Aceleración de la visión por computadora utilizando GPUs, aplicación a fútbol de robots

Ignacio Eguinoa
Facultad de Informática, UNLP

Resumen

A lo largo del trabajo se recorren todos los aspectos relacionados con el uso de arquitecturas GPU para aplicaciones de visión por computadora.

En primer lugar se muestran las propiedades de esta arquitectura usada para propósito general. Luego se explican en detalle como OpenCV permite acelerar algunas de sus funcionalidades usando esta arquitectura. La misma funcionalidad se muestra desde el punto de vista del usuario, al cual la librería le permite utilizar la arquitectura de forma transparente a la implementación subyacente.

Finalmente se ejemplifica el proceso de desarrollo para aplicaciones que utilizan este tipo de arquitecturas a través de OpenCV. El desarrollo que se muestra, independientemente de los resultados logrados, permite conocer como es el procedimiento que se debe seguir para evaluar cualquier posiblidad de acelerar calculos sobre imagenes utilizando dispositivos GPU.

Índice general

Capítulo 1

Introduccion

1.1. Estructura del trabajo

En lo que resta de este primer capítulo 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.

El capítulo 2 se centra en la libreria OpenCV, donde se describe en primer lugar la estructura general y las funcionalidades que provee. Luego se describe el módulo específico que implementa funcionalidades aceleradas mediante procesadores gráficos. En la última parte del capítulo se detallan las implementaciones de algunas funcionalidades caracterísicas del módulo, conectando los conceptos vistos de la arquitectura con la interfaz de la librería que el usuario utiliza de forma transparente.

El capítulo 3 muestra el desarrollo experimental asociado al 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 el algoritmo, 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

Modern GPU accelerators has become powerful and featured enough to be capable to perform general purpose computations (GPGPU). It is a very fast growing area that generates a lot of interest from scientists, researchers and engineers that develop computationally intensive applications. Despite of difficulties reimplementing algorithms on GPU, many people are doing it to check on how fast they could be. To support such efforts, a lot of advanced languages and tool have been available such as CUDA, OpenCL, C++ AMP, debuggers, profilers and so on.

Capítulo 2

La librería OpenCV

2.1. Generalidades

OpenCV [1] es una librería open-source que se distribuye bajo licencia BSD. Reúne una gran cantidad de funcionalidades asociadas a la visión por computadoras, incluyendo estructuras para almacenamiento de datos y algoritmos de bajo y alto nivel para el procesamiento de imágenes (desde transformaciones que se aplican directamente sobre las imágenes hasta reconocimiento de rostros y seguimiento de objetos).

La librería esta compuesta por distintos módulos que proveen funcionalidades independientes entre si. Entre los más conocidos están:

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

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

Módulo video
: Incluye algoritmos que se aplican sobre elementos de video para estimar movimientos, seguimiento de objetos y para sustraer el fondo.

Módulo highgui
: Interface para captura de video. Además contiene funcionalidades para definir interfaces gráficas de usuario.

La librería esta implementada usando el lenguaje C++ y define un espacio de nombres(cv) bajo el cual se puede acceder tanto a las estructuras de datos como a las operaciones provistas por los distintos módulos. En los siguientes recuadros se muestran algunos ejemplos de funcionalidades:

Código 2.1: Módulo core

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

```
1  #include "opencv2/core/core.hpp"
2  cv::Mat img;    //Estructura de datos para imagenes
3  cv::absdiff(src1,src2,dst)  //Diferencia absoluta entre src1 y src2
4  cv::transpose(src,dst)  //Calcula la transpuesta de la matriz src
5  cv::dft(src,dst,flags, nonzeroRows) //Discrete Fourier transformation
```

Código 2.2: Módulo imgproc

```
1  #include "opencv2/imgproc/imgproc.hpp"
2  cv::threshold(src,dst,thresh,maxval,type) //aplica un umbral a los valores del input
3  cv::Canny(InputArray image, OutputArray edges,.... ) //aplica el algoritmo de
       Canny para buscar bordes en la imagen
4  cv::cvtColor(src,dst,code,...) //convierte la imagen src de un espacio de colores a
       otro
```

El módulo core es probablemente el más importante, en parte por contener los algoritmos de más bajo nivel para manipular imágenes sobre los cuales se sustentan el resto de las operaciones pero, principalmente, porque contiene las definiciones para las estructuras de datos que provee la librería para almacenar las imágenes.

