Low-Cost Image and Video Processing Using High-Performance Middleware in Single-Board Computers with Open Internet Standards

C. Pérez, *Member, IEEE*, M. Cleva, D. Liska, C. da Fonseca, and D. Aquino

***Abstract*— Image processing is becoming ubiquitous in many activities. This kind of systems use industry-standard libraries, such as OpenCV, and GPGPU techniques such as CUDA and OpenCL. Nowadays, these are being ported to many computing platforms, offering significant processing power even in devices with limited resources. However, the only model that is truly ubiquitous, is the web itself. Modern browsers feature quite complex internals and offer sophisticated development and profiling tools, in order to offer the best user experience. Introduction of HTML5 allows Realtime video and image manipulation, in browser space, without any plugin. In addition, Wasm (web assembly) Javascript execution engine provides fastest possible performance by means of highly customized compiler and runtime, in almost any browser, including embedded ones. This paper presents an image processing system, architected as a modular web application, using only Raspberry PIs with a compact but fast middleware server, that performs all image operations in browser space by means of web assemblies. All components, including database support, can run in a single board, providing image and video processing speeds that match, or surpass, their native compiled C counterparts on the same platform. This solution has a very low cost, that fits with emerging markets, making it ideal for LATAM scenarios.**

***Index Terms*—Computer vision, Embedded software, Image processing, WebRTC, Video processing.**

I. Introducción

La visión computacional está ganando una amplia aceptación debido a la proliferación de cámaras embebidas de bajo costo, y de bibliotecas de procesamiento de imágenes y visión computacional, disponibles con licencias permisivas de software libre. La aparición de dispositivos de cómputo muy abordables, de diversos factores de forma, desde teléfonos móviles inteligentes hasta sistemas embebidos en computadores de placa única, han acelerado la convergencia tecnológica en estos escenarios. Con los diseños tradicionales, las bibliotecas de procesamiento de imágenes se proporcionan junto a una solución compilada de cliente inteligente o pesado, sobre premisa, que debe instalarse en cada dispositivo, donde el formato ejecutable debe coincidir con el compatible en cada plataforma seleccionada.

Si bien el código nativo compilado ofrece el mejor rendimiento posible en casi todas las plataformas, especialmente en dispositivos de recursos de cómputo limitados, también suele introducir ciertas complejidades en el ciclo de vida del software. Por ejemplo, la restricción a una determinada plataforma de desarrollo, lo que condiciona a los siguientes elementos: herramientas de diseño, compiladores, herramientas de despliegue en plataformas disímiles, emuladores de dispositivos para depuración en plataformas cruzadas, sincronización de las versiones desplegadas en escenarios distribuidos, etc. Por lo tanto, sería deseable una aplicación web pura, para procesamiento de imágenes, cuyos extremos cliente y servidor sean capaces de ejecutar en sistemas embebidos de bajo costo y con recursos restringidos. Este enfoque web soluciona el despliegue en los clientes, centraliza el control del sistema, minimiza el mantenimiento del software, y reduce los requisitos del extremo cliente a un simple navegador de internet. Recientemente, los avances en el rendimiento de Javascript hicieron que tal solución fuese posible, basada puramente en estándares HTTP, permitiendo que el código Javascript supere en rendimiento a sus contrapartes de clientes inteligentes, incluyendo a binarios nativos de C o C++.

En este trabajo se propone una arquitectura distribuida cliente-servidor que es compatible, en todos sus componentes, con el computador de placa única (SBC, por sus siglas en inglés) Raspberry PI. La solución propuesta es de bajo costo, con requisitos de *instalación cero* y de *huella de software nula*, ya que no se necesita instalación previa en el extremo cliente, y cuyo extremo servidor puede alojarse junto al cliente en la misma placa física, incluyendo el gestor de base de datos. Consta de una aplicación web para Linux Raspbian (el sistema operativo oficial para el computador de placa única Raspberry PI, en adelante RPI), la cual comprende un servidor HTTP de muy baja huella de software, corriendo con la modalidad "en-proceso", que puede atender en simultáneo peticiones de páginas web y de Web API en modo REST. Sobre el mismo Raspbian se ejecuta un gestor de bases de datos relacionales que proporciona el soporte de *back-end*. Debido a su naturaleza distribuida, el sistema se puede desplegar en varias combinaciones de cliente y servidor, en forma modular. Este diseño libera los recursos del extremo servidor porque ejecuta todas las tareas de manipulación de imágenes en el espacio del cliente, el cual puede reducirse a ser sólo un navegador de internet, compatible con HTML5. Este middleware se puede ejecutar en casi cualquier factor de forma del dispositivo de cómputo, desde grandes servidores

C.A. Pérez, M.S. Cleva, D.O. Liska, C.R. da Fonseca and D. Aquino are with CINAPTIC, Centro de Investigación Aplicada en Tecnologías de Información y Comunicaciones at Universidad Tecnológica Nacional, Facultad Regional Resistencia, French 414 (3500) Resistencia, Chaco, Argentina.

Corresponding author: C.A. Pérez (e-mail: cperez@ca.frre.utn.edu.ar).

hasta computadores de placa única (de aquí en adelante, SBC, por sus iniciales en idioma inglés), debido a su código compatible con los sistemas operativos Windows, MacOS y Linux, incluyendo ediciones embebidas. También ofrece escalabilidad horizontal, ya que sus componentes pueden distribuirse físicamente en un arreglo de placas conectadas a una misma red, y permite aumentar la cantidad de placas de adquisición de imágenes o video en forma arbitraria.

El presente trabajo está estructurado de la siguiente forma: en la Sección 2 se ofrece una revisión de los trabajos previos, y se dan detalles sobre la selección de SBCs como la plataforma para el despliegue del sistema. En la Sección 3 se presenta la arquitectura propuesta. En la Sección 4 se discuten los resultados obtenidos. Por último, en la Sección 5 se presentan las conclusiones y posibles mejoras.

