Acelerando la Comparación de Huellas Dactilares basadas en Agrupaciones Deformables de Minucias

Andrés Jesús Sánchez[1], Luis Felipe Romero[2] y Siham Tabik[3]

Resumen— **El reconocimiento de huellas dactilares es considerado como uno de los métodos de acreditación biométrica más utilizado en la actualidad. La identificación de una huella requiere realizar la comparación de sus minucias con todas las minucias que conforman cada huella perteneciente a una base de datos. Los algoritmos de comparación de huellas más avanzados son muy costosos desde el punto de vista computacional, e ineficientes cuando trabajan sobre bases de datos de grandes dimensiones. En este trabajo, se han incluido diversos métodos para acelerar el algoritmo DMC (el método de comparación de huellas dactilares más preciso basado únicamente en minucias). En particular, se han reescrito en C++ las funciones del algoritmo con mayor carga computacional; se ha creado una librería estática en C++ donde se ejecuta el algoritmo de comparación DMC modificado y que se conecta con el código original en C# utilizando para ello un proyecto de librería de clases de CLR. La solución reimplementa funciones críticas tales como la cuenta del número de bits con valor uno mediante la inclusión de una librería de *PopCount* en C++ y el uso del cuadrado de la distancia Euclídea para el cálculo de la vecindad de las minucias. Los resultados experimentales muestran una reducción significativa del tiempo de ejecución de las funciones optimizadas dentro del algoritmo DMC. Por último, se presenta como trabajo futuro una nueva estrategia de procesamiento paralelo de los datos de las huellas, en la que se tiene en cuenta la jerarquía de memoria.**

Palabras clave— **Procesamiento de Huellas Dactilares, Optimización de Caché, Interoperatividad de los Lenguajes de Programación.**

I. Introducción

La tarea del reconocimiento de huellas dactilares ha sido ampliamente estudiada en estos últimos años. La razón principal por la que se utiliza esta parte del cuerpo humano para identificar a las personas, reside en el hecho de que las huellas dactilares son únicas y no cambian con el paso del tiempo. Por consiguiente, no existen dos personas con las mismas huellas, convirtiéndolas en rasgos perfectos de identificación. Los elementos principales que definen a las huellas dactilares, basados en los surcos y crestas que las forman, son conocidos como minucias. Estas minucias se encuentran normalmente localizadas en los finales o bifurcaciones de las crestas. Por todo ello, en la mayoría de los algoritmos de procesamiento de huellas dactilares, la comparación entre las mismas se realiza obteniendo la similaridad existente entre la totalidad de minucias que forman las huellas a contrastar.

La implementación de los algoritmos de comparación de huellas dactilares se suele realizar utilizando para ello lenguajes de programación de alto nivel como pueden ser C#, Java o Python [1]. Esta elección se realiza en base a la gran cantidad de herramientas de programación disponibles en estos lenguajes y sus entornos de trabajo, que facilitan la implementación de los algoritmos a los investigadores. Sin embargo, la mayor desventaja de estos lenguajes y otros similares reside en la baja velocidad de ejecución de los códigos, en contraste con los lenguajes de programación de bajo nivel. Por ejemplo, en C#, cuando un código escrito con este lenguaje es compilado, las líneas de código son inicialmente traducidas a un lenguaje intermedio denominado *Common Intermediate Language* (CIL) y después a código máquina, en tiempo de ejecución, gracias a la herramienta del entorno de trabajo de .NET *Common Language Runtime* (CLR). Esta es la razón principal por la que este tipo de lenguaje es también conocido como lenguaje manejado, provocando mayor carga de trabajo durante la ejecución.

Este artículo se centra en el estudio de uno de los algoritmos más precisos de comparación de huellas digitales basado en el procesamiento de las minucias, propuesto por Medina-Pérez et al. [2] y que será denotado como "DMC" a lo largo de este trabajo. Este método fue inicialmente implementado en C# y emplea los siguientes tres algoritmos:

1. El método *Minutia Cylinder-Code* [3] basado en estructuras tridimensionales de datos, denominadas *Binary Cylinder Codes*, formados a partir de las posiciones y ángulos que caracterizan a las minucias utilizando estructuras locales [4]. DMC emplea esta representación de las minucias en el algoritmo de comparación de huellas.
2. El método *Minutiae Discrimination* [5] calcula un valor de calidad para cada minucia basado en la consistencia en las direcciones de las minucias vecinas a cada una de ellas.
3. El método *Deformable Minutiae Clustering* [2], [6] empleado para evitar la pérdida de datos provocada por la deformación de la huella durante su captura. Los resultados de similaridad son obtenidos sumando los pesos de cada par de minucias coincidentes encontrados al mezclar las

