Análisis del Proceso de Aceleración vía Hardware en *LIBSVM*: Máquinas de Soporte Vectorial Optimizadas

Lucas Leiva
Facultad de Ingeniería
Universidad FASTA
Facultad de Ciencias Exactas
UNICEN
Pinto 399, Tandil, 7000
lleiva@exa.unicen.edu.ar

Jordina Torrents-Barrena
Departamento de Ingeniería
Informática y Matemáticas,
Universidad Rovira i Virgili
Av. Paisos Catalans 26,
Tarragona, Spain
jordina.torrents@urv.cat

Martín Vázquez
Facultad de Ingeniería
Universidad FASTA
Facultad de Ciencias Exactas
UNICEN
Pinto 399, Tandil, 7000
mvazquez@exa.unicen.edu.ar

Domènec Puig
Departamento de Ingeniería
Informática y Matemáticas
Universidad Rovira i Virgili
Av. Paisos Catalans 26,
Tarragona, Spain
domenec.puig@urv.cat

Elías Todorovich
Facultad de Ingeniería
Universidad FASTA
Facultad de Ciencias Exactas
UNICEN
Pinto 399, Tandil, 7000
etodorov@exa.unicen.edu.ar

Resumen

Las Máquinas de Soporte Vectorial (SVM) son una herramienta de aprendizaje común utilizada ampliamente gracias a su alta precisión en tareas de clasificación. El uso de las SVM en tiempo real a través de sistemas embebidos resulta desafiante debido a la complejidad de los cálculos que se requieren. Por tanto, se hace presente la necesidad de migrar las SVM a plataformas hardware con el fin de alcanzar un alto rendimiento con costes y consumos de energía bajos. El presente artículo proporciona un estudio exhaustivo del proceso de aceleración de la librería LIBSVM mediante hardware. Dicha librería será utilizada durante la clasificación binaria de características, tomando como caso de estudio la identificación de masas mamarias. Se presenta además una optimización que ofrece una aceleración del 57% respecto de algoritmo original. Por último, se sugieren algunas direcciones claves que optimizarán el proceso de aceleración en futuras investigaciones.

1. Introducción

Actualmente, la metodología aplicada por las Máquinas de Soporte Vectorial (SVM) es ampliamente utilizada por la comunidad científica para el entrenamiento supervisado de clasificadores. Su popularidad se atribuye a los buenos resultados obtenidos en una gran variedad de problemas tales como la detección de objetos [1], el reconocimiento de voz [2], la clasificación de imágenes, y el diagnóstico médico [3], entre otros. El estándar SVM se define como un clasificador supervisado no-probabilístico, binario y lineal que añade capacidades de clasificación no lineales mediante el truco del *kernel*. Éste, transforma dicho problema a un espacio dimensional de características superior de modo que sus clases devienen linealmente separables [4]. Las implementaciones actuales que referencian a las SVM (por ejemplo, *LIBSVM* [5], *SVMTorch* [6], *SVM*^{Light} [7]) ofrecen una gran variedad de propiedades que permiten hacer uso de distintos *kernels*, de la clasificación multiclase y la validación cruzada.

Los métodos de aprendizaje supervisado normalmente constan de las siguientes fases: el entrenamiento o aprendizaje, y la predicción o clasificación. La primera etapa construye el modelo SVM optimizado, cuya base serán los vectores de soporte (SV) identificados a partir de los datos de entrenamiento [8]. Posteriormente, los SV son utilizados en la segunda fase para predecir la clase de los datos de entrada.

Con la eclosión de nuevos objetivos exigentes y desafiantes que requieren de una mayor precisión para ser resueltos, los problemas de clasificación deben abordar una elevada complejidad computacional. La dificultad del problema aumenta con los datos de entrenamiento disponibles y, por tanto, con el tamaño de los vectores de entrada. Por otra parte, las técnicas como el ajuste de parámetros y la validación cruzada, permiten mejorar la clasificación, incrementando los costes computacionales. En algunos casos, el tiempo de procesamiento para la formación de un modelo SVM puede ser del orden de días [9]. Por lo tanto, es crucial acelerar el proceso de entrenamiento de dicho modelo sin sacrificar la exactitud en los resultados.

Existe un creciente interés por explotar las SVM en muchos sistemas de detección embebidos, así como en aplicaciones de procesamiento de imágenes. El modelo resultante de la fase de entrenamiento es computacionalmente caro y consume mucho tiempo, especialmente para los problemas a gran escala, lo que hace vital su aceleración. Dichos problemas requieren arquitecturas de hardware específicas y dedicadas a cumplir con restricciones como la utilización de recursos limitados y un bajo consumo de energía [5]. Este hecho ha motivado gran cantidad de investigadores a estudiar tanto la aceleración de las SVM en hardware como el uso de plataformas de computación paralela.

El hardware específico y reconfigurable es prometedor para acelerar los cálculos, y proporciona computación de alto rendimiento (HPC) a bajo coste y consumo de energía [10]. *Field-Programmable Gate Arrays* (FPGA) son dispositivos reconfigurables de procesamiento potentes y altamente paralelos que se utilizan para la consecución de HPC en sistemas embebidos con la utilización eficiente de los recursos de hardware. Recientemente, las FPGA han mostrado mejoras significativas de rendimiento superando los *General-Purpose-Processors* (GPP) en una creciente gama de áreas [11-13].

Las unidades de procesamiento gráfico (GPU) también ofrecen una plataforma alternativa para la computación de alto rendimiento [14]. La disponibilidad de librerías de código abierto como OpenCV ayudan a alcanzar un tiempo de desarrollo mucho más rápido para las GPU que para FPGA. Sin embargo, para los algoritmos más complejos que utilizan matrices compartidas y muchos accesos a memoria, las GPU reducen su rendimiento debido a las limitaciones de acceso a memoria. A pesar de que las GPU disminuyen el coste y el tiempo de desarrollo en comparación con las FPGA, éstas son inferiores en términos de consumo energético [15]. Por otra parte, las implementaciones existentes para GPU son difíciles de mapear debido al hardware fijo y las limitaciones en cuanto a los recursos disponibles (menos memoria, registros, caché y núcleos). En consecuencia, dichas implementaciones son complejas de programar en entornos integrados, motivando un movimiento hacia las implementaciones en FPGA [16].

