Reconocimiento de imágenes utilizando GPUs, aplicacion a fútbol de robots

Ignacio Eguinoa
Facultad de Informática, UNLP

14 de febrero de 2015

Resumen

En este trabajo se desarrollan conceptos relacionados con la vision por computadoras utilizando unidades de procesamiento gráfico(GPUs). Se incluye una descripción de la libreria OpenCV, incluyendo el módulo que implementa aceleración mediante GPUs. Además, se realiza una desarrollo práctico que consiste en reimplementar una libreria para el procesamiento de imágenes provenientes de fúbol de robots. Se plantean variaciones en esta libreria que aceleran distintos pasos del procesamiento de imagenes utlizando una GPU. Las distintas variantes son evaluadas utilizando un sistema de pruebas y los resultados analizados en base a las caracterísicas de la arquitectura.

Índice general

Capítulo 1

Introduccion

1.1. Estructura del trabajo

En lo que resta de este primer capitulo se realiza una introducción a la arquitectura GPU. El objetivo es dar una idea general de las caracterísicas que posee y las posiblidades que ofrece tanto para procesamiento de gráficos como para cómputo de propósito general

En el capitulo 2 se describe de forma detallada la libreria OpenCV, principalmente los módulos y funciones que se utilizarán luego en el desarrollo.

El capitulo 3 contiene el desarrollo experimental del trabajo. En primer lugar se describe el contexto de la aplicación y la implementacion existente para el procesamiento de imágenes de fútbol robot(libreria bottracker). Luego se plantean modificaciones sobre esta librería, utilizando el módulo de GPU provisto por OpenCV. Se hacen evaluaciones de las distintas modificaciones y se analizan los resultados basandose en los conceptos explicados en los capitulos previos.

1.2. La arquitectura GPU

Capítulo 2

Libreria OpenCV

2.1. Introducción

La libreria OpenCV [1] reune una gran cantidad de algoritmos asociados a la vision por computadoras. Es una libreria open-source que se distribuye bajo una licencia BSD. is an open-source BSD-licensed library that includes several hundreds of computer vision algorithms.

La librería esta compuesta por distintos modulos que proveen funcionalidades independientes entre si. Algunos modulos comunes son:

core: Define funciones básicas utilizadas por los demás módulos y una estructura de datos que se utiliza para almacenamiento de imágenes(detallada mas adelante)

imgproc: contiene algoritmos para aplicar a imagenes (filtros, transformaciones, conversiones de colores, etc)

video: incluye algoritmos para estimar movimientos, seguimiento de objetos y para sustraer el fondo.

highgui: interface para captura de video.

El modulo core es el más importante por la utilidad de ... La principal función de la librería OpenCV es procesar imágenes, las cuales en cualquier sistema de cómputo se encuentran almacenadas como matrices numéricas. El módulo core contine una interface propia que es utilizada para el manejo de imagenes (entrada y salida) en todas las funciones de la libreria. Presenta una forma simple y segura de manejar este tipo de información y es, por lo tanto, una de las partes centrales de OpenCV.

Inicialmente, OpenCV fue implementado usando el lenguaje C y se usaban estructuras de memoria propias del lenguaje para manejar las imágenes. El problema de esto es que todo el proceso de alocar y desalocalr el espacio correspondiente se debe hacer de forma manual y depende de quien está utilizando la librería. A partir de la version 2.0 de OpenCV, el código se extendió usando el lenguaje C++ (aprovechando la compatibilidad entre ambos lenguajes) y en este proceso se introdujo una interface llamada Mat que apunta a automatizar

[1] Open Source Computer Vision Library `http://opencv.org`

todo el manejo de memoria. De esta forma, los programas que utilizaban la librería OpenCV se hacen mas simples de desarrollar y manejar, incluso para programas de gran tamaño. La mayoria de las funciones de OpenCV realizan la alocacion de memoria para el output automáticamente. Además, si el input consiste en un objeto Mat ya instanciado entonces el espacio de memoria de este es reutilizado

Mat es basicamente una clase con dos partes de datos: el encabezado de la matriz(contiene informacion tal como el tamaño de la matriz, el metodo usado para almacenarla, la direccion de memoria, etc) y un puntero a la matriz conteniendo los valores de los pixels (que puede tomar cualquier dimension, dependiendo del metodo usado para almacenar). El tamaño del header de la matriz es constante pero el tamaño de la matriz en si puede variar de imagen a imagen.

