Cálculo Entrenamiento con HPC

Avalos Leonel, Callapiña López Juan, Crescente Maximiliano, Zurdo Misael

[1]Universidad Nacional de La Matanza,
Departamento de Ingeniería e Investigaciones Tecnológicas,
Florencio Varela 1903 - San Justo, Argentina
leonelavalos.95@gmail.com, guillekallap@gmail.com,
maxi.crescente10@gmail.com, misaelzurdo@gmail.com

Resumen. El siguiente Trabajo de Investigación que nuestro grupo va a presentar está basado en el estudio para mejorar el tiempo de respuesta y procesamiento de cálculos que presenta nuestro proyecto actual a presentar. Este estudio se desarrolla a partir de la tecnología HPC para paralelizar la distribución de datos y lograr un óptimo reconocimiento facial por medio del sistema biométrico planteado.

Palabras claves: Sistema Embebido, HPC, OpenMP, Android, entrenamiento imágenes, CUDA.

1 Introducción

En el presente estudio se aborda el problema de la velocidad del reconocimiento de imágenes como puede ser rostros. Para solucionar esta cuestión se plantean los siguientes objetivos:

1. Encontrar un concepto o propiedad matemática que permita el reconocimiento de imágenes que, además de cumplir con las condiciones deseables de robustez a la hora del análisis, posea una estructura de operación que facilite su optimización mediante la aplicación de técnicas de paralelización de las tareas necesarias.
2. Estudiar y seleccionar las mejores tecnologías de programación que nos permitan desarrollar una herramienta aplicando la idea anterior de la forma más óptima posible.
3. Idear un algoritmo con este concepto y codificarlo con las tecnologías seleccionadas tanto de forma secuencial como paralela.
4. Generar un plan de pruebas y comparar los resultados obtenidos para confirmar que el algoritmo cumple su función de manera óptima.

Se realizan múltiples comparaciones entre el rasgo biométrico a identificar y los patrones almacenados en la base de datos. El sistema de identificación deberá establecer la identidad del sujeto sino por el contrario, indicar que no se encuentra en la base de datos. Existen múltiples tecnologías que facilitan la escritura y el desarrollo de programas capaces de realizar ejecución de procesos paralelos:

Threads: Estas técnicas se basan en la paralelización de tareas, de forma que la carga de trabajo pueda repartirse entre distintos recursos.
SIMD: La idea básica de esta técnica consiste en realizar la misma operación sobre múltiples datos de forma simultánea.
CUDA (Arquitectura Unificada de Dispositivos de Cómputo) hace referencia tanto a la plataforma de cómputo paralelo como al modelo de programación creado por NVIDIA. Esta forma de computación paralela hace uso de la GPU (Unidad de Procesamiento Gráfico) aprovechando la arquitectura multinúcleo con el objetivo de aumentar el rendimiento de cómputo. El lenguaje utilizado es una extensión del lenguaje C y C++ que permite implementar el procesamiento de tareas y datos de forma paralela.
OpenMP es una API de multiprocesamiento de memoria compartida que se seleccionó para administrar el nivel de paralelismo host tal como es:

- Multiplataforma
- Una interfaz C / C ++
- Portátil y escalable
- Utilizado por OpenCV1

OpenMP usa directivas de preprocesador para indicar bloques paralelos de código.
Dicho ésta introducción, nuestro objetivo es utilizar Viola & Jones que es un algoritmo de detección de objetos comúnmente utilizado que es bien conocido por su rendimiento de detección de rostros donde éste algoritmo extendido se implementó como parte de la biblioteca OpenCV usando tecnología HPC.

2 Desarrollo

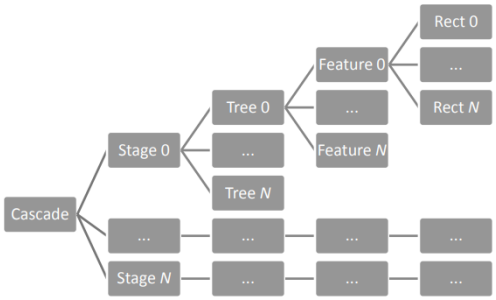
La implementación de OpenCV del detector de objetos Viola & Jones se utilizó como referencia para la implementación de CUDA y para que coincida con la precisión, se requiere una precisión doble para algunos cálculos utilizando tarjetas de mayor poder computacional. Desde un nivel alto, el flujo del programa Viola & Jones es el siguiente:

- Leer y preprocesar la imagen usando OpenCV
- Crear imágenes integrales
- Copiar imágenes integrales en el dispositivo
- Clasificar la imagen en el dispositivo
- Copiar las caras detectadas de vuelta al host y visualizar los resultados

3 Explicación del algoritmo

Datos de entrenamiento

OpenCV proporciona un conjunto de detectores preentrenamiento para ojos, caras frontales, caras de perfil, parte inferior del cuerpo, parte superior del cuerpo y todo el cuerpo en forma de archivos XML. El detector de objetos OpenCV carga el nombre del archivo para que un detector entrenado lo use para la clasificación. Este archivo se lee en el inicio de la aplicación y se usa para configurar estructuras de datos para representar la cascada que está compuesta de un conjunto de etapas, cada una con un número determinado de árboles. Los árboles están compuestos de características que contienen un número determinado de rectángulos.



Preproceso

OpenCV se usa para leer y preprocesar la imagen. La imagen se lee y se convierte en una matriz de datos en escala de grises de 8 bits por píxel. OpenCV realiza la ecualización del histograma sobre toda la imagen antes de generar la imagen integral, aunque no implementada en CUDA. Sin embargo, la ecualización del histograma es un proceso paralelizable siendo una buena implementación para CUDA.

Creación de Contexto

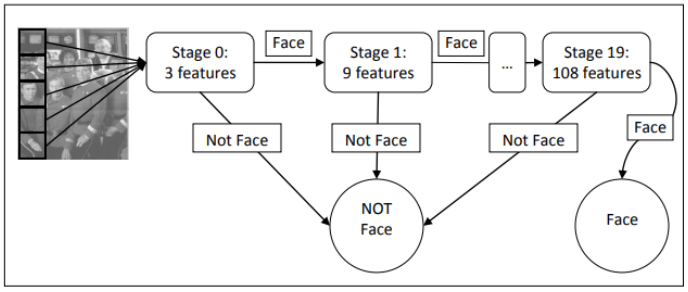
Un contexto CUDA es comparable a un proceso de CPU, rastrea todos los recursos y acciones, lo que permite a CUDA limpiarse automáticamente. Debe ser creado para cada GPU antes de que se puedan realizar llamadas CUDA en una aplicación. Para evitar que el contexto sea creado durante la llamada de clasificación, se escribió una función para crear el contexto en el lanzamiento de la aplicación.

Pseudocodigo
cascada
Etapa 0
Árbol 0
Característica 0
Rect 0
...
... Rect N
... Característica N
Árbol N
... ...
Etapa N ... ...

Como la implementación de CUDA de Viola & Jones es multi-GPU, también se requiere algo de trabajo adicional en esta función.

- Determinar la cantidad de GPU en el sistema
- Crear un hilo de host para cada dispositivo usando OpenMP
- Asociar un dispositivo CUDA con cada uno de los hilos de host
- Habilitar copia cero en cada dispositivo
- Crear un contexto CUDA para cada dispositivo bajo el hilo de host asociado

Imagen Integral



La imagen integral puede ser rápida y fácil generado con un *bucle for anidado* sobre las columnas y filas de la imagen de entrada. El rendimiento del procesamiento paralelo ofrece una aceleración mínima y en realidad causan una desaceleración cuando se trabaja con imágenes más pequeñas. Por esta razón, la imagen integral es construido en el host y luego copiado en el dispositivo. Es usada para calcular los valores de las características de detección de rostros. Como las características no son consecutivas, no es posible unir el acceso a la memoria con la imagen integral. Por lo que se la vincula a la memoria caché, aunque no admite datos de doble precisión. Para atar la imagen cuadrada integral de la memoria caché de texturas, primero debe convertirse en datos *CUDA int2 tipo*. Esto se hace mediante el uso de una unión en la que se puede establecer o leer una matriz doble o interna:

```
union doubleToint2
{
      double     d;
      signed int i[2];
};
```

Estos valores luego son convertidos a valores de doble precisión por el núcleo cuando se leen desde el caché de texturas usando las funciones intrínsecas de CUDA y se almacena la raíz cuadrada de la imagen cuadrada integral

```
static __inline__ __device__ double fetch_double(texture<int2, 2> t,
                                                 int x,
                                                 int y)
{
    int2 v = tex2D(t, x, y);
    return __hiloint2double(v.y, v.x);
}
```

Corrección Lumínica

Para proporcionar una clasificación precisa, las ventanas secundarias de prueba también deben estar normalizadas. Para una escala determinada, la desviación estándar sigue siendo la misma para una ventana secundaria en cada etapa. En lugar de calcular la desviación estándar en cada etapa, puede almacenarse en caché a la memoria global después de la primera etapa y simplemente volver a leer en etapas posteriores, como tal esta implementación tuvo un efecto insignificante en el rendimiento general.

