Optimización de algoritmos de visión por computadora utilizando GPUs, aplicación a fútbol de robots

Ignacio Eguinoa
Facultad de Informática, UNLP

Resumen

A lo largo del trabajo se recorren distintos 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 introduce la librería OpenCV y se explica en detalle como implementa la aceleración de algunas de sus funcionalidades utilizando arquitecturas altamente paralelas.

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 correctamente cualquier posiblidad de acelerar cálculos sobre imágenes utilizando dispositivos GPU.

Índice general

Capítulo 1

Introducción

1.1. Estructura del trabajo

En lo que resta de este primer capítulo se realiza una introducción a la arquitectura GPU y la plataforma de programación CUDA. El objetivo es dar una idea general de las características 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 basándose en conceptos explicados en los capítulos previos.

1.2. El procesador GPU

Originalmente, las operaciones matemáticas necesarias para graficar objetos tridimensionales eran resueltas por el procesador principal (CPU). A medida que las aplicaciones gráficas (principalmente juegos) fueron evolucionando, fue necesario delegar estas tareas a un procesador especializado para poder alcanzar rendimientos aceptables. De este modo surgieron los procesadores gráficos o GPU, presentes en cualquier placa de video actual. Estos procesadores implementan una arquitectura orientada específicamente al tipo de operaciones que deben ejecutar. Dado que la mayoría consiste en la aplicación de una simple transformación sobre una gran cantidad de píxeles o vértices tridimensionales sin interdependencias de los resultados, la arquitectura tiene, como objetivo principal, permitir una ejecución altamente paralelizada. Más adelante, con la aparición de la programabilidad de estos procesadores, se dio un gran paso evolutivo. Se hizo posible definir porciones de código que fueran ejecutadas por el GPU, para transformar de manera programática vértices y píxeles. Estos pequeños programas fueron

denominados shaders, dado que se los utilizaba principalmente para generar efectos complejos de iluminación.

Originalmente, el lenguaje de programación utilizado era primordialmente Cg (C for Graphics), desarrollado por NVIDIA. Este lenguaje ad-hoc incluía tipos de datos especializados (como color, vértice, etc.) y conceptos propios del renderizado (como las texturas). Con el tiempo fueron apareciendo nuevos lenguajes similares (GLSL, HLSL)[25], a medida que se desarrollaban nuevos y más poderosos procesadores gráficos. Con el tiempo se hizo evidente que esta programabilidad podía ser explotada para utilizar el GPU como un co-procesador matemático. Esta práctica se dio a conocer como GPGPU o General Processing on the GPU. Dado que los elementos del lenguaje estaban fuertemente orientados a aplicaciones gráficas, era necesario darles otras interpretaciones más abstractas (un color podía ser interpretado como tres variables de punto flotante, una textura como una matriz de datos, etc.). Esto era propenso a confusiones y a la vez