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 contiene una interface propia que es utilizada para el manejo de imágenes (entrada y salida) en todas las funciones de la librería. 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 desalocar el espacio correspondiente se debe hacer de forma manual y queda en manos de quien está utilizando la librería. A partir de la version 2.0 de OpenCV, el código se extendió para usar el lenguaje C++ (aprovechando la compatibilidad entre ambos lenguajes) y en este proceso se introdujo una interface llamada Mat que apunta a automatizar todo el manejo de memoria. De esta forma, los programas que utilizaban la librería OpenCV se hacen más simples de desarrollar y manejar, incluso aquellos programas de gran tamaño. Además, la mayoría de las funciones de OpenCV realizan la alocación de memoria necesaria para el resultado de forma automática y, si la entrada consiste en un objeto Mat ya instanciado, entonces el espacio de memoria de éste es reutilizado.

Mat es básicamente una clase con dos partes de datos: el encabezado de la matriz(contiene información tal como el tamaño, el método usado para almacenarla, la dirección de memoria, etc) y un puntero a la matriz conteniendo los valores de los pixeles (que puede tomar cualquier dimensión, dependiendo del método usado para almacenar). El tamaño del encabezado es constante pero el tamaño de la matriz en si puede variar de imagen a imagen.

OpenCV utiliza un sistema para mantener el número de referencias por cada imagen, el cual permite que todo el proceso de copia y pasaje por parámetro involucre solo modificaciones en el encabezado (se debe especificar explícitamente si se quiere hacer una copia real de los datos de la matriz). Además, permite

liberar el espacio ocupado por una imagen una vez que ya no existen referencias a ésta.

Dadas estas propiedades, el manejo de imágenes es naturalmente transparente para el usuario, realizando además una manipulación eficiente de los datos.

2.2. Paralelismo y cómputo heterogéneo

Históricamente, la forma mas simple de aumentar la performance en la ejecución de un algoritmo era esperar a que avance el proceso natural de mejora en los transistores(Ley de Moore). Al cabo de un tiempo, esto resultaba en un aumento de la velocidad del reloj en los dispositivos. Cuando este aumento ocurria, las aplicaciones se acelearaban automaticamente sin necesidad de modificar el código. A medida que aumentaba la densidad de transistores también aumento la perdida de energia en forma de calor y esto puso un límite en el número de transistores que pueden integrarse eficientemente. La eficiencia energetica puso un limite en la potencia individual de los transistores de forma que un aumento en la densidad de estos no se traduce en una mejora directa de la performace en la CPU.

El proceso de mejora continuó permitiendo cada vez mas transistores por unidad de superficie, existiendo dos formas comúnes de traducir esta mejora en una *****Implementacion**** aprovechable

En primer lugar esta la paralelización, que consiste en crear mas unidades idénticas en lugar de intentar hacer unidades mas rápidas y potentes.

La otra forma de aprovecharlo es mediante especialización, es decir, construir hardware específico para realizar una cierta clase de funciones de forma más eficiente.

Ciertos campos de aplicacion, que desarrollan continuamente aplicaciones con requerimientos intensivos de computo, tienen la necesidad de obtener constantemente mayores capacidades por parte del hardware lo que lleva a combinar estas dos ideas,es decir, utilizar conjuntamente múltiples unidades de procesamiento(CPUs) a la par de dispositivos específicos de aceleración, concepto denominado cómputo paralelo heterogéneo.

Sin embargo, la evolución que existió hasta hace algún tiempo donde las aplicaciones mejoraban su rendimiento a la par del hardware sin necesidad de modificar las implementaciones ya no es algo posible y, hoy en día, es necesario adaptar los algoritmos a los nuevos modelos de cómputo y el hardware heterogéneo.

La visión por computadoras, y el procesamiento de imágenes en general, es una de las áreas que requiere constantemente mayores capacidades de computo por parte del hardware. Además, muchos de los algoritmos pertenecientes a este campo se caracterízan por ser ****embarrassingly parallel****..... .... por lo tanto son ****candidatos usuales para hacer uso**** del computo paralelo heterogeneo, haciendo uso de dispositivos con las arquitecturas altamente paralelas como las GPUs.