Evaluación de árbol

La formación del detector OpenCV produce un archivo XML que describe las etapas en cascada, los árboles, las características y los umbrales. Cada etapa se compone de un conjunto de árboles, los árboles se componen de características que a su vez se componen de rectángulos.

```
<!-- root node -->
<feature>
      <rects>
            <_>2 7 16 4 -1.</_>
            <_>2 9 16 2 2.</_>
      </rects>
      <tilted>0</tilted>
</feature>
<threshold>4.3272329494357109e-003</threshold>
<left_val>0.0383819006383419</left_val>
<right_node>1</right_node>

<!-- node 1 -->
<feature>
      <rects>
            <_>8 4 3 14 -1.</_>
            <_>8 11 3 7 2.</_>
      </rects>
      <tilted>0</tilted>
</feature>
<threshold>0.0130761601030827</threshold>
<left_val>0.8965256810188294</left_val>
<right_val>0.2629314064979553</right_val>
```

Los campos rects proporcionan x, y, ancho, alto y peso para cada rectángulo en la característica. El campo inclinado especifica si la función se gira (1) o no (0). Las características se evalúan tomando la suma de los rectángulos:

$$f = \sum_{i=1}^{\text{AllRects}} \text{RecSum}(\text{rects}_i) * c_i ,$$ donde c es el peso asociado con el rectángulo.

Si se evalúa que la característica es menor que el umbral ajustado, se toma la rama izquierda; de lo contrario, se toma la rama derecha. Si se toma la rama izquierda, el campo con el prefijo "left_" es inspeccionado. Si es un valor, el valor se devuelve como el valor del árbol. Sin embargo, si se trata de un nodo, el nodo secundario (característica) se evalúa y ese resultado se devuelve como el valor del árbol. El

mismo concepto se usa si se toma la rama derecha, excepto que se inspecciona el campo con el prefijo "right_". Sin embargo, se realiza muy poco cálculo a lo largo de cada ruta de bifurcación, lo que aumenta la posibilidad de que el compilador utilice predicción de bifurcación en lugar de bifurcación.

Etapa de Evaluación

```
<!-- stage 0 -->
<trees>
      ...
</trees>
<stage_threshold>0.3506923019886017</stage_threshold>
<parent>-1</parent>
<next>-1</next>


<!-- stage 1 -->
<trees>
      ...
</trees>
<stage_threshold>3.4721779823303223</stage_threshold>
<parent>0</parent>
<next>-1</next>
```

Los valores de árbol dentro de una etapa se acumulan para compararse con el umbral de la etapa. Si la suma de la etapa es mayor que el umbral, la ventana secundaria se clasifica como una cara por la etapa y pasa a la etapa siguiente para una clasificación posterior. La implementación de CUDA procesa todas las ventanas secundarias en paralelo, por lo que se ejecuta una sola etapa de cascada a la vez y por último lanza un núcleo CUDA después de que la etapa uno esté realizado. Para ahorrar tiempo de procesamiento, OpenCV realiza dos pasadas sobre la imagen; el primero de los cuales evalúa las etapas cero y uno. El segundo paso sobre la imagen evalúa las etapas restantes en las ventanas secundarias que no fueron marcadas en el primer pase.

