DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Abril de 2019 — Organización del Computador II

Integrante	LU	Correo electrónico
Sansone, Agustín	99/17	agustinsansone7@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Resumen

En este trabajo contruiremos reconocedores de dígitos en los lenguajes *C* y *Assembly x64* para la base de datos provista por MNIST[1] (Modified National Institute of Standards and Technology). En particular, utilizaremos la técnica de **kNN**[2] (*k-Nearest Neighbors*) para detectar qué dígito aparece representado en una imagen dada, comparándola con un conjunto de imágenes de entrenamiento. También, emplearemos la técnica de **PCA**[3] (*Principal Component Analysis*) con la finalidad de reducir además el tiempo de cómputo del método anterior.

Dado el gran orden de operaciones vectoriales y matriciales involucrados en ambos métodos, reimplementaremos parcialmente el código con el objetivo de poner en práctica lo aprendido en la materia sobre **SIMD**[4] (*Single Instruction, Multiple Data*). A partir de esto, esperamos una significante mejora en los tiempos de ejecución en todos los casos. Con la experimentación, se buscará encontrar una cantidad adecuada de vecinos a considerar y componentes principales a evaluar, para *kNN* y *PCA*, respectivamente. Además, vamos a querer corroborar de manera empírica las optimizaciones de las operaciones paralelas que buscamos con las implementaciones que utilizan *SIMD*.

Palabras Clave: Reconocedor de dígitos, kNN (*K-Nearest Neighbors*), PCA (*Principal Component Analysis*), método de la potencia con deflación, SIMD (*Single Instruction, Multiple Data*).

[1] http://yann.lecun.com/exdb/mnist/
[2] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
[3] https://en.wikipedia.org/wiki/Principal_component_analysis
[4] https://en.wikipedia.org/wiki/SIMD

2. Introducción

El problema que resuelve este trabajo se basa en poder reconocer dígitos con precisión y velocidad. Esto tiene usos tanto teóricos como prácticos. Para lograrlo, presentaremos y desarrollaremos dos algoritmos, *kNN* y *PCA*. Con el fin de ponerlos a prueba, contaremos con una base de datos de 70.000 imágenes en blanco y negro. Cada instancia (imagen) de la base de datos se encontrará representada inicialmente por una imagen de 28x28; y a cada uno de sus píxeles le corresponderá un número entero entre 0 y 255.

2.1. Base de datos

Como a los algortimos no les importa mucho la imagen en sí para reconocerla, sino la secuencia de números entre 0 y 255 que la componen, por consiguiente, podemos darle como input un vector de 784 ($= 28^2$) posiciones, en vez de la imagen de 28x28 píxeles. Luego, en vez de la base de datos conste de varias imágenes de 28x28, podemos hacer que sean muchas secuencias de vectores de 784. De este modo, el algoritmo no tiene que transformar cada imagen a un vector cada vez que lee la base de datos.

De las 70.000 imágenes, utilizaremos 60.000 para entrenamiento y 10.000 para testing. De esta forma, contaremos con los archivos *train_ entries.txt* y *test_ entries.txt*, que contendrán 60.000 y 10.000 líneas compuestas por 784+1 números, respectivamente. El primer número corresponderá al dígito que efectivamente estaba escrito en la imagen, y los otros 784 números serán los que conformaban esa imagen. En la carpeta Test one img se encuentra un script para reconstuir cualquier imagen a formato *.png* por si efectivamente queremos verla.

El primer paso en estos métodos será guardar la base de datos en dos estructuras. Como los píxeles son números entre 0 y 255 y $256 = 2^8$, basta con 8 bits (1 byte) para guardar cada uno. Entonces cada imagen será, en principio, un vector de 784 bytes. De este modo, tendremos un vector de 60.000 de estos y otro de 10.000, para la data de entrenamiento y testing, respectivamente.

2.2. Metodología

Para poder clasificar una imagen usamos el algoritmo *kNN*. Este consiste en, a partir del conjunto de entrenamiento, buscar las k imágenes más cercanas (para alguna definición de cercanas) y quedarnos con la que más veces apareció de esas k más cercanas.

