando estas funciones intrínsecas de Intel y se midió el tiempo de ejecución de estas pruebas para determinar si es eficiente o no implementar un algoritmo de ordenamiento basado en SIMD.**

# Introducción

La programación paralela se puede llevar a cabo de muchas maneras, por ejemplo, con hilos, que estos están dentro de cada Core de un procesador, haciendo que cada hilo haga una tarea para luego unir los resultado, con Cores físicos, que es casi lo mismo que los threads y finalmente lo que utilizaremos ahora con intrínsecas del procesador, en este caso de Intel. Las intrínsecas si bien están consideradas dentro de la programación paralela, no lo son tanto como los threads o Cores de los procesadores, ya que no pueden hacer 2 tareas simultaneas, pero lo que si pueden hacer es realizar una operación sobre un conjunto de datos en 1 solo ciclo de procesador. Entonces por ejemplo antes hacer una suba de 8 números, que nos tomaría 5 ciclos de procesador, ahora con la intrínseca correcta nos puede tomar hasta 1 solo ciclo realizar esta suma, lo que se traduce en optimización de tiempo. Entonces el experimento que haremos será el de ordenar una cantidad de números utilizando estas intrínsecas para “optimizar” el tiempo de ordenamiento.

# Componentes del entorno de pruebas

Para la realización de este trabajo se utilizará el servidor facilitado por la universidad de Valparaíso, que tiene los siguientes componentes técnicos.

CPU 🡪 Intel ® Xeon ® Platinum 8260 CPU @ 2.4GHz

Sockets(s) 🡪 16

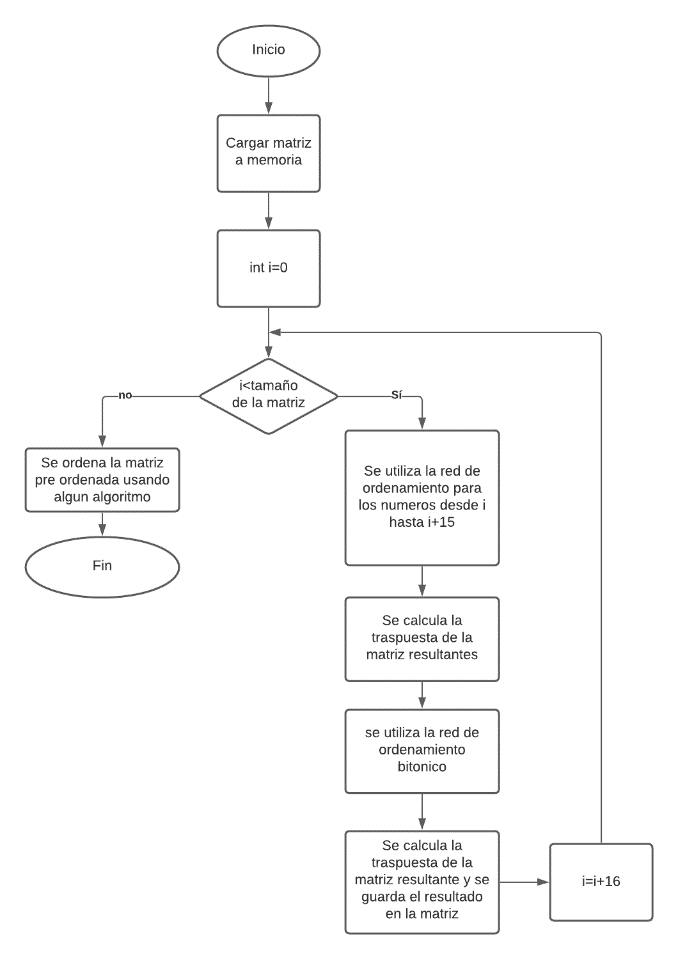
Core(s) por socket 🡪 1

Thread(s) por core 🡪 1

Arquitectura 🡪 x86\_64

# Desarrollo

Para la realización de este algoritmo se utilizaron redes de ordenamiento, ordenadores bitonicos, y redes de ordenamientos bitonico, de esta manera el arreglo total se puede preordenar en grupos de 16 números ordenados entre si, para aligerar la carga de la función que finalmente lo ordenara.



**Figura 1.** En este diagrama de flujo se puede ver el funcionamiento del algoritmo a alto nivel

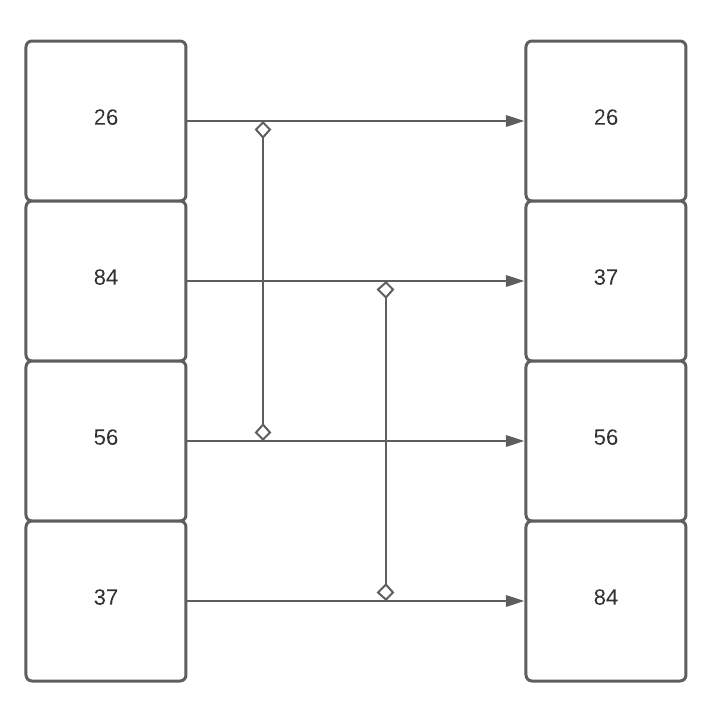
## Sorting network

Una red de ordenamiento es un conjunto de pasos para que un numero fijo de pasos y números sean ordenados. En este caso usaremos una red de ordenamiento de 4 números, pero al momento de implementarla para utilizar vectores ordenaremos 16 números en vez de solo 4, en la misma cantidad de ciclos que si fueran 4 números.

Para explicarlo mejor se utilizará un ejemplo.

### Paso 1

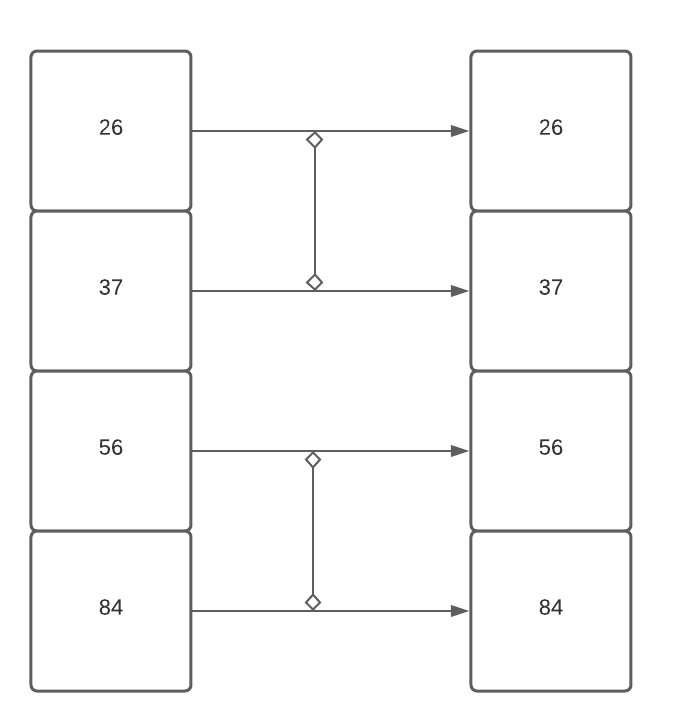
Se tienen 4 números {26,84,56,37} y el paso es comparar la posición 1 con la posición 3, y la posición 2 con la posición 4, con la operación máximo y mínimo, esto quiere decir que a un lado de la comparación siempre dejara el numero menor, y al otro lado siempre dejara el valor mayor, así como se puede observar en la figura a continuación.