Como se explicó en el capitulo 1, la funcion inicial de la gpu era renderizar imagenes a partir de escenas y, con el tiempo, la gran versatilidad de estos dispositivos llevo a que se implementen funciones completamente distintas que mapeaban correctamente con la arquitectura. En la introducción de esta sección se mencionó que los algoritmos asociados a la visión por computadora eran tareas

que generalmente mapeaban de forma natural con dispositivos GPU. Esto, sin embargo, no es una conoincidencia, ya que la vision por computadora resuelve el problema inverso al cual se apuntaba especificamente al diseñar las GPUs. De esta forma, mientras el pipeline gráfico original transforma la descripción de una escena en pixeles, la visión por computadora transforma los pixeles en información útil de alto nivel.

En el caso de OpenCV, las caracteristicas intrinsecamente paralelas de algunos de sus algoritmos son explotadas utilizando dos plataformas muy conocidas para paralelismo y computo heterogeneo: -Por un lado utiliza el framework OpenCL, el cual permite escribir programas destinados a ser ejecutados sobre plataformas que contienen unidades de CPUs y otro tipo de procesadores, generalmente utilizados como aceleradores de calculo, tales como graphics processing units (GPUs), digital signal processors (DSPs) y field-programmable gate arrays (FPGAs). -Otras implementaciones utilizan la plataforma CUDA, un modelo de programacion creado por NVIDIA y solo implementado en las GPUs de su linea. Si bien parece un tanto restrictivo, esta plataforma tuvo un gran exito y ha dominado el area de programacion heterogenea en los ultimos años.

Ademas de este conjunto de funciones altamente paralelizables, muchas otras tareas asociadas al campo de la vision por computadora sin dificiles de adaptar a estas arquitecturas debido a que contienen segmentos secuenciales dependientes entre si. Estos algoritmos, por lo tanto, no se adaptan correctamente a las GPUs, no hacen un uso eficiente de esta y suelen ser mas fáciles de implementar(incluso obteniendo mejor performance) sobre CPUs.

Muchos algoritmos de alto nivel están compuestos de diversas subtareas, algunas de las cuales son facilmente paralelizables (y por lo tanto pueden ser aceleradas mediante GPUs), y otras tienen caracteristicas naturalmente secuenciales(y por lo tanto corren mas eficientemente sobre CPUs) La combinacion de componentes que corren sobre CPU y componentes que corren sobre GPU en una misma tarea global tiene al menos dos fuentes importantes de ineficiencia: Una de ellas es la sincronización: cuando una subtarea depende de los resultados de otra, ésta última debe esperar a que la otra termine para comenzar. La segunda fuente de ineficiencia corresponde al overhead asociado a la transferencia de datos ente las memoria de la CPU y la GPU. Este problema es particularmente importante en los algoritmos de visión por computadora ya que operan principalmente sobre imagenes, lo cual involucra mover grandes cantidades de datos. Estos aspectos deben ser tenidos en cuenta cuando se intenta acelerar una aplicacion mediante programacion heterogenea, ejecutando tareas sobre CPU y GPU en un mismo procedimiento.

2.2.1. El modulo gpu

Como se mencionó en la sección anterior, CUDA es la principal plataforma para programacion heterogenea, fundamentalmente por el dominio de NVIDIA en el mercado de GPUs. Es por esto que a fines de 2010, cuando se comenzo a implementar el módulo gpu de OpenCV, se utilizó esta plataforma para desarrollarlo. Junto con la gran popularidad de los dispositivos NVIDIA, la plataforma CUDA contiene una importante comunidad de usuarios, conferencias específicas, plublicaciones asociadas, y muchas herramientas y librerías desarrolladas específicamente, lo cual facilita en gran parte el avance en proyectos open-source tales como OpenCV.

En 2011 fue lanzada la primer versión del módulo, el cual contiene todas las funcionalidades aceleradas mediante GPU (reimplementaciones de algoritmos que ya se encuentraban entre los otros módulos)

Este nuevo módulo es totalmente consistente con la versión en CPU, es decir, la forma de llamar a las funciones implementadas para la nueva arquitectura es equivalente a la forma en que se utilizaban las funciones desarrolladas para CPU. Esto hace que sea muy facil de adaptar el código para utilizarlo.