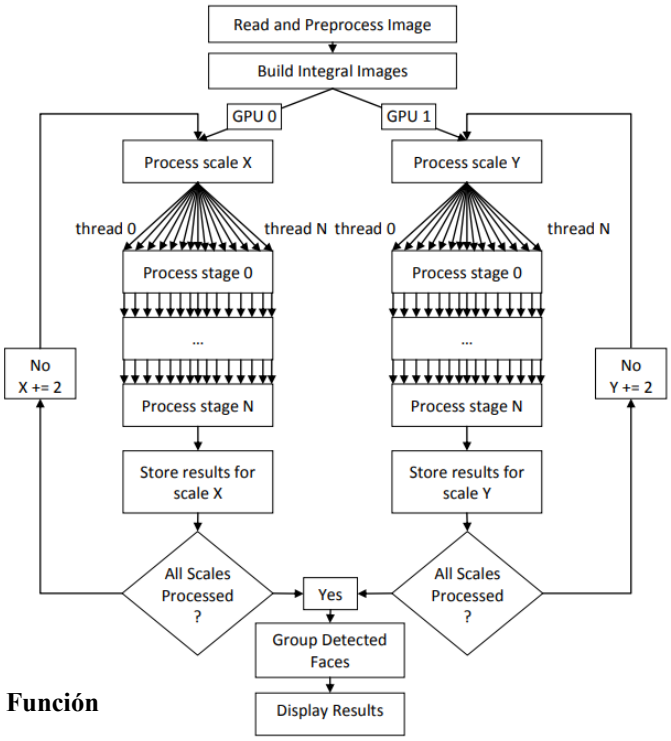
Escalado de Funciones

Luego de procesar todas las etapas en cascada, la ubicación y las dimensiones de todas las caras detectadas se almacenan. La ventana se escala, y el clasificador se vuelve a ejecutar sobre la imagen. A medida que la ventana se escala, el tamaño del paso entre ventanas también.

El redondeo imparcial simplemente redondea cualquier valor fraccionario hacia el entero par más cercano. Afortunadamente, esto simplemente escala la ubicación de la función, la altura y el ancho. Al escalar las entidades por un valor fraccionario, las coordenadas y las dimensiones se moverán a valores no enteros. Estas nuevas posiciones pueden causar un cambio en la relación de área, dando como resultado diferencias entre los datos de entrenamiento y prueba.

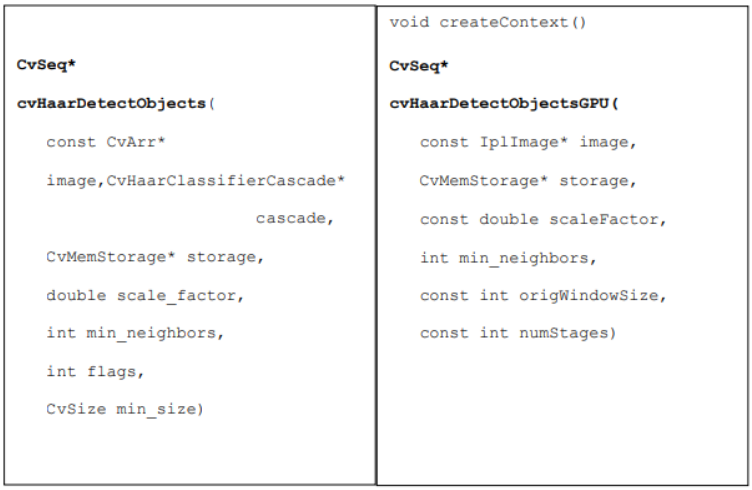
```
sum0 = 0

FOR each rectangle
     newX = origX * scale
     newW = origW * scale
     newY = origY * scale
     newH = origH * scale

     if (first rectangle)
          area0 = newW * newH
     else
          sum0 += origWeight * newW * newH

rectangle0 weight = -sum0 / area0
```

Este código se implementó como parte de los núcleos de la etapa de CUDA y se ejecuta si la escala >1. Las iteraciones de escala son independientes por lo que ofrecen un nivel adicional de paralelismo que puede ser fácilmente explotado por un sistema multi-GPU.



Llamadas a Función

El contexto debe crearse en el inicio de la aplicación a través de una llamada a createContext(). La principal diferencia en la llamada a la función se debe a que la cascada no se pasa a la función CUDA porque está "codificada" en los núcleos CUDA. Además, OpenCV permite pasar un conjunto de indicadores al clasificador.



```
CvSeq*
cvHaarDetectObjects(
   const CvArr*
   image,CvHaarClassifierCascade*
                        cascade,
   CvMemStorage* storage,
   double scale_factor,
   int min_neighbors,
   int flags,
   CvSize min_size)
```

```
void createContext()
CvSeq*
cvHaarDetectObjectsGPU(
   const IplImage* image,
   CvMemStorage* storage,
   const double scaleFactor,
   int min_neighbors,
   const int origWindowSize,
   const int numStages)
```

Llamada a la función de detección OpenCV. // Llamada a la función de detección CUDA.