****

**Figura 2.** Primer paso sorting network

### Paso 2

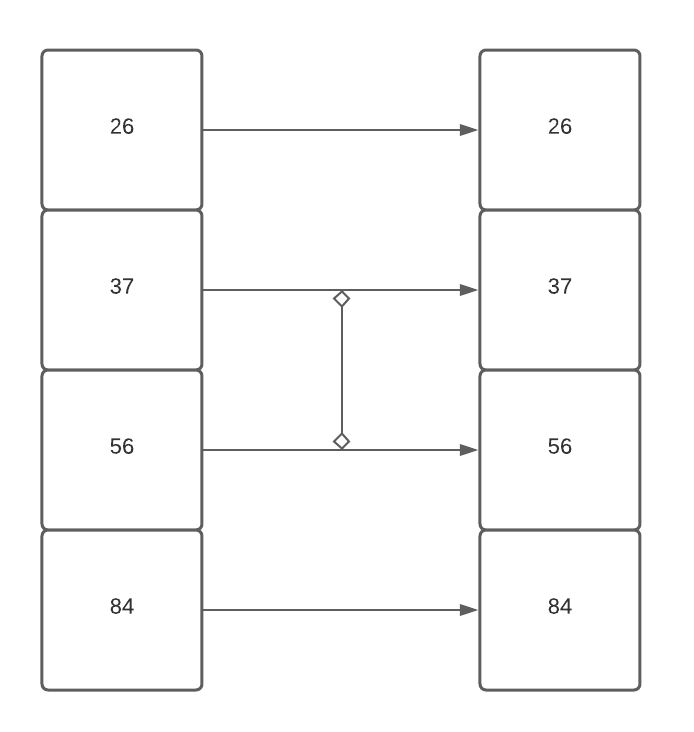
Luego con el resultado hacemos la operación máximo/mínimo entre las posiciones 1 y 2, y entre las posiciones 3 y 4 como se muestra a continuación.



**Figura 3.**  Segundo paso sorting network

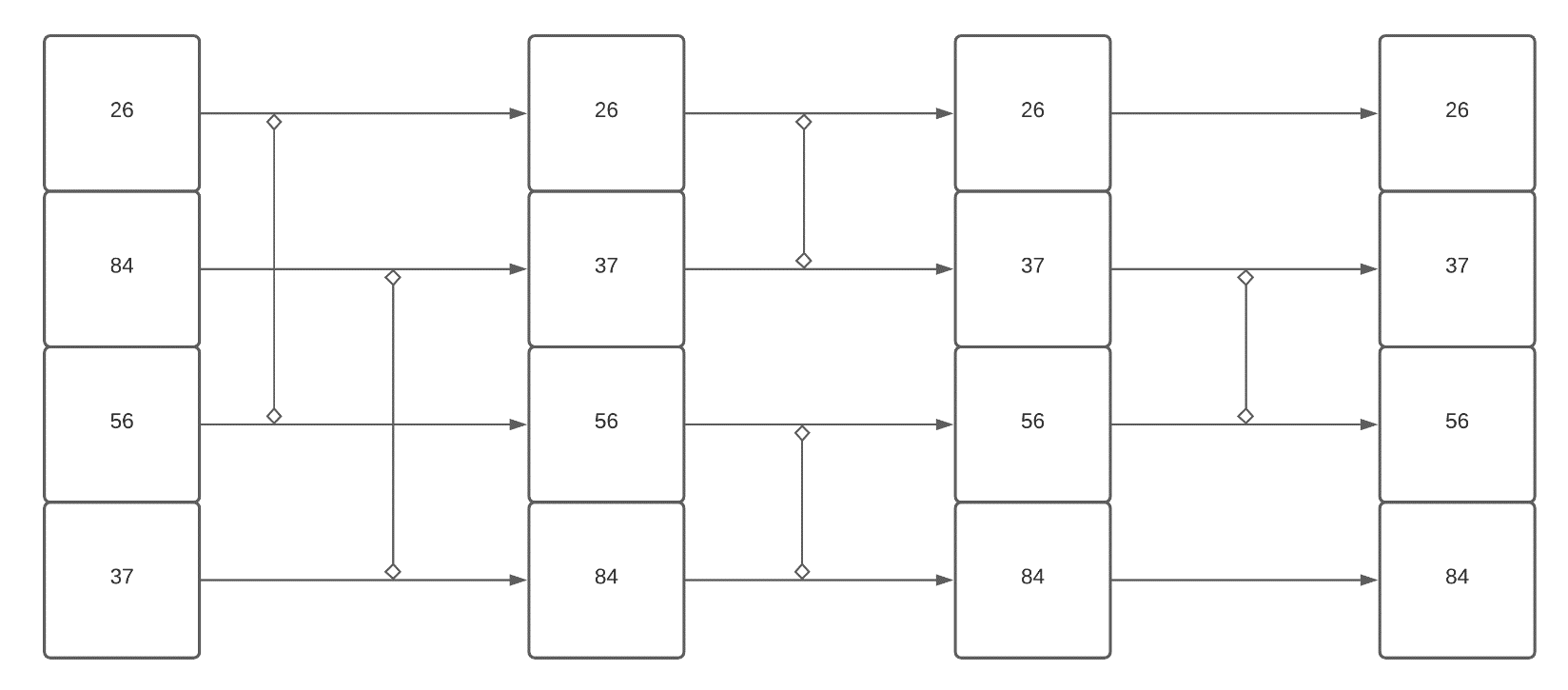
### Paso 3

Finalmente se hace la comparación con los números “del medio” ósea los que están en las posiciones 2 y 3 y con esto nuestro conjunto de 4 números quedaran ordenados.



**Figura 3.**  Tercer paso sorting network

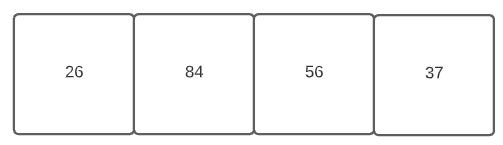
### Vista complete de la sorting network



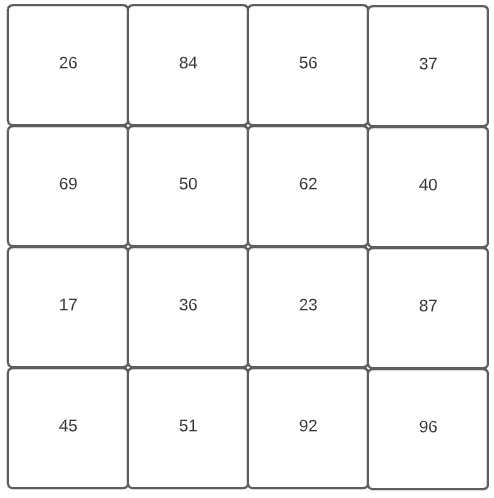
**Figura 3.**  sorting network completa

## explicacion vectorial sorting network

Para implementar esta sorting network con procesamiento vectorial deberíamos tener 1 numero en cada vector para poder hacer las comparaciones, ya que no podemos realizar operaciones sobre el mismo vector, entonces tenemos que tomar 4 vectores para poder realizar estas operaciones. Pero como estos son vectores de 128bits pueden almacenar hasta 4 números, por lo que ordenaremos 4 grupos de 4 números al mismo tiempo.

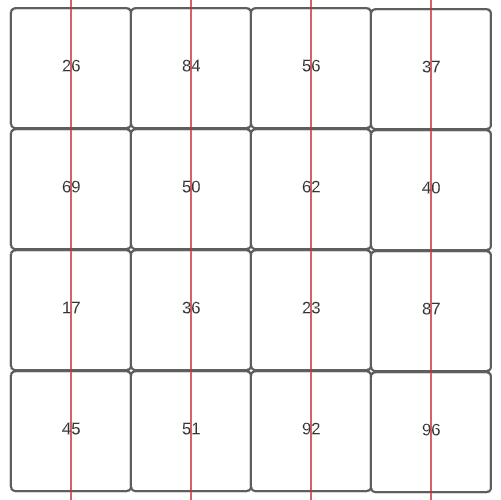


**Figura 3.**  Vector de ejemplo

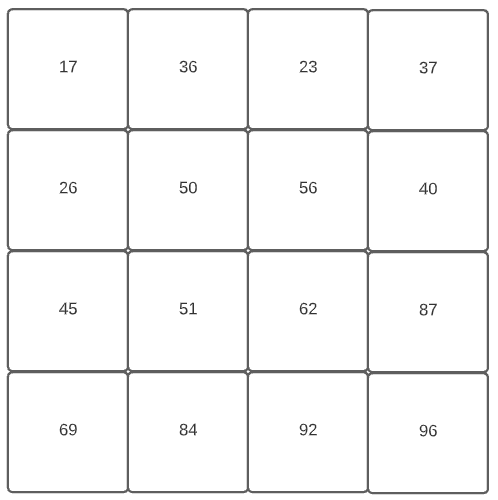


**Figura 3.**  4 vectores posicionados como una matriz 4x4

De esta manera al aplicar la sorting network de manera vectorial ordenaríamos columnas de números y no filas, pero lo haríamos en 5 operaciones siempre.