Este algoritmo tiene la desventaja de que sus tiempos de corrida son muy sensibles a la dimensión de la imagen. Una alternativa para contrarrestar este problema es la técnica de *PCA*. El principal objetivo de este método es poder reducir la dimensión de las imágenes, para luego poder aplicar *kNN* sobre imágenes de dimensiones más chicas.

En este trabajo práctico, además de comparar ambas técnicas entre sí, nos interesará particularmente compararlas con sus respectivas implementaciones en *ASM*, para corroborar si las optimizaciones logradas usando instrucciones de *SIMD* tienen realmente un impacto en la performance.

3. Desarrollo

3.1. Objetivos

Buscaremos ver que :

- Los métodos logren efectivamente reconocer las imágenes.
- Utilizar *SIMD* mejore los tiempos de ejecución.

3.2. *kNN*: k-Nearest Neighbors

Para el algoritmo de *kNN*, contamos con un conjunto de entrenamiento de 60.000 vectores de 784 posiciones y una imagen desconocida x que queremos reconocer. Lo que haremos primero será buscar los k vectores del conjunto de entrenamiento más *cercanos* a x.

Pero, ¿cómo calculamos los vectores más *cercanos*? Si los vectores de la base de datos tuvieran dos posiciones, podríamos pensarlos como puntos en el plano; en este caso la distancia entre dos puntos $P = (x_1, x_2)$, $Q = (y_1, y_2)$ se calcularía como:

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

Si tuvieran tres posiciones, es la distancia entre dos puntos del espacio, que se calcularía como (siendo ahora $P = (x_1, x_2, x_3)$, $Q = (y_1, y_2, y_3)$):

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2}$$

Análogamente funciona para cualquier cantidad de posiciones[5]. En particular, para vectores de 784 posiciones, si $P = (x_1, x_2, ..., x_{784})$, $Q = (y_1, y_2, ..., y_{784})$, su distancia estará dada por:

$$\sqrt{\sum_{i=1}^{784} (x_i - y_i)^2}$$

Con esta definición, una vez encontrados los k vectores más cercanos, el resultado se corresponderá con la moda de estos k *vecinos* seleccionados (es decir, el que más veces aparece).

La intuición de este método es tomar como resultado de nuestra predicción el dígito más predominante de cierto entorno (definido por k) en un espacio 784-dimensional.

3.3. *PCA*: Principal Component Analysis

Como mencionamos, esta técnica sirve para reducir la dimensión de las muestras (en este caso imágenes) y así aplicar *kNN* sobre una base de dimensiones más chicas. Para lograr

[5] https://en.wikipedia.org/wiki/Euclidean_distance

esto, primero se calcula la matriz de covarianza[6] M en base al conjunto de entrenamiento. Esta matriz se define como $M = X^t X$, donde $X \in \mathbb{R}^{60,000 \times 784}$ es la matriz que contiene en la i-ésima fila al vector $(x_i - \mu)/\sqrt{60{,}000 - 1}$, μ es el promedio coordenada a coordenada de los datos y x_i es la i-ésima imágen de la base de entrenamiento.

A partir de esto, se computan los primeros α autovectores[7] de M (es decir, los α autovectores asociados a los autovalores más grandes en módulo). En la siguiente sección veremos detalladamente cómo.

Una vez computados estos primeros α autovectores $v_1, v_2, .., v_\alpha$ estamos en condiciones de calcular la transformación característica. Esta se define como $tc(x) = (v_1 x, v_2 x, \ldots, v_\alpha x)^t$. Es decir, que dada una imagen x de 784 posiciones, esta nos devuelve una de sólo α posiciones.

Una vez tranformada la base de datos y la imagen a reconocer con esta función, sólo queda aplicar *kNN* sobre esta base transformada.

Respecto al tipo de datos de la implementación, notemos que como los autovalores de M son números reales, la transformación característica transormará nuestro vector de enteros (números entre 0 y 255) en reales, por lo que ya no podremos aprovechar esta propiedad. Es por esto que ahora guardaremos el resultado de aplicar la transformación a una imagen en un vector de α *floats* en vez uno de 784 bytes.

3.4. Cálculo de Autovalores y Autovectores

Para calcular los primeros α autovectores de la matriz M, utilizaremos el ***método de la potencia***[8] *con deflación* y conseguiremos una buena aproximación de estos.

