Automated Generation of Optimized Code Implementing SVM models on GPUs

O. Castro and I. F. Vega

***Abstract*—Deploying Support Vector Machine (SVM) models is challenging, since these contain complex math and logic, and also can be composed of a large number of real-valued coefficients; therefore, when performed by humans, the deployment is a slow and error-prone process. From the premise that the process of translating a machine learned predictive model into source code is deterministic and, hence, automatable, this paper's main contribution is demonstrating that it is possible to automate the generation of source code able to efficiently implement SVM models on Graphic Processing Units (GPU), and thus facilitate their operationalization. This research presents: guidelines for the automatic source code generation and efficient execution of SVM models, including code generation for specialized architectures such as GPUs using the Computed Unified Device Architecture (CUDA) platform/language; experimental evidence showing that the resulting source code implements these models efficiently; and a detailed description of the generated source code from sequential up to an optimized version of parallel code. Experiments with a large data set sized up to 9 GB were conducted to show the proposal's feasibility and scalability; such experiments showed an average speed-up of 112.71 times over the sequential, CPU-based, execution of SVMs; additionally, demonstrated a speed-up of 6.2 with respect to other SVM modeling tools running on GPU.**

***Index Terms*—Machine Learning, Support Vector Machines, GPU, CUDA, Code Generation.**

I. INTRODUCCIÓN

Las máquinas de soporte vectorial (SVM, por sus siglas en inglés de *Support Vector Machine*) son un conjunto de técnicas de aprendizaje máquina de tipo supervisado [1], propuestas por Cortes y Vapnik en 1995 [2]. Un modelo de SVM describe un conjunto de vectores, usualmente de alta dimensionalidad, cuya interpretación geométrica es la de hiperplanos que separan a las clases. Los modelos de SVM han demostrado ser valiosas herramientas para resolver gran variedad de problemas de regresión y clasificación en distintas áreas de la ciencia y la ingeniería (ver, por ejemplo: [3], [4], [5], [6], [7]).

Luego de conocerse la efectividad demostrada por las SVM en tareas predictivas de regresión y clasificación, se han desarrollado diversas herramientas de software que facilitan la construcción y la validación de este tipo de modelos. Algunas de las herramientas de tipo *open source* más usadas para la construcción de modelos de SVM son R y Python, aunque existen también muchas herramientas comerciales útiles en esta tarea.

O. Castro, Universidad Autónoma de Sinaloa, Culiacán, Sinaloa, México, oscarcastro@uas.edu.mx

I. F. Vega, Universidad Autónoma de Sinaloa, Culiacán, Sinaloa, México, ifvega@uas.edu.mx

Corresponding author: Inés F. Vega López.

Por lo general, los modelos predictivos como las SVM son construidos con la intención de integrarlos en aplicaciones de software. Los datos generados por sistemas en producción se utilizan como entrada a modelos predictivos para generar predicciones, destinadas a automatizar acciones o apoyar la toma de decisiones. El proceso de incorporar un modelo predictivo a software en producción suele conocerse como despliegue (*deployment*, en inglés). Un modelo de SVM es, esencialmente, una función matemática. Al proceso de aplicar esta función sobre nuevos datos se le llama predicción (*scoring*). El valor de un modelo se encuentra en la precisión de sus predicciones y en la aplicación de éstas a un proceso productivo.

Ya incorporados a un sistema en producción, el generar predicciones al ejecutar modelos de SVM puede ralentizar el sistema al convertirse en un cuello de botella y afectar el desempeño global del sistema. Las principales razones son dos: por un lado, los sistemas actuales generan una gran cantidad de datos que deben ser procesados y evaluados a fin de obtener una predicción por parte de la SVM (éste es un escenario cada vez más común, de manera genérica denominado *big data*); por otro lado tenemos la alta demanda de recursos computacionales requeridos por este tipo de modelos. Con el objetivo de reducir tal ralentización, la comunidad científica ha encontrado un remedio en el uso de hardware especializado; con muy buenos resultados, destaca la propuesta de utilizar unidades de procesamiento gráfico (GPU, *Graphic Processing Unit*). Las GPU, al inicio sólo orientadas a realizar procesamiento de gráficos, han tomado gran relevancia para acelerar cómputo, en particular operaciones de álgebra lineal. Esto resulta conveniente, pues muchos tipos de modelos predictivos inferidos mediante algoritmos de aprendizaje máquina (*machine learning*), en particular los modelos de SVM, son funciones matemáticas compuestas principalmente por operaciones del álgebra lineal. Por otro lado, y como ventaja adicional, lenguajes de programación como CUDA (*Compute Unified Device Architecture*) y OpenCL (*Open Computing Language*) facilitan de manera significativa la construcción de aplicaciones de software ejecutables en las GPU.

La presente investigación permitió identificar la existencia de dos alternativas para hacer el despliegue de modelos predictivos: 1) la codificación manual del modelo; 2) la conexión, vía un flujo de trabajo secuencial (mejor conocido por el término en inglés *pipeline*), del sistema operacional con alguna herramienta de modelado. Hasta donde se sabe y al momento de escribir este artículo, no existe una herramienta capaz de generar código fuente libre de dependencias para su ejecución en el procesador central (CPU, *Central Processing Unit*) o

GPU a partir de la descripción formal de un modelo de SVM. Existe la posibilidad de aprovechar herramientas analíticas en la construcción de modelos de SVM empleando las GPU y de alguna manera conectar dichas herramientas con el software en producción; si es el caso, la herramienta de modelado debe estar disponible en un servidor, lista para recibir datos y regresar predicciones en un esquema tipo cliente-servidor, donde el cliente es el software en operación y el servidor es la herramienta de modelado. Una ventaja importante de tal esquema es su relativa facilidad de construcción. Las desventajas más relevantes de dicho esquema son las siguientes: 1) el servidor se convierte en un agente ralentizante del sistema y un punto crítico de fallo; 2) se requiere una conexión de red para la transferencia de datos (la conexión de red no siempre está disponible en algunos sistemas en producción y, si la hay, podría también ser ralentizante si el sistema procesa un alto volumen de datos); 3) las herramientas de modelado suelen ser diseñadas con el fin de generar prototipos rápidamente, no para procesar altas demandas de predicciones; 4) la administración y el mantenimiento de un sistema así se vuelven complicados porque existe una dependencia tecnológica hacia las herramientas de modelado [8].