Los progresos recientes en FPGA hacen que sea posible la inclusión de núcleos de procesador en un solo chip, lo que garantiza una mayor flexibilidad en el diseño de sistemas integrados y multiprocesador de alto rendimiento [17]. Además, las modernas herramientas de desarrollo publicadas recientemente permiten diseñar de forma simple sistemas integrados mediante lenguajes de alto nivel. En consecuencia, las FPGA se han convertido en una opción atractiva para la aceleración sustancial de los cálculos intensivos producidos en las SVM.

El trabajo presentado contribuye a los usuarios de la librería LIBSVM, respecto a la necesidad de aceleración. En este trabajo se realiza un análisis exhaustivo de la librería para la clasificación, siguiendo una metodología aceleración de algoritmos en hardware. Además demuestra la factibilidad de migración a hardware de las rutinas con mayor latencia.

El presente artículo está organizado como sigue. La sección 2 presenta el estado del arte en aceleraciones hardware de máquinas de soporte vectorial, haciendo hincapié en soluciones de entrenamiento y clasificación. En la sección 3 se describe el estudio realizado utilizando herramientas de profiling y de análisis de memoria con el fin de acelerar las rutinas de clasificación de la librería LIBSVM. El caso de estudio aplicado se basará en la detección de masas mediante mamografías de rayos X. A su vez, se detalla una optimización del código C presente en la librería, que permite aumentar el rendimiento. Por último, se incluye una primera solución de la migración hardware a partir de una herramienta de síntesis de alto nivel. Finalmente, la sección 4 detalla las conclusiones extraídas y los trabajos futuros pendientes.

2. Estado del arte

En esta sección se presenta una revisión de la literatura existente. Concretamente, se han citado las implementaciones hardware en FPGA más significativas para las SVM. Los artículos presentados se han dividido básicamente en dos grupos: aceleración en la fase de entrenamiento o clasificación. Por tanto, cada grupo muestra las técnicas hardware usadas para mejorar y optimizar la fase seleccionada.

2.1. Implementaciones SVM: Entrenamiento

Una gran gamma de metodologías para el entrenamiento de SVM ha sido presentadas en la literatura, basándose en el método común de descomposición optimización secuencial mínima (SMO) [18]. T. Kuan et al. [19] propusieron un diseño de

circuitería totalmente funcional basado en dicho algoritmo para acelerar la fase de aprendizaje. La arquitectura se compone de tres módulos con circuitos principales que funcionan durante la ejecución del SMO mediante un bloque de memoria y uno de caché. El controlador fue diseñado a partir de una máquina de estados finitos.

Por otra parte, K. Cao et al. [20] mostraron una arquitectura digital escalable y paralela para la formación de una SVM basada en el algoritmo SMO. El objetivo era superar la falta de flexibilidad presentada en las aplicaciones embebidas anteriores. Una versión modificada del algoritmo SMO tradicional [21] se adoptó en el sistema digital, donde múltiples unidades de procesamiento trabajaban en paralelo. El tamaño de la memoria y el número de unidades de procesamiento eran ajustables, para poder lograr una arquitectura escalable manejando diferentes tamaños dependiendo del problema.

J. Filho et al. [22] diseñaron una arquitectura de propósito general y dinámicamente reconfigurable con el fin de soportar diferentes tamaños de conjuntos de entrenamiento. También basada en el método SMO, la arquitectura permitió los intercambios modulares mediante la reconfiguración. El *kernel* propuesto en [23] fue utilizado en el sistema empleando el método *Coordinate Rotation Digital Computer* (CORDIC) [24]. Éste, se basa en operaciones de desplazamiento y adición que fueron usadas durante la implementación del *kernel* mencionado. La metodología creada alcanzaba el 22,38% de ahorro en área aceptando el tiempo de penalización necesario para la reconfiguración.

Posteriormente, se propuso otro sistema de co-diseño hardware / software a través del algoritmo SMO. Su uso se enfocó a sistemas embebidos para la identificación del habla [25]. El diseño modular creado en [19] fue explotado y mejorado para solucionar el cuello de botella generado durante el entrenamiento del algoritmo SMO en hardware. Otras etapas, incluyendo el pre-procesamiento, la extracción de características y el análisis por votación fueron programados en un procesador ARM (código C embebido). Además, un nuevo mecanismo de datos fue creado para mejorar la eficiencia en la comunicación y transmisión de estos entre el software y el hardware (aproximadamente un 5% de reducción en el tiempo de entrega). En comparación con el código C en ARM embebido, el sistema propuesto disminuye el tiempo de entrenamiento hasta un 90% con una tasa de identificación del 89'9%.

Además, se presentó otro sistema de co-diseño hardware / software para la aceleración de la fase de aprendizaje basado en un método de descomposición distinto [26]. El algoritmo *Hybrid Working Set* (HWS) fue creado a partir de la metodología de descomposición extendida en [27], aprovechando los valores del *kernel* en caché y la naturaleza rápida de convergencia con el fin de disminuir el número de iteraciones. A su vez, se propuso una arquitectura coprocesador completamente escalable que consistía en una red de núcleos para conseguir el paralelismo necesario en los cálculos del *kernel*. Como consecuencia, se logró una aceleración del 25x a partir del coprocesador implementado (32 núcleos) con una sola CPU Core i5. También se minimizó el número de iteraciones en un 50% y 60% respecto a los programas de software LIBSVM y SVM^{light} gracias a la implementación del algoritmo de conjunto de trabajo híbrido (HWS). En conclusión, este nuevo diseño del coprocesador y el algoritmo HWS consiguió un aumento de velocidad del 15x y 23x en comparación con las librerías mencionadas anteriormente.