En general, dada una matriz A con $|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$ autovalores correspondientes a los autovectores $w_1, w_2, ..., w_n$ que conforman una base, el algoritmo de *Método de la potencia*, que computa el autovalor más grande en módulo y su respectivo autovector, se define como:

Algorithm 1 Calcula w_1 y λ_1 utilizando el método de la potencia

function MetodoDeLaPotencia(A)
  $v \leftarrow x_0$
  for $n \in [1 \ldots$ CantidadIteraciones$]$ **do**
    $v \leftarrow \frac{Av}{\|Av\|_2}$
  $\lambda \leftarrow \frac{v^t A v}{v^t v}$
  return v, λ

donde CantidadIteraciones es una constante definida previamente y x_0 es un vector inicial cualquiera.

[6] https://en.wikipedia.org/wiki/Covariance_matrix
[7] https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors
[8] https://en.wikipedia.org/wiki/Power_iteration

Este método es correcto, ya que luego de la iteración número k tendríamos que el valor actual de v (aunque normalizado) es:

$$A^k x_0 = A^k \sum_{i=1}^{n} \alpha_i w_i = \sum_{i=1}^{n} \alpha_i A^k w_i = \sum_{i=1}^{n} \alpha_i \lambda_i^k w_i$$
$$= \lambda_1^k \sum_{i=1}^{n} \alpha_i \left(\frac{\lambda_i}{\lambda_1}\right)^k w_i$$

Pero $\left(\frac{|\lambda_i|}{|\lambda_1|}\right)^k$ tiende a 0 $\forall i = 2 \ldots n$ ya que $|\lambda_1| > |\lambda_i|$. Entonces, $A^k x_0$ (normalizado) tenderá a w_1, que es el autovector que queríamos encontrar. Y para hallar su autovalor asociado, simplemente calculamos $\lambda_1 = \frac{w_1^t A w_1}{w_1^t w_1}$, como hace el algoritmo mostrado.

Notemos que este método funciona para conseguir el autovector asociado al autovalor más grande en módulo, pero a nosotros nos interesan los primeros α. Lo que se puede hacer, para conseguir el resto, es usar este método repetidamente modificando la matriz A. Es decir, la idea es que si A tiene los autovalores

$$|\lambda_1| > |\lambda_2| > \ldots |\lambda_n|$$

al aplicar el *Método de la potencia*, obtenemos efectivamente λ_1 y w_1; pero después de esto modificamos la matriz A para que quede con autovalores

$$|\lambda_2| > |\lambda_3| > \ldots |\lambda_n| > |\lambda_1'|$$

y al aplicar ahora el método obtener λ_2 y w_2; y así sucesivamente hasta conseguir los primeros α.

Esto lo podemos hacer efectivamente modificando A de la siguiente forma:

$$A := A - \lambda_1 \frac{w_1 w_1^t}{w_1^t w_1}$$

ya que

$$\left(A - \lambda_1 \frac{w_1 w_1^t}{w_1^t w_1}\right) w_1 = A w_1 - \lambda_1 \frac{w_1 w_1^t w_1}{w_1^t w_1} = \lambda_1 w_1 - \lambda_1 w_1 = 0$$

y dado un w_i con $i \neq 1$

$$\left(A - \lambda_1 \frac{w_1 w_1^t}{w_1^t w_1}\right) w_i = A w_i - \lambda_1 \frac{w_1 w_1^t w_i}{w_1^t w_1} = \lambda_i w_i - 0 = \lambda_i w_i$$

pues w_i es linealmente independiente con w_1 dado que pertenecen los dos a la misma base. A este procedimiento se le llama *deflación*.