La principal contribución de la investigación aquí presentada es demostrar que es posible automatizar la generación de código fuente capaz de implementar eficientemente modelos de SVM en las GPU para así facilitar su puesta en operación. Además, muestra cómo, a partir de un algoritmo base, puede generarse código optimizado para utilizar características avanzadas de las GPU. El resto de este artículo se organiza de la siguiente manera: la Sección II presenta el trabajo relacionado con generación y uso de modelos de SVM en las GPU; la Sección III explica la implementación eficiente de modelos de SVM; la Sección IV muestra los resultados de la evaluación experimental; por último, la Sección V expone la discusión y las conclusiones del presente trabajo.

II. TRABAJO RELACIONADO

La literatura científica reporta gran variedad de propuestas para sacar provecho de las GPU, tanto al momento de entrenar modelos de SVM como para su uso en tareas de regresión y clasificación. Destaca, por ejemplo, el trabajo de Catanzaro et al. [9], donde los autores demuestran la factibilidad de paralelizar el algoritmo de entrenamiento y predicción de SVMs mediante el uso de una GPU. Los resultados obtenidos son comparados con la popular biblioteca LibSVM [10] y logran un mejor desempeño, tanto al momento de entrenar como al momento de predecir. Los experimentos de los autores reportan ser de 5 a 32 veces más rápidos en el tiempo de entrenamiento, y de 120 a 150 veces más rápidos en la generación de predicciones al compararse con la herramienta LibSVM.

Otro ejemplo es el de Carpenter [11], quien muestra una herramienta denominada cuSVM para generar modelos de SVM con ayuda de una GPU a través de código CUDA. Con respecto a LibSVM, el aumento de velocidad reportado en sus experimentos es consistente con lo reportado por Catanzaro et al., pues logra ser de 13 a 73 veces más rápido durante el entrenamiento de los modelos, y de 22 a 172 veces más rápido en la predicción. Asimismo, Li et al. [12] proponen GPUSVM, una herramienta dedicada al entrenamiento de modelos tipo SVM que adopta CUDA como lenguaje de programación para su implementación en las GPU. La evaluación experimental presentada por los autores equipara GPUSVM con LibSVM; según sus resultados, su herramienta puede ser hasta 20 veces más rápida que LibSVM en el entrenamiento de modelos, y hasta 80 veces más rápida en la ejecución de predicciones.

ThunderSVM, herramienta desarrollada por Wen et al. [13], construye modelos de SVM, así en GPU como en CPU. ThunderSVM fue desarrollada en C/C++, pero tiene interfaces compatibles con Python, R y Matlab. Esta herramienta puede ser instalada para ejecutarse en una CPU multinúcleo o en una GPU. Cuando un modelo ha sido entrenado con ThunderSVM, es posible guardarlo en un formato propietario y luego cargarlo para generar predicciones. Según reporta la evaluación experimental presentada, al compararse con LibSVM, ThunderSVM es aproximadamente 100 veces más rápida si se ejecuta en una GPU; mientras que, al utilizar sólo una CPU, ThunderSVM es 10 veces más rápida que LibSVM.

Sin duda, los trabajos aquí descritos son contribuciones valiosas al estado del arte de las SVM. Estas propuestas muestran claramente los beneficios prácticos de aplicar una arquitectura especializada como las GPU a tareas de entrenamiento o predicción con modelos de SVM. Las diversas propuestas reportan mejoras considerables en los tiempos de ejecución al usar las GPU, aunque vale la pena enfatizar que los incrementos del rendimiento dependen de diversos factores, por ejemplo: las prestaciones del equipo de cómputo (RAM, CPU, GPU, etc.), el conjunto de datos y el tamaño del modelo.

Los trabajos relacionados aquí reseñados pueden llevar a cabo las tareas de entrenamiento y predicción de modelos de SVM, es decir, se trata de propuestas que generan modelos, mas para utilizarse en producción dependen de la misma herramienta con la cual el modelo fue generado. La propuesta del presente trabajo, por otro lado, se ocupa de automatizar el proceso de generar código fuente libre de dependencias e incorporar los modelos a los sistemas en producción para que funcionen como pieza integral de estos y así además generen valor durante el proceso de toma de decisiones. A diferencia de lo encontrado en el estado del arte, la propuesta presentada en este artículo se centra en generar código fuente destinado a la tarea de predicción, así como automatizar el despliegue de modelos de SVM para su ejecución eficiente sirviéndose de una GPU; esto permite integrar un componente de software ligero, pues no se lleva a producción toda una herramienta analítica completa. De acuerdo con los avances consultados en la literatura científica, se trata de la primera propuesta orientada a generar automáticamente código fuente capaz de realizar el cómputo de las predicciones de modelos de SVM en una GPU.