Ya se ha resaltado la importancia que tiene el manejo de datos cuando se utilizan dispositivos tipo GPU para la aceleración del cómputo, por lo tanto, al desarrollar una extensión de la librería que contenga este tipo de implementaciones es necesario tener muy en cuenta como se hace el manejo de datos.

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.

In the GPU module the container cv::gpu::GpuMat stores the image data in the GPU memory and does not provide direct access to the data. If users want to modify the pixel data in the main program running on the GPU, they first need to copy the data from GpuMat to Mat.

De esta forma, el módulo GPU esta diseñado como una extension de la vista del host que tiene la API. Este diseño permite que el usuario controle explícitamente la transferencia de datos entre la memoria de CPU y GPU. Si bien esto implica tener que explicitar una pequeña porción de código adicional para ejecutar el código sobre la GPU (no se logra simplemente llamando a funciones del módulo gpu, también se deben transferir los datos), resulta en un forma flexible y eficiente de ejecutar las operaciones.

A continuación se muestra un ejemplo sencillo de portación de un código que corre exclusivamente en CPU a uno que utiliza funcionalidades del módulo GPU.

Código 2.3: Aplicar threshold a imagen en gris - CPU

```
1  #include <iostream>
2  #include "opencv2/opencv.hpp"
3
4  int main (int argc, char* argv[])
5  {
6      try
7      {
8          cv::Mat dst;
9          cv::Mat src = cv::imread("file.png", CV_LOAD_IMAGE_GRAYSCALE);
10         cv::threshold(src, dst, 128.0, 255.0, CV_THRESH_BINARY);
11         cv::imshow("Result", dst);
12         cv::waitKey();
13     }
14     catch(const cv::Exception& ex)
```

```
15      {
16          std::cout << "Error:␣" << ex.what() << std::endl;
17      }
18      return 0;
19  }
```

En las líneas 8 y 9 se declaran dos instancias de la interface Mat, dst y src, y se inicializa esta última con una imagen almacenada y leida en escala de gris.

En la linea 10 se aplica la función threshold (la misma operación se encarga de alocar el espacio necesario para la imagen resultante en la variable dst, la cual no habia sido instanciada. El resultado queda almacenado en la memoria del host, asociado a la variable dst, y puede mostrarse en la linea 11 usando la funcion imshow(esta función, correspondiente al módulo highui muestra una imagen dentro de una ventana).

A continuación se muestra como varía el mismo código para poder ejecutar la aplicación del threshold sobre una GPU.

Código 2.4: Aplicar threshold a imagen en gris - GPU

```
1   #include <iostream>
2   #include "opencv2/opencv.hpp"
3   #include "opencv2/gpu/gpu.hpp"
4
5   int main (int argc, char* argv[])
6   {
7       try
8       {
9           cv::Mat src_host = cv::imread("file.png", CV_LOAD_IMAGE_GRAYSCALE);
10          cv::gpu::GpuMat dst, src;
11          src.upload(src_host);
12          cv::gpu::threshold(src, dst, 128.0, 255.0, CV_THRESH_BINARY);
13          cv::Mat result_host = dst;
14          cv::imshow("Result", result_host);
15          cv::waitKey();
16      }
17      catch(const cv::Exception& ex)
18      {
19          std::cout << "Error:␣" << ex.what() << std::endl;
20      }
21      return 0;
22  }
```

En la línea 9 se instancia una variable conteniendo la imagen en la memoria del host. Para poder aplicar la funcion threshold utilizando la GPU primero se deben transferir los datos para que sean accesibles por ésta. En la línea 10 se declaran las variables que contendran los datos de entrada y salida en la GPU (src y dst respectivamente). En la línea 11 se indica explícitamente la transferencia de los datos desde la memoria del host a la memoria global de la GPU (funcion upload). Luego de aplicar la operación sobre la GPU(línea 12) es necesario descargar el resultado si se quiere ,por ej., mostrarlo por pantalla. Para esto existen diferentes formas, en la línea 13 del código anterior se ve que se hace simplemente la asignación del contenido de la variable en GPU a una nueva variable sobre la memoria del host(esto resulta muy transparente para el usuario pero implica un alto costo de transferencia en el caso de imagenes de tamaño considerable). Otra forma de hacerlo es mediante la funcion download, la cual puede usarse de forma sincrónica o asincrónica.