3.4.1. Almacenando la información

En la implementación de los métodos explicados utilizaremos las siguientes estructuras para guardar la información tanto de la base de datos como de otros cómputos que requieran ser calculados:

```
struct info{
    int index; //Indice en la base de datos
    u_int8_t entry[entry_size]; //Data de la img de 28*28
    int value; //Que numero dice la img
    float dif_actual; //Que tan lejos esta esta img
    float entry_PCA[max_alpha]; // tc(entry)
} train_data[train_size], test_data[test_size];

int alpha;
float mu[entry_size];
float X_transpose[entry_size][train_size];
float M[entry_size][entry_size];
float M_eigenvectors[max_alpha][entry_size];
```

Figura 1: Structs para almacenar la información

Por imagen, contaremos con un struct *info*, que guardará todo lo que podemos llegar a necesitar computar sobre esta:

- En *index* guardamos el índice de la imagen en la base de datos.
- *entry* será el vector de 784 bytes que contrendrá toda la información de la imagen original.
- En *value* guardaremos el dígito que efectivamente estaba escrito en esa imagen.
- En *dif_ actual* guardaremos la diferencia de esa imagen contra la que estamos clasificando cuando usemos *kNN*. Nótese que, como la diferencia tiene una raíz cuadrada, no deberíamos usar tipos enteros.
- En *entry_ PCA* guardamos el resultado de aplicar la transformación característica a esa imagen. Como explicamos en la subsección de *PCA*, será conveniente utilizar *floats* para guardarlo.

Para almacenar toda la base de entrenamiento y tesing, tendremos un vector *train_ data* de 60.000 y otro *test_ data* de 10.000 de esta estructura, respectivamente.

Para *PCA* además tenemos que computar la transformación característica, que conlleva aún más cálculos:

- En *alpha* guardamos el valor α que enunciamos.
- En *mu* guardamos μ. Nótese que cada coordenada corresponde a un promedio de coordenadas, por lo que difícilmente sea un entero, razón por la cual usamos *floats*.

- $X\_$ *transpose* es la transpuesta de la matriz X que explicamos. Como esta depende de μ, también vamos a usar *floats*.
- En M guardamos X^tX. Y como depende de X, también usamos *floats*.
- En $M\_$ *eigenvectors* guardamos los α autovectores asociados a los primeros autovalores de M. Claramente aquí también utilizaremos tipos de datos no enteros.

3.5. Optimizaciones

3.5.1. Mejoras para *kNN* con *SIMD*

Recordemos que *kNN* tiene que ordenar las imágenes de la base de entrenamiento según su cercanía a la imagen que queremos clasificar para así quedarse con las k más cercanas.

Para poder ordenarlas, primero tenemos que calcular efectivamente la distancia entre cada imagen de entrenamiento y la que vamos a clasificar, cálculo que definimos como:

$$\sqrt{\sum_{i=1}^{784}(x_i - y_i)^2}$$

Para esto, podemos restar con *psubd* las dos imagenes para calcular los $x_i - y_i$, luego hacer los cuadrados con *pmulld* para tener los $(x_i - y_i)^2$ y después hacer las sumas con *paddd* para llegar a $\sum_{i=1}^{784}(x_i - y_i)^2$. Finalemente hacemos una raíz con *sqrtps* para obtener el resultado. Recordemos que las imágenes son vecrores de 784 bytes, y como en un XMM entran 16 bytes, podemos levantar de a 16 bytes, que sería de a 16 píxeles en simultáneo. Entonces para este cómputo, bastará con hacer $784/16 = 49$ pedidos a memoria por imagen.

3.5.2. Mejoras para *PCA* con *SIMD*

Primero tenemos que calcular la matriz de covarianza M:

Para esto el primer paso es computar μ, el promedio de las coordenadas de todas las imágenes. Podemos arrancar con el vector *mu* en 0 (con la función *fijar_ASM*), luego recorremos todas las imágenes de entrenamiento y vamos sumando su *entry* (con la función *sumar2_ASM*). Notar que bastará con $784/16 = 49$ pedidos a memoria por imagen, ya que podemos levantar de a 16 píxeles. Una vez hecha toda la sumatoria, para que sea un promedio debemos dividir todo por el total (usamos la función *dividir_ASM*). Pero como este vector lo queremos en *floats*, tendremos que hacer la conversión (con *cvtdq2ps*), para luego poder promediar las coordenadas.

Ahora, para calcular M, notemos que:

$$M_{ij} = fila_i(X^t) * columna_j(X) = fila_i(X^t) * fila_j(X^t)$$

Por lo que para calcular cada celda de M, tendremos que realizar una multiplicación vectorial. Notar que las filas de $X\_$ *transpose* son vectores de 60.000 *floats*. Por esto, procesaremos de

a $16/4 = 4$ *floats* en simultáneo. Luego, lo único que hay que hacer es multiplicar y sumar los XMM, para esto usaremos las instrucciones *mulps* y *addps*.