III. GENERACIÓN DE CÓDIGO FUENTE

Con el propósito de generar código de manera automatizada, se diseñó una herramienta que, a partir de una descripción formal de un modelo de SVM, extrae la información necesaria

y realiza diversas transformaciones algorítmicas a fin de implementar el modelo en una arquitectura destino. En esta sección se describen los principales componentes de tal herramienta: un *front-end*, representación intermedia y un *back-end*. La Fig. 1 muestra el diseño general de la herramienta de generación de código.

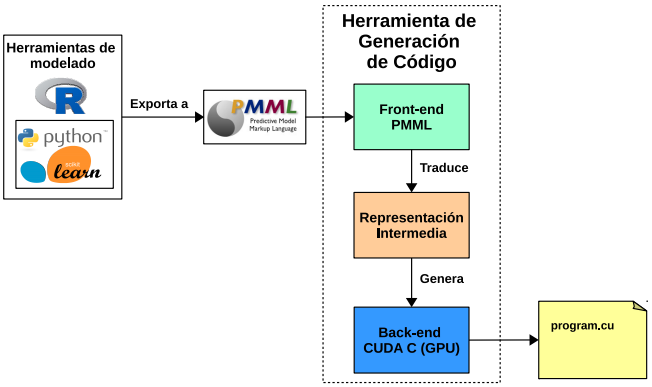

Fig. 1. Diseño de la herramienta de generación de código, utilizando PMML para describir modelos de SVM como entrada. Un modelo en PMML es traducido a una representación intermedia y posteriormente se genera código para implementar en una GPU.

A. *Front-end*

Los modelos de SVM suelen construirse con herramientas de modelado como R [14] o Python (Scikit-learn) [15], las cuales tienen bibliotecas basadas principalmente en LibSVM; cada herramienta de modelado tiene su propia manera de representar un modelo de SVM. Con el objetivo de no depender de una sola herramienta de modelado como entrada a la herramienta de generación de código, se optó por utilizar PMML (*Predictive Model Markup Language*) como formato de representación de modelos; PMML es un estándar de representación de modelos predictivos basado en XML compatible con diversas herramientas. Cabe mencionar, es posible construir modelos de SVM con diferentes herramientas analíticas y exportarlas a PMML para su posterior uso.

El *front-end* traduce un modelo de SVM en formato PMML a una representación intermedia propia; se extrae la información relevante de una descripción formal del modelo de SVM; el contenido de dicha descripción es determinado por el algoritmo de inferencia usado durante el entrenamiento o la construcción del modelo. Aunque de momento se emplea PMML como formato de entrada, es posible extender la funcionalidad de la herramienta para hacerla compatible con otros formatos de representación de modelos de SVM. El proceso de traducción es máquina a máquina, entonces, se simplifican algunos pasos de verificación y análisis de la representación del modelo de SVM.

B. *Representación Intermedia*

A partir de la traducción de un modelo de SVM a la representación intermedia, se obtiene una descripción que contiene los siguientes elementos: nombre de las variables de entrada, tipos de variables de entrada, tipo de *kernel*, número de vectores de soporte, coeficientes, sesgo (*bias*) y las etiquetas de salida. Se definió una representación intermedia capaz de contener todos los elementos de un modelo de SVM. Aquí, es preciso destacar que la estructura general de los modelos de SVM es siempre la misma y, por ende, se puede generalizar una estructura donde se incluya todo lo necesario para posteriormente utilizarla como entrada para los diferentes *back-ends* que generan código compatible con un lenguaje destino particular.

La representación intermedia está definida por los elementos o variables de una SVM y por las operaciones de álgebra lineal realizadas por los diferentes tipos de *kernel* de modelos de SVM. La función matemática del modelo puede ser expresada de manera general como se muestra en la Ecuación 1.

$$d(\vec{x}) = \sum_{i=1}^{m} \alpha_i K(\vec{S_i}, \vec{x}) + b \tag{1}$$

Donde $\vec{x}$ es un vector que representa la entrada del modelo, tiene n dimensiones y caracteriza el objeto de interés, razón por la cual muchas veces es denominado *vector característico*. Por su parte, $\vec{S}$ representa los m vectores de soporte, cada uno de dimensión n. Se usa $\vec{S_i}$ para representar el *i-ésimo* vector en $\vec{S}$. La variable α representa un vector de tamaño m conteniendo coeficientes y b es un escalar que representa el sesgo. La función *kernel* de la SVM es representada por $K()$; esta función se define al momento de entrenar el modelo y puede tomar alguna de las formas indicadas en la Tabla I.

TABLA I
LISTA DE LOS SVM KERNELS MÁS UTILIZADOS Y SU ECUACIÓN

SVM Kernel	Ecuación
Lineal	$K(\vec{x}, \vec{z}) = \vec{x} \cdot \vec{z}$
Polinomial	$K(\vec{x}, \vec{z}) = (\gamma * (\vec{x} \cdot \vec{z}) + c)^d$
Radial	$K(\vec{x}, \vec{z}) = e^{-\gamma * \|\|\vec{x} - \vec{z}\|\|^2}$
Sigmoid	$K(\vec{x}, \vec{z}) = tanh(\gamma * (\vec{x} \cdot \vec{z}) + c)$

Nota: γ, c y d son valores determinados por el algoritmo de entrenamiento.

C. *Back-end*

El *back-end* es un módulo de software que recibe como entrada una representación intermedia de un modelo de SVM. El *back-end* analiza la representación intermedia y genera como salida el código fuente que implementa la SVM en el lenguaje especificado. El diseño modular de la herramienta de generación de código permite agregar de manera sencilla más componentes *back-end* para diferentes lenguajes de programación y/o arquitecturas de cómputo. A continuación, se describe el diseño del código secuencial que implementa un modelo de SVM, mismo código tomado como base para generar código a implementar en la GPU.