2.2. Aceleracion por paralelismo

In the past, an easy way to increase the performance of a computing device was to wait for the semiconductor processes to improve, which resulted in an increase in the clock speed of the device. When the clock speed increased, all applications got faster without the programmer modifying them or the libraries that they relied on. Unfortunately, those days are over. As transistors get denser, they also leak more current, and hence are less energy efficient. Improving energy efficiency has become an important priority. The process improvements now allow for more transistors per area, and there are two primary ways to put them to good use. The first is via parallelization: creating more identical processing units instead of making the single unit faster and more powerful. The second is via specialization: building domain-specific hardware accelerators that can perform a particular class of functions more efficiently. The concept of combining these two ideas—that is, running a CPU or CPUs together with various accelerators—is called heterogeneous parallel computing.

High-level computer-vision tasks often contain subtasks that can be run faster on special-purpose hardware architectures than on the CPU, while other subtasks are computed on the CPU. The GPU (graphics processing unit), for example, is an accelerator that is now available on every desktop computer, as well as on mobile devices such as smart phones and tablets.

Como se dijo en el capitulo previo, la funcion inicial de la gpu era renderizar imagenes a partir de escenas. Con el tiempo la generalizacion en las aplicaciones llevo a que se implementen funciones totalmente distintas. Por ej. mediante el modulo de gpu de la libreria OpenCV se estan realizando la funcion opuesta, que es entender las escenas a partir de imagenes. ((ver filmina 11 de la presentacion))

Computer vision is one of the tasks that often naturally map to GPUs. This is not a coincidence, as computer vision solves the inverse of the computer graphics problem. While graphics transforms a scene or object description to pixels, vision transforms pixels to higher-level information. GPUs contain lots of similar processing units and are very efficient in executing simple, similar subtasks such as rendering or filtering pixels. Such tasks are often known as "embarrassingly parallel," because they are so easy to parallelize efficiently on a GPU.

Many tasks, however, do not parallelize easily, as they contain serial segments where the results of the later stages depend on the results of earlier stages. These serial algorithms do not run efficiently on GPUs and are much easier to program and often run faster on CPUs. Many iterative numerical optimization algorithms and stack-based tree-search algorithms belong to that class.

Since many high-level tasks consist of both parallel and serial subtasks, the entire task can be accelerated by running some of its components on the CPU and others on the GPU. Unfortunately, this introduces two sources of inefficiency. One is synchronization: when one subtask depends on the results of another, the later stage needs to wait until the previous stage is done. The other inefficiency is the overhead of moving the data back and forth between the GPU and CPU memories—and since computer-vision tasks need to process lots of pixels, it can mean moving massive data chunks back and forth. These are the key challenges in accelerating computer-vision tasks on a system with both a CPU and GPU.

2.2.1. El modulo gpu

Dada la expansion en el uso de las arquitecturas GPU, se comenzó a implementar un modulo adicional que contiene optimizaciones realizadas sobre GPU. El modulo fue lanzado en el 2011, contiene algunos algoritmos que ya estan implementadas en diversos modulos de OpenCV y que fueron reimplementados con el fin de obtener una aceleracion extra mediante esta arquitectura.

El manejo de estructuras de datos es importante cuando interviene codigo sobre la gpu ya que la transferfencia es una parte relevante.

Por su parte, el modulo gpu define la siguiente clase:

class gpu::GpuMat Base storage class for GPU memory with reference counting. Its interface matches the Mat interface with the following limitations: no arbitrary dimensions support (only 2D) no functions that return references to their data (because references on GPU are not valid for CPU) no expression templates technique support

All GPU functions receive GpuMat as input and output arguments. This allows to invoke several GPU algorithms without downloading data. GPU module API interface is also kept similar with CPU interface where possible. So developers who are familiar with Opencv on CPU could start using GPU straightaway.

2.2.2. Transformacion de imágenes

Una de las funcionalidades mas simples para acelerar mediante el uso de paralelismo es la aplicacion de una funcion sobre cada uno de los pixeles de una imagen, generando una nueva imagen con la salida de esta funcion. Si la funcion es independiente entre los pixeles es muy simple pensar que los calculos se pueden realizar totalmente en paralelo.