[1]Dpto. de Arquitectura de Computadores, Univ. Málaga, e-mail: ajsanchez@ac.uma.es.
[2]Dpto. de Arquitectura de Computadores, Univ. Málaga, e-mail: felipe@uma.es.
[3]Dpto. de Ciencias de la Computación e Inteligencia Artificial, Univ. Granada, e-mail: siham@ugr.es.

agrupaciones de minucias similares. A su vez, el peso de cada par de minucias (q, p) se calcula evaluando las minucias incluidas en la agrupación coincidente en la que (q, p) está localizada en el centro. Los resultados estadísticos de la comparación de las huellas dactilares siguen las doctrinas de evaluación propuestas por Cappelli et al. [7].

Nosotros proponemos distintas técnicas de aceleración para el algoritmo de comparación de huellas dactilares DMC. En primer lugar, se presenta el análisis de rendimiento del algoritmo de comparación para determinar las funciones con el mayor coste computacional. Posteriormente, el algoritmo de comparación DMC se traduce a código no manejado implementando una serie de optimizaciones dentro de una librería de C++, que es conectada con el código C# mediante un proyecto de librería de clases del CLR. Con respecto a dichas mejoras, se ha optimizado la operación de cuenta de bits con valor uno que era realizada dentro del algoritmo *Minutia Cylinder-Code* gracias a la inclusión de una librería de C++. Por otro lado, el cálculo de la consistencia de la dirección de las minucias dentro del método *Minutiae Discrimination* se ha mejorado realizando modificaciones sobre las operaciones con el mayor coste computacional.

Este artículo está organizado de la siguiente forma. La sección II introduce la estructura del algoritmo de comparación DMC junto con los resultados obtenidos al realizar el análisis de rendimiento. La sección III describe las técnicas de optimización implementadas sobre el algoritmo de comparación DMC. La sección IV presenta los resultados experimentales. La sección V propone un nuevo planteamiento para mejorar el rendimiento de la memoria caché, junto con el procesamiento en paralelo del algoritmo de comparación. Finalmente, en la sección VI se analizan los resultados obtenidos y el enfoque de trabajo futuro.

II. Estructura y perfil de rendimiento del algoritmo DMC

El algoritmo de comparación de huellas dactilares conocido como DMC fue diseñado a partir de la implementación de tres métodos de comparación independientes cuyos procedimientos pueden ser resumidos en los siguientes apartados:

1. Se definen Q y T como las listas de minucias pertenecientes a las huellas dactilares consulta y plantilla, respectivamente, de una determinada base de datos. Cada minucia $\mathbf{q} \in Q$ es contrastada con todas las minucias $\mathbf{t} \in T$ en base a la similaridad existente entre sus respectivos descriptores. En nuestro caso de análisis, el descriptor utilizado para las minucias es un vector de 1280 bits que almacena la información referente a las posiciones y ángulos de todas las minucias vecinas que se encuentren dentro de una circunferencia de radio r, centrada en la minucia de estudio. Por consiguiente, dos minucias similares deben tener una gran cantidad de bits con la misma posición y valor dentro de sus respectivos vectores, de la forma que, si se realiza la operación XNOR entre estos descriptores como se muestra en la figura 3, el resultado obtenido será un nuevo vector con unos en las posiciones coincidentes. Por ello, la función denominada como *PopCount* es necesaria para realizar esta operación de cuenta de bits con valor uno, resultado que es utilizado para obtener el valor de similaridad entre minucias. Cada par de minucias coincidentes $(\mathbf{q}, \mathbf{t})$ es posteriormente incluido dentro de un nuevo vector de pares de minucias similares (A), y ordenado en orden decreciente en función del valor de similitud obtenido.
2. Para cada minucia $\mathbf{q} \in Q$ se calcula un valor de calidad en función de la consistencia existente entre las direcciones de las minucias que se encuentran dentro de su vecindario, y para alcanzar este objetivo, es necesario previamente realizar el cálculo de las tres minucias más cercanas a la minucia de estudio, procedimiento ejecutado por la función *UpdateNearest* dentro del algoritmo DMC. Asimismo, la misma operación se realiza para cada minucia $\mathbf{t} \in T$. Por último, se forman dos nuevos vectores que contienen todas las minucias de cada huella junto con sus correspondientes valores de calidad.
3. A partir de la lista ordenada A, se determinan agrupaciones de pares de minucias similares para cada par de minucias $(\mathbf{q}, \mathbf{t})$, junto con un valor de peso obtenido teniendo en cuenta los pares de minucias similares dentro de la agrupación similar en la que $(\mathbf{q}, \mathbf{t})$ está situada en el centro. Después, todas las agrupaciones dentro de ambas huellas son comparadas para encontrar pares de minucias globales. Posteriormente, se realiza una nueva búsqueda centrada en cada minucia $\mathbf{q} \in Q$ y $\mathbf{t} \in T$, utilizando el método *Thin Plate Spline* para encontrar nuevos pares