Una vez calculada la matriz M, lo que nos interesa es calcular sus primeros α autovectores. Como ya enunciamos, para esto utilizamos el *Método de la Potencia con Deflación*. Para la parte de *Método de la Potencia*, empezando con un vector x_0, necesitamos calcular suficientes veces $x_0 := \frac{Mx_0}{\|Mx_0\|_2}$. Para calcular Mx_0, como M es una matriz de 784x784 y x_0 es un vector de 784, hay que hacer 784 multiplicaciones vectoriales, operación que ya mencionamos cómo optimzar. Luego, sólo resta normalizar el vector, que equivale a dividir el vector por la norma (operación que también ya contemplamos). Y calcular la norma es lo mismo que una multiplicación vectorial con el mismo vector, sólo que al final se retorna la raíz cuadrada del resultado. Para la parte de *Deflación*, estando en el i-ésimo paso, simplemente hay que recalcular:

$$M := M - \lambda_i \frac{w_i w_i^t}{w_i^t w_i}$$

Este cálculo involucra multiplicar, dividir y restar vectores, cosa que ya entendemos cómo optimizar. Como ahora estamos trabajando con *floats*, utilizaremos intrucciones para operar con $16/4 = 4$ valores simultáneamente.

Recién calculados los primeros α autovectores $v_1, v_2, ..., v_\alpha$ estamos en condiciones de aplicarle la *transformación caracterísica* a cada imagen de la base de entrenamiento y a la imagen que querramos predecir. Recordemos que se define como $tc(x) = (v_1 x, v_2 x, \ldots, v_\alpha x)^t$. Notemos que resulta de hacer α multiplicaciones vectoriales, con la salvedad de que primero tenemos que convertir x a un vector de *floats* ya que los autovectores son de *floats* como lo es el resultado que queremos obtener. Haremos las conversiones y las mutiplicaciones intercaladamente con la función *mult_ vectorial2* para tratar de optimizar lo máximo posible.

Una vez transformada toda la base de entrenamiento y la imagen a reconocer con la transformación característica, sólo resta aplicar *kNN* en la nueva base. Notemos que ahora tenemos vectores de α *floats* en vez de 784 bytes, por lo que para calcular la distancia entre dos imágenes estaremos procesando de a $16/4 = 4$ posiciones.

4. Resultados

4.1. Experimentación

En esta sección buscaremos valores adecuados de k y α para kNN y PCA, respectivamente. Luego compararemos cada técnica con sus variantes en C y ASM, además de entre sí. Para esto, también compilaremos todos los códigos escritos en C con y sin O3. Esperamos que aún así las implementaciones en ASM con instrucciones *SIMD* sigan siendo superiores que las optimizaciones provistas por O3 en cuanto a tiempos de ejecución.

4.1.1. *kNN*: C vs *ASM*

Para encontrar un valor de k admisible, utilizaremos kNN para clasificar las 10.000 imágenes de testing. Esto lo haremos para 2, 5, 20, 50, 200, 500, 2.000, 5.000 y 20.000 vecinos, ya que estos valores de k son suficientemente espaciados y representativos. A continuación presentamos para los distintos valores el accuracy, es decir, el porcentaje de aciertos que tuvo el método:

k	Accuracy
2	96.270 %
5	96.880 %
20	96.250 %
50	95.340 %
200	92.890 %
500	90.390 %
2.000	82.810 %
5.000	74.260 %

A partir de lo anterior, observamos que con $k = 5$ alcanzamos el mejor resultado con un acuraccy de casi el 97 %, o sea que con este valor se pudo reconocer válidamente el 97 % de las 10.000 imágenes. Por esto conservaremos este valor de k para los próximos experimentos.

Ahora que ya fijamos este valor de k, podemos hacer una comparación entre la implementación en C con y sin O3 y ASM. Seguidamente mostramos cuánto tomó correr las 10.000 instancias de testing para cada caso:

Implementación	Tiempo (en segundos)
C sin O3	2605.1470
C con O3	948.7025
ASM	875.0909

De esto último, podemos decir que la implementación en *ASM* tomó efectivamente menos tiempo que la de C con O3. Sin embargo, el factor que más resalta es la diferencia entre *ASM* y *C* sin optimizaciones. Para este caso, el algoritmo implementado en *ASM* con instrucciones *SIMD* llega a tardar menos de un 30 % del tiempo que toma en *C*.