Finalmente, S. Wang et al. [28] explotaron la tecnología *Run Time Reconfiguration* (RTR) de las FPGA para potenciar el entrenamiento online de las LS-SVM. El diseño propuesto fue dividido en dos partes intercambiables entre sí utilizando el RTR y el diseño modular, alcanzando una alta paralelización. La primera fase consiste en formular la matriz del *kernel* aplicando un método de interpolación lineal por partes. La segunda fase resuelve el problema de los mínimos cuadrados a través de una modificación de la descomposición de Cholesky. Ésta, mejora los altos requisitos de memoria y el tiempo de latencia causado por las operaciones de raíces cuadradas. Los resultados experimentales ilustran una velocidad de 6 a 218x en comparación con la implementación Xeon CPU. En cuanto al análisis del coste temporal, la arquitectura propuesta demostró ser adecuada para problemas a gran escala con más de 1000 muestras, debido al alto tiempo de reconfiguración.

2.2. Implementaciones SVM: Clasificación

En cuanto al grupo de clasificación, varias técnicas han sido desarrolladas para acelerar el proceso de clasificación *online* en placas FPGA. El modelo SVM fue entrenado *offline* extrayendo datos para ser utilizados mediante hardware durante la clasificación *online*.

R. Patil et al. [29] utilizaron la arquitectura de matriz sistólica para implementar un clasificador SVM multiclase en Xilinx Virtex-6 FPGA con el fin de reconocer expresiones faciales. Además, la técnica de reconfiguración parcial basada en diferencias fue usada para la optimización de energía en el diseño FPGA. Una reducción de energía del 3-5% fue conseguida después de la reconfiguración mediante el uso de herramientas de diseño digital de Xilinx.

La técnica *Dynamic Partially Reconfigurable* (DPR) fue explotada por H. Hussain et al. [30] con el objetivo de crear una SVM para la clasificación de *microarrays* de datos en aplicaciones bioinformáticas. Dicho clasificador hacia uso de una arquitectura en matriz sistólica (4

bloques principales), pero mediante una placa FPGA antigua. Finalmente, se logró una aceleración del 85x a través de una aplicación de software equivalente a las GPP. Para poder modificar el núcleo de la SVM con diferentes parámetros, se aplicó DPR que era 8x más rápida que la reconfiguración de todo el dispositivo FPGA.

M. Ruiz-Llata et al. [31] desarrollaron un diseño hardware-FPGA para la clasificación y regresión mediante SVM. El sistema propuesto adoptó la función *hardware friendly kernel* presentada en [23], cuyo objetivo era simplificar significativamente el diseño en la fase de clasificación, evitando el uso de multiplicaciones computacionalmente intensivas y proporcionando un buen rendimiento en comparación con el *kernel* gaussiano tradicional. La arquitectura también hace uso del algoritmo iterativo CORDIC [24]. El sistema de clasificación SVM implementado utiliza un 75% de la lógica FPGA (Cyclone II) y una memoria externa para almacenar los vectores de soporte que conducen a 2ms de limitación en la velocidad de clasificación, con una tasa de error del 4%.

Por otra parte, Y. Ago et al. [32] presentaron un nuevo enfoque para la utilización eficaz de los *Digital Signal Processor* (DSP) en cascada y los bloques RAM embebidos en FPGA. El propósito era diseñar una arquitectura DSP completa para acelerar la clasificación con SVM. El núcleo del procesador hacia uso de 768 DSP y 800 bloques de RAM implementados en Xilinx Virtex-6 FPGA. Finalmente, el clasificador SVM se componía de 760 vectores de soporte e implementaba tres tipos de funciones del *kernel*: sigmoidea, polinómica y radial (RBF). Los resultados experimentales mostraron un alto rendimiento de 2.89×10^6 veces por segundo en la predicción de un espacio de características de 128 dimensiones funcionando a 370.096 MHz.

Una de las herramientas más comunes es el *Xilinx System Generator* que ofrece un modelado de sistemas de alto rendimiento y la generación automática de código Matlab / Simulink. D. Mahmoodi et al. [33] utilizaron dicho generador para diseñar e implementar una arquitectura hardware simple compuesta por un clasificador SVM de 3 clases binarias. La fase de entrenamiento se llevó a cabo en Matlab usando el modelo extraído mediante la librería *LIBSVM*. A continuación, la fase de test se ejecutó a través de la combinación en serie y en paralelo de bloques y funciones del generador del sistema, alcanzando una arquitectura del clasificador SVM paralela y simultánea. Además, el *CORDIC block* del sistema generador fue explotado para la implementación de la función exponencial. Los resultados de la simulación en FPGA demostraron una frecuencia máxima de 202.840 MHz para la clasificación lineal y un 98.67% de precisión para la no lineal, con un descenso considerable del tiempo de cálculo en comparación con la implementación en Matlab.

Finalmente, C. Kyrkou et al. [34] extendieron su trabajo previo de aceleración en cascada [35] proponiendo una arquitectura híbrida y optimizada con el método de reducción de hardware y evaluación de respuesta. Dicho proceso de evaluación hacía uso del modelo generado por una red neuronal (NN) para la clasificación en cascada de respuestas obtenidas en etapas anteriores. El objetivo era eliminar muestras antes de la etapa final de clasificación, mejorando su velocidad. Además, la arquitectura empleaba el descriptor *local binary pattern* (LBP) con el fin de extraer características antes de la etapa final, mejorando la precisión de detección. La arquitectura presentada fue implementada en Spartan-6 FPGA y orientada a la detección de rostros usando imágenes de 800 x 600 en alta resolución. El diseño híbrido creado logró un procesamiento en tiempo real de 40 fps con el 80% de precisión, así como un 25% y 20% de reducción en el área y potencia de pico, respectivamente. La precisión total de clasificación sólo se redujo en un 1%.