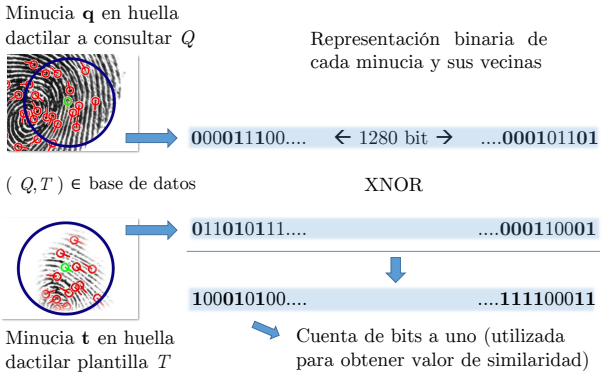

Fig. 1. Ejemplo de la operación de cuenta de bits con valor 1 empleada en el primer paso del algoritmo de comparación. Los bits a 1 resultantes de la aplicación de la operación XNOR sobre dos descriptores de minucias, son contados y utilizados para calcular el valor de similaridad entre minucias.

de minucias similares [8], que habrían podido ser descartados de los anteriores procesamientos debido a posibles deformaciones en la captura de las huellas.

4. El valor total de similaridad se obtiene como resultado de sumar los pesos de todos los pares de minucias coincidentes.

TABLA I

Análisis de rendimiento con estructura de árbol de llamadas del algoritmo DMC de comparación de huellas dactilares. Las funciones se encuentran listadas en orden descendente en base al número de muestras exclusivas de CPU obtenidas. BCC define a Binary Cylinder-Code y DMC-BCC.Match a la función principal donde se ejecuta el algoritmo de comparación.

Función	Inclusivas	Exclusivas
clr.dll	2096	2094
mscorlib.ni.dll	1685	1187
Map	1242	358
UpdateNearest	315	231
DMC-BCC.Match	6193	209
Compare	381	205
BCC.Match	309	189
MatchMinutiae	281	107
BCC.PopCount	95	95

Una vez que el algoritmo de comparación ha sido analizado, el siguiente objetivo se basa en determinar las funciones de mayor coste computacional dentro de los métodos mencionados. Por consiguiente, se ha llevado a cabo un análisis de rendimiento sobre el algoritmo DMC y para ello se ha empleado la herramienta *Performance Profiler* de Visual Studio 2017 que ofrece dos tipos de resultados: muestras inclusivas y exclusivas de CPU. Para nuestro estudio, se han tenido en cuenta únicamente las muestras exclusivas para identificar las funciones con la mayor carga computacional y el menor número de lineas de código. Esto es posible ya que este tipo de muestras se determinan teniendo en cuenta únicamente las obtenidas dentro de la función de estudio, sin contar las funciones que son llamadas desde ésta, por lo son usadas para determinar los cuellos de botella en los algoritmo de estudio. La tabla I recoge las funciones con el mayor coste computacional ordenadas en orden decreciente en base a las muestras exclusivas de CPU obtenidas para cada una. Las funciones resaltadas son aquellas que se encuentran implementadas dentro del algoritmo de comparación, mientras que el resto son herramientas utilizadas por el entorno de trabajo .NET. A partir de estos resultados, el estudio se centra en las funciones *UpdateNearest* y *PopCount*, debido a la gran cantidad de muestras exclusivas obtenidas en contraste con el pequeño número de líneas de código que contienen. En la sección siguiente, se presentarán las optimizaciones realizadas sobre las dos funciones mencionadas.