D. *Código Base*

El código base no hace uso de hardware especializado, se presenta aquí con propósitos ilustrativos y para establecer una línea base de comparación que permita la evaluación de la

ejecución del código generado para las GPU. El Algoritmo 1 muestra el pseudocódigo de naturaleza secuencial; su parte central es un ciclo, que implementa la sumatoria contenida en la Ecuación 1. Durante cada iteración del ciclo principal se evalúa la función *kernel* de la SVM; función cuyo resultado (un escalar) se multiplica por el *i-ésimo* coeficiente descrito en el modelo, y el resultado se acumula en una variable; al final, este resultado acumulado se suma al sesgo. El algoritmo aquí descrito representa un modelo de clasificación binaria: el resultado final es una etiqueta que indica si el vector de entrada es clasificado como A o B, dependiendo del signo del valor calculado (líneas 8 a la 11). Sin embargo, este mismo algoritmo puede utilizarse orientado a tareas de regresión si se retorna el valor de la variable D y no una etiqueta.

Algoritmo 1: pseudocódigo del algoritmo de clasificación para una ejecución secuencial de un modelo de SVM.

Input: un vector $X \leftarrow \{x_1, x_2, \ldots, x_n\}$ de valores numéricos.
Output: una cadena que contiene la etiqueta de predicción de la SVM.

```
1  S ← una matriz de vectores de soporte de r × n;
2  C ← un vector de coeficientes con r elementos;
3  D ← 0;
4  for i ← 0 to r do
5  |  D ← D + K(X, S_i) × C_i;
6  end
7  D ← D + b;
8  if D > 0 then
9  |  return "A";
10 else
11 |  return "B";
12 end
```

IV. GENERACIÓN DE CÓDIGO PARA LA GPU EN CUDA C

En esta sección se presenta el diseño del *back-end* destinado a generar código que implementa modelos de SVM para su ejecución eficiente en una GPU. A fin de generar código ejecutable en una GPU, se optó por CUDA C, plataforma que facilita expresar el paralelismo en dispositivos GPU mediante extensiones de los lenguajes C y C++.

Cuando se usa CUDA para desarrollar código ejecutable en la GPU se deben definir dos tipos de funciones: esclavo y maestro. Las funciones tipo esclavo (llamadas CUDA *kernels*) se ejecutan en hilos y el cómputo de estos hilos se distribuye entre los núcleos de procesamiento de la GPU. Cada hilo ejecuta instrucciones de la función esclavo y varios hilos se ejecutan de manera paralela; a cada hilo se le asigna un identificador único (*thread id*), el cual permite diferenciarlos. La función maestro se encarga de reservar y liberar la memoria principal y la memoria del dispositivo (GPU), copiar datos hacia/desde el dispositivo, además de llamar las funciones esclavo [16].

Al llamar las funciones esclavo se debe indicar la cantidad de hilos que van a ejecutar dicha función. En el ambiente CUDA se aplican dos parámetros numéricos denominados *grid* y *bloque*. El prototipo para llamar a una función CUDA C es el siguiente:

```
nombreFuncion<<<grid, bloque>>>(param...)
```

El bloque representa un grupo de hilos a ejecutarse en paralelo; los hilos de un mismo bloque se pueden comunicar mediante memoria compartida. El *grid* es un grupo de bloques y cada bloque contiene la misma cantidad de hilos. La cantidad total de hilos en la cual se va a ejecutar una función, se obtiene al multiplicar el valor del *grid* por el valor de bloque.

Los valores de los vectores de soporte, coeficientes, sesgo, y algunos elementos de las funciones *kernel* son constantes durante la ejecución de la SVM. Por lo tanto, las primeras instrucciones de la función maestro asignan memoria en el dispositivo con la función `cudaMalloc()` y copian estos valores desde la memoria principal al dispositivo mediante la función `cudaMemcpy()`. Al terminar todos los cálculos, y antes de finalizar el programa, se libera la memoria asignada a cada variable con la función `cudaFree()`.

A diferencia del código base donde sólo hay una función de predicción, el código a generar en CUDA C tiene dos funciones: esclavo y maestro. La función esclavo realiza cómputo en los núcleos de la GPU y la función maestro se encarga de coordinar la distribución de los datos a las funciones esclavo. La función de la versión secuencial recibe un vector y retorna una predicción. El código a ejecutar en la GPU se crea de forma que favorezca la generación de predicciones en lote, tratando de minimizar la copia de datos; esto se debe a que la copia de datos entre la memoria principal y la memoria del dispositivo puede ralentizar el proceso.

La función maestro recibe una matriz M, que representa los datos de entrada, y un vector O, donde se almacenan las predicciones. Se diseñó así el código debido a la arquitectura de ejecución de las GPU, pues éstas siguen el paradigma SIMD (*Single Instruction Multiple Data*), donde un conjunto de datos es procesado en paralelo. Cada columna de la matriz M representa el valor de una variable independiente, y cada fila un vector que representa un objeto sobre el cual se va a realizar una predicción. El vector O contiene las predicciones y el número de elementos de O es igual al número de filas de la matriz M.

La manera como se realiza el cómputo para obtener las predicciones se describe a continuación. Se distribuye el cómputo de tal forma que cada hilo realice los cálculos para generar una predicción; es decir, en cada hilo se genera una predicción por fila de la matriz de entrada. Se lanza la mayor cantidad de hilos posible, de acuerdo con la memoria disponible del dispositivo. Si el tamaño del conjunto de vectores a los cuales se les va a generar una predicción es mayor a la memoria del dispositivo, el conjunto total se parte en lotes de un tamaño menor o igual a la memoria disponible del dispositivo. El código de la función esclavo para el cómputo del modelo de SVM se muestra en el Algoritmo 2.