3. Análisis de la librería *LIBSVM*

La aceleración hardware propuesta para la librería *LIBSVM* se realiza siguiendo la metodología presentada en [36]. Esta metodología establece como primera instancia la migración, optimización y la partición HW/SW, en donde se caracterizan los recursos necesarios para la ejecución del software contemplando el consumo de memoria, los periféricos e interfaces involucrados y la estimación del consumo del procesador. Realizar este tipo de análisis resulta útil para determinar el segmento de programa con mayor consumo de tiempo o memoria, y a partir de los resultados obtenidos tomar las acciones pertinentes para obtener una mejora efectiva.

La librería puede ser utilizada en diferentes áreas de aplicación, pero se toma como caso de estudio un sistema de identificación de tumores mamarios a través de la clasificación de texturas basadas en píxeles [37]. Los resultados obtenidos pueden ser generalizados en otros casos de estudio.

Si bien el sistema comprende dos etapas: entrenamiento y predicción, se considera la premisa que el sistema se encuentra en un estado operativo, con un modelo previamente entrenado y verificado. Por este motivo, se descarta el análisis de tiempos relacionados con la fase de entrenamiento.

En la Figura 1, se presenta el pseudo-código referente a la etapa de predicción. Éste, se corresponde con el ejecutado e implementado en Matlab, resaltando las funciones de interés para el posterior análisis.

```
Función principal (run.m)
 1. Configuración variables
 2. Configuración parámetros filtro de Gabor
 3. Creación directorios temporales
 4. Extracción de características
    (fcngmsdfeat.m)
 5. Generación Caso de test
 6. Clasificación de características +
    validación experimental
    (classificationSVM.m)
       a. Carga del modelo
       b. Preparar modelo de test
       c. Uso del modelo para la
          clasificación (svmpredict.c)
       d. Almacenamiento resultados
```

Figura 1. Pseudo-código del algoritmo en Matlab.

Dentro de los parámetros de configuración se establecen las rutas a los archivos donde se encuentran las imágenes a analizar, así como también del modelo entrenado. La configuración de los parámetros del filtro de Gabor, permite escoger el tipo de frecuencia (baja o alta), escalas, orientaciones, tamaño de ventana (estrechamente relacionado con el radio promedio de los tumores), el número de características (mediana, desviación estándar, sesgo y curtosis), y el kernel a utilizar (no lineal, lineal, polinomial, sigmoidea). Luego se realiza la creación de los directorios donde se almacenarán los archivos temporales, se extraen las características para la imagen que se desea clasificar mediante la función *fcngmsdfeat.m,* y se generan los archivos necesarios para el caso de test. La ejecución continúa con la llamada a la función *classificationSVM.m* en el paso 6, la cual realiza la carga y preparación del modelo, la clasificación mediante la llamada a la función *svmpredict* y el almacenamiento de los resultados.

Respecto a la función *svmpredict*, ésta es una rutina MEX, correspondiente a la librería *LIBSVM*. Está escrita en lenguaje C, y previamente compilada para un mejor desempeño temporal.

3.1. Análisis de tiempos de ejecución en Matlab

El análisis se llevó a cabo utilizando la herramienta de *profiling* contenida en Matlab. El *profiling* de código fue realizado sobre la evaluación de una mamografía de la Base de Datos Mini-MIAS [38] considerando un modelo SVM entrenado previamente. Este banco de imágenes contiene un conjunto de mamografías en formato *png* con una resolución de 1024 x 1024 píxeles. Las características utilizadas fueron: mediana y desviación estándar. Para esto, se almacenó un modelo entrenado con 36 imágenes y durante la ejecución del programa para la prueba el modelo fue recuperado desde archivo. Esta acción evita el cálculo de los vectores característicos involucrados en el conjunto de entrenamiento, así como también el entrenamiento mismo.

La imagen utilizada para las pruebas se corresponde con la imagen "imb001" de la Base de Datos Mini-MIAS.

Las pruebas se realizaron sobre los 4 tipos de kernels disponibles en la librería *LIBSVM*: no lineal (radial), lineal, polinómico y sigmoideo. El programa fue ejecutado sobre una PC con un procesador Intel I5 de cuádruple núcleo, con 4Gb de memoria RAM. La versión de Matlab utilizada fue 7.12.0 (R2011a) y el sistema operativo Windows 7.

Los parámetros utilizados durante la generación de los modelos se describen a continuación:

- **No Lineal:** C = 0.0312; gamma = 64
- **Lineal:** C = 0.5
- **Polinómico:** C = 1; coef0 = 0; grado = 1
- **Sigmoideo**: C = 0.0312; gamma = 64; coef0 = 0

La Tabla 1 presenta los resultados obtenidos en cada una de las pruebas, y se encuentran expresados en segundos, representando el tiempo consumido durante la ejecución para cada una de las funciones. Las funciones presentadas se corresponden únicamente con las 4 funciones con mayor retardo.

Tabla 1. Análisis de las funciones con mayor consumo de tiempo.

	No lineal	Lineal	Polinómico	Sigmoideo
svmpredict (archivo MEX)	3345.1	1925.4	2616.5	1855.1
classificationSVM	16.1	4.8	9.3	3.7
run	4.6	5.6	8.6	4.7
fcngmsdfeat	1.4	1.7	2.4	1.5
Tiempo total de ejecución	3368.7	1939.6	2639.3	1868.2

Si bien, los resultados presentados contienen el *overhead* introducido por el *profiling*, en todos los casos se observa que el mayor retardo de ejecución se encuentra asociado a la rutina de predicción de la librería SVM (*svmpredict*), indicando que el tiempo asociado al resto de la ejecución en código Matlab es de aproximadamente 20 segundos. Por ello, el tiempo consumido por el desarrollo en Matlab puede ser considerado irrelevante en contraste con el introducido por la función mencionada.