Este mismo esquema se puede aplicar para operaciones que tiene como entrada pixeles correspondientes en 2 imagenes. De

Capítulo 3

Trabajo experimental

3.1. Futbol robot

El trabajo consiste en un sistema de visión por computador para el reconocimiento de objetos en un partido de Futbol de robots. El futbol de robots se puede definir como una competición de tecnología robótica de avanzada en un espacio contenido Este trabajo se acota a una categoría particular de la competencia, en la cual los equipos se componen de 5 robots controlados por un sistema centralizado. Los robots se identifican por el color de sus 'camisetas'. Cada robot debe llevar el color designado para su equipo y puede llevar otros colores para identificar los robots dentro de un equipo. No puede tener el color del equipo contrario en ningun lugar de su camiseta. Ademas, ningun robot puede tener el color caracterísico de la pelota(naranja) en su camiseta. Todo el procesamiento se realiza desde un sistema central y no se permite la intervencion de humanos a menos que el juego este detenido. El sistema central dispone de las imágenes tomadas por una unica camara central situada sobre el campo de juego. Una forma de definir el sistema de control es dividir el problema en las siguientes áreas: Reconocimiento del campo: usando la imagen de la cámara, y posiblemente información anterior, se determina la posición, velocidad y orientación de los robots de ambos equipos y de la pelota. Planificación de las acciones de los robots: Se determina las acciones a tomar con objeto de lograr el objetivo de trasladar la pelota al objetivo. Control de los robots: Se usa un sistema de comunicación inalámbrico para mover los robots de acuerdo a la estrategia definida.

3.2. Detección en tiempo real

Dentro de las etapas definidas en la seccion anterior hay diversos procesos que requieren de tiempo: latencia de la camara(desde que se toma la imagen hasta que se comienza a procesar), latencia de la deteccion, latencia de definicion de estrategia, latencia de comunicación. El objetivo es que la toma de decisiones y la ejecucion de estas se realice en un tiempo donde la imagen sobre la cual se estan tomando las decisiones sea representativa del estado actual del campo. La realidad es que el sistema está en continuo cambio, aún cuando los tiempos de latencia sean muy chicos, por lo tanto se realizan 2 aproximaciones:

-Modificar los algoritmos de toma de decisiones para tener en cuenta que el sistema ha cambiado desde que se tiene la informacion, posiblemente usando informacion anterior. -disminuir lo mas posible la latencia en todos los pasos del procesamiento, de forma tal que la imagen sea lo mas actual posible.

En un sistema de tiempo real duro, el procesamiento solo sería válido si se completan todas las etapas antes de que se capture la proxima imágen, la cual invalida el estado descrito por la imagen anterior.

3.3. Implementacion existente

El trabajo esta centrado en el primer paso del procesamiento, la etapa de reconocimiento del campo. En este paso, se recibe una imágen actual del estado del campo y a partir de esta se deben detectar las posiciones de los robots de cada equipo y de la pelota.

Para realizar esto disponemos de una libreria que permite procesar frames capturados a partir del video de un juego. La libreria permite detectar los elementos de cada cuadro (robots y pelota) devolviendo la información relevante.

El proceso de detección no se va a detallar nuevamente, se puede encontrar en detalle en Ref. [1, capitulo 5] En esta sección solo se verá en forma general los pasos y se detallan brevemente las funcionalidades que seran optimizadas usando gpu.

3.3.1. La libreria bottracker

La clase central de la libreria es bot_tracker [1]. Esta clase necesita ser instanciada y configurada con los parametros necesarios(imagen de background, colores, etc) para poder realizar todo el proceso de detección.

3.4. Implementaciones sobre GPU

Como se explicó previamente, una de las aproximaciones para que el procesamiento de imágenes se adapte a la realidad cambiante del juego de futbol es disminuir la latencia en todos los pasos que involucra este procesamiento. La aceleración de esta etapa mediante el uso de GPU tiene este objetivo. Si la optimizacion lograda mediante el uso de funciones implementadas sobre GPU no supera el tiempo extra para transferir los datos hacia/desde la memoria GPU, entonces se reducirá el tiempo total requerido para esta etapa y las decisiones que se tomaran a continuación estarán basadas en información mas actual.