El Algoritmo 3 muestra un pseudocódigo de la función maestro. Debido a que una de las principales acciones ralentizantes es la copia de datos entre la memoria principal y la

Algoritmo 2: pseudocódigo de la función esclavo para la GPU, en la cual se calcula una predicción por hilo. En donde r es la cantidad total de vectores de soporte y n es el número de elementos de cada vector de soporte.

Input: X, S, C, b, T y l, en donde:
X ←vector con los valores de entrada,
S ←un vector representando los vectores de soporte,
C ←un vector representando los coeficientes,
b ←valor escalar que contiene el sesgo,
T ←vector donde se almacenan los resultados,
l ←valor escalar que representa la cantidad de vectores característicos a procesar (tamaño del lote).
Output: no aplica, se almacena el resultado en T

```
. tid ←identificador de hilo;
if tid < l then
    idx ← tid × n ;                /* Índice de X */
    D ← 0;
    for i ← 1 to r do
        D ← D + K(X_idx, S_i) × C_i;
    end
    D ← D + b;
    if D < 0 then
        T_tid ← −1;
    else
        T_tid ← 1;
    end
end
```

Algoritmo 3: pseudocódigo de la función maestro que se encarga de organizar los datos por lotes e invocar las funciones esclavo para su ejecución en la GPU.

Input: M, O y L, donde:
M ←representa los datos de entrada en una matriz de $L \times n$, cada fila representa un vector sobre el cual se realiza una predicción,
O ←vector de longitud L donde se almacenan las predicciones,
L ← cantidad de vectores característicos sobre los que se va a calcular una predicción.
Output: no aplica, se almacena el resultado en O

```
.
S ←vector representando los vectores de soporte;
C ←vector representando los coeficientes;
Asignar memoria en el dispositivo para las variables S y C con cudaMalloc();
Copiar S y C de la memoria principal al dispositivo con cudaMemcpy();
free ←espacio libre de la memoria del dispositivo;
Obtener el tamaño del lote menor a free y la cantidad de lotes necesarios para obtener las L predicciones.
ex ←cantidad de lotes (número de ejecuciones)
l ←cantidad de vectores a calcular por lote;
b ← 1024;                      /* tamaño de bloque */
g ← l/b;                       /* tamaño del grid */
if (l mod g) ≠ 0 then
    g ← g + 1;
end
T ←vector donde se guardan los resultados parciales de la función esclavo;
M' ←vector en donde se guarda un número l de filas de manera temporal para procesar un lote;
Asignar memoria en el dispositivo para las variables T y M' con cudaMalloc();
for i ← 0 to ex do
    offset ← i × l;
    M' ← guardar las l filas correspondientes al lote_i de la matriz de entrada en modo vector fila;
    Copiar M' de la memoria principal al dispositivo con cudaMemcpy();
    svmKERNEL<<< g, b >>>(M', S, C, b, T, l);
    Copiar T del dispositivo a la memoria principal y guardar en el vector T' con cudaMemcpy();
    for k ← 1 to l do
        if (T'_k + b) > 0 then
            O_{k+offset} ← "A";
        else
            O_{k+offset} ← "B";
        end
    end
end
Liberar memoria en el dispositivo asignada a las variables S, C, T y M' con cudaFree();
```

memoria del dispositivo, se optó por un diseño que minimiza la cantidad de llamadas a la función `cudaMemcpy()`. En el Algoritmo 3 inicialmente se declaran variables, se asigna memoria en el dispositivo y se copian los valores a ser constantes durante la ejecución, como los vectores de soporte y los coeficientes. Posteriormente, se calcula el espacio libre de la memoria del dispositivo, y a partir de ese valor se definen un tamaño de lote (por el número de vectores característicos) y la cantidad de lotes (o ejecuciones) necesarias para cubrir todo el conjunto de datos.

Con base en la cantidad de lotes a procesar, se define un ciclo principal, donde se realiza lo siguiente: 1) se copia un lote de datos M' al dispositivo; 2) se invoca la función `svmKERNEL()`; 3) se copian los resultados del cómputo de la función esclavo a la memoria principal en el vector T'; y 4) de manera secuencial se itera el vector T', durante cada iteración se compara el *k-ésimo* elemento con un umbral para asignarle la etiqueta A o B, asignación luego almacenada en el vector O. Fuera del ciclo, se libera la memoria asignada a las variables dinámicas declaradas (en la memoria principal y en la GPU). Al final, el vector O contiene las etiquetas resultantes y, al ser pasado por parámetros, puede utilizarse en el programa desde donde fue llamada la función maestro.

La cantidad de hilos con la cual se invoca la función esclavo se calcula a partir de la cantidad de vectores a procesar por lote. Se definió un valor constante para la cantidad de hilos por bloque, en el Algoritmo 3 se le asigna un valor de 1024; sin embargo, el número de hilos por bloque puede ser distinto y, de acuerdo con la documentación de CUDA [17], se recomienda que sea un múltiplo de 32. Se asigna un valor al tamaño del *grid* (que define la cantidad de bloques) al dividir la cantidad de vectores a procesar por lote entre el número de hilos por bloque; si el resultado de esta división no es un entero, se toma el entero inmediato superior. La última llamada a la función esclavo podría lanzar hilos de más, pero esto no afecta en mayor medida el rendimiento del programa; además, en la función esclavo hay una validación para que los hilos extra no realicen ningún cálculo.

Cabe aquí enfatizar lo siguiente: a diferencia de las herramientas listadas en la Sección II, el código generado por la herramienta desarrollada en este trabajo es autocontenido.