Con el motivo de obtener resultados más precisos y descartar retrasos inducidos por el sistema operativo durante las ejecuciones anteriormente realizadas, se realizó el *profiling* de la ejecución de 10 predicciones para cada uno de los modelos, analizando el tiempo consumido por la función MEX (*svmpredict*) con respecto al tiempo total. En la Tabla 2 se presentan estos tiempos, expresados en segundos, junto con los respectivos promedios, y el porcentaje del tiempo total de ejecución consumido por la rutina en conflicto.

Tabla 2. Porcentajes de tiempo consumido por la función MEX respecto del total.

	Tiempo total	Tiempo función MEX	Porcentaje de ocupación
No Lineal	2742.6	2726.0	99.4%
Lineal	1671.7	1659.4	99.2%
Polinómico	1836.2	1821.2	99.1%
Sigmoideo	1913.3	1891.2	98.9%

De los resultados obtenidos, se descarta la hipótesis de presencia de casos aislados inducidos por el sistema operativo durante la ejecución. Además, se contempla un alto consumo (superior al 98%) independientemente del kernel escogido. Se puede establecer mediante la ley de *Amdahl* que la aceleración de esta rutina puede resultar auspiciosa, siendo que la aceleración (*Acc*) para el peor de los casos, estará definida por:

$$Acc = \frac{1}{0.011 + \frac{0.989}{A_{local}}}$$

Esto permite inferir que la aceleración teórica máxima del sistema, a través de la aceleración de la función *svmpredict*, es de 90X.

Por otra parte, el origen de esta latencia puede estar directamente asociado a un alto consumo de memoria o de procesador. De esta forma, se realizó un el análisis de memoria utilizando la herramienta de *profiling*. Los resultados se detallan en la Tabla 3.

Tabla 3. Análisis de funciones con mayor consumo de memoria.

	Pico de memoria	Memoria asignada
run	210844.00 Kb	934576.00 Kb
classificationSVM	210844.00 Kb	411476.00 Kb
fcngmsdfeat	194692.00 Kb	501684.00 Kb
labelpixel_miniMIAS	32452.00 Kb	77640.00 Kb
filter2	8352.00 Kb	197280.00 kb
svmpredict (archivo MEX)	5768.00 Kb	5768.00 Kb

Los datos, tanto los picos de memoria, como la memoria asignada para cada una de las funciones, descartan que la latencia esté asociada con un alto consumo de memoria por parte de la función *svmpredict*.

Para asegurar esta hipótesis, se analizó el número de fallos de página durante la ejecución del programa mediante el monitor de rendimiento de Windows. Se observa en la Figura 2 que los errores de página asociados a Matlab (amarillo), se mantienen bajos durante la ejecución del programa, con respecto a los errores de páginas totales (verde). Esto permite afirmar que el cuello de botella de la función *svmpredict* no se encuentra en los datos, sino en el procesamiento.

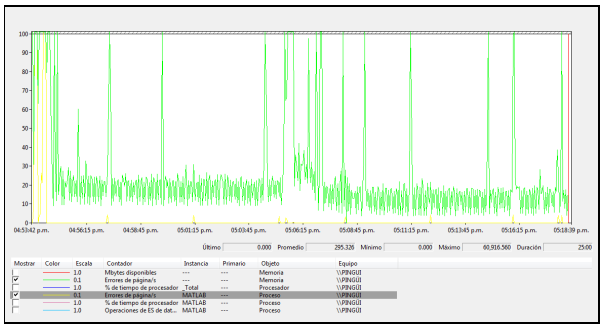

Figura 2. Análisis de consumo de memoria de la aplicación.

3.2. Ejecución en *LIBSVM*

La ejecución de la función *svmpredict*, como se mencionó anteriormente, se corresponde con una función MEX, descripta en C que contiene declaraciones propias de Matlab. Esto hace que el código no sea portable a otros compiladores GNU.

En la Figura 3 se presenta un pseudo-código del camino crítico de la ejecución de la rutina, detallando las funciones involucradas. En este caso, se presta especial atención a los lazos, ya que estos son los que generalmente definen la latencia de ejecución de un programa.

```
void mexFunction(…)
→ svm_predict (…)
    por cada vector a predecir
    // testing_instance_number iteraciones
    {
        // cálculo de kvalues
        → svm_predict_values(…)
            por cada support vector del modelo
            // l iteraciones
            {
              → Kernel::k_function(…)
                → dot(…)
                      por cada valor del vector
                      a predecir
                      // sv_size iteraciones
                      {
                          PRODUCTO ESCALAR
                      }
            }
            // predicción
    }
```

Figura 3. Pseudo-código la función MEX de predicción.

La función *svm_predict* se encuentra contenida en el archivo *svmpredict*, y es compilada como una función

MEX, permitiendo de esta manera tener un mejor rendimiento en cuanto al tiempo de ejecución realiza la clasificación o regresión para un conjunto de testeo. En esta función, cada una de los vectores característicos es clasificado mediante la llamada a la función *svm_predict_values*, la cual retorna los valores de decisión (etiqueta de clasificación) de un vector de testeo *x*, para un modelo dado. Dependiendo del kernel seleccionado, se aplica la función de distancia pertinente (*kernel::k_function*), el cual en todos los casos se basa en el cálculo del producto escalar (función *dot*) entre el vector de soporte del modelo, y el vector a clasificar.

En primera instancia se determina una complejidad algorítmica de *O(testing_instance_number * l * SV_size)*. En nuestro caso de estudio, se tiene que el valor de *testing_instante_number* se corresponde con el total de pixeles a analizar (357.355), el número de vectores de soporte (*l*) del modelo es de 86.402, y la dimensión de los vectores es 48 (6 escalas * 4 orientaciones * 2 características). De esta manera, el número total de multiplicaciones y acumulaciones que realiza el algoritmo es $1{,}481 \times 10^{12}$, lo que significa 1 billón y medio de operaciones.