III. Nuevas optimizaciones

El algoritmo de comparación DMC fue implementado utilizando C# junto al entorno de trabajo de .NET y por lo tanto, el primer paso para la optimización reside en su traducción, incluyendo a las funciones previamente determinadas a partir del análisis de rendimiento, a un lenguaje de programación no manejado y de bajo nivel pero preservando la estructura original de la solución. Por ello, se seleccionó C++ para reemplazar a C# como el lenguaje de programación empleado, debido a su mejor rendimiento y al hecho de que es también un lenguaje de programación orientado a objetos (OOP, por sus siglas en inglés) lo que resulta en menos problemas de compatibilidad durante la traducción. Más tarde, se tenía que lograr la conexión entre el proyecto original en C# y una nueva librería implementada en C++, para poder compartir los datos y resultados entre ellos y para lograrlo, dos herramientas del entorno de trabajo de .NET fueron estudiadas: *Platform Invoke* utilizando métodos externos en "C" [9] y C++/CLI *Wrapper* [10]. Ambas estrategias aceleran las zonas de código que son traducidas a C++, sin embargo, el segundo método permite la creación de objetos, la utilización de *name-spaces* y el paso de valores, y además habilita el ensamblado de este tipo de proyectos no manejados junto con otros de tipo manejado.

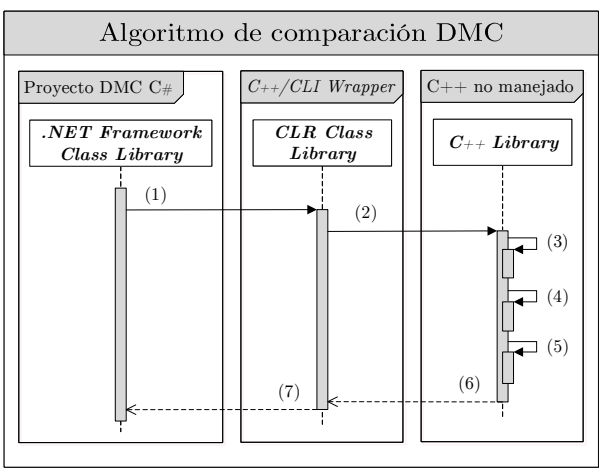

Fig. 2. Estructura del algoritmo de comparación de huellas dactilares DMC modificado que incluye una librería de clases de CLR junto a otra librería no manejada de C++ donde se ejecuta el algoritmo original.

La figura 2 muestra el algoritmo de comparación DMC modificado tras utilizar la estrategia C++/CLI *Wrapper*, donde han sido incluidos dos nuevos proyectos: una librería de clases de CLR y una librería no manejada de C++. La primera está escrita utilizando código C++/CLI que ofrece muchas herramientas del entorno de trabajo de .NET para trabajar con memoria manejada y no manejada dentro del mismo archivo. Gracias a ello, todos los datos pertenecientes a las huellas dactilares cargados dentro del proyecto de C# pueden ser transferidos y almacenados dentro de la librería de

clases de CLR con pequeñas transformaciones (1). Más tarde, estos datos pueden ser exportados de una forma sencilla a la librería de C++ llamando a las funciones definidas en este último archivo (2). Posteriormente, la librería de C++ se encarga de ejecutar el algoritmo de comparación DMC con las nuevas optimizaciones, que incluye los siguientes métodos: *Minutia Cylinder-Code Creation* (3), *Minutia Quality Computation* (4) y *Clustering Computation and Merging* (5). Por último, todos los resultados obtenidos son devueltos a la librería de clases de CLR (6) y después, al proyecto de C# para realizar la evaluación de los mismos (7).

Dentro del primer paso del algoritmo de comparación, se tiene que emplear la operación de contar bits a uno para obtener el valor de similaridad entre minucias de las huellas dactilares. Esta operación calcula el número de unos contenidos dentro de un determinado valor y se encuentra implementada dentro de la función denominada *PopCount*, utilizando el método SWAR. Esta técnica realiza la cuenta de 2 valores dentro de la palabra, para después agregar la cuenta de dúos a un cuarteto de estos y más tarde, bytes dentro de un registro de 64 bits para finalmente sumar todos los bytes juntos. A pesar de que este método logra un buen rendimiento, se buscaba implementar la mejor estrategia existente para la cuenta de unos, en función de las características del ordenador sobre el que se está ejecutando el algoritmo de comparación DMC. Por ello, se ha implementado la librería de C++ desarrollada por Wojciech Mula et al. [11] que escoge, en función de las características del procesador y del tamaño del vector de estudio, el algoritmo óptimo para calcular el número de bits a uno que contiene.

Por otro lado, centrando el estudio en el segundo paso del algoritmo de comparación, es necesario calcular las minucias vecinas para cada una de las minucias del total que forman la huella dactilar, utilizando para ello el valor de la distancia Euclídea. Los resultados de la raíz cuadrada realizada en esta operación pueden ser obtenidos de dos formas: mediante el acceso a una tabla preestablecida de valores que contiene los resultados de la raíz cuadrada para los primeros 1024 números naturales o, si el valor no se encuentra en dicha tabla, mediante la realización de la operación raíz cuadrada gracias a la librería de clases matemáticas de C#. Con la finalidad de reducir el tiempo de ejecución necesario para ambos métodos, se ha empleado en cambio el cuadrado de la distancia Euclídea para el cálculo de la vecindad de cada minucia.