Las banderas incluyen:

- CV_HAAR_DO_CANNY_PRUNING - Causa Detección de bordes caídos en la imagen antes de la clasificación
- CV_HAAR_SCALE_IMAGE - Hace que la imagen cambie de tamaño en lugar de escalar las ventanas y características.
- CV_HAAR_FIND_BIGGEST_OBJECT - Hace que la detección encuentre el objeto más grande en la imagen.
- CV_HAAR_DO_ROUGH_SEARCH - Usado junto con CV_HAAR_FIND_BIGGEST_OBJECT, hace que la función descarte al objeto candidato del tamaño más pequeño una vez que encuentra un objeto en la escala actual. Esto puede mucho disminuir el tiempo de procesamiento y también puede disminuir la precisión.

4 Pruebas que pueden realizarse

A la hora de realizar pruebas, se podrán observar mejoras tanto en la velocidad de procesamiento de imágenes debido al uso de algoritmos CUDA, la corrección lumínica, la segmentación de etapas y el escalado de funciones además del armado a través de los llamados a funciones. Nuestra aplicación guardará la imagen en la base de datos con lo cual podrá comparar el ingreso de una persona al establecimiento donde al comienzo se darán los permisos autorizados correspondiendo al criterio del

usuario en poder. Al crecer el tamaño de la base debido a las imágenes que se van registrando para el ingreso, la paralelización se encarga de la fase de entrenamiento de los mismos facilitando configuración de estructura de datos a una velocidad mayor como la que podía apreciarse anteriormente sin el uso de HPC mejorando en performance. De esta manera, optimizamos las solicitudes otorgando flexibilidad al sistema y cual sea la situación puede tener menor tasa de error debido al algoritmo empleado evitando el ingreso incorrecto a personas.

5 Conclusiones

Los algoritmos CUDA requieren la asignación de memoria del dispositivo y las copias antes de que se pueda realizar cualquier procesamiento. Para comparar las implementaciones de CPU y CUDA de manera precisa y justa, estos procesos se incluyen en el tiempo de CUDA. El contexto CUDA debe crearse antes de que se puedan realizar llamadas CUDA. Sin embargo, el contexto debe crearse solo una vez en el lanzamiento de la aplicación y no tendrá efecto en las llamadas de clasificación posteriores. Debido a esto, el tiempo de creación del contexto CUDA no está incluido en estos resultados por lo que podríamos decir que a lo largo de la ejecución mejoraría nuestro proyecto a velocidad de procesamiento y el objetivo estaría cumplido. El compilador de OpenMP ha resultado ser altamente eficiente en su tarea de paralelización, aunque también OpenCV ha proporcionado el soporte requerido de forma eficiente y sencilla. La tasa de errores que se manejaba puede ser mejorada al usar CUDA tal como se ha dicho en la explicación del algoritmo.
La biometría presenta ciertos inconvenientes y limitaciones, por ejemplo, la variación de las características biométricas a la hora de su adquisición o el rechazo social por violación a la privacidad.
Fue bastante útil el conocimiento previo a estas tecnologías durante nuestra cursada en la cátedra ya que nos facilitó el comienzo del camino a investigador tanto por el Proyecto grupal y los temas vistos como HPC que nos resultaron interesantes ya que siempre vemos que el hardware potencia a un computador, pero nunca como funcionaba por dentro y las mejoras a realizar del mismo.
Quizás un próximo trabajo de investigación a pensar o implementar sería sobre algún algoritmo que pueda ser útil para el reconocimiento de iris y añadir a este Proyecto.

6 Referencias

1. González Pérez, C.: Detección y seguimiento de objetos por colores en una plataforma Raspberry PI. In: Madrid (2016).
2. Martín, A., Serrano, M.: Aceleración con GPU de Algoritmos de matching para reconocimiento biométrico mediante patrón de iris. In: Universidad Carlos III de Madrid Link: https://e-archivo.uc3m.es/bitstream/handle/10016/26537/TFG_Alejandro_Martin-Serrano_Martin_2014.pdf?sequence=1&isAllowed=y (2014).
3. NVIDIA. CUDA Toolkit v9.2.88 [Online]. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model (2018).