II. Antecedentes

A. API de Imágenes y Sistemas Embebidos

Tanto para clientes inteligentes como para aplicaciones web, el procesamiento de imágenes siempre fue una tarea demandante. Por un lado, las CPU son rápidas y tienen circuitos dedicados para proporcionar alta velocidad con datos de alta precisión, donde el factor limitante de la CPU es el ancho de banda de la memoria. Por otro lado, las GPU de escritorio cuentan con cientos, o incluso miles, de núcleos de cálculo, con un ancho de banda considerablemente mayor que las CPU. Las optimizaciones de GPU incluyen aritmética de baja precisión, por lo que la microelectrónica de cada núcleo gráfico es más simple y económica. Las bibliotecas de gráficos son abstracciones de las funciones de bajo nivel de la GPU [1]. Al exponer estas funciones como una API, se vuelven accesibles desde otros programas y procesos. El paralelismo intrínseco en las GPU las convirtió en el coprocesador ideal para tareas no gráficas que involucren algoritmos altamente paralelizables. Los primeros ejemplos de este uso se presentaron en 2003 [2]. Con el tiempo, se agregó una vía de comunicación inversa, que permite a las GPUs devolver los datos procesados a los programas y aplicaciones comunes, creando así el nuevo paradigma GPGPU (unidad de tarjeta gráfica de procesamiento general). Entre las principales API de GPGPU podemos citar OpenGL, CUDA, y OpenVino, las cuales han empezado a migrarse a sistemas embebidos.

OpenGL es una abstracción de las diferentes tarjetas de video en una sola API de código abierto, la cual es compatible, en modo experimental, con placas como las Raspberry PI.

CUDA es un framework de cómputo paralelo para desarrollar algoritmos en las tarjetas de video de la compañía NVIDIA. Debido a esta última restricción, CUDA no está disponible en placas de bajo costo como la RPI, sino en SBC de la compañía, como la Jetson TX1. Esta tarjeta fue evaluada en [3] para procesamiento de video en sistemas embebidos críticos, concluyendo que, a pesar de su potencia de cálculo, apenas podía sostener una tasa de 30 FPS para las tareas de disparidad estéreo 3D, aprendizaje profundo con CaffeNet y reconocimiento de señales viales. Su sucesor, el modelo Jetson TX2, fue evaluado en [4], comparando su rendimiento con tarjetas de escritorio Nvidia GTX 1080Ti, y gráficos integrados Intel 630, donde se pudo verificar que la solución de Intel ejecutaba seis veces más velozmente que el máximo reportado por el fabricante, y que la Nvidia TX2, por el contrario, tenía un rendimiento 50% menor al reportado. Los autores concluyeron que el software era crítico, debido que Intel proporcionaba una optimización de vectores N-dimensionales que estaba ausente en las soluciones CUDA.

Por último, OpenVINO (acrónimo de inferencia visual y optimización de redes neuronales) es un conjunto de herramientas para las GPUs integradas de los procesadores Intel Core, de 6ª generación en adelante. Basada en el SDK de Visión Computacional de la compañía, OpenVINO se introdujo en 2018, y se extendió su compatibilidad a sistemas de tipo embebidos como el Intel Movidius® Neural Compute Stick, modelos 1 y 2, y a sistemas FPGA (matrices de compuertas programables en el campo). La microelectrónica FPGA puede configurar su interconexión en tiempo de ejecución. En 2019, OpenVINO fue puesto a prueba con frameworks de aprendizaje profundo y tareas de clasificación de señales viales, utilizando FPGAs en vez de GPUs [5], donde los autores concluyen que la optimización extrema de paralelismo en FPGAs compensa sus limitados recursos. Comparada con la GPU, la FPGA obtuvo mayor eficiencia energética global, mayor rendimiento en las tareas de clasificación e igual rendimiento en las tareas de verificación. Si bien OpenCV es compatible con OpenVINO, se debe tener en cuenta que las plataformas de Intel no han proliferado en Latinoamérica como sí lo han hecho las plataformas de open hardware y software de SBC, como la RPI. A pesar de que esta última tiene considerablemente menos potencia de cálculo que los FPGA y los Neural Compute Sticks de Intel, los factores de determinantes del éxito del RPI son su amplia disponibilidad de hardware a costo relativamente contenido, y su libre disponibilidad de software.

B. Procesamiento de Imágenes en Navegadores

Con respecto a los navegadores, antes de la aparición de HTML5, el procesamiento de imágenes en línea tenía que efectuarse usando un Applet Java o un componente Adobe® Flash [6]. Con cualquiera de ellos, la instalación de un complemento de navegador era obligatoria. En 2008 se propuso HTML5, y con la introducción de nuevas APIs, los navegadores compatibles pudieron ejecutar aplicaciones web complejas, escritas sólo en JavaScript. Por ejemplo, una nueva API de audio y video permitió aceptar transmisiones en vivo desde dispositivos como cámaras u otro navegador, y una nueva API para la conversión de base64 permitió codificar o decodificar mapas de bits. Para procesar una imagen, se agregó nueva marcación de elementos de documento, por ejemplo, *canvas*, que permitieron generar mapas de bits en tiempos de ejecución. Los objetos del modelo documento (objetos DOM), pueden manipularse utilizando JavaScript puro, el cual se ejecuta en casi todos los navegadores modernos sin cambios, evitando así la instalación de complementos. HTML5 alcanzó el estado de estándar recomendado en diciembre de 2014 [7], con una rápida adopción en los principales navegadores.

En 2010 se presentó un punto de referencia preliminar utilizando HTML5 y la biblioteca ImageJ [6], utilizando tres mecanismos: un complemento ImageJ, una macro ImageJ y un código HTML5 equivalente con JavaScript. Se utilizó una imagen de 350x336 píxeles, las pruebas se ejecutaron solo en

MacOS utilizando 5 navegadores diferentes. Como resultado, el código basado en el lienzo HTML5 fue 25 veces más rápido que el código de macro ImageJ, debido a la capacidad de leer y escribir todos los píxeles en una sola llamada. Además, la ejecución de JavaScript ya estaba muy optimizada para ese entonces.