El código generado debe ser compilado y puede ser llamado desde otros programas; es decir, es un código fuente libre de dependencias porque, una vez generado, éste puede utilizarse sin depender de otros programas o bibliotecas adicionales. Por su parte, la función maestro está diseñada para poder procesar un conjunto de datos cuyo tamaño supere la memoria disponible de la GPU.

V. Evaluación Experimental

Esta sección presenta una descripción detallada de los experimentos realizados, se describen: la configuración del hardware empleado, el conjunto de datos de prueba y los resultados obtenidos.

A. Plataforma Experimental

Todos los experimentos fueron realizados con una computadora tipo estación de trabajo (*workstation*) con procesador Intel Xeon W-2133 con 6 núcleos a una frecuencia de operación de 3.60 GHz y 64 GB en RAM; el sistema operativo elegido fue Linux Ubuntu 18. Las características de la GPU empleada se muestran en la Tabla II.

TABLA II
MODELO Y CARACTERÍSTICAS DE LA TARJETA GRÁFICA UTILIZADA EN LOS EXPERIMENTOS

Especificaciones	
Modelo	GeForce GTX 1080
Núcleos	$2,560$
RAM GDDR5X (GB)	8
Arquitectura	Pascal
Frecuencia de la memoria (Gb/s)	10
Ancho de banda de memoria (GB/s)	320

El lenguaje de programación elegido fue C, complementado con CUDA C para el uso de la GPU. Se efectuaron experimentos en cuatro ambientes distintos, donde se tomó el tiempo transcurrido del cómputo de las predicciones de un modelo de SVM. Los ambientes en los cuales se realizaron los experimentos fueron los siguientes: 1) el entorno R; 2) el código base secuencial generado por la herramienta producto de esta investigación; 3) el código generado por esta misma herramienta para ser ejecutado en una GPU; y 4) ThunderSVM para ejecutar modelos de SVM en las GPU. La comparación entre el entorno R y el código base secuencial permite tener un punto base de evaluación el rendimiento del código generado. La comparación entre el código generado para una GPU y ThunderSVM permite demostrar/analizar el rendimiento del código generado. Cada experimento se realizó 30 veces, tomando el tiempo transcurrido de generación de predicciones con un modelo de SVM. Se reporta aquí el tiempo de ejecución promedio obtenido después de haber eliminado el tiempo más alto y el más bajo.

B. Datos de Prueba y Construcción del Modelo de SVM

En la validación experimental se utilizó el conjunto de datos *Synthetic Financial Dataset for Fraud Detection*, el cual contiene datos sintéticos de transacciones móviles de dinero [18]; su tamaño es de 493.50 MB, con $6'362,620$ registros y 11 variables (la variable objetivo define si una transacción es fraudulenta o no). Se generó el modelo de SVM con la herramienta R y el paquete KernLab [19]; se creó un modelo de SVM con *kernel* tipo radial. Los parámetros de entrenamiento usados fueron seleccionados por *grid search* (búsqueda de la mejor combinación de parámetros), $\sigma = 8.347$ y $C = 64$; después de un análisis del conjunto de datos, se eligieron seis variables para la formulación del modelo resultante, que tiene una precisión de $97\,\%$ y está constituido por $1,609$ vectores de soporte. El modelo tiene seis variables independientes: una categórica y cinco numéricas; la variable categórica tiene cinco categorías, las cuales son tratadas como variables independientes numéricas y, entonces, los vectores son de dimensión 10. Por lo tanto, al final, los vectores de soporte pueden ser representados por un vector de una longitud de $16,090$ o matriz de $1,609 \times 10$.

Con el objetivo de evaluar el comportamiento de los modelos y el código generado cuando los datos sobrepasan los límites de memoria de las GPU, el conjunto de datos se replicó en diferentes factores de escala. Se recurrió a factores de escala de 1, 2, 4, 8, 16 y 20 para los experimentos de la tarea de predicción de modelos de SVM. El modelo de SVM fue generado con el software R y posteriormente exportado a PMML. Mediante la herramienta propuesta, y con el modelo PMML como entrada, se generó código ejecutable en un solo hilo con el lenguaje C y ejecutable en la GPU con código en el lenguaje CUDA C. Con ThunderSVM es posible construir modelos de SVM, no obstante, debido a su naturaleza, el modelo resultante no es exactamente el mismo que el creado por R, aunque se utilicen los mismos parámetros y el mismo conjunto de datos. Por ende, con el propósito de hacer una comparación justa, el mismo modelo construido en R fue traducido a formato ThunderSVM. Se desarrolló un *script* de R y de Python para ThunderSVM a fin de leer los datos, generar las predicciones para cada uno de los ambientes y medir el tiempo transcurrido de generación de predicciones. También se desarrolló un programa en C que lee el conjunto de datos, invoca una función para generar predicciones con el código base en C o para la GPU empleando CUDA C, y mide el tiempo transcurrido en la generación de las predicciones. Con la función `clock_gettime()`, se midió el tiempo transcurrido en los programas en código C y CUDA C.