IV. Resultados experimentales

Los experimentos fueron ejecutados empleando el procesador Intel i5-8600K con una frecuencia de ejecución de 3,60 GHz. Dicho procesador posee seis memorias caché L1 y L2 de 32 y 256 kB, respectivamente, junto a una única unidad de memoria caché L3 de 9 MB. Además, se emplea una memoria RAM de 8GB (DDR4 con doble canal). Los resultados de tiempo han sido obtenidos empleando la clase *Stopwatch* del entorno de trabajo .NET en C# y la librería *C++11 Chrono* en la librería de C++.

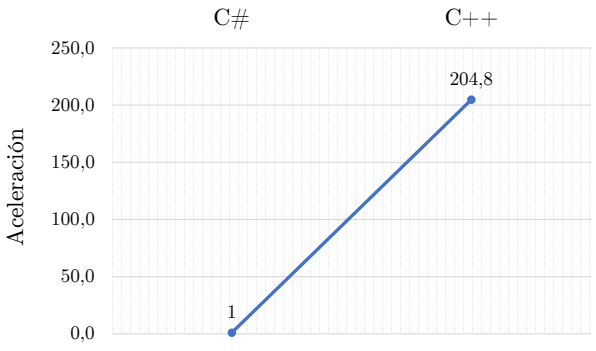

Fig. 3. Resultados de tiempo y aceleración por iteración de la operación de cuenta de bits a uno dentro de la función *PopCount*, donde se comparan los desempeños de los algoritmos de SWAR en C# y la librería rápida de C++.

Con respecto a la optimización llevada a cabo sobre la operación de cuenta de bits a uno, se ha empleado una librería rápida de C++ dentro de la función *PopCount* en detrimento del algoritmo SWAR original de C#. Los resultados obtenidos tras la ejecución de ambos métodos se muestran en la figura 3. El tiempo de ejecución del método de comparación DMC por iteración de la operación de cuenta de bits a uno, fue reducido de 90,176 microsegundos en C# hasta 0,440 microsegundos en C++ sobre un vector cuyo tamaño aproximado es de 160 bytes (20 valores de 8 bytes). Por lo tanto, los tiempos de ejecución de la función se han llegado a mejorar hasta en 204,8 veces tomando como base el tiempo obtenido en C#. Por otro lado, debido a que el tamaño del vector es menor de 512 bytes, la nueva librería emplea un algoritmo denominado como *unrolled pop count* para realizar la operación de cuenta de bits. Si el tamaño del vector fuera superior a 512 bytes se emplearía un algoritmo basado en instrucciones AVX2, no obstante, esta situación no se cumple en este caso, siendo estudiada como trabajo futuro con la finalidad de mejorar el desempeño del algoritmo sin incrementar el tiempo de ejecución del mismo..

En relación con la función *UpdateNearest*, se ha reemplazado el uso de la distancia Euclídea por el cuadrado de la misma (SED, por sus siglas en inglés), eliminando con ello la necesidad de realizar la operación raíz cuadrada y mejorando los resultados de tiempo, que son mostrados en la figura 4. Se ha logrado una aceleración para esta función de hasta 2,83 al emplear el cuadrado de la distancia Euclídea, en contraste con la aceleración de 2,33 conseguida utilizando la distancia Euclídea estándar, todo ello tomando como referencia el tiempo de ejecución previamente calculado en C#.

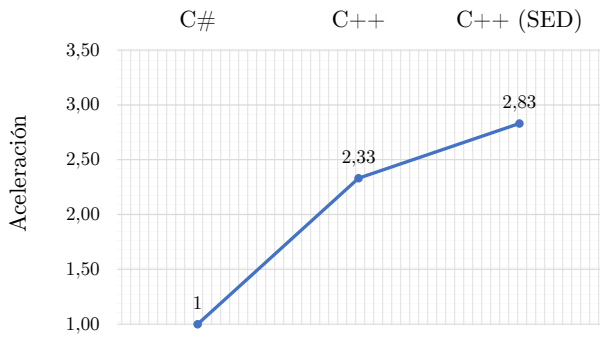

Fig. 4. Resultados de tiempo y aceleración por iteración de la función *UpdateNearest* para tres situaciones diferentes: el método original en C#, el método original traducido a C++ donde se emplea la distancia Euclídea estándar y el método optimizado que implementa el cuadrado de la distancia Euclídea (SED) en C++.