La idea de esta sección es ir planteando modificaciones individuales en el algoritmo implementado usando funciones sobre CPU. El objetivo de esto es tener un conjunto de implementaciones distintas que implementen funciones independientes sobre GPU, ejecutando el resto sobre CPU. De esta forma se puede realizar una analisis en funcion de la operacion que se esta optimizando y no del contexto en el cual ocurre. Finalmente se plantea una version donde se implementan sobre GPU todas las funciones posibles (version mas optimizada) y es la que se usa para comparar la optimizacion lograda en el contexto de la deteccion de imagenes para futbol de robot.

3.4.1. Algoritmos

En las secciones anteriores se describio el codigo para la deteccion de imagenes en el cual se basa este trabajo. Hay 3 operaciones de este proceso que han sido implementadas en el modulo gpu de OpenCV. Las cuales se corresponden con el proceso de deteccion de blobs descrito en ref. [1, capitulo 5.1]. Repasemos en primer lugar cuales son estas 3 operaciones(visto en el capitulo de implementacion existente) y cuales son sus posiblidades de paralelizacion:

Conversion a escala de grises: En el caso de una imagen definida en RGB, la funcion que se debe calcular es:
$resultado(x, y) = 0{,}299 \cdot R(x, y) + 0{,}587 \cdot G(x, y) + 0{,}114 \cdot B(x, y)$

Es decir, hay que realizar una ecuacion lineal sobre los valores de R, G y B de cada pixel. Esto se realiza naturalmente con 2 for anidados

El valor resultante es, como se ve, independiente entre las posiciones de la matriz, lo que lleva a pensar que es un procesamiento muy simple de paralelizar.

Hay una version paralelizada en:

se puede ver que se llama a esta funcion 1 vez por cada columnas??? y dentro desta funcion el segundo for esta paralelizado usando la directiva #pragma unroll (linea 145) sobre un for que itera sobre el numero de filas .

La funcionalidad esta tambien implementada sobre gpu. El template utilizado para el kernel es:

Listing 3.1: gpu/include/opencv2/gpu/device/detail/transform_detail.hpp

```
1  template <typename T, typename D, typename UnOp, typename Mask>
2  __global__ static void transformSimple(const PtrStepSz<T> src, PtrStep<D> dst, const
       Mask mask, const UnOp op)
3  {
4    const int x = blockDim.x * blockIdx.x + threadIdx.x;
5    const int y = blockDim.y * blockIdx.y + threadIdx.y;
6
7    if (x < src.cols && y < src.rows && mask(y, x))
8      {
9        dst.ptr(y)[x] = op(src.ptr(y)[x]);
10     }
11 }
```

Los parámetros src y dst se corresponden con la imagen original y el resultado luego de la transformación.

La operacion aplicada sobre cada pixel(op, linea 9) es, en este caso, la funcion lineal que se vio previamente.

El kernel es llamado de tal forma que se genera un thread por cada pixel de la imagen. En las lineas 4 y 5 se definen valores de x e y para mapear cada thread con un par (x,y). Los threads que caen en valores de x mayores que el ancho de la imagen o en valores de y mayores que el largo no tendran ningun pixel sobre el cual trabajar, entonces el thread hace un return instantáneamente.

A continuación se muestra el template utilizado para generar los threads llamando al kernel anterior.

Listing 3.2: gpu/include/opencv2/gpu/device/detail/transform_detail.hpp

```
1
2  template <typename T, typename D, typename UnOp, typename Mask>
```

```
3  static void call(PtrStepSz<T> src, PtrStepSz<D> dst, UnOp op, Mask mask,
       cudaStream_t stream)
4    {
5      typedef TransformFunctorTraits<UnOp> ft;
6
7      const dim3 threads(ft::simple_block_dim_x, ft::simple_block_dim_y, 1);
8      const dim3 grid(divUp(src.cols, threads.x), divUp(src.rows, threads.y), 1);
9
10     transformSimple<T, D><<<grid, threads, 0, stream>>>(src, dst, mask, op);
11     cudaSafeCall( cudaGetLastError() );
12
13     if (stream == 0)
14         cudaSafeCall( cudaDeviceSynchronize() );
15   }
```

En las lineas 7 y 8 se definen las variables que indican el tamaño del bloque y el grid para ejecutar el kernel sobre la gpu, los cuales tienen tamaños estandar definidos por la librería.