En 2011, se introdujo WebGL (biblioteca de gráficos web), que definió una API de JavaScript para representar gráficos en 3D sobre cualquier navegador web. Esta biblioteca permitía usar la aceleración de GPU por medio de un código JavaScript que llamaba a objetos nativos OpenGL u OpenGL ES (móviles).

En 2013, se presentó un entorno de ejecución independiente y sin navegador para WebGL [8], donde la salida no se representaba en un objeto lienzo (canvas) del documento HTML5, sino en una pantalla normal. El código fuente de JavaScript ejecutaba llamadas OpenGL / OpenGL ES, y el hardware subyacente mostraba las imágenes de salida directamente en la pantalla del equipo. Sin embargo, no admitía multiplicidad de lienzos, se ignoraba la salida de texto de la función JavaScript, se dejaron fuera muchas características de HTML5 y sólo se tomaron en cuenta las algunas opciones del objeto lienzo. Todas estas deficiencias se resolverían incorporando un navegador a la solución.

En 2012, se plantearon varias advertencias sobre WebGL y el rendimiento de los gráficos del navegador [9]. La velocidad máxima de fotogramas en ciertos navegadores, como Google Chrome, quedó limitada a la frecuencia de actualización del hardware, 60 FPS en la mayoría de los casos. Además, la medición de los tiempos transcurridos mediante funciones JavaScript tenía varios inconvenientes. Por ejemplo, la base de tiempo se cuantificó en 4 ms, por lo que los lapsos más cortos no son mensurables, por lo tanto, no era posible lograr una velocidad superior a 250 FPS. Los navegadores basados en Chromium cuentan con un búfer de comandos, que contiene los comandos de graficación, los cuales se ejecutan en la próxima actualización de la pantalla, lo que distorsiona los tiempos de llamada a las funciones de graficación. Para resolver esto, en [9] se sugirió el siguiente procedimiento: comenzar a dibujar los objetos, y en cada iteración agregar nuevos objetos, mientras se mide el número de fotogramas generados por segundo. Cuando esta tasa de FPS comienza a disminuir, el proceso se detiene y las cifras obtenidas darán rendimientos relativos entre navegadores y plataformas.

En 2015, se presentó una propuesta para utilizar WebGL como motor de visualización para Big Data [10]. El objetivo era permitir visualizaciones de datos interactivas y basadas en la web, sin el costo adicional de tener que instalar y ejecutar software. Sin embargo, dado que la mayor parte del cálculo del espacio de datos debe realizarse en WebGL, la operación para transferir datos desde el navegador a la GPU era bastante costosa. Se sugirió el preprocesamiento en el extremo servidor, siempre que el método para cargar de datos sea veloz en forma consistente, por ejemplo, mediante la carga de datos binarios usando un socket web HTML5.

En cuanto al rendimiento del software, el lenguaje y las técnicas de compilación son primordiales: en 2019, las ganancias de optimización esperadas son: al reescribir un algoritmo de Python en C, la ganancia es del 4700%. Por paralelización adecuada, 700%. Al optimizar la explotación de caché, 2000%. Y mediante el uso de extensiones SIMD, se pueden ejecutar hasta 16 operaciones por instrucción, produciendo un total de 62.000 veces más velocidad en comparación con el fragmento de Python interpretado tradicional [11].

En cuanto a las bibliotecas de procesamiento de imágenes, el estándar de facto es OpenCV [9], escrito en C++. La última versión es 4.1.1 [12] con 2500 algoritmos. Se ha portado a casi todas las plataformas de escritorio y móviles convencionales, así como a sistemas integrados y verticales, que van desde código nativo sistemas embebidos hasta teléfonos inteligentes comerciales. Tiene interfaces en constante evolución para OpenCL, CUDA y OpenVINO, donde la biblioteca OpenVX complementa la de OpenCV. También se ha utilizado con fines educativos, debido a su amplia documentación, amplia disponibilidad y licencia de código abierto [13] .

En 2015, se presentó SIMD.js [14], un diseño e implementación de funciones SIMD, acrónimo en inglés de “instrucción única para múltiples datos”, compatible con la web, con extensiones de lenguaje y soporte de compilador que agregó paralelismo vectorial de granularidad fina a JavaScript. Las extensiones se compararon con el rendimiento y el consumo de energía, utilizando sólo plataformas Intel x86. Con vectores SIMD de ancho 4, los ahorros de energía fueron de aproximadamente 25% a 55%. Las ganancias de rendimiento variaron desde 2x hasta 7x (ganancia super lineal). Esto proporcionó, por primera vez, portabilidad e independencia de la arquitectura para un lenguaje de alto nivel.

En 2017 se introdujo *OpenCV.js* [15]. OpenCV.js es una extensa biblioteca de JavaScript generada a partir de fuentes C / C ++. La misma fue propuesta como bloque fundacional para la plataforma Open Web al año siguiente [16]. El objetivo de Opencv.js era proporcionar velocidades de corrida casi nativas, por medio de dos modelos de ejecución altamente optimizados, ya disponibles por defecto en navegadores regulares: *asm.js* y *Wasm* (ensamblado web), entendiendo por ensamblado la mínima unidad de distribución de código binario. *Asm.js* [17] es un subconjunto de JavaScript de bajo nivel, diseñado para mejorar las velocidades de procesamiento en los navegadores. El código C ++ se compila a código intermedio usando LLVM [18], luego se efectúa una transcompilación a asm.js usando *Emscripten* [19]. La biblioteca así obtenida debe referenciarse en el código Javascript de la página web que lo utiliza, para que sus funciones se puedan llamar desde los scripts normales de la página. Sin embargo, muy recientemente, se alertó sobre el problema de analizar y compilar grandes archivos JavaScript, como OpenCV.js, que constituye una restricción de procesamiento en sistemas de escasos recursos de cómputo, especialmente en los procesadores móviles de rango medio [20]. En previsión a esto, *Wasm* se introdujo en 2017 [21], como un estándar que especifica un formato binario y una sintaxis del ensamblador correspondiente, proporcionando al motor de ejecución de JavaScript la velocidad del código nativo. El código Wasm se obtiene compilando el código asm.js usando la herramienta denominada *Binaryen*, la cual genera un código

altamente optimizado que corre en un motor de ejecución especial denominado máquina de pilas. Se ha reportado, en [22], que puede superar en rendimiento al código binario compilado convencional de C ++.