V. Análisis de rendimiento y propuestas de mejora

Los algoritmos de reconocimiento de huellas dactilares normalmente realizan la comparación entre una huella latente y todas las huellas almacenadas dentro de una determinada base de datos de huellas, obteniendo con ello un grupo de huellas similares que, posteriormente, serán analizadas con mayor detenimiento por un experto en este campo. Esta técnica permite reducir el número total de huellas a analizar, obteniendo un pequeño grupo de éstas cuya cantidad es determinada previamente por el experto. Sin embargo, el tiempo necesario para desempeñar esta tarea de comparación depende en gran medida del tamaño de la base de datos.

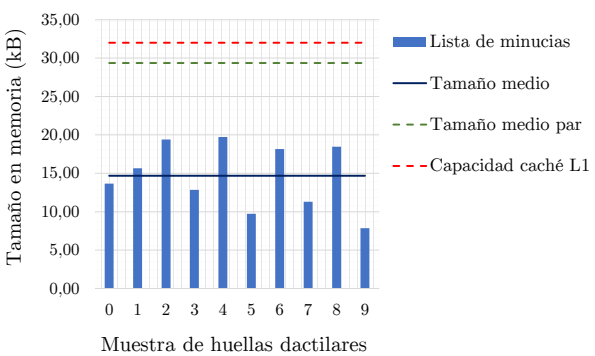

Fig. 5. Tamaños de las listas de minucias, en memoria manejada, para una muestra de 10 huellas dactilares de la base de datos FVC2004_DB1A. Se muestra el tamaño medio de una lista y de un par de listas de minucias calculadas para la muestra seleccionada, junto con la capacidad de la memoria caché de datos L1.

Por otro lado, cada muestra tomada de una huella dactilar, ya sea de tipo rodada, latente o impresa, no tiene por qué contener un número fijo de minucias, debido a la considerable complejidad que tiene la toma de muestras de huellas dactilares, que implica varios aspectos como las variaciones de presión, el área (muestra parcial o total), grandes distorsiones no lineales, entre otras variables [12]. Por lo tanto, si la huella a consultar y la de plantilla de la base de datos contienen una gran cantidad de minucias cada una, la cantidad total de datos con los que se trabaja durante la comparación de ambas huellas necesitará más espacio de memoria del disponible dentro de la memoria caché de datos L1, incrementando con ello los fallos de caché y aumentando por consiguiente el tiempo de ejecución [13]. Esta situación se presenta en la figura 5, donde se muestra que el tamaño medio de un par de huellas dactilares se encuentra muy cercano al máximo espacio disponible dentro de la memoria caché L1, estos resultados han sido obtenidos a partir de la ejecución y análisis del algoritmo de comparación DMC sobre la base de datos de huellas FVC2004_DB1A.

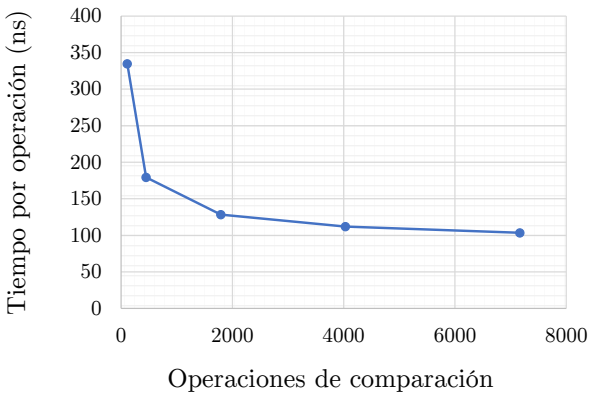

Fig. 6. Tiempo por operación de comparación entre minucias, en nanosegundos, con respecto al número total de operaciones a realizar. Las comparaciones entre huellas dactilares que posean una cantidad reducida de minucias necesitan un tiempo de procesamiento mayor que aquellas que tienen un gran número de minucias a comparar.

Por ello, se ha analizado el desempeño de los dos primeros pasos del algoritmo de comparación DMC al calcular el valor de similitud existente entre diversas huellas dactilares que contienen diferente número de minucias a comparar. Los resultados obtenidos, en términos de aceleración, se pueden observar en la figura 6. En ella se muestra el tiempo medio necesario para realizar una operación de comparación entre dos minucias en función del número total de comparaciones a realizar. El tiempo medio por operación de comparación decrece conforme aumenta el número de minucias a analizar y por ende, la cantidad de operaciones de comparación. Posiblemente este hecho se deba a las optimizaciones llevadas a cabo por el compilador en lo referente a los bucles de procesamiento del algoritmo.