**Figura 3.**  Matriz con las columnas marcadas por líneas rojas



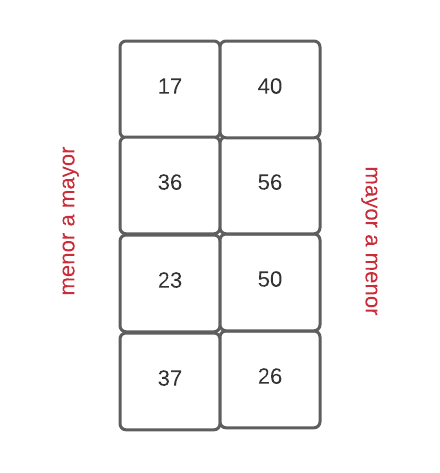
**Figura 3.**  Matriz después de aplicar la sorting network de manera vectorial

Ahora tenemos una matriz con solo las columnas ordenadas de menor a mayor en ellas, sin ninguna relación entre columnas.

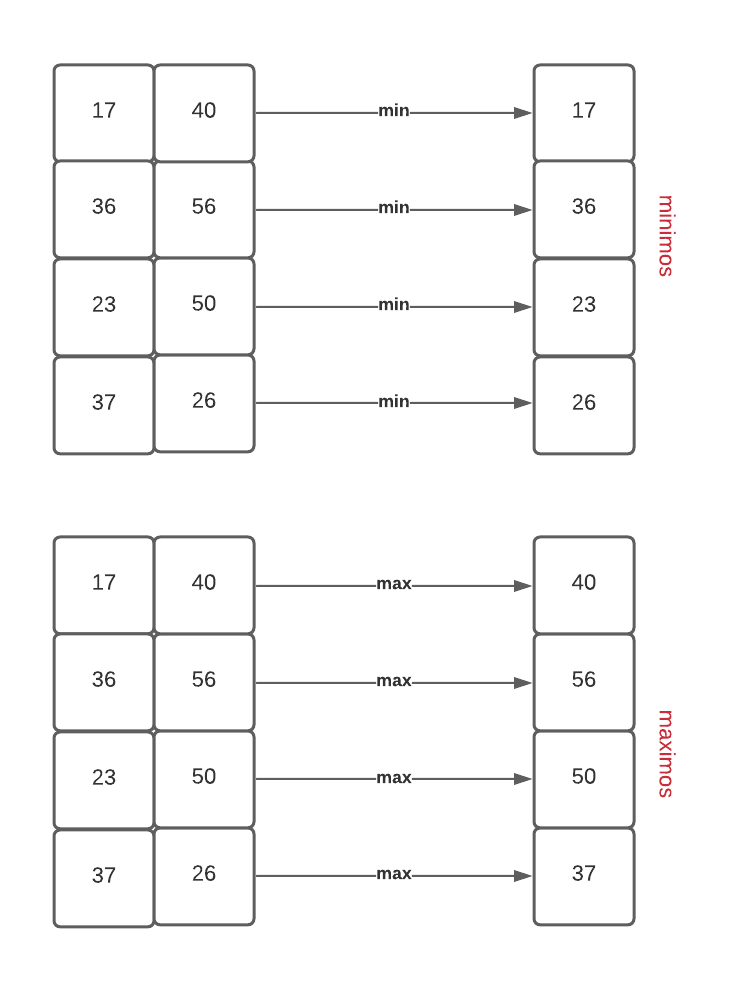
## explicacion del bitonic sorter

### explicacion bitonic sorter paso 1

Un bitonic sorter ordenador bitonico es un algoritmo que es capaz de ordenar 8 números que estén preordenados, un grupo de 4 de menor a mayor y los otros 4 de mayor a menor. De esta manera este algoritmo será capaz de ordenar los 8 números siempre en la misma cantidad de pasos. Luego de tener los 2 vectores con 4 números cada uno ordenados de la forma correcta hacemos operaciones de máximo y mínimo sobre ambos vectores.



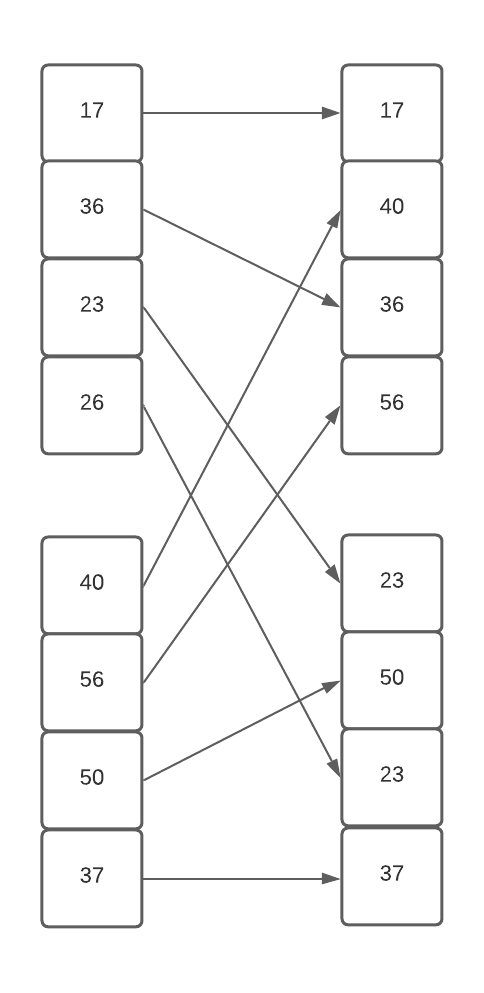
**Figura 3.**  Vectores ordenados



**Figura 3.**  Vectores después de la operación máximo y mínimo

### Implementación bitonic sorter paso 2

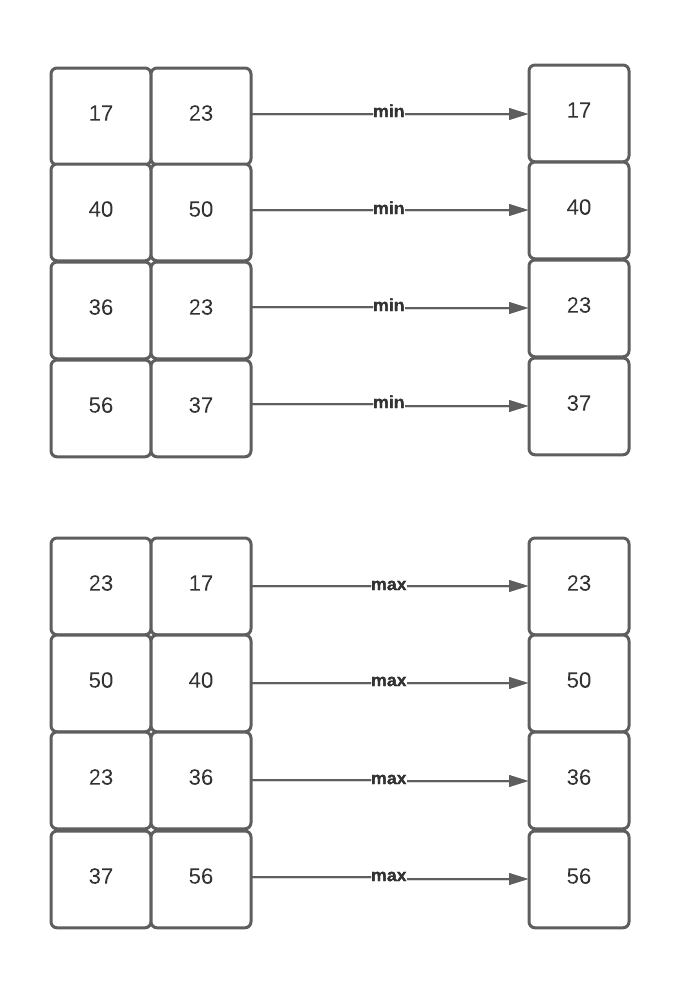
Luego de el primer paso tenemos que mover de posición los vectores, ya que necesitamos comparar valores correspondientes al mismo vector, y esto no lo podemos realizar, así que para trabajar estos vectores solo colocan los primeros 2 números de ambos vectores en un único vector y los últimos 2 números del otro vector en otro único vector.