OpenCV.js materializa todas las ventajas propuestas en [9]. Dado que Wasm se ejecuta en una máquina de pila personalizada, que se ejecuta en el proceso del navegador, éste es más bien una extensión natural de JavaScript, y no un complemento externo o plugin. En 2018, se presentaron en [16] las cifras de evaluación comparativa de OpenCV.js, utilizando tres algoritmos: Canny para la detección de bordes, Cascadas de Haar para la detección de rostros, e Histograma de Gradientes como extractor de características. Las pruebas se ejecutaron en un equipo con Intel i7-377, 8GiB de memoria RAM y Ubuntu 16.04 como sistema operativo. En estos ensayos, asm.js y Wasm se desempeñaron en forma similar. Tomando como referencia el rendimiento de OpenCV compilado en C ++ igual a 1.0, las cifras obtenidas con Wasm fueron: 0.6 para Canny, 0.65 para detección de rostros y 0.58 para detección de personas. En resumen, Wasm era más rápido que el código nativo de C ++, por un significativo margen.

En 2019, se realizaron pruebas de rendimiento utilizando los navegadores Chrome, Edge, Edge Developer, Firefox y Brave, en cinco dispositivos informáticos: una PC con Windows, una placa Raspbian RPI y tres teléfonos inteligentes Android. Se utilizó una función de color a escala de grises en OpenCV.js con imágenes de 480p. Todo navegador, luego de descargar la biblioteca OpenCV.js de la red, debe evaluarla y compilarla. Se encontró que la principal demora es la evaluación del script OpenCV.js, seguida de ancho de banda y latencia de la red [23]. En dicho trabajo se introdujeron los conceptos de estado transitorio y estado estable, tomando como medida el tiempo consumido para procesar cada cuadro, durante los primeros 100 cuadros, de un flujo de video, capturados por una cámara web estándar. Como conclusión, se estableció que el navegador Brave fue el mejor para este tipo de aplicaciones, con el valor agregado que podía ejecutar en sistemas de placa única, como el Raspberry PI.

Hasta dónde llega el conocimiento de los autores de este documento, a la fecha no se ha presentado un sistema de imágenes construido en torno OpenCV.js que ejecute completamente en computadores de placa única, de código abierto, como la RPI, con estándares abiertos de internet.

C. Computador de Placa Única y la Raspberry Pi

Una computadora de placa única, o SBC, es una computadora completa en un solo circuito físico, con microprocesador, memoria de acceso aleatorio, memoria masiva, subsistemas de E / S, etc. Los modelos actuales son lo suficientemente potentes como para ejecutar SO y cargas de trabajo convencionales, por ejemplo, Hay varias propuestas para clústeres de computación de productos básicos de alto rendimiento, construidas en estos tableros [22].

Al momento de escribir este artículo, una docena de familias SBC están disponibles comercialmente. Las placas Raspberry Pi (RPI) fueron preseleccionadas para este trabajo, debido a su bajo costo, compatibilidad con Linux / Windows, amplia conectividad, comunidad de usuarios sólida y adopción mundial. RPI puede ser percibido conceptualmente como un manejador de dispositivo de tarjeta de video (driver), rodeado de un sistema operativo, lo que la transforma en un dispositivo muy adecuado para procesamiento de imágenes. Se seleccionó el modelo RPI 3B + [23], ya que el modelo RPI 4B presentaba, al momento de escribir este trabajo, varios problemas con el sistema de alimentación [24].

El RPI 3B + tiene una GPU Broadcom Videocore IV, un diseño de 12 núcleos con reloj a 400 MHz, cada núcleo puede ejecutar instrucciones independientes, admite un ancho de vector SIMD de 16 elementos y DMA. El máximo rendimiento teórico y el consumo de energía son 24 GFLOP y 7.5 vatios respectivamente. Sin embargo, las mejores cifras obtenidas en los puntos de referencia actuales son aproximadamente el 50% de ese máximo [25].

III. Un Sistema de Visión Basado en HTTP y Placas de Cómputo

A. Objetivos de Diseño

Los objetivos de diseño son los siguientes: las placas RPI deben utilizarse como dispositivos informáticos, lo que garantiza bajo costo, alto rendimiento, con bajo consumo de energía, sin piezas móviles, que acepten fuentes de poder estándar de móviles. Respecto del software, la mayor parte de la complejidad debe residir en el extremo servidor, ya que el extremo cliente debe ser uno de configuración cero y huella cero, pero ambos deben poder ejecutarse en una misma tarjeta RPI si fuese necesario. Se debe utilizar un RDBMS para almacenar las imágenes adquiridas en tarjetas de memoria microSD. El software de middleware debe ejecutarse lo más rápido posible y debe exponer una API REST estándar, lo que permite que las soluciones de terceros accedan fácilmente a las funciones del sistema. El middleware y la base de datos deben ser compatibles simultáneamente con Windows, las principales distribuciones de Linux, Apple MacOS y sistemas de placa única como el RPI. Por último, el procesamiento debe realizarse en extremo cliente y debe funcionar en navegadores móviles.

B. Justificación de las Decisiones de Diseño

Para el desarrollo de aplicaciones, se evaluaron Java y .NET Core. Ambas plataformas son muy similares en características y [24] prestaciones, pero ASP.NET Core fue seleccionado por su servidor HTTP simplificado *Kestrel*, que es compatible con Raspbian, Windows, MacOS y Linux, por tener un IDE que se puede ejecutar en todas estas plataformas, por disponer del lenguaje CSharp que es compatible con Raspbian, Windows, MacOS y Linux, por ofrecer tres bibliotecas de paralelismo que suman con doce clases paralelas, y por su capacidad de ofrecer aplicaciones de consola basadas en caracteres, así como aplicaciones web. El middleware puede atender peticiones de páginas web, de API web REST y admite comunicaciones en tiempo real con llamadas a procedimiento remoto (RPC), de servidor a clientes, a través de JavaScript mediante la biblioteca SignalR. [25]

Para el soporte back-end de RDBMS, se seleccionó PostgreSQL 9.6 debido a su estabilidad en Raspbian, compatibilidad entre diferentes plataformas, soporte robusto para datos complejos y esquema de licencia simple y abierto.