Se propone la implementación de una estructura paralela para la ejecución del algoritmo de comparación DMC. La figura 7 muestra esta estructura, donde cada *core* de la CPU tendrá a su disposición una copia de la huella a consultar y realizará la comparación entre ésta y un bloque de huellas pertenecientes a la base de datos. Este bloque podrá estar formado por huellas completas o parciales, dependiendo del espacio total que ocupe la comparación en memoria, por ejemplo, F5-S1 y F5-S2 de

la figura anterior representan la división de la quinta huella dactilar de la base de datos en dos grupos de minucias para ser procesadas en dos *cores* diferentes. El tamaño de este bloque de procesamiento será calculado de forma que todos los datos referentes a las minucias puedan estar almacenados perfectamente, y sin desbordamientos, dentro de cada memoria caché L1 al realizar los dos primeros pasos del algoritmo de comparación DMC. Si se produce una coincidencia entre minucias, el valor de similaridad será almacenado para calcular posteriormente la coincidencia total entre huellas. Por ello, esta propuesta intenta solventar dos problemas muy comunes entre los algoritmos de comparación de huellas dactilares: la mala escalabilidad de estos programas para bases de datos de grandes dimensiones y el bajo rendimiento de la memoria caché durante los cálculos.

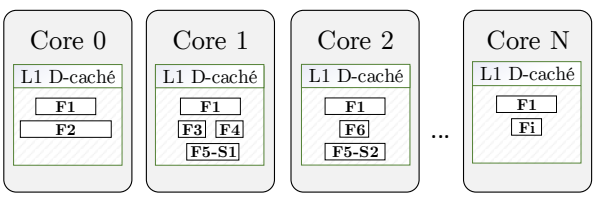

Fig. 7. Estructura del algoritmo paralelo de comparación DMC propuesto donde cada hilo de ejecución posee una copia de la huella a consultar ($F1$) para ser comparada con un bloque de huellas dactilares seleccionadas de la base de datos de forma que la totalidad de los datos puedan ser almacenados dentro de cada memoria caché L1.

Por último, se han realizado una serie de pruebas sobre las dos primeras etapas del algoritmo para obtener un resultado aproximado de la escalabilidad del código, y que se muestra en la figura 8. Para el procesamiento paralelo de las huellas se han empleado las directivas que ofrece OpenMP para C++ tomando como huella a consultar la primera de la base de datos FVC2004_DB1A frente a la totalidad de huellas que la componen (800 muestras). Como se puede observar en el gráfico, la primera parte del código escala correctamente empleando 2, 4 y 6 procesadores.

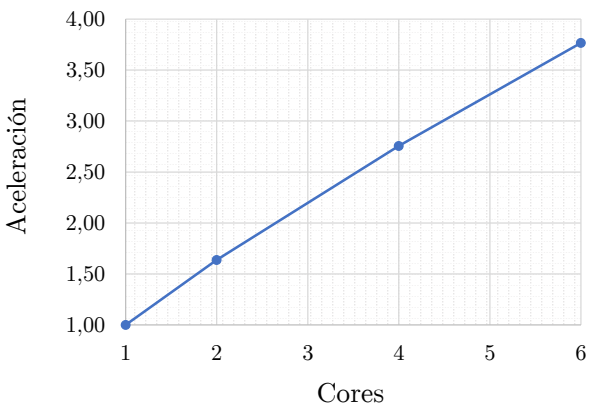

Fig. 8. Resultados de aceleración al paralelizar la comparación de huellas para las dos primeras etapas del algoritmo DMC, empleando la base de datos FVC2004_DB1A.