Como se ve en este ejemplo, las llamadas a ejecutar la función son equivalentes en CPU y GPU (solo cambia el espacio de nombres cv por cv::gpu). Sin embargo, se debe modificar el contexto de la llamada para asegurarse que los datos estén en el lugar correcto para ejecutar cada operación.

2.2.2. Transformacion de imágenes

Una de las funcionalidades mas simples para acelerar mediante el uso de paralelismo es la de aplicar una función sobre cada uno de los pixeles de una imagen, generando una nueva imagen con el resultado de esta función La forma secuencial de resolver esto sería recorrer todos los valores de la matriz, utilizando dos for anidados. Sin embargo, si la aplicación de la funcion es independiente entre los pixeles es muy simple pensar que los cálculos se pueden realizar totalmente en paralelo.

Usando los conceptos de CUDA y la arquitectura GPU introducidos en el capitulo 1, OpenCV implementa esta funcionalidad de la siguiente forma:

Código 2.5: 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   }
```

En el código anterior se muestra el template utilizado para el kernel:

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) depende en cada caso de la funcion que se quiera aplicar sobre el input.

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.

Código 2.6: 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.

A modo de comparación se muestra cómo el mismo codigo es implementado usando el framework OpenCL:

Código 2.7: imgproc/src/opencl/cvtcolor.cl

```
1
2   __kernel void RGB2Gray(__global const uchar * srcptr, int src_step, int src_offset, __global
        uchar * dstptr, int dst_step, int dst_offset, int rows, int cols)
3   {
4     int x = get_global_id(0);
5     int y = get_global_id(1) * PIX_PER_WI_Y;
6
7     if (x < cols)
8     {
9         int src_index = mad24(y, src_step, mad24(x, scnbytes, src_offset));
10        int dst_index = mad24(y, dst_step, mad24(x, dcnbytes, dst_offset));
11
12        #pragma unroll
13        for (int cy = 0; cy < PIX_PER_WI_Y; ++cy)
14        {
15            if (y < rows)
16            {
17              __global const DATA_TYPE* src = (__global const DATA_TYPE*)(srcptr +
                    src_index);
18              __global DATA_TYPE* dst = (__global DATA_TYPE*)(dstptr + dst_index);
19              DATA_TYPE_4 src_pix = vload4(0, src);
20  #ifdef DEPTH_5
21              dst[0] = fma(src_pix.B_COMP, 0.114f, fma(src_pix.G_COMP, 0.587f, src_pix.
                    R_COMP * 0.299f));
22  #else
23              dst[0] = (DATA_TYPE)CV_DESCALE(mad24(src_pix.B_COMP, B2Y, mad24
                    (src_pix.G_COMP, G2Y, mul24(src_pix.R_COMP, R2Y))), yuv_shift);
24  #endif
25              ++y;
26              src_index += src_step;
27              dst_index += dst_step;
28            }
29        }
30    }
31  }
```

De forma general, este código se llama 1 vez por cada columna de la matriz y, dentro de esta función, el segundo for esta paralelizado usando la directiva #pragma unroll (linea 145,primitiva para paralelizar las iteraciones) aplicada a un for que itera sobre el numero de filas .

En el ejemplo anterior la función de transformación era aplicada sobre un solo dato de entrada. El mismo esquema de paralelizacóon se puede utilizar

para operaciones que tiene como entrada pixeles correspondientes a distintas imagenes. El proceso es muy similar al caso anterior (transformación unaria), pero ahora se trabaja con dos imágenes que se reciben por parámetro.

El template utilizado ahora para el kernel es:

Código 2.8: 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   }
```

Como se ve en la línea 10, ahora la funcion (op) recibe por parámetro dos elementos de imágenes distintas.

Las funciones de transformación mostradas en esta sección son solo algunas de las funcionalidades que estan implementadas en el módulo GPU en OpenCV. Se tomaron como ejemplo en primer lugar por su simplicidad a la hora de ser paralelizadas pero además porque estas funciones abstractas de transformación generalizan la implementación de distintas operaciones que se utilizarán para el desarrollo experimental en la segunda parte de este trabajo.