Para la biblioteca de procesamiento de imágenes en RPI, se evaluaron los siguientes escenarios, con todos los componentes corriendo en una placa única:

1. OpenCV 4 nativo con Python 3.5.3. Un enfoque de cliente inteligente tradicional y eficaz, que no cumple con las restricciones al carecer de huella de software nula.
2. OpenCV 4 nativo, que se accede desde el código administrado de .NET, corriendo en el servidor. Dado que no hay una portabilidad de OpenCV a .NET Core ARM32, las clases de interoperabilidad serían muy complejas, lo que disminuiría el rendimiento.
3. OpenCVSharp para .NET Core. Esto carece de una compatibilidad en tiempo de ejecución ARM32 adecuada, sólo existe compatibilidad con Ubuntu Linux, en las pruebas la versión de 32 bits se bloqueó sobre Raspbian.
4. OpenCV.js en el explorador HTML5 mediante asm.js o Wasm. Fue elegido porque satisface todas las restricciones. El navegador puede adquirir imágenes de webcam a través de la API WebRTC HTML5 [26] y ejecuta OpenCV.js en modo Wasm. El cliente está escrito en HTML + JavaScript, y puede llamar directamente a las funciones Wasm.

C. *Arquitectura Propuesta*

La Fig. 1 muestra un diagrama de bloques del sistema propuesto, utilizando una placa RPI. Kestrel enruta las solicitudes por tipo de recurso. Si el cliente solicita una página web, se enruta a la página Razor coincidente, si solicita un recurso de API web, se enruta al controlador que administra la persistencia, lo que admite operaciones CRUD mediante verbos HTTP. Los clientes externos también pueden acceder directamente a la base de datos, para tareas de administración o replicación. Se puede acceder a las páginas web desde cualquier navegador convencional, admitiendo navegadores en placa RPI, en PC y en teléfono móvil inteligente. El archivo OpenCV.js, de casi 10 MiB, se sirve como recurso estático. Los roles de RPI son servidor de base de datos, servidor web y controlador de API web. No se realiza ninguna manipulación de imagen en el lado del servidor. Cualquier dispositivo con navegadores convencionales puede acceder y utilizar funciones OpenCV, simplemente accediendo a una página web. No se necesitan procedimientos de compilación ni de instalación en el lado del cliente, por lo que satisface el requisito de "huella cero de software". Dado que es un sistema distribuido, todos los componentes se pueden ejecutar en una sola placa RPI, o bien se puede implementar una separación física de componentes de cliente y de servidor. Por ejemplo, un RPI puede desempeñar el rol de adquisición de imágenes, con un explorador HTML5 y una cámara web, mientras que otro RPI puede desempeñar el rol de servidor corriendo el middleware y el gestor de bases de datos relacionales, RDBMS. Incluso se puede llegar a separar el extremo servidor con un tercer dispositivo RPI que ejecute una instancia RDBMS dedicada. En tal caso, se sugiere una conexión Ethernet cableada entre la placa de middleware y la placa de base de datos, para evitar la latencia y la superposición de canales que podrían surgir en las redes Wi-Fi, especialmente en la banda de 2,4 GHz. En cualquier combinación seleccionada, PostgreSQL proporciona replicación y acceso a la interfaz de nivel de llamada desde clientes de terceros, lo que permite la integración, por ejemplo, con los principales sistemas ERP. Sin embargo, en algunos casos será deseable un enfoque de placa única, por ejemplo, en la automatización de una máquina compacta, a fin de lograr un único dispositivo de adquisición, procesamiento, transmisión y almacenamiento de imágenes. Dado que el middleware se puede ejecutar en Windows, MacOS, Linux y Raspbian, y el cliente solo comprende un navegador compatible con HTML5, se garantiza el escalado vertical y horizontal del sistema. Usando la API WebRTC, es posible soportar múltiples cámaras en un solo dispositivo RPI. Esto permite asociar cámaras seleccionadas a diferentes secuencias, que pueden ser procesadas por diferentes lienzos de la página web.

En cuanto a la seguridad, el middleware expone el tráfico HTTPS de forma predeterminada y cumple requisitos de nivel empresarial.

Las placas RPI no superan los 7 vatios a plena carga, lo que permite utilizar cargadores de batería USB de iones de litio de bajo costo, como el modelo [27], que pueden proporcionar hasta 2,5 horas de tiempo de ejecución a plena capacidad. Raspbian y puede funcionar en modo “headless” (sin pantalla), el cual aumenta la eficiencia del lado servidor porque deshabilita la interfaz de usuario y los subsistemas relacionados con el vídeo, liberando esos recursos para las tareas de middleware y base de datos.

En cuanto a las comunicaciones, se utiliza Wi-Fi. Dado que la placa RPI 3B+ tiene un adaptador Gigabit Ethernet que se halla limitado a 300 MBPS debido a compartir circuitos con USB, se comprobó que un Wi-Fi AC (5Ghz) brinda un ancho de banda similar y una conexión de buena calidad.

IV. Rendimiento Comparativo del Sistema

Las mediciones de rendimiento se presentan en la Tabla 1. Para medir los métodos de JavaScript, se utilizó la API de rendimiento de JavaScript. Para los eventos que se desencadenan antes de la carga del script, se usó un medidor de alta frecuencia, integrado en el navegador, que permite medir tiempos por debajo del milisegundo. Se seleccionaron 12 eventos por ciclo de vida de página, con cuatro combinaciones de cliente/servidor distintas que comprenden una placa RPI, un teléfono inteligente (smartphone) y un ordenador portátil.