**Figura 3.**  Imagen de la explicación del paso 2

### explicacion bitonic sorter paso 3

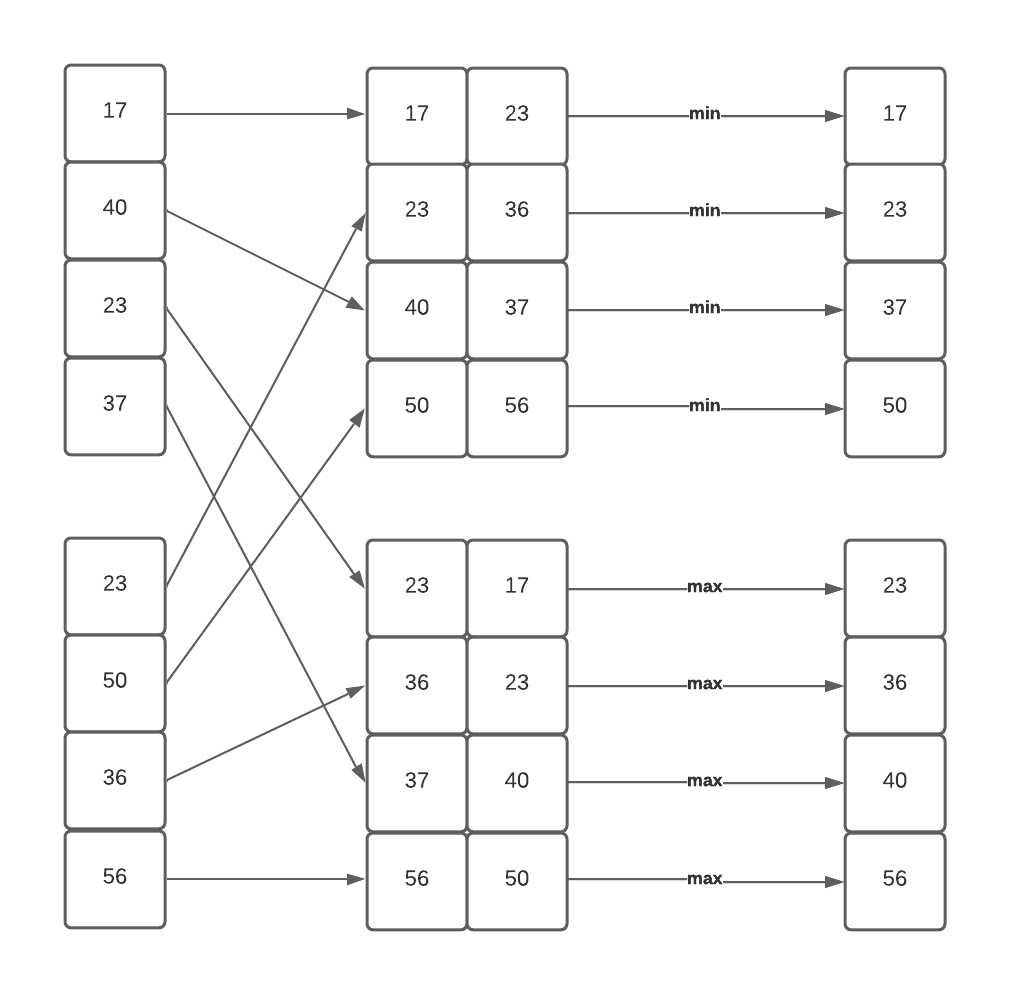
Luego aplicamos la operación máxima/mínimo sobre ambos vectores.



**Figura 3.**  Imagen de la explicación del paso 3

### explicacion bitonic sorter paso 4

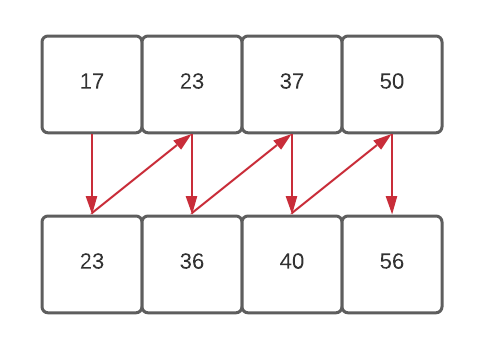
Y luego repetimos lo hecho en el paso 2 y 3



**Figura 3.**  Imagen de la explicación del paso 3

### explicacion bitonic sorter paso 5

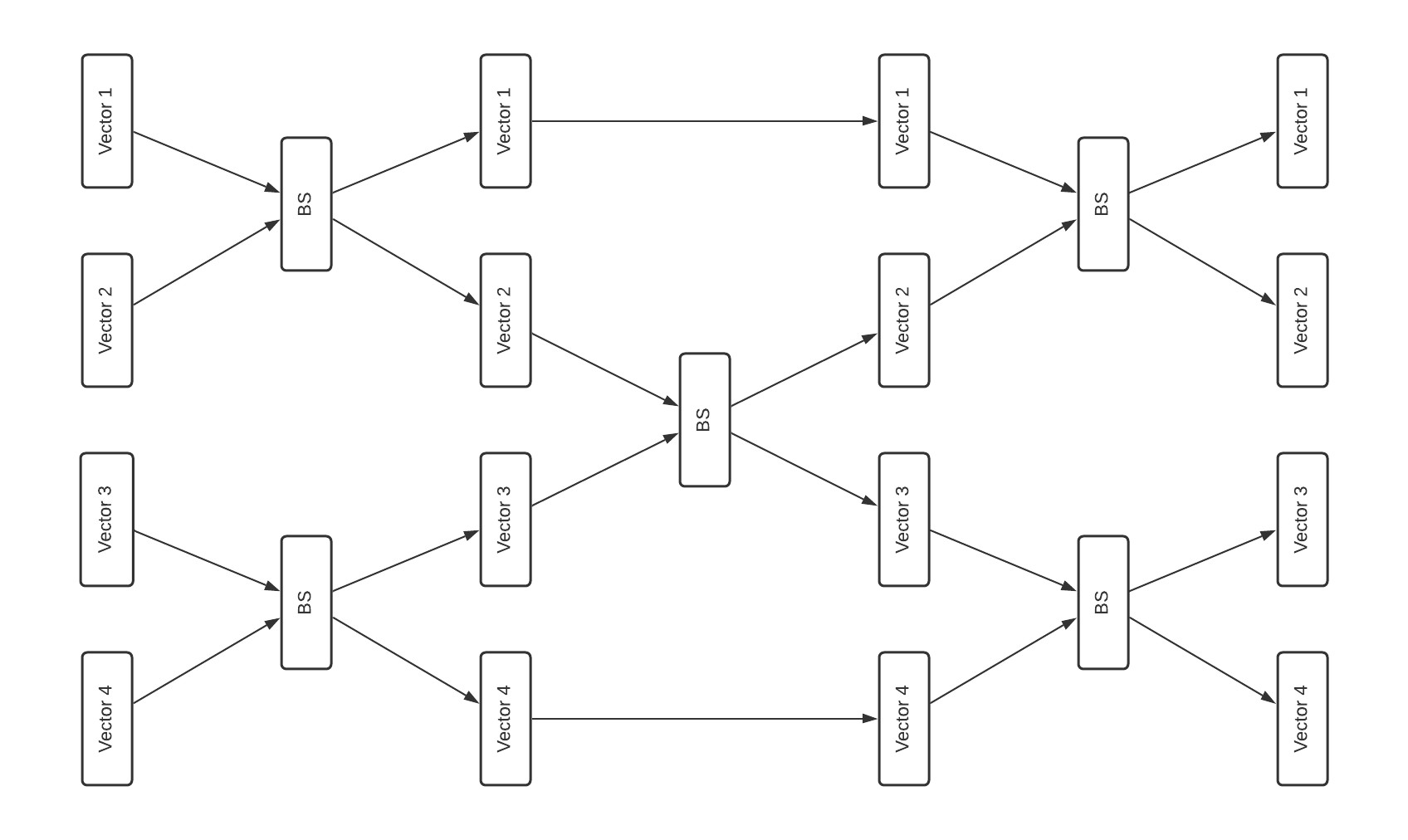
Aquí puede parecer que no están ordenador, pero si lo están ya que hay que verlos en un orden especifico, como se puede observar en la siguiente figura. Y con esto tenemos terminado el bitonic sorter