Si bien los resultados teóricos establecen que el cuello de botella se encuentra presente por el gran número de iteraciones realizadas por la librería, se procedió al cálculo experimental de tiempos durante la ejecución de la función *svmpredict*. Debido que se trata de un ejecutable que es invocado por Matlab, en este caso el *profiler* de Matlab utilizado anteriormente no cuenta con el acceso al código fuente en C, ya que es una función compilada y no interpretada. De esta forma para el *profiler* la función es considerada como una caja negra, y no resulta útil para el análisis deseado.

Se analizaron diferentes alternativas que permitieran realizar el *profiling*, considerando diferentes herramientas, las cuales resultaron no factibles porque la mayoría de los *profilers* o bien son propietarios y requieren de la compra de licencias, o bien no son compatibles con el sistema operativo Windows. Analizando las alternativas existentes, se escogió la presentada en [39], en donde los tiempos son calculados a partir del reloj del sistema mediante la incorporación de código fuente.

De esta forma se analizó el tiempo asociado a la ejecución total de la función MEX, el tiempo asociado al bucle presente en la función *svm_predict_values(…)* y el tiempo consumido por una iteración de este bucle. Los resultados obtenidos son:

- **Tiempo total de ejecución**: 1910.125 segundos.
- **Tiempo total del bucle**: 1910.015 segundos.
- **Tiempo por iteración**: 0.005 – 0.006 segundos.

Estos resultados permiten deducir que el problema principal de performance se encuentra durante la ejecución del bucle presente en la función *svm_predict*, y más precisamente en el gran número de iteraciones que posee, tratándose de un problema de cómputo secuencial.

3.3. Optimización de la librería

A partir de los resultados obtenidos, se observa que un gran numero de vectores del conjunto generado a partir de una imagen mamográfica, poseen valores nulos. La razón de esto se debe a que dentro de la imagen existen zonas con texturas homogéneas, como puede apreciarse en la Figura 4. De esta forma, resulta auspiciosa la optimización software que descarte el procesamiento de estos vectores.

Figura 4. Imagen de entrada al sistema.

Esta información es computada originalmente por la librería *LIBSVM* sin tener en cuenta estos aspectos. De esto se propone realizar la modificación, considerando el análisis de solamente aquellos vectores que posean información relevante. El pseudo-código de la Figura 5, muestra la modificación realizada al previamente presentado.

```
→ svm_predict (…)
    por cada vector a predecir no nulo
    // testing_instance_number iteraciones
    {
      // cálculo de kvalues
      → svm_predict_values(…)
        por cada support vector del modelo
        // l iteraciones
        {
          → Kernel::k_function(…)
           → dot(…)
               por cada valor del vector a
               predecir
               // sv_size iteraciones
               {
                  PRODUCTO ESCALAR
               }
        }
        // predicción
    }
```

Figura 5. Pseudo-código de la función de predicción optimizado.

Con esta optimización, el bucle para la imagen "imb001" solo se realiza para 156.867 vectores (43% del

total original). El tiempo con sumido en este caso es de 1212.933 segundos, lo que representa un aceleración del 57%.

3.4. Migración a HW y partición HW/SW

Se propone para la migración hardware de la librería, el uso de una herramienta de síntesis de alto nivel (HLS) para diseño digital. El objetivo principal de estas herramientas es mejorar el tiempo de salida al mercado en los diseños de alta complejidad y reducir los costos de desarrollo. Las herramientas de HLS se han convertido en una alternativa a especificaciones de diseño de transferencia entre registro (RTL), aumentando el nivel de abstracción. Es importante señalar que el proceso de verificación también se mejora significativamente, ya que automatiza el proceso de descripción RTL. Otra ventaja de HLS sobre RTL está en la forma en que gestiona la sincronización, los tipos de datos y las interfaces. HLS infiere temporización desde el diseño de forma automática, proporciona tipos de datos para el modelado de los datos presentes en los sistemas digitales; y también es compatible con clases de interfaz / estructuras que se pueden especificar indirectamente por *pragmas* o directivas de diseño, o crear instancias directamente en función del lenguaje de alto nivel seleccionado. La tercera ventaja de HLS es que hace que una exploración del espacio de diseño mucho más productiva. El diseñador puede hacer rápidamente un ajuste a su micro-arquitectura de código fuente *SystemC*, o para las directivas de diseño en C / C++ y evaluar el desempeño, el área, y los impactos de energía [40].

Para la aceleración hardware de *LIBSVM* se escogió como plataforma una FPGA Virtex7 de Xilinx (xc7vx980), y el software Vivado HLS v14.2 [41].

Si bien la librería posee una gran flexibilidad en cuanto a configuraciones, se pretende lograr una aceleración aplicada al caso de estudio en cuestión, por lo que el código fue simplificado significativamente realizando un análisis de cobertura de código fuente de forma manual.

Dentro de la descripción de las rutinas asociadas a la clasificación existen construcciones no soportadas por la herramienta de HLS que debieron ser modificadas. En este sentido las estructuras propietarias de Matlab *mxArray*, se modificaron a estructuras del tipo *double[]*. También debieron establecerse como constantes todos aquellos bucles cuyas iteraciones dependían del valor de una variable. Este cambio también debió ser realizado en las construcciones de control tipo *while*.

Los punteros a variables del tipo arreglo debieron modificarse por los propios arreglos. Esta modificación también involucra a la administración de memoria, ya que la asignación dinámica de memoria no es soportada en Vivado HLS, y el tamaño de los arreglos debe estar establecido en tiempo de compilación. Asimismo, las estructuras definidas como punteros dobles fueron modificadas a estructuras del tipo matriz.

El diseño fue validado a través de la creación de un *testbench*, que carga desde archivo el modelo y los vectores a clasificar, y ejecuta la rutina de clasificación. Los resultados fueron contrastados con los obtenidos por la ejecución de la librería desde Matlab, demostrando el correcto resultado en los cálculos.