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 central para futbol robot, esto es, la etapa de reconocimiento del campo. En este paso, se recibe una imagen 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 [1] que permite realizar el procesamiento de estos frames. La funcion central de la libreria recibe por parametro una imagen correspondiente al cuadro actual del estado del campo y detecta a partir de este los elementos de interes (robots y pelota) devolviendo la información relevante.

El proceso de detección no se va a explicar nuevamente pero se puede encontrar en detalle en Ref. [1, capitulo 5] En esta sección se veran en forma general los pasos del procesamiento y se detallaran las funcionalidades 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 acutalmente 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. Este es un buen algoritmo para evaluar objetivamente las ventajas de la optimizacion mediante gpu ya que es suficientemente simple para optimizar pero a la vez tiene la complejidad que brinda un codigo real en el cual parte de este tiene la posibilidad de ser ?descargado?? a la gpu y otra parte del codigo debe mantenerse en la CPU(porque no tiene sentido y/o no esta implementado), de esta forma existen distintas combinaciones de gpu/cpu que deberan ser evaluadas en cuanto a performance.

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. Este tipo de procesamiento concuerda con la transformacion unaria explicada en el capitulo 2.xxxx La libreria OpenCV optimiza la conversion a escala de grises utilizando el template para transformaciones de imagenes explicado previamente.

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.

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

En primer lugar se evaluaron todos los pasos iniciales del procesamiento (sobre los cuales se trabajo en la aceleracion) de forma individual. Dado que al trabajar con GPUs se debe tener en cuenta la transferencia desde/hacia la memoria global del dispositivo, estos movimiento de datos se consideraron como un paso mas. Las pruebas se realizaron utilizando el mismo context de ejecucion detallado en la seccion del programa cliente. Se evaluaron los valores promedio de cada operacion detallada para todos los frames del video de prueba. De esta forma se obtuvieron los siguientes tiempos promedio:

	GPU	CPU	Aceleracion
Transferencia frame(RGB) a GPU	707 μs	-	
Conversión a escala de grises	220 μs	590 μs	2.7x
Calculo de la dif. con la imagen del fondo	28 μs	197 μs	7x
Calculo de matriz binaria	50 μs	110 μs	2.2x
Transferencia matriz de valores binarios	215 μs	-	

Como se ve en la tabla, algunas de las operaciones se alcancan valores de aceleracion muy altos, lo que hace pensar que el resultado general del tiempo de ejecucion seria favorable. Sin embargo, si hacemos la suma de estos valores (son solo valores promedio) de tiempo para el algoritmo sobre GPU y el mismo sobre CPU veremos que el total es mayor cuando intentamos derivar calculos en la GPU.

Analizando un poco mas en detalle se ve que el primer paso de conversion no provee una gran mejora en el tiempo de ejecucion y sin embargo requiere que se transfiera a la gpu la imagen inicial que esta almacenada en un formato RGB (implica 3 colores...1 byte por cada color....bla bla). Si se hace este paso sobre la CPU , la transferencia de una imagen con la misma cantidad de pixeles pero en escala de grises tiene un tiempo promedio de 215 μs. Si bien en este caso la modificacion no alcanza para mejorar el tiempo correspondiente a hacer todo el calculo sobre cpu, es util para ver que la forma de analizar la aceleracion

mediante dispositivos GPU es haciendo una evaluacion de todos los tiempos asociados, tanto de computo como de transferencia. Esto no siempre es fácil, con un caso bastante simple como este donde tenemos 7 combinaciones que parecen ser razonables. A continuación se evaluan todas ellas en el contexto del algoritmo y se ven los tiempos resultantes de toda la ejecucion.

A continuación se muestra los resultados de la ejecucion del algoritmo global para las combinaciones mencionadas ....