La adquisición de imágenes se estableció en 1024x768 píxeles con profundidad de color RGB de 8 bits, utilizando una cámara web USB 2.0 MS HD 5000. La columna A muestra los valores del cliente y del servidor que se ejecutan en una sola placa RPI 3B+. La columna B muestra los valores de un smartphone (Samsung S9+ con Android 9), donde el rol del smartphone es el cliente que accede al middleware de RPI. Los eventos 1, 2 y 3 no se pudieron medir debido a la falta de perfiles de rendimiento en el navegador Android. La columna C muestra los valores para el cliente y el servidor que se ejecutan en un equipo portátil con un procesador Intel i7 7700HQ y 32 GiB RAM. La columna D muestra los valores para el equipo portátil como el cliente que accede al middleware RPI. El tamaño del archivo OpenCV.js afecta a los tiempos de carga, incluso cuando los componentes se ejecutan localmente. La evaluación de scripts fue la tarea más exigente en el navegador RPI (más de 5s). Sin embargo, puesto que la evaluación ocurre solamente una vez por la carga de la página, una configuración apropiada de la

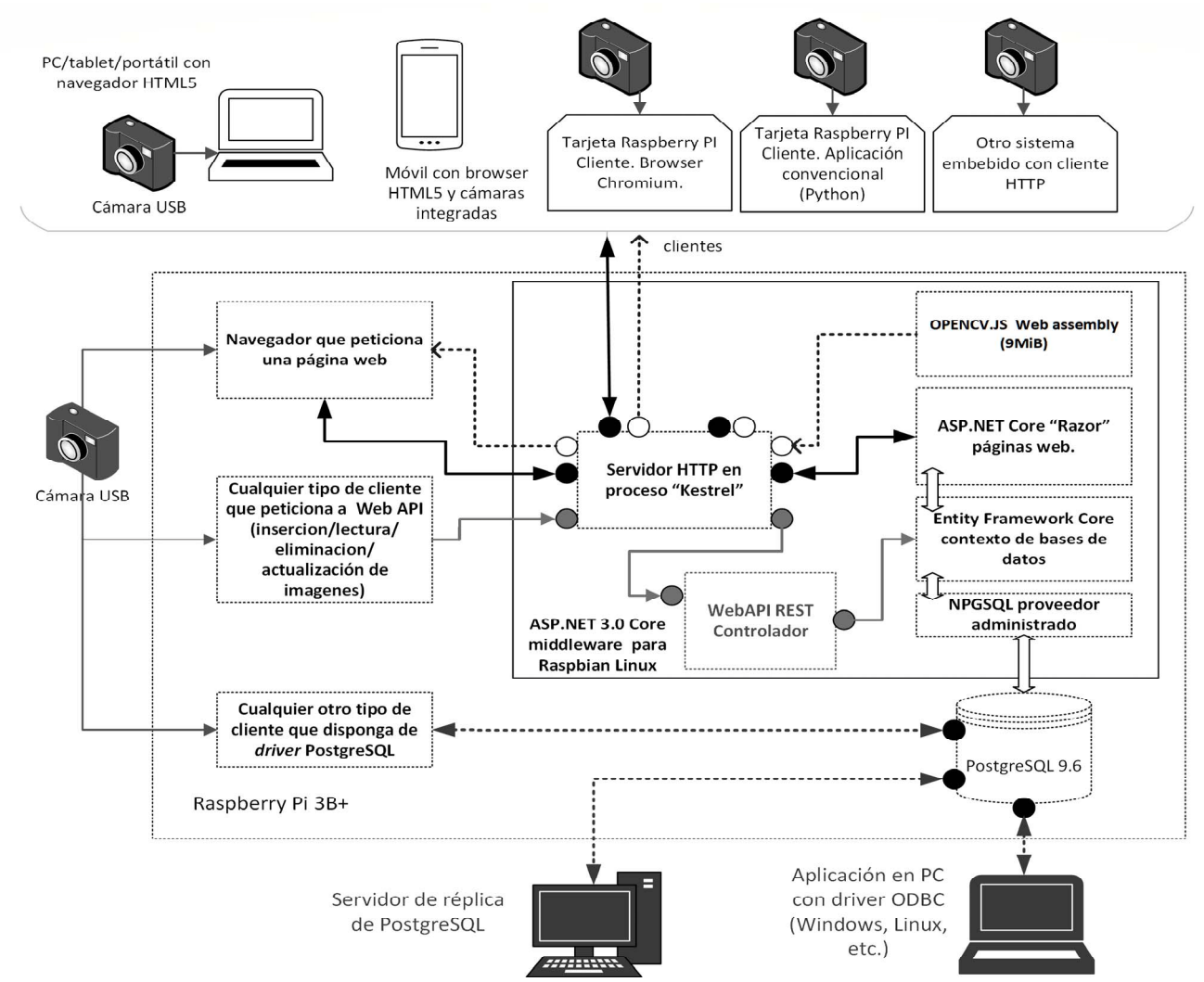

Fig. 1. Diagrama en bloques de la arquitectura (ejemplo con 1 placa RPI).

TABLA I
MEDIDAS DEL RENDIMIENTO

Evento de interés	Combinaciones de cliente y servidor			
	A. ***RPI 3B+*** ***Cliente y servidor***	***B.*** ***Cliente Android, servidor RPI 3B+***	***C.*** ***PC i7 7700HQ*** ***Cliente y servidor***	***D.*** ***Cliente PC i7 7700HQ servidor RPI 3B+***
1. Biblioteca OpenCV.js cargada desde el servidor	2800 ms	n/a	434 ms	4410 ms
2. Evaluación del script OpenCV.js una vez cargado	5250 ms	n/a	443.92 ms	321.82 ms
3. Compilación de ensamblados web OpenCV.js	139.63 ms	n/a	130 ms	19.2 ms
4. Primer fotograma capturado de la secuencia de webcam	39 ms	7 ms	14 ms	17 ms
5. Imagen de lienzo 2D a matriz OpenCV	14 ms	12 ms	2 ms	32 ms
6. 1ª adecuación y envío (marshalling) de imágenes e inserción de bases de datos	3717 ms	704 ms	2232 ms	5976 ms
7. 2ª adecuación y envío (marshalling) de imágenes e inserción de bases de datos.	800 ms	540 ms	189 ms	2230 ms
8. Asignación de matrices OpenCV.js	0 ms	0 ms	0 ms	0 ms
9. Conversión de OpenCV.js a tonos de gris	20 ms	5 ms	2 ms	1 ms
10. Representación de la imagen en grises	71 ms	23 ms	9 ms	9 ms
11. Detector de bordes Canny	157 ms	29 ms	17 ms	14 ms
12. Representación en lienzo del detector de bordes.	87 ms	24 ms	6 ms	6 ms