En la linea 10 se ve la llamada al kernel que realiza la transformación.

Para utilizar esta aceleracion el paso de conversion se realiza ahora de la siguiente forma:

```
cv::gpu::cvtColor(d_frame, d_gray0, CV_BGR2GRAY);
```

Diferencia absoluta: En este paso lo que se hace es calcular la diferencia absoluta entre el valor de cada pixel del frame(en escala de grises) y el valor del pixel correspondiente del fondo(tambien convertido a escala de grises). Este proceso involucra un calculo aritmetico simple entre valores para cada pixel, y es totalmente independiente uno de otro, por lo que es totalmente paralelizable. Sin embargo, el costo de este paso (que crece con el tamaño de la imagen) no es tan significativo ya que solo se debe hacer 1 calculo simple para cada posicion de la matriz.

El proceso de paralelizacion es muy similar al caso anterior (conversion a escala de grises), pero en este caso se trabaja con 2 imagenes que se reciben por parametro. La libreria utiliza un template muy similar al anterior pero cuyo objetivo es generalizar la aplicacion de transformaciones binarias (es decir, que generan una salida a partir de 2 imagenes de entrada). Este template es:

Listing 3.3: modules/cudev/include/opencv2/cudev/grid/detail/transform.hpp

```
1  template <class SrcPtr1, class SrcPtr2, typename DstType, class BinOp, class
       MaskPtr>
2  __global__ void transformSimple(const SrcPtr1 src1, const SrcPtr2 src2, GlobPtr<
       DstType> dst, const BinOp op, const MaskPtr mask, const int rows, const int
       cols)
3    {
4      const int x = blockIdx.x * blockDim.x + threadIdx.x;
5      const int y = blockIdx.y * blockDim.y + threadIdx.y;
6
7      if (x >= cols || y >= rows || !mask(y, x))
8        return;
9
10     dst(y, x) = saturate_cast<DstType>(op(src1(y, x), src2(y, x)));
11   }
```

Para utilizar la funcion sobre gpu, el codigo queda:

```
cv::gpu::absdiff(d_gray0, d_backgroundGray, d_gray1);
```

Conversion a valores binarios: Este paso también involucra una operacion simple sobre cada pixel. Implica puntualmente realizar la comparacion del valor en cada pixel con un valor de threshold y asignarle 1 o 0 según sea mayor, menor o igual que este. La funcion que hay que calcular es:

$$resultado(x,y) = \begin{cases} 1 & if\ input(x,y) > threshold \\ 0 & if\ input(x,y) \leq threshold \end{cases}$$

```
cv::gpu::threshold(d_gray1, d_gray1, threshold, 255, CV_THRESH_BINARY);
```

Estas 3 funcionalidades reciben por parámetros imagenenes ya alocada en la memoria de la gpu, y el resultado tambien quedará almacenado en esta memoria. Si se quiere utilizar esta funcionalidad como parte del procesamiento de una imagen sobre cpu se debe primero enviar la imagen a la memoria de la gpu (metodo upload visto en el cap. 2) y luego descargar los resultados nuevamente a la memoria cpu (metodo download cap. 2).

Se debe tener en cuenta que el tiempo de procesamiento y por lo tanto el tiempo de aceleracion logrado dependen del tamaño de la imagen?? ya que una imagen mayor tendra mas posibilidades de paralelizacion si es que la arquitectura lo permite.

Todas estas propiedades se analizarán en las proximas secciones.

3.4.2. Sistema de pruebas

Para realizar las pruebas se reutilizo gran parte del programa cliente descrito en [1, capitulo 4].

Una diferencia relevante es que, cuando se procesa el video de entrada, cada frame es cargado en la gpu y enviado como parametro al modulo de procesamiento. Una vez que se hace toda la deteccion de blobs sobre gpu se descargan los resultados a la memoria de cpu

3.4.3. Resultados

3.4.4. Hardware utilizado

Capítulo 4

Conclusiones y trabajo futuro

Bibliografía

[1] Ignacio Jaureguiberry. Reconocimiento de imágenes en fútbol de robots - informe de trabajo final para tiempo real, 2011.