VI. Conclusiones

Se han presentado dos estrategias para mejorar el tiempo de ejecución del algoritmo de comparación DMC propuesto por Medina-Pérez et al. [2]. Para ello, se ha probado que el primer paso para la reducción del tiempo de procesamiento del algoritmo DMC reside en la traducción desde C# a C++ de las funciones con mayor coste computacional, debido a la mayor carga de trabajo que induce el código C# por ser un lenguaje manejado. Para alcanzar este objetivo, se ha implementado una librería de clases de CLR dentro del código original del DMC para trabajar como conector entre el proyecto original de C# y el nuevo archivo de C++ no manejado, permitiendo con ello la transferencia de los datos de las huellas dactilares en ambas direcciones. El código C++ no manejado se incluye dentro de una librería estática de C++ (.lib) que contiene el algoritmo optimizado de comparación de huellas DMC reescrito en dicho lenguaje de bajo nivel. Con respecto a las optimizaciones realizadas sobre el algoritmo, se ha implementado una librería rápida de C++ dentro de la función *PopCount* reduciendo significativamente el tiempo a emplear para realizar la operación de cuenta de valores a uno hasta 204,8 veces, en contraste con el método SWAR implementado en C#. En relación con la función *UpdateNearest*, se han probado dos alternativas, dentro de la nueva librería de C++, de optimización para el cálculo de la vecindad de cada minucia: la traducción del mismo algoritmo implantado en C# a C++ que emplea el uso de la distancia Euclídea, y esta traducción empleando el cuadrado de la misma (SED). Estas estrategias han mejorado 2,33 y 2,83 veces, respectivamente, el tiempo de ejecución obtenido para el código de C#. Como último aspecto, se ha propuesto un nuevo enfoque para mejorar el rendimiento del algoritmo de comparación DMC basado en una estructura paralela de procesamiento donde cada hilo de ejecución realizará la comparación entre la copia de la huella a consultar que tiene almacenada y los bloques de huellas completas o parciales procedentes de la base de datos, que podrán ser almacenadas en su totalidad dentro del espacio disponible en la memoria caché L1 durante la realización de los dos primeros pasos del algoritmo. Además, se ha mostrado que, centrando el estudio en el algoritmo DMC, el tamaño de los datos necesarios para realizar la comparación entre dos huellas dactilares es, en algunos casos, mayor que el espacio disponible dentro de la memoria cache L1. Esta situación reduce previsiblemente el rendimiento de los cálculos debido a los fallos de caché y la mala escalabilidad del algoritmo para bases de datos que contienen gran cantidad de huellas. No obstante, estos problemas podrían ser atenuados utilizando la nueva propuesta que emplea la estructura paralela de procesamiento. Por último, se han mostrado los resultados relativos a la aceleración obtenida para las dos primeras etapas del algoritmo DMC, empleando para ello la estructura paralela propuesta.

Agradecimientos

El presente trabajo ha sido financiado por el Proyecto TIN2016-80920-R perteneciente al Plan Nacional de Investigación destinado al estudio de arquitecturas de altas prestaciones para aplicaciones intensivas en datos, y por la Universidad de Málaga (Campus de Excelencia Internacional Andalucía Tech).

Referencias

[1] M.A. Medina-Perez, et al., *Introducing an experimental framework in C# for fingerprint recognition, MCPR 2014*, pp. 132–141, Springer, Cham, 2014.

[2] M.A. Medina-Perez, et al., *Latent fingerprint identification using deformable minutiae clustering, Neurocomputing*, vol. 175, pp. 851–865, 2016.

[3] R. Cappelli, M. Ferrara, D. Maltoni, *Minutia cylinder-code: A new representation and matching technique for fingerprint recognition, IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 12, pp. 2128–2141, 2010.

[4] X. Jiang, Y. Wei-Yun, *Fingerprint minutiae matching based on the local and global structures, Pattern Recognition*, vol. 2000, no. 2, pp. 1038–1041, 2000.

[5] K. Cao, et al., *Fingerprint matching by incorporating minutiae discriminability, IJCB 2011*, pp. 1–6, 2011.

[6] M.A. Medina-Perez, et al., *Improving fingerprint verification using minutiae triplets, Sensors*, vol. 12, no. 3, pp. 3418–3437, 2012.

[7] R. Cappelli, et al., *Performance evaluation of fingerprint verification systems, IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 1, pp. 3–18, 2006.

[8] A.M. Bazen, H.G. Sabih, *Fingerprint matching by thin-plate spline modelling of elastic deformations, Pattern Recognition*, vol. 36, no. 8, pp. 1859–1867, 2003.

[9] Using C++ Interop (Implicit PInvoke), *https://msdn.microsoft.com/en-gb/library/2x8kf7zx.aspx*, Último acceso: 16 Febrero 2018.

[10] .NET Programming with C++/CLI (Visual C++), *https://msdn.microsoft.com/en-gb/library/68td296t*, Último acceso: 22 Mayo 2018.

[11] W. Mua, N. Kurz, D. Lemire, *Faster population counts using AVX2 instructions, The Computer Journal*, vol. 61, no. 1, pp. 111–120, 2017.

[12] A.K. Jain, J. Feng, *Latent fingerprint matching, IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 88–100, 2011.

[13] M. Kowarschik, C. Weiss, *An overview of cache optimization techniques and cache-aware numerical algorithms, Algorithms for Memory Hierarchies*, pp. 213–232, Springer, Berlin, 2011.