Como fue establecido en la sección 3.3, la latencia mayor del sistema se encuentra presente en la función *svm_predict.c* de la librería. Por este motivo se asume que ésta es la función a acelerar. Como una primera aproximación a la aceleración, se aplican las directivas *IN_LINE* a las funciones invocadas por la función en cuestión (*svm_predict_values*, *kernel_function*, *dot*), se aplica además la directiva *HLS_ARRAY_MAP* a los parámetros de la función, y la directiva *PIPELINE* a los bucles comprendidos dentro del código fuente.

Los resultados de la síntesis demuestran que los tiempos de respuesta de la función sin aplicar las directivas, respecto de haberlas aplicado, logra una aceleración de 8.56X.

El tiempo de respuesta de la función en HLS es de 47.979 segundos, y si bien se encuentra por encima en un factor de aproximadamente 20 respecto al tiempo de respuesta de la librería ejecutado en PC, el conjunto de las directivas aplicadas se trata de un conjunto inicial, cuyos resultados fueron prósperos respecto a la versión original sintetizada en Vivado HLS. Se debe considerar que esta latencia surge de la diferencia entre las frecuencias de operación de las plataformas utilizadas, siendo que la CPU posee una frecuencia de 2.5GHz, mientras que en la se utilizó una cristal de 100 MHz.

4. Conclusiones y trabajo futuro

Muchas de las aplicaciones que requieren de un clasificador pretenden que la respuesta sea entregada en un corto plazo. En particular, el caso de estudio utilizado en este trabajo, demuestra que el uso de un clasificador implementado por una máquina de soporte vectorial puede ser el cuello de botella del sistema. En este sentido, se analizó el sistema completo siguiendo una metodología, diagnosticando que la problemática se encuentra asociada a la implementación de la librería utilizada (*LIBSVM*).

Además, se logró determinar el problema existente dentro de la librería, permitiendo generalizarlo en otros campos de aplicación, ya que se observó que la mayor latencia del sistema se encuentra asociada con el número de vectores de soporte contenidos dentro del modelo, el número de vectores a clasificar y las dimensiones de dichos vectores. Por tanto, se recomienda a los diseñadores de sistemas de clasificación con

requerimientos de tiempo real prestar especial atención a dichos factores.

Por otra parte, se logró realizar una mejora significativa respecto al código original, mediante una modificación de la librería que permitió aumentar el rendimiento en un 57%. Se realizaron las adaptaciones respectivas del código C++ para ser compatible con Vivado HLS, y se realizó una primera síntesis lógica, generando una descripción RTL a partir de la adaptación del código original, demostrando la factibilidad de la migración de la librería a hardware.

Se pretende continuar realizando optimizaciones en diversos aspectos del hardware generado, de manera de obtener una mayor aceleración, y aplicar la metodología utilizada para la aceleración de la etapa de clasificación, en la etapa de entrenamiento, permitiendo cubrir todas las necesidades de los usuarios de la librería.

5. Agradecimientos

Este trabajo está financiado parcialmente por la Universidad FASTA, en el marco del proyecto "Aceleración de algoritmos de procesamiento de imágenes en FPGA (APIF)", Expediente 99-2014.

6. Referencias

[1] Bui, T. Q., Vu, T. T., & Hong, K. S. "Extraction of sparse features of color images in recognizing objects", 2016, *International Journal of Control, Automation and Systems*, *14,* 2, pp. 616-627, 2016.

[2] Chavhan, Y. D., Yelure, B. S., & Tayade, K. N., "Speech emotion recognition using RBF kernel of LIBSVM", In *2nd International Conference on Electronics and Communication Systems (ICECS),* pp. 1132-1135, 2015.

[3] Cortes, C., and Vapnik, V., "Support-vector networks", *Machine Learning*, 20, pp. 273-297, 1995.

[4] Qiang, Y., & Li, Y., "Research of Image Segmentation on Pulmonary Nodules in PET-CT Image Based on LIBSVM", In *International Journal of Digital Content Technology and its Applications*, *7,* 4, pp-764, 2013.

[5] Chang, C.-C., and Lin, C.-J., "LIBSVM: a library for support vector machines", 2001, Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[6] Collobert, R., and Bengio, S., "SVMTorch: Support Vector Machines for Large-Scale Regression Problems", *Journal of Machine Learning Research*, 1, pp. 143-160, 2001, Software available at http://bengio.abracadoudou.com/SVMTorch.html.

[7] Joachims, T. "SVM^{light} Support Vector Machine", Software available at http://svmlight.joachims.org.

[8] Afifi, S., M., GholamHosseini, H., and Sinha, R., "Hardware Implementations of SVM on FPGA: A State-of-the-Art Review of Current Practice", *International Journal of Innovative Science, Engineering & Technology (IJISET)*, 2, 11, pp. 733-752, 2015.

[9] Athanasopoulos, A., Dimou, A., Mezaris, V., and Kompatsiaris, I., "GPU Acceleration for Support Vector Machines", *In WIAMIS 2011*, 2011.

[10] Véstias, M., P., "High-Performance Reconfigurable Computing Granularity", *Encyclopaedia of Information Science and Technology*, pp. 3558-3567, 2015.

[11] Wielgosz, M., Jamro, E., Zurek, D., and Wiatr, K., "FPGA Implementation of the Selected Parts of the Fast Image Segmentation", *in Studies in Computational Intelligence*, 390, pp. 203-216, 2012.

[12] Hussain, H., M., Benkrid, K., and Seker, H., "The Role of FPGAs as High Performance Computing Solution to Bioinformatics and Computational Biology Data", *AIHLS2013*, pp. 102, 2013.

[13] Nagarajan, K., Holland, B., George, A., D., Slatton, K., C., and Lam, H., "Accelerating Machine-Learning Algorithms on FPGAs using Pattern-Based Decomposition", *Journal of Signal Processing Systems*, 62, pp. 43-63, 2011.