**Figura 3.**  Imagen de la explicación del paso 3

## explicacion del bitonic merge network

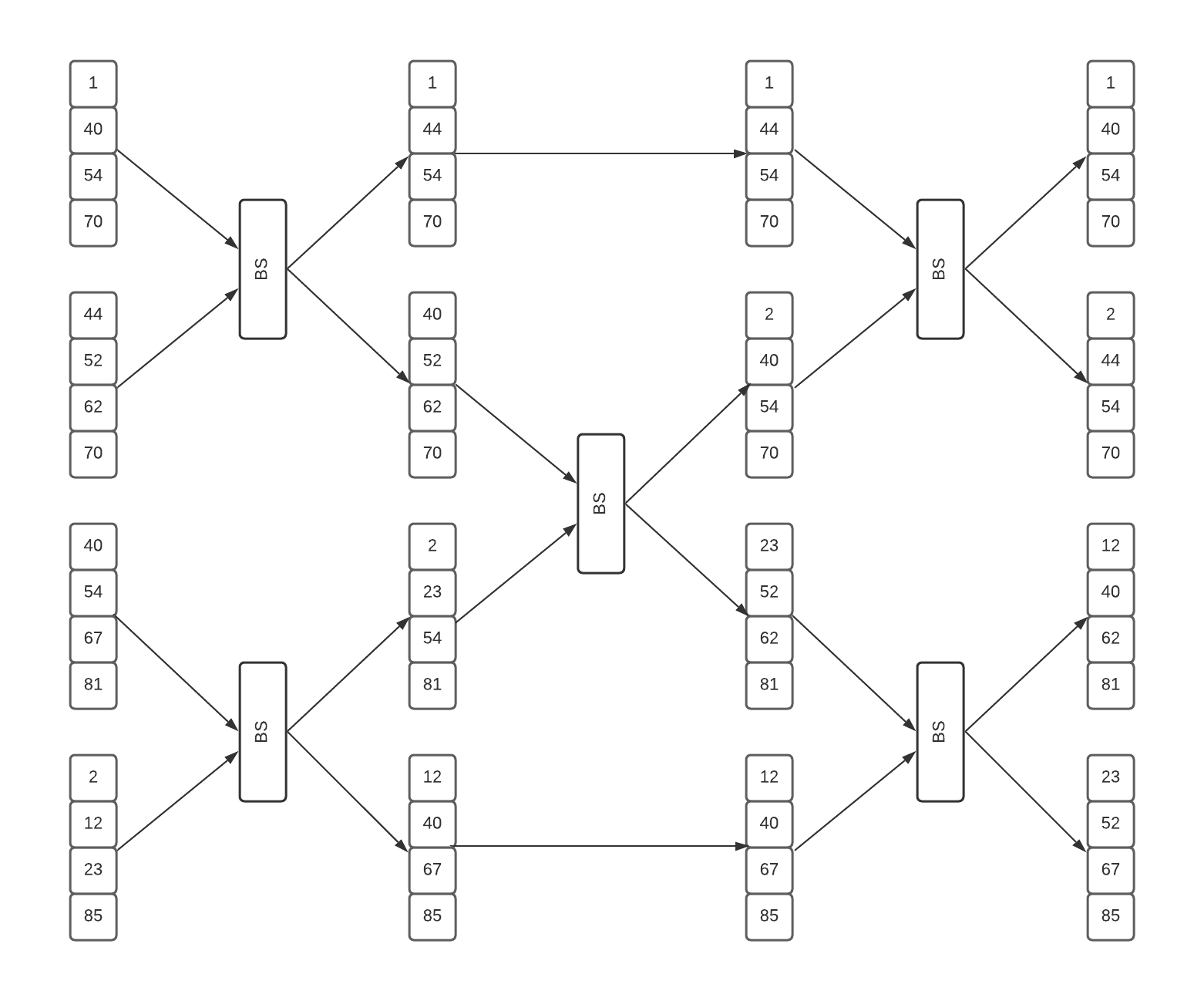
Esta es una red donde se utilizan los bitonic sorter, donde con 5 operaciones se pueden ordenar los valores en 4 vectores con números, cabe destacar que en la entrada debe cumplir las mismas condiciones que para un bitonic sorter, ósea que los vectores vengan ordenados en grupos de 4 independientes. Entonces a grandes rasgos y en forma de resumen se puede ver como funciona la bitocnic merge network en el siguiente diagrama donde BS es bitonic sorter.



**Figura 3.**  Imagen de la explicación del paso 3

### Ejemplo con bitonic merge network paso 1

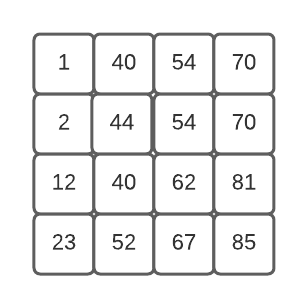
Se ingresan 4 vectores ordenados independientemente y se aplica un bitonic sorter entre el primero y el segundo y entre el tercero y el cuarto.



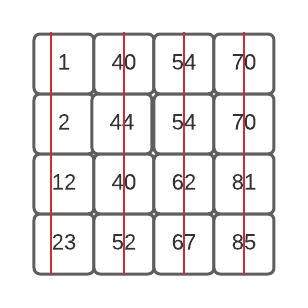
**Figura 3.**  Imagen de la explicación del paso 3

### Ejemplo con bitonic merge network paso 2

Con esto nos damos cuenta que si vemos los números mirando hacia abajo no están ordenados, pero si los colocamos como vectores hacia la derecha y colocamos los vectores uno debajo del otro nos damos cuenta que si están ordenados, pero están ordenados por columnas.



**Figura 3.**  Imagen de la explicación del paso 3



**Figura 3.**  Imagen de la explicación del paso 3

### Ejemplo con bitonic merge network paso 3 (análisis)

Como podemos ver siguen ordenados, así que aquí tenemos 2 opciones, una seria sacar la matriz traspuesta de esta matriz para que los números queden ordenados dentro de cada vector, pero esto se puede o no hacer, ya que para la implementación que se hará no es necesario, puede ser necesario para que sea mas fácil la implementación, pero si no se hace se gana un poco de velocidad al procesar, ya que se será una operación que no se hará, y al momento de implementar el calculo de una matriz traspuesta de con intrínsecas se necesitan 8 operaciones, entonces por cada operación nos ahorramos esas 8 operaciones.

## Implementación

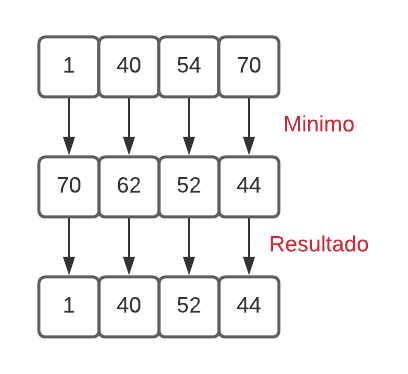
Para la implementación de estos algoritmos utilizaremos el lenguaje c++, específicamente utilizaremos las intrínsecas de Intel. A continuación de explicar cada intrínseca que se utilizara dentro del programa.

### \_mm\_setr\_epi32(int e3, int e2, int e1, int e0)

Esta intrínseca es una de las mas importantes dentro de el programa ya que con esta se colocan los valores que habrán dentro de cada vector para luego poder procesarlos, la función retorna un \_\_m128i ósea un registro de 128 bits de enteros sin signo.

### \_mm\_min\_epi32(\_\_m128i r1, \_\_m128i r2)

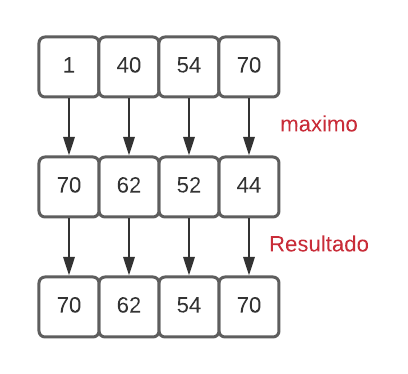
Esta intrínseca obtiene los valores menores de ambos vectores y los coloca en 1 vector como resultado.