4.1.2. *PCA*: C vs ASM

Para buscar un valor de α apropiado, aplicamos *PCA* a todas las imágenes de testing con el valor de k igual a 5 como habíamos fijado. Seguidamente se muestra el accuracy para valores de α iguales a 4, 16, 40, 160 y 400. Elegimos estos en particular, porque están suficientemente distribuidos y son representativos:

α	Accuracy
4	64.140 %
16	94.370 %
40	97.200 %
160	97.370 %
400	97.030 %

A partir de lo anterior vemos que, aunque no por mucho, el mejor valor de α de los probados es el de 160, alcanzando un accuracy que ronda el 97 %, valor que se corresponde con el accuracy al que habíamos llegado para *kNN* con el mejor valor de k en el experimento anterior.

Una vez fijado este valor de α igual a 160, podemos comparar los tiempos de aplicar la técnica de *PCA* a todas las imágenes de testing para la implementación en C con y sin O3 y ASM:

Implementación	Tiempo (en segundos)
C sin O3	1146.9885
C con O3	848.8649
ASM	685.0909

A partir de lo anterior, vemos que la implementación en *ASM* con *SIMD* toma también menos tiempo que la de C con O3. A su vez, la diferencia con C sin O3 es aún mayor, como era de esperar. En este caso, la implementación con *ASM* tomó menos de un 60 % del tiempo de ejecución que requirió la misma en *C*.

4.1.3. *kNN* vs *PCA*

Habiendo encontrado valores de k y α aceptables y analizado *kNN* y *PCA* por separado, podemos realizar una comparación entre ambos métodos:

Implementación	Tiempo kNN (seg)	Tiempo PCA (seg)
C sin O3	2605.1470	1146.9885
C con O3	948.7025	848.8649
ASM	875.0909	685.0909

En cuanto a la confiabilidad de predicción, vimos que ambos alcanzaron un accuracy que ronda el 97 %. Esto es remarcable para la técnica de *PCA*, ya que habiendo reducido

las dimensiones de las imágenes, no parece haber perdido información. Por otro lado, en cuanto a tiempos de ejecución sí notamos claras diferencias. Observamos que en todos los casos, *PCA* toma menos tiempo que *kNN*. Esto era de esperar ya que el principal objetivo de la técnica de *PCA* es justamente transformar la base de datos a una más pequeña para ahorrarse cómputos al aplicar *kNN*. Por último, notamos que las implementaciones en *ASM* con instrucciones *SIMD* siempre tomaron menos tiempo que las de *C* con y sin O3. A su vez, C compilado con O3 tardó menos que sin O3, aunque esto era previsible.

4.2. Conclusión

Luego de revisar los resultados anteriores, podemos concluir que para esta base de datos k igual a 5 es un valor admisible para la técnica de *kNN*, alcanzando un accuracy de casi un 97 %.

Similarmente, con este k fijado, llegamos a que un α igual a 160 es un valor favorable para la técnica de *PCA*, obteniendo un accuracy que supera el 97 %.

A su vez, hemos corroborado empíricamente que las implementaciones en *ASM* utilizando instrucciones de *SIMD* superan ampliamente los tiempos de ejecución de sus respectivas implementaciones en el lenguaje *C*. Para *kNN*, logramos bajar el tiempo de ejecución a un 30 %, mientras que para *PCA* conseguimos llegar correrlo en un 60 % del tiempo que tomaba en *C*. Además de esto, corroboramos que compilar con O3 también mejora notablemente los tiempos de ejecución; sin embargo, nuestras optimizaciones con *SIMD* siguieron siendo superiores.

Como último comentario, quiero agregar que este trabajo me permitió profundizar y poner en práctica gran parte del conocimiento aprendido en Organización del Computador II. Llegamos a implementar efectivamente clasficadores de dígitos funcionales y eficientes. Además, pudimos corroborar objetivamente las notables mejoras que se pueden alcanzar utilizando *SIMD*.