memoria caché del navegador, o un patrón de diseño SPA (aplicación de una sola página) puede resolver esto. Otra tarea exigente fue la primera inserción de imágenes en la base de datos, ya que la creación en memoria de la cadena de objetos de persistencia es un proceso muy costoso. La velocidad y el retardo de la red también contribuyen, en la columna D de la Tabla 1, el PC portátil rápido demoró casi 6 s. para insertar la primera imagen adquirida y 2 s. para las inserciones posteriores. Con un solo RPI, las inserciones posteriores produjeron tiempos mucho más cortos, se observó casi una ganancia de velocidad de cinco veces para la segunda imagen (col. A). Para la detección de bordes Canny, con una sola placa RPI, se obtuvieron 8 fotogramas por segundo, lo que hace que esta solución sea factible para aplicaciones industriales o de IoT que no demanden muchos cuadros por segundo.

V. CONCLUSIONES

En este artículo, se presenta una arquitectura web modular para el procesamiento de imágenes, compatible con computador de placa única. Se creó y probó un sistema distribuido de bajo costo y alto rendimiento para el procesamiento de imágenes, integrando placas Raspberry PI, middleware de código abierto más reciente, ensamblado web OpenCV.js y un RDBMS de código abierto como back-end. El sistema produjo puntuaciones que son lo suficientemente aceptables para IoT o incluso aplicaciones industriales. Dado que OpenCV.js tiene grandes ventajas como ubicuidad y configuración cero en el lado del cliente, sin pérdida de rendimiento en comparación con su homólogo de código nativo, esta arquitectura de bajo costo y alto rendimiento es ideal para proporcionar subsistemas de visión computacional en mercados emergentes. La arquitectura permite separar los roles en varias tarjetas RPI, habilitando al servidor middleware a correr el modo “headless” si fuese necesario, para utilizar la RPI como el componente principal de un sistema más complejo que demande procesamiento de imágenes. La investigación futura incluye la optimización de la evaluación de scripts en placas RPIs, utilizando un esquema de carga de scripts incremental por demanda. Y, mediante la combinación de la generación bajo demanda de ensamblados web con comunicaciones HTTP en tiempo real, utilizando web sockets HTML5 de conexión persistente, los clientes estarán siempre conectados al servidor middleware, lo que permitirá transferir las imágenes desde cámaras remotas sin latencia de la red. Este escenario puede lograr un balanceo de carga de trabajo adaptativo, entre clientes y servidores.

REFERENCIAS

[1] H. Fassold, "Computer Vision on the GPU – Tools, Algorithms and Frameworks," in *IEEE International Conference on Intelligent Engineering Systems*, Budapest, 2016.

[2] M. Harris, W. Baxter, T. Scheuermann y A. Lastra, «Simulation of cloud dynamics on graphics hardware,» de *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware - HWWS 2003*, San Diego, 2003.

[3] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, D. F. Smith, A. Berg y S. Wang, «An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads,» de *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Pittsburgh, PA, USA, 2017.

[4] T. Amert, N. Otterness, M. Yang, J. H. Anderson y D. F. Smith, «GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed,» de *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[5] Z. Lin, M. Yih, J. M. Ota, J. D. Owens y P. Muyan-Özçelik, «Benchmarking Deep Learning Frameworks and Investigating FPGA Deployment for Traffic Sign Classification and Detection,» *IEEE Transactions on Intelligent Vehicles,* vol. 4, nº 3, pp. 385-395, September 2019.

[6] K.-U. Barthel y K. Schulz, «Image processing in the browser using HTML5,» *Proceedings of the ImageJ User and Developer Conference 2010,* pp. 14-18, 2010.

[7] Pieters, Simon. Opera Software ASA., «HTML5 Differences from HTML4,» World Wide Web Consortium, 9 12 2014. [En línea]. Available: https://www.w3.org/TR/html5-diff/.

[8] N. Baek, «A Standalone WebGL Supporting Architecture,» *International Journal of Computer and Information Engineering,* vol. 7, nº 2, pp. 180-183, 2013.

[9] I. Heikkinen, «How to measure browser graphics performance,» Google, 07 2012. [En línea]. Available: https://developers.google.com/web/updates/2012/07/How-to-measure-browser-graphics-performance.

[10] A. Sarikaya y M. Gleicher, «Using WebGL as an Interactive Visualization Medium,» de *1st Workshop on Data Systems for Interactive Analysis IEEE VIS 2015*, Chicago, IL. USA., 2015.

[11] J. L. Hennesy y D. A. Patterson, «A New Golden Age for Computer Architecture,» *Communications of the ACM,* vol. 62, nº 2, pp. 48-60, February 2019.

[12] A. Aleknhin, D. Kurtaev, M. Shabunin y V. Tuzov, «OpenCV 4.1.1,» [En línea]. Available: https://opencv.org/opencv-4-1-1/. [Último acceso: 30 6 2019].

[13] A. Kozlowski y A. Królak, «Teaching image processing and pattern recognition with the Intel OpenCV library,» de *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2009*, 2009.

[14] I. Jibaja, P. Jensen, N. Hu, M. R. Haghighat, J. McCutchan, D. Gohman, S. M. Blackburn y K. S. McKinley, «Vector Parallelism in Javascript: language and compiler support for SIMD,» de *International Conference on Parallel Architecture and Compilation (PACT 2015)*, San Francisco, CA. USA., 2015.