[14] Eklund, A., Dufort, P., Forsberg, D., and LaConte, S., M., "Medical Image Processing on the GPU-Past, Present and Future", *Medical Image Analysis*, vol. 17, pp. 1073-1094, 2013.

[15] Fowers, J., Brown, G., Cooke, P., and Stitt, G., "A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications", *in Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 47-56, 2012.

[16] Maghazeh, A., Bordoloi, U., D., Eles, P., and Peng, Z., "General Purpose Computing on Low-Power Embedded GPUs: Has It Come of Age?", in 2013 *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, pp. 1-10, 2013.

[17] Zynq-7000 All Programmable SoC. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[18] Platt, J., "Fast Training of Support Vector Machines Using Sequential Minimal Optimization", *Advances in Kernel Methods-Support Vector Learning*, vol. 3, 1999.

[19] Ta-Wen, K., Jhing-Fa, W., Jia-Ching, W., Po-Chuan, L., and Gaung-Hui, G., "VLSI Design of an SVM Learning Core on Sequential Minimal Optimization Algorithm", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, pp. 673-683, 2012.

[20] Cao, K.-k., Shen, H.-b., and Chen, H.-f., "A Parallel and Scalable Digital Architecture for Training Support Vector Machines", *Journal of Zhejiang University SCIENCE C*, vol. 11, pp. 620-628, 2010.

[21] Keerthi, S., S., Shevade, S., K., Bhattacharyya, C., and Murthy, K., R., K., "Improvements to Platt's SMO Algorithm for SVM Classifier Design", *Neural Computation*, vol. 13, pp. 637-649, 2001.

[22] Filho, J., G., Raffo, M., Strum, M., and Chau, W., J., "A General-Purpose Dynamically Reconfigurable SVM", *in 2010 VI Southern Programmable Logic Conference (SPL)*, 2010, pp. 107-112.

[23] Anguita, D., Pischiutta, S., Ridella, S., and Sterpi, D., "Feed-Forward Support Vector Machine without Multipliers", *IEEE Transactions on Neural Networks*, vol. 17, pp. 1328-1331, 2006.

[24] Andraka, R., "A Survey of CORDIC Algorithms for FPGA Based Computers", *in Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pp. 191-200, 1998.

[25] Jhing-Fa, W., Jr-Shiang, P., Jia-Ching, W., Po-Chuan, L., and Ta-Wen, K., "Hardware/Software Co-design for Fast Trainable Speaker Identification System Based on SMO", *in 2011 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 1621-1625, 2011.

[26] Venkateshan, S., Patel, A., and Varghese, K., "Hybrid Working Set Algorithm for SVM Learning with a Kernel Coprocessor on FPGA", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2014.

[27] Joachims, T., "Advances in kernel methods", *Chapter: Making large-scale support vector machine learning practical*, pp. 169-184, 1999.

[28] Shaojun, W., Yu, P., Guangquan, Z., and Xiyuan, P., "Accelerating On-Line Training of LS-SVM with Run-Time Reconfiguration", *in the International Conference on Field-Programmable Technology (FPT)*, pp. 1-6, 2011.

[29] Patil, R., Gupta, G., Sahula V., and Mandal, A., "Power Aware Hardware Prototyping of Multiclass SVM Classifier Through Reconfiguration" *in 2012 25th International Conference on VLSI Design (VLSID)*, pp. 62-67, 2012.

[30] Hussain, H., M., Benkrid, K., and Seker, H., "Reconfiguration-Based Implementation of SVM Classifier on FPGA for Classifying Microarray Data" *in 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 3058-3061, 2013.

[31] Ruiz-Llata, M., Guarnizo, G., and Yébenes-Calvino, M., "FPGA Implementation of a Support Vector Machine for Classification and Regression" *in The 2010 International Joint Conference on Neural Networks (IJCNN)*, pp. 1-5, 2010.

[32] Ago, Y., Nakano, K., and Ito, Y., "A Classification Processor for a Support Vector Machine with Embedded DSP Slices and Block RAMs in the FPGA" *in IEEE 7th International Symposium on Embedded Multicore Socs (MCSoC)*, pp. 91-96, 2013.

[33] Mahmoodi, D., Soleimani, A., Khosravi, H., and Taghizadeh, M., "FPGA Simulation of Linear and Nonlinear Support Vector Machine" *Journal of Software Engineering and Applications*, vol. 4, pp. 320-328, 2011.

[34] Kyrkou, C., Bouganis, C., S., Theocharides, T., and Polycarpou, M., M., "Embedded Hardware-Efficient Real-Time Classification with Cascade Support Vector Machines", *IEEE Transactions on Neural Networks and Learning Systems*, 2015.

[35] Kyrkou, C., Theocharides, T., and Bouganis, C., S., "An Embedded Hardware-Efficient Architecture for Real-Time Cascade Support Vector Machine Classification" *in 2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, pp. 129-136, 2013.

[36] Pedre, S., Krajník, T., Todorovich, E., & Borensztejn, P. "Accelerating embedded image processing for real time: a case study" in *Journal of Real-Time Image Processing*, vol. 11, no 2, pp. 349-374, 2016.

[37] Torrents-Barrena, J., Puig, D., Ferre, M., Melendez, J., Diez-Presa, L., Arenas, M., & Marti, J. "Breast masses identification through pixel-based texture classification", *In International Workshop on Digital Mammography*, pp. 581-588, 2014.

[38] The mini-MIAS database of mammograms: http://peipa.essex.ac.uk/info/mias.html

[39] Altman, Y. M. *Accelerating MATLAB Performance: 1001 tips to speed up MATLAB programs*. CRC Press, 2014.

[40] H.-Y. Liu, L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis", *DAC*, pp50-55, 2013

[41] Xilinx, "Vivado Design Suite User Guide: High-Level Synthesis", v2014.2, 2014, [Online]. Available: www.xilinx.com.