El *kernel* del modelo de SVM entrenado fue radial, su ecuación es descrita en la Tabla I. La parte central de esa ecuación es una operación norma vectorial cuadrada entre $\vec{x}$ y $\vec{y}$. La operación de norma vectorial está definida por la Ecuación 2. La Ecuación 3 muestra una versión simplificada de la norma vectorial al eliminar la raíz cuadrada y su elevación al cuadrado. Sin embargo, la Ecuación 3 puede ser sustituida por la Ecuación 4. Mientras programar la Ecuación 3 requiere realizar todos los cálculos en tiempo de ejecución, programar la Ecuación 4 permite reducir la cantidad de operaciones de la siguiente manera: el producto punto de $\vec{x} \cdot \vec{x}$ del vector de entrada sólo se realiza una vez, el producto punto de $\vec{z} \cdot \vec{z}$ se puede realizar en tiempo de compilación porque los vectores de soporte son conocidos (sólo es necesario copiar un vector de resultados) y lo que se ha de calcular en tiempo de ejecución es la multiplicación del valor constante dos por el resultado

del producto punto de $\vec{x} \cdot \vec{z}$.

$$||\vec{x} - \vec{z}||^2 = (\sqrt{(x_0 - z_0)^2 + \ldots + (x_n - z_n)^2})^2 \quad (2)$$

$$||\vec{x} - \vec{z}||^2 = (x_0 - z_0)^2 + \ldots + (x_n - z_n)^2 \quad (3)$$

$$||\vec{x} - \vec{z}||^2 = (\vec{x} \cdot \vec{x}) - 2(\vec{x} \cdot \vec{z}) + (\vec{z} \cdot \vec{z}) \quad (4)$$

C. *Resultados*

La Fig. 2 muestra un gráfico de barras con los resultados de las ejecuciones de los experimentos llevados a cabo con la herramienta R y el código generado para ejecución secuencial en C. Estos resultados permiten tener un punto de comparación inicial del código generado para GPU; tal comparación se realizó con los factores de escala de datos 1, 2, 4 y 8. Se puede observar que los experimentos hechos con el código generado para el lenguaje C son en promedio 1.4 veces más rápidos que la ejecución en R; esto se debe a que R es un lenguaje interpretado, en tanto el código en C, por su parte, es compilado.

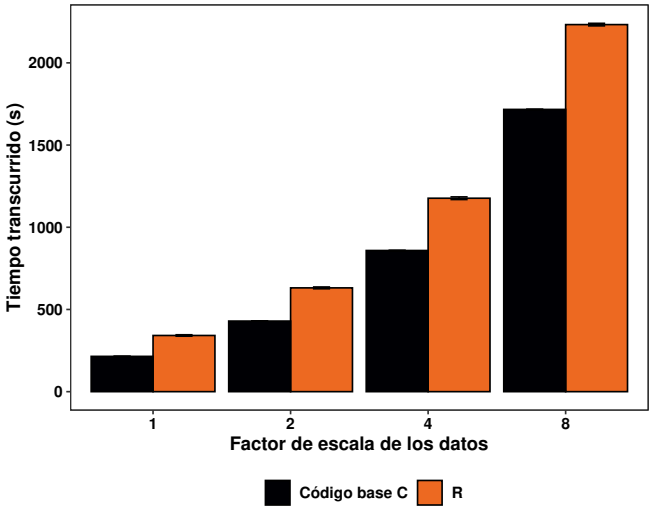

Fig. 2. Resultado de los experimentos, contrastando el tiempo de ejecución transcurrido de un modelo de SVM utilizando R contra el de uno empleando el código generado para ejecución secuencial en C.

La Fig. 3 muestra un gráfico de barras con los resultados de las ejecuciones de los experimentos efectuados mediante la GPU con la herramienta ThunderSVM y código generado para CUDA C. Se realizaron experimentos con factores de escala de datos 1, 2, 4, 8, 12, 16, y 20, superando la capacidad de memoria RAM de la GPU. Los experimentos muestran que la ejecución del código generado es en promedio 6.2 veces más rápida que cuando se usa ThunderSVM.

La Tabla III expone los tiempos de ejecución en segundos por cada ambiente y cada factor de escala de los datos. Si realizamos una comparación directa, los experimentos con ThunderSVM son en promedio 19.75 veces más rápidos que R, mientras las ejecuciones del código generado por la herramienta para la GPU aquí presentada son, en promedio, 112.71 veces más rápidas en contraste con las realizadas usando R.

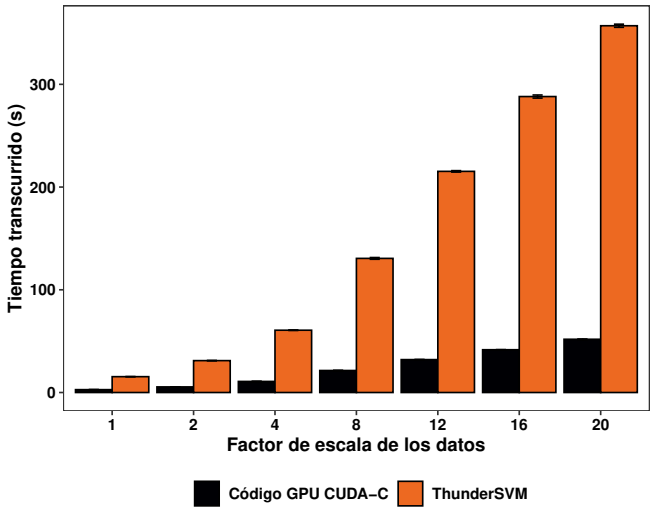

Fig. 3. Resultado de los experimentos utilizando la GPU GTX 1080, reportando el tiempo transcurrido de ejecución de un modelo de SVM empleando la herramienta ThunderSVM y usando el código generado para CUDA C.

TABLA III
RESULTADOS DE LOS TIEMPOS DE EJECUCIÓN DE UN MODELO DE SVM EN LOS DIFERENTES AMBIENTES Y FACTOR DE ESCALA DE LOS DATOS

	Ambiente / Tiempo de ejecución en segundos			
Escala	**R**	**Base C**	**CUDA C**	**ThunderSVM**
1	341.75	214.70	2.80	15.43
2	631.32	429.42	5.47	31.05
4	1,176.63	858.72	10.79	60.65
8	2,232.68	1,716.74	21.43	130.56
12	-	-	32.04	215.27
16	-	-	41.66	288.10
20	-	-	51.85	357.02

VI. CONCLUSIONES

El presente trabajo demuestra que es posible, a partir de la concepción formal de un modelo de SVM, generar código fuente de manera automática para su despliegue. También, que el código generado es altamente eficiente y aprovecha el paralelismo de dispositivos como las GPU. Asimismo, se encontró que el principal factor ralentizante es la copia de datos entre la memoria principal y la memoria de la GPU; así pues, el código generado se orienta a minimizar las llamadas a funciones de copia de datos.