**Figura 3.**  Imagen de la explicación del paso 3

### \_mm\_max\_epi32(\_\_m128i r1, \_\_m128i r2)

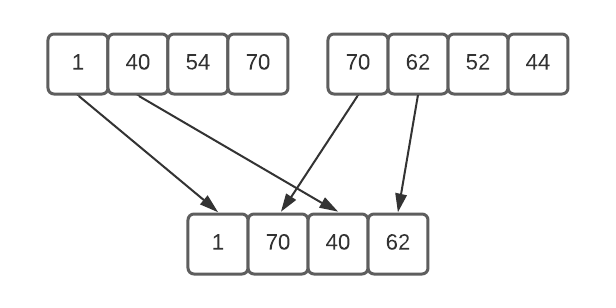
Esta intrínseca obtiene los valores mayores de ambos vectores y los coloca en 1 vector como resultado.



**Figura 3.**  Imagen de la explicación del paso 3

### \_mm\_unpackhi\_epi32 (\_\_m128i r1, \_\_m128i r2)

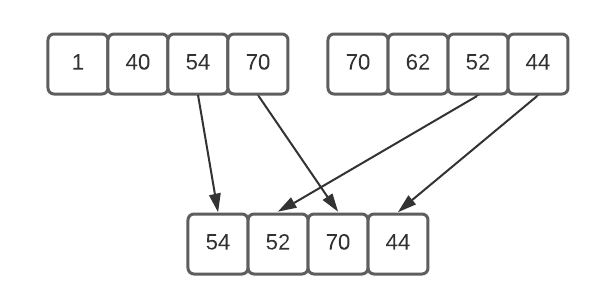
Esta intrínseca nos retorna los bits más significativos de 2 vectores de la siguiente manera.



**Figura 3.**  Imagen de la explicación del paso 3

### \_mm\_unpacklo\_epi32 (\_\_m128i r1, \_\_m128i r2)

Esta intrínseca nos retorna los bits menos significativos de 2 vectores de la siguiente manera.



**Figura 3.**  Imagen de la explicación del paso 3

### \_mm\_shuffle\_epi32 (\_\_m128i r1, \_MM\_SHUFFLE(pos3,pos2,pos1,pos0))

Esta intrínseca lo que hace es cambiar la posición los números dentro del registro esta recibe por parámetro 1 registro \_\_m128i y un \_MM\_SHUFFLE(pos3,pos2,pos1,pos0) que lo que hace esto es decir en la posición que tomaran los valores del registro.

### \_mm\_extract\_epi32 (\_\_m128i r1, int pos1)

Esta intrínseca retorna el valor que hay en la posición de un registro.

# Pruebas realizadas

Para realizar estas pruebas se modifico un poco le código para que este realizara 100 repeticiones para poder sumar los tiempos de ejecución.

## Algoritmo para las pruebas

Se utilizo el algorirmo por defecto de c++ que corresponde a un merge sort que su complejidad es de nlog(n)

## Como se ejecutaron las pruebas

Para ejecutar las pruebas se colocaron timers en ciertas partes claves del código, para medir el tiempo que tardaba en realizar estas secciones, los timer que se utilizaron fueron:

1. Para medir cuanto tiempo se demoraba en cargar el archivo a memoria
2. Para medir el tiempo de la function sort de c++
3. Para medir cuanto tiempo tardaba en hacer el procesamiento vectorial el algoritmo
4. Para medir cuanto tiempo se tardaba la función sort de c++ después de aplicar el algoritmo vectorial
5. Para medir cuanto tiempo tardaba Shell sort en ordenar los números aplicando el procesamiento vectorial

## Como obtuvieron valores promedio

Como a partir de una sola ejecución no podemos obtener conclusiones, se modificó el código para que todo el algoritmo se ejecutara una cantidad x de veces, para el concepto de la realización de las pruebas se ejecuto el algoritmo 1000 veces

## Que tamaño tuvieron las pruebas

Se hicieron pruebas para distintos tamaños de arreglos, para los cuales cada vez se demoraba más la ejecución del código, se hicieron pruebas para las siguientes cantidades de números:

# Resultados de las pruebas

Como resultado general de las pruebas s obtuvo los tiempos mostrados en tabla 1. Cabe destacar que los números en esta tabla están en milisegundos

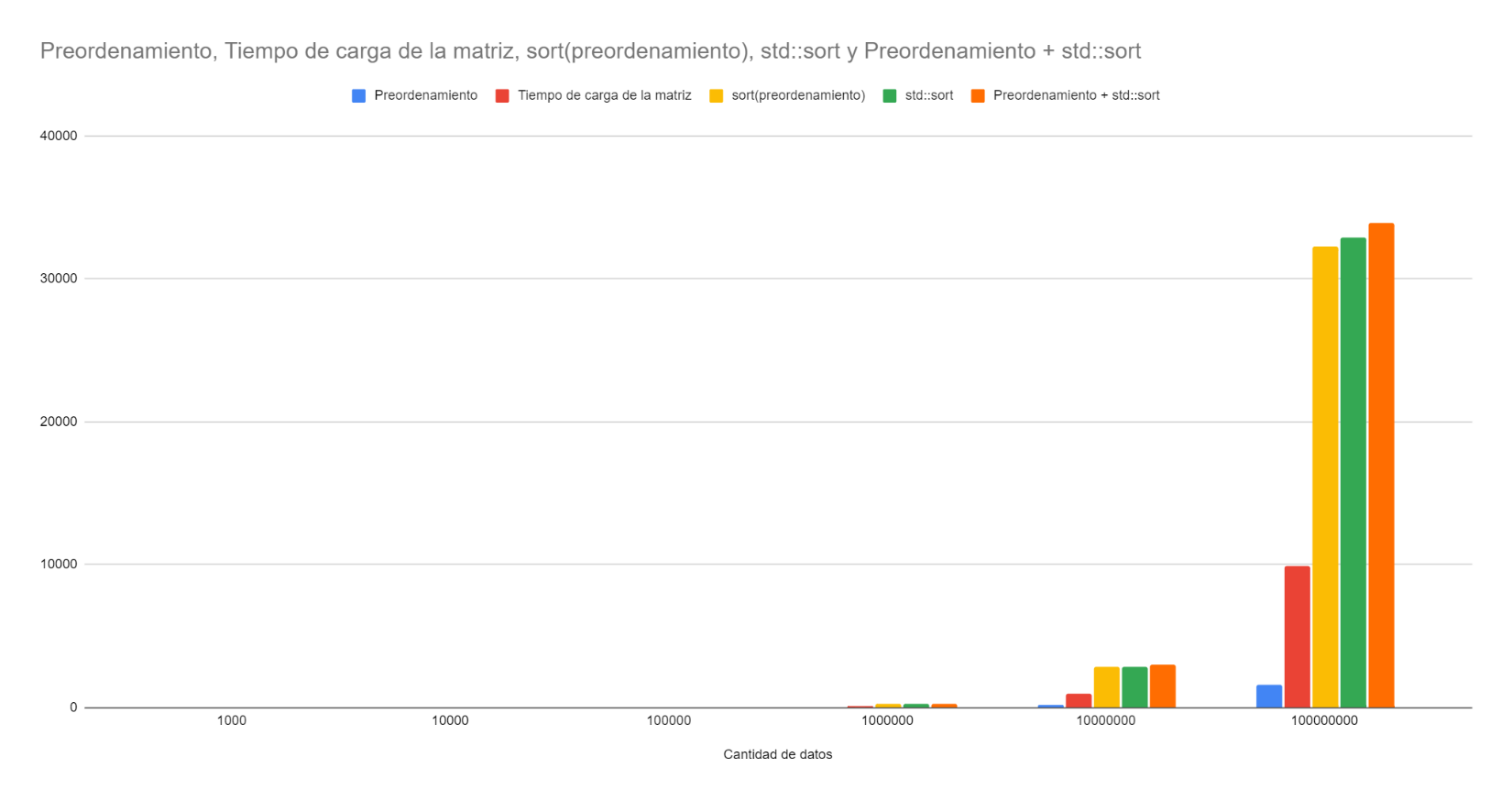
**Tabla 1.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cantidad de datos | Preodenamiento vectorial | Tiempo de carga de la matriz | Ordenamiento de los datos preordenados | Ordenamiento de c++ | Tiempo de ordenamiento total (perodenamiento+ sort c++) |
|  | 0 | 0 | 0 | 0 | 0 |
|  | 0 | 0.06 | 1 | 1 | 1 |
|  | 1.01 | 7.17 | 20..04 | 20.45 | 21.05 |
|  | 15.32 | 81.91 | 244.94 | 249.71 | 260.26 |
|  | 158.58 | 999.63 | 2835.51 | 2880.35 | 2994.09 |
|  | 1595.71 | 9928 | 32303.9 | 32880.8 | 33899.6 |