Ejecucion completa en CPU	3026.08 μs
Haciendo la conversion a esc. grises en GPU	3342.34 μs
Haciendo solo el calculo de la dif. absoluta en GPU	3233.43 μs
Haciendo conversión a esc. de grises + dif. absoluta en GPU	3332.71 μs
Haciendo solo el calculo de matriz binaria en GPU	3318.82 μs
Haciendo el calculo de dif. absoluta + calc. de matriz binaria en GPU	3276.13 μs
Haciendo los 3 pasos sobre la GPU	3315.95 μs

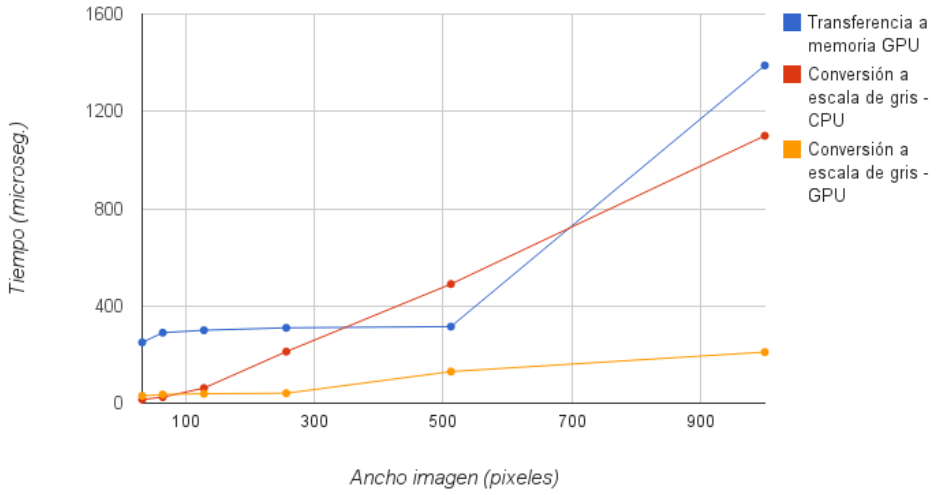
Analizando de forma general estos valores se puede ver que los tiempos en los que se hace el calculo de conversion a escala de grises (que requieren subir el frame en RGB a la gpu) son los que tienen mayor diferencia con respecto al valor de CPU(1).

Hasta acá hemos visto un ejemplo de como analizar una situacion en donde tenemos computo hibrido (cpu - gpu). Si bien el mejor analisis es haciendo las medidas correspondientes, como se menciono esto solo es posible para casos simples donde solo hay unas pocas operaciones en juego. En la realidad es mas comun hacer un analisis global de las variables que afectan positiva y negativamente la aceleracion global mediante gpu, principalmente se deben analizar el tamaño de las imagenes sobre las cuales se aplican las operaciones y cuales son las operaciones que se aplican. Una imagen de mayor tamaño implica una mayor transferencia pero en muchos casos permite una mayor capacidad de paralelizacion, principalmente en funciones que se aplican independientemente sobre todos los pixeles. En estos caso, dada la gran cantidad de unidades de procesamiento que tienen las GPUs se podrá obtener una mayor aceleracion.

Para mostrar esto se realizaron una serie de pruebas con un conjunto de imagenes de distintas

3.4.4. Hardware utilizado

AMD Opteron 6320 2.8GHz 64 Gb Memoria RAM NVIDIA Tesla M2090


1600
1200
800
400
0
Tiempo (microseg.)
100
300
500
700
900
Ancho imagen (pixeles)
Transferencia a memoria GPU
Conversión a escala de gris - CPU
Conversión a escala de gris - GPU

Capítulo 4

Conclusiones y trabajo futuro

En cuanto al desarrollo de software utilizando el modulo gpu..... Es dificil sacar una conclusion real?? acerca de la mejora en la performance que se obtiene con el modulo gpu ya que, para esto, se deberia separar el overhead relacionado con la synchronization y la transferencia de datos. Obviamente se obtendran mejoras mas relevantes para imagenes mayores y cuando mayor cantidad de procesamiento se puede hacer teniendo los datos sobre la gpu. Como principio general, como se menciona en [2] , los desarrolladores deberian probar diferentes combinaciones de procesamiento sobre CPU y GPU, medir los tiempos de procesamiento, y luego decidir cual es la combinacion que brinda la mejor performance

Bibliografía

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

[2] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Real-time computer vision with opencv. *Communications of the ACM*, 55(6): 61–69, 2012.