A través de los experimentos se demuestra que el código generado en CUDA C que implementa el modelo de SVM en la GPU es en promedio 112.71 veces más rápido que R, es 78.74 veces más rápido si se compara con la ejecución del código base generado en C. En una comparación con una herramienta como ThunderSVM, que también utiliza la GPU, la ejecución del código generado para GPU y objeto de esta investigación es en promedio 6.2 veces más rápida. El código, tal cual se genera, puede ser aplicado en diferentes modelos de GPU, no es necesario realizar cambios o adecuaciones. En adición, el crecimiento de los tiempos de ejecución no se ve afectado si los datos sobrepasan el tamaño de la memoria del dispositivo, ya que, de acuerdo con los resultados, el

crecimiento en los tiempos de respuesta es lineal con respecto a la cantidad de datos procesados.

AGRADECIMIENTOS

Este trabajo no hubiera sido posible sin el apoyo financiero otorgado a través de la beca de posgrado del Consejo Nacional de Ciencia y Tecnología de México (CONACYT) y el apoyo otorgado a través del proyecto número 291772 del fondo sectorial CONACYT-INEGI. Asimismo fue esencial el apoyo de la Universidad Autónoma de Sinaloa.

REFERENCIAS

[1] H. Yu, "Support Vector Machine," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. New York, NY: Springer New York, 2016, pp. 1–4.

[2] C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep 1995.

[3] P. Chinas, I. Lopez, J. A. Vazquez, R. Osorio, and G. Lefranc, "SVM and ANN Application to Multivariate Pattern Recognition Using Scatter Data," *IEEE Latin America Transactions*, vol. 13, no. 5, pp. 1633–1639, May 2015.

[4] L. P. D. Bosque and S. E. Garza, "Prediction of Aggressive Comments in Social Media: an Exploratory Study," *IEEE Latin America Transactions*, vol. 14, no. 7, pp. 3474–3480, July 2016.

[5] G. J. G. Uribe, D. E. R. Guevara, and P. A. R. Gallego, "Estimation of residential natural gas consumption in Medellín-Antioquia," *IEEE Latin America Transactions*, vol. 16, no. 3, pp. 819–822, March 2018.

[6] R. Eskandarpour and A. Khodaei, "Leveraging accuracy-uncertainty tradeoff in svm to achieve highly accurate outage predictions," *IEEE Transactions on Power Systems*, vol. 33, no. 1, pp. 1139–1141, Jan 2018.

[7] A. Zendehboudi, M. Baseer, and R. Saidur, "Application of support vector machine models for forecasting solar and wind energy resources: A review," *Journal of Cleaner Production*, vol. 199, pp. 272 – 285, 2018.

[8] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden Technical Debt in Machine Learning Systems," in *Proceedings of the International Conference on Neural Information Processing Systems*, vol. 2. Cambridge, MA, USA: MIT Press, 2015, pp. 2503–2511.

[9] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast Support Vector Machine Training and Classification on Graphics Processors," in *Proceedings International Conference on Machine Learning*. New York, NY, USA: ACM, 2008, pp. 104–111.

[10] C.-C. Chang and C.-J. Lin, "LIBSVM: A Library for Support Vector Machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[11] A. Carpenter, "CUSVM: A CUDA Implementation of Support Vector Classification and Regression," 01 2009.

[12] Q. Li, R. Salman, E. Test, R. Strack, and V. Kecman, "GPUSVM: A Comprehensive CUDA Based Support Vector Machine Package," *Open Computer Science*, vol. 1, no. 4, pp. 387–405, 2011.

[13] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen, "ThunderSVM: A fast SVM library on GPUs and CPUs," *The Journal of Machine Learning Research*, vol. 19, no. 1, pp. 797–801, 2018.

[14] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2018. [Online]. Available: https://www.R-project.org/

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *The Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 11 2011.

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," in *Proceedings of the International Conference and Exhibition on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 2008, pp. 16:1–16:14.

[17] NVIDIA Corporation, *CUDA C Programming Guide*, 2018, version 9.2.

[18] E. Lopez-Rojas, A. Elmir, and S. Axelsson, "PaySim: A Financial Mobile Money Simulator for Fraud Detection," in *Proceedings of the European Modeling and Simulation Symposium*. Dime University of Genoa, 2016, pp. 249–255.

[19] A. Karatzoglou, A. Smola, K. Hornik, and A. Zeileis, "kernlab – An S4 Package for Kernel Methods in R," *Journal of Statistical Software*, vol. 11, no. 9, pp. 1–20, 2004.

Oscar J. Castro-López is a PhD student in Information Science at the Universidad Autónoma de Sinaloa (Culiacán, Mexico). He received his M.Sc. degree in Sciences and Information Technologies from the Universidad Autónoma Metropolitana (Mexico City, Mexico). His main research interests are machine learning deployment, data science, compilers, and software engineering.

Inés F. Vega-López has been a professor of Computer Science at the Universidad Autonoma de Sinaloa (Culiacán, Mexico) since 2004. He received his PhD degree in Computer Science from the University of Arizona in 2004. His current research interests include high performance database systems, large-scale data analytics, and machine learning.