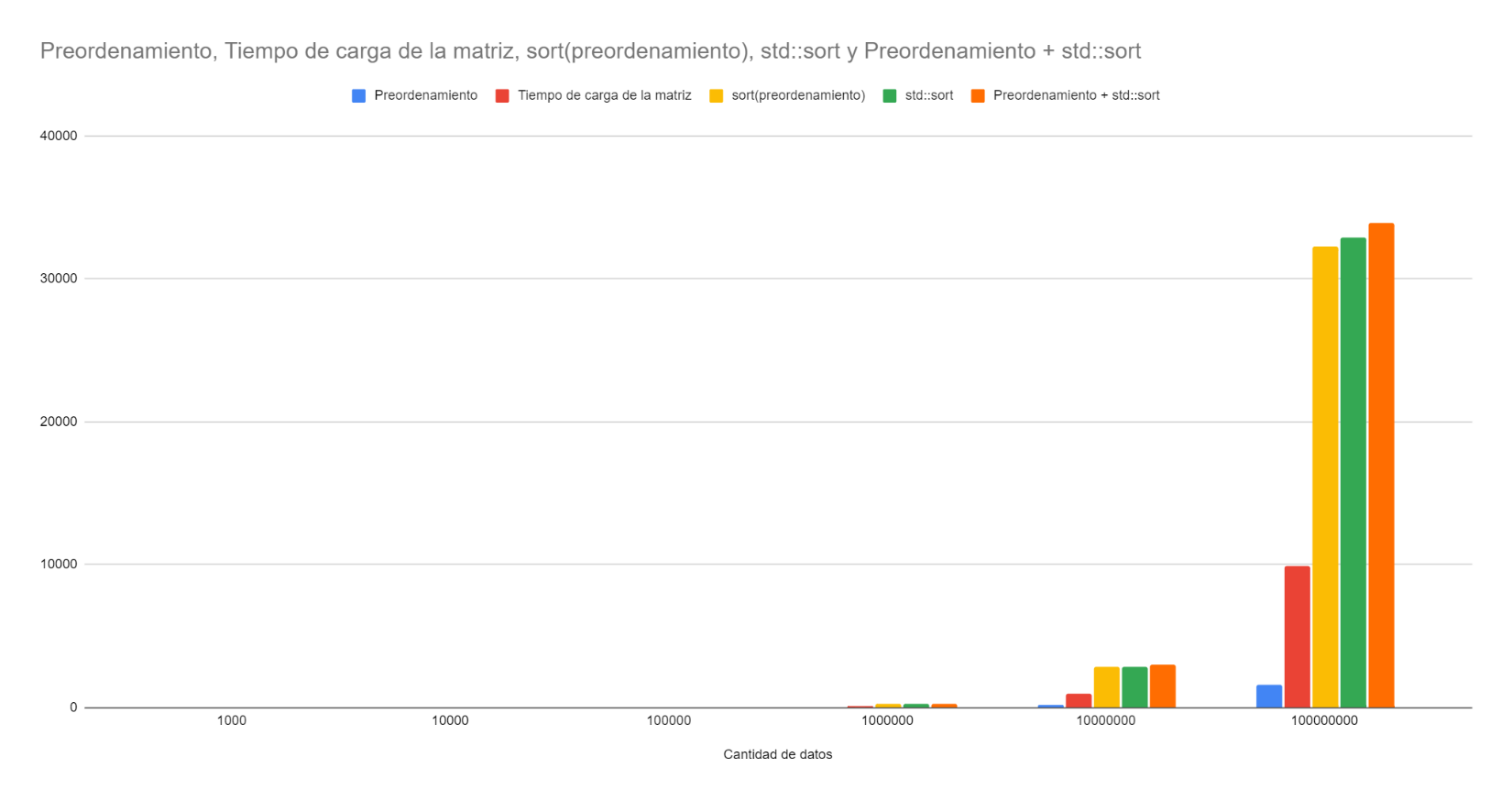
Viendo los resultados de las pruebas podemos determinar que el procesamiento vectorial es prácticamente directamente proporcional a la cantidad de datos, esto se debe a que siempre hace la misma cantidad de operaciones para grupos de 16, así que, si por ejemplo en trabajar los 16 datos se demora 2 milisegundos, para trabajar 32 se demorara 4, y así sucesivamente. En cambio, cuando se utiliza la función sort de c++ al tener que trabajar moviendo números en la memoria cuando hace las comparaciones, a veces después de esas comparaciones no es necesario realizar estos movimientos en memoria, lo cual puede ahorrar tiempo. Pero además se puede observar que cuando se realiza la operación vectorial antes de hacer un sort con merge sort implementado por c++ el ordenamiento es un poco más rápido, de hecho, para el algoritmo se demora medio segundo menos en ordenarlo luego de aplicarle el ordenamiento vectorial. Pero aun así en ningún caso aplicar el algoritmo vectorial hace que el ordenamiento sea mas eficiente, ya que, si o si se tiene que pasar por el ordenamiento vectorial, puede ser que quizá se pueda optimizar un poco el procesamiento vectorial para reducir el tiempo de ejecución, para que este tiempo se haga la diferencia, pero para eso tendríamos que reducir el tiempo a 1/3 y un poco más del tiempo que se demora la ejecución vectorial en el código.

En los siguientes gráficos se mostrarán los datos gráficamente para poder compararlos de manera más sencilla. De hecho, se mostrarán gráficos, eliminando los datos mas grandes para poder visualizar los datos mas pequeños.

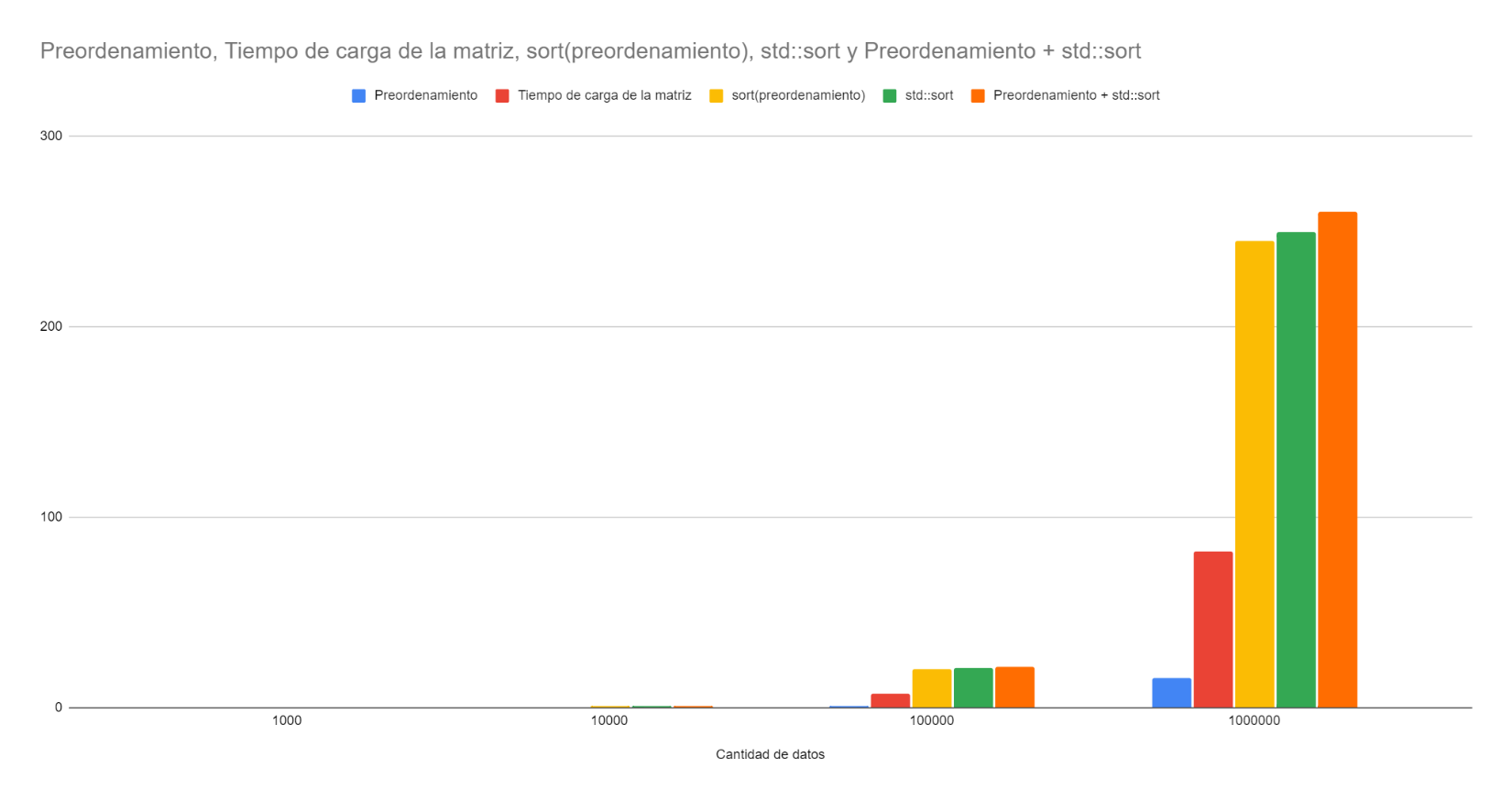
Además, cabe aclarar que algunos tiempos son marcados como “0” ya que el tiempo es tan pequeño que no se alcanza a medir en milisegundos