[15] OpenCV.org, "opencv.js," [Online]. Available: https://docs.opencv.org/master/opencv.js. [Accessed 30 04 2019].

[16] A. Sajjad Taheri, N. Hu y M. R. Haghighat, «Opencv.js: Computer vision processing for the Open Web Platform,» de *Proceedings of 9th ACM Multimedia Systems Conference (MMSys'18)*, New York, 2018.

[17] D. Herman, L. Wagner and A. Zakai, "asm.js - an extraordinarily optimizable, low-level subset of JavaScript," Mozilla.org, 18 08 2014. [Online]. Available: http://asmjs.org/. [Accessed 2019 04 30].

[18] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization. CGO 2004.*, San Jose, CA. USA., 2004.

[19] A. Zakai, "Emscripten: an LLVM-to-JavaScript compiler," in *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011*, Portland, OR. USA., 2011.

[20] A. Osmani, «Javascript Start-up Optimization,» Google Corp., 29 05 2019. [En línea]. Available: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/javascript-startup-optimization/. [Último acceso: 30 07 2019].

[21] A. Haas, A. Rossberg, D. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai and J. F. Bastien, "Bringing the Web up to Speed with WebAssembly," in *ACM SIGPLAN Programming Language Design and Implementation - PLDI 2017*, Barcelona, 2017.

[22] B. Malle, N. Giuliani, P. Kieseberg y A. Holzinger, «The Need for Speed of AI Applications. Performance Comparison of Native vs. Browser-based,» 11 02 2018. [En línea]. Available: https://berndmalle.com/assets/pdfs/Malle--the_need_for_speed.pdf. [Último acceso: 20 07 2019].

[23] C. A. Perez, «Measuring Opencv.js performance with Wasm execution engine in desktop, embedded and mobile browsers,» de *Jornadas Argentinas de Informatica JAIIO 2019*, Salta, Argentina, 2019 (in press).

[24] K. Kronis y M. Uhanova, «Performance Comparison of Java EE and ASP.NET Core Technologies for WebAPI development,» *Applied Computer Systems,* pp. 37-44, 05 2018.

[25] A. Choudhry y A. Premchand, «Real Time Apps Using SignalR,» de *International Journal of Computer Trends and Technology (IJCTT)*, New Delhi, India, 2014.

[26] Web Realtime Communications Organization, "Web RTC native APIs," Google, Inc., [Online]. Available: https://webrtc.org/native-code/native-apis/. [Accessed 30 4 2019].

[27] TP Link Corporation, «Ally Series 15600 mAh High Capacity Power Bank,» TP Link Corporation, 2019. [En línea]. Available: https://www.tp-link.com/en/home-networking/mobile-accessory/tl-pb15600/. [Último acceso: 22 07 2019].

[28] S. J. Johnston, P. J. Basford, C. S. Perkins, H. Herry, P. F. Tso, D. Pezaros, R. D. Mullins, E. Yoneki, S. J. Cox y J. Singer, «Commodity single board computer clusters and their applications,» *Future Generation Computer Systems,* vol. 89, pp. 201-212, 2018.

[29] Raspberry Pi Foundation, «Raspberry Pi 3 Model B+,» [En línea]. Available: https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf. [Último acceso: 20 07 2019].

[30] R. Amadeo, «Raspberry Pi admits to faulty USB-C design on the Pi 4,» Ars Technica, 9 07 2019. [En línea]. Available: https://arstechnica.com/gadgets/2019/07/raspberry-pi-4-uses-incorrect-usb-c-design-wont-work-with-some-chargers/. [Último acceso: 25 07 2019].

[31] R. Longbottom, «Raspberry Pi 3B+ 32 bit and 64 bit Benchmarks and Stress Tests,» 09 2018. [En línea]. Available: https://www.researchgate.net/publication/327467963_Raspberry_Pi_3B_32_bit_and_64_bit_Benchmarks_and_Stress_Tests. [Último acceso: 26 07 2019].

[32] T. Deseyn, "GitHub - redhat-developer/kestrel-linux-transport: Linux Transport for Kestrel," Github, [Online]. Available: https://github.com/redhat-developer/kestrel-linux-transport. [Accessed 30 04 2019].

[33] I. Culjak, D. Abram, T. Pribanic, D. Hrvoje and M. Cifrek, "A brief introduction to OpenCV," in *2012 Proceedings of the 35th International Convention MIPRO*, Opatija, Croatia, 2012.

[34] https://docs.opencv.org/master/utils.js. [Accessed 30 04 2019].

[35] I. Culjak, D. Abram, T. Pribanic, D. Hrvoje y M. Cifrek, «A brief introduction to OpenCV,» de *2012 Proceedings of the 35th International Convention MIPRO*, Opatija, Croatia, 2012.

Carlos Alejandro Pérez. He is a fulltime Research Professor in CINAPTIC[1]. He has a degree in Electrical Mechanical Engineering and is currently VP of IEEE Computational Intelligence Society, Argentina Chapter, with interests in A.I., Business Intelligence, Control Systems, Web and Mobile development and Imagery Processing.

Mario Sergio Cleva. He is a fulltime Research Professor in CINAPTIC[1]. He has a degree in Physics and belongs to the National System of Science and Technology, with interests in Image processing, Physics and Measurement Systems by imagery.

Diego Orlando Liska. He is a fulltime Research Professor in CINAPTIC[1]. He has a degree in Electrical Mechanical Engineering, with interests in Physics, Thermodynamics and Industrial Automation and Security.

Claudio Rodrigues da Fonseca. He is a part-time Research professor in CINAPTIC. Center for Applied Research in Information and Communication Technologies.UTN Facultad Regional Resistencia. French 414, Resistencia, Province of Chaco, Argentina, , he has a degree in Information Systems Engineering, with interests in Industrial Automation, Robot Programming and Mobile development.

Dominga Concepción Aquino. She is a fulltime Research Professor in CINAPTIC[1], she has a degree in Information Systems Engineering, with interests in Theory of Control and Simulation.