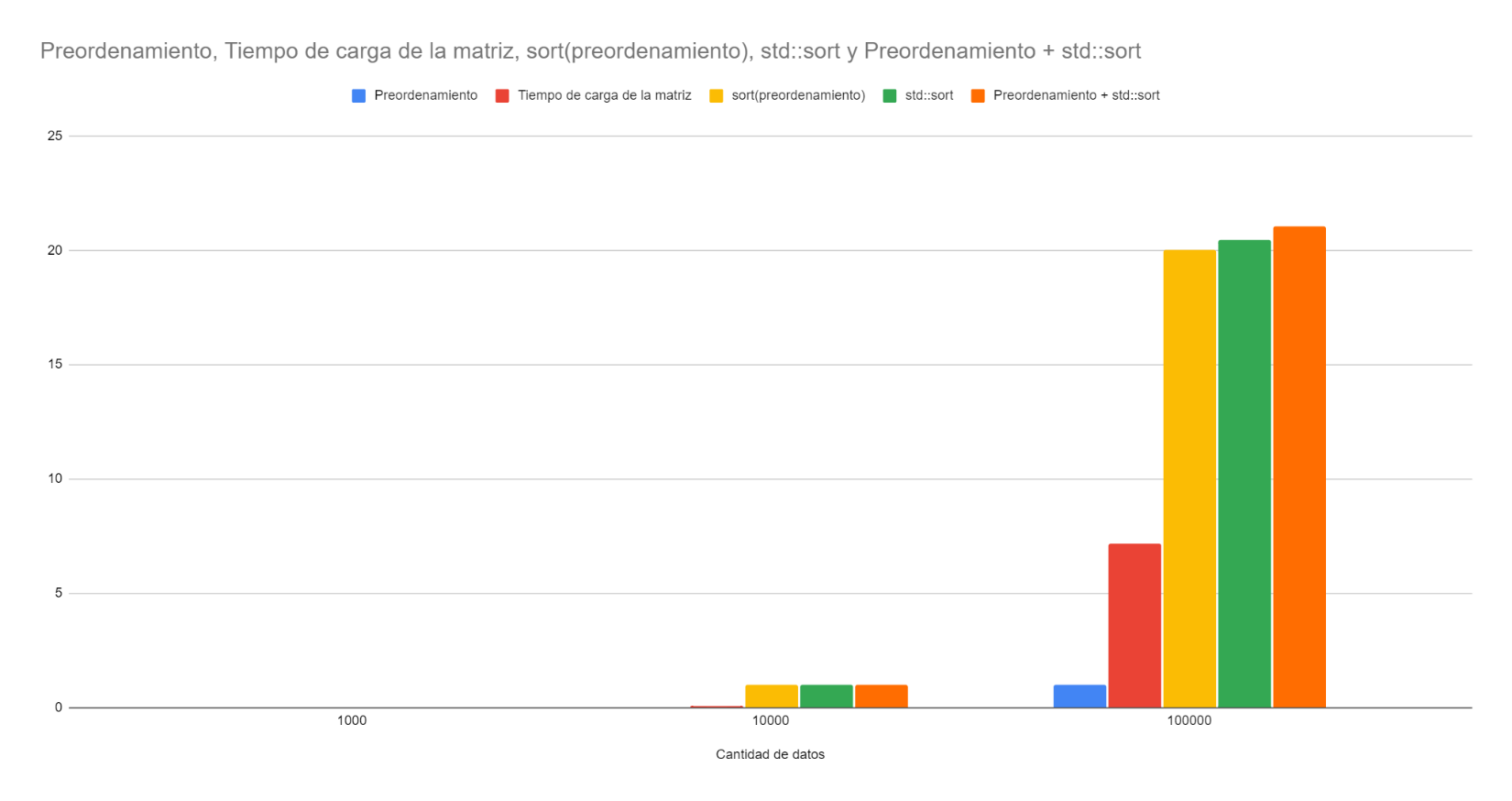
**Figura 2.** En este diagrama de flujo se puede ver el funcionamiento del algoritmo a alto nivel



**Figura 3.** En este diagrama de flujo se puede ver el funcionamiento del algoritmo a alto nivel



**Figura 4.** En este diagrama de flujo se puede ver el funcionamiento del algoritmo a alto nivel



**Figura 5.** En este diagrama de flujo se puede ver el funcionamiento del algoritmo a alto nivel

## Calculo de speedup

Si bien el cálculo de speedup se utiliza para comparar el tiempo de ejecución de un algoritmo con distintas cantidades de Cores, al final es una métrica de comparación entre tiempos que nos dice en porcentaje cuanto mejor es un algoritmo que otro hablando de tiempo de ejecución, por lo tanto al ser una métrica de comparación de tiempo podemos aplicarla para determinar la eficiencia de este algoritmo de la siguiente forma:

(Tiempo de ejecución normal/ tiempo de ejecución con procesamiento vectorial)-1

Teniendo esto la formula nos queda en

Para datos 🡪 (1/1)-1=0 🡪0%

Para datos 🡪 (20.45/21.05)-1=-0.028 🡪-2.8%

Para datos 🡪 (249.71/260.26)-1=-0.04 🡪 -4%

Para datos 🡪 (2880.35/2994.09)-1=-0.037 🡪 -3.7%

Para datos 🡪 (32880/33899.6)-1=-0.030 🡪 -3%

Entonces podemos ver que los porcentajes son negativos , y varían según la cantidad de datos, esto quiere decir que es menos eficiente que la función predeterminada de c++ para ordenar un arreglo de números.

# Conclusiones

El procesamiento vectorial nos puede ayudar de muchas maneras ya que se puede trabajar mucho más rápido con algunos datos al trabajar con 4 números en 1 solo ciclo de procesador, pero de la misma manera puede ser llamado un arma de doble filo, ya que al estar subiendo y bajando números a la memoria vectorial puede retrasar el procesamientos en algún algoritmo, por lo tanto para aplicarlo había que determinar si es factible sacrificar ese tiempo de transferencia a memoria vectorial o no, en el caso de el problema que tenemos presente no es muy factible ya que el tiempo vectorial mas el tiempo que se necesita para ordenar los números preordenados es mayor a lo que demora el algoritmo predeterminado de c++, por lo tanto para esto no es recomendable aplicar procesamiento vectorial para ordenar números. Pero por lo que se puede ver en el estudio de speedup se ve que el porcentaje de ineficiencia va subiendo, pero llegado a un punto comienza a bajar, no se puede saber con precisión si seguirá bajando ya que no se hizo pruebas como mas de diez mil millones de datos pero se puede estimar que para muchos mas datos, esto puede comenzar a subir la eficiencia hasta ganarle a la función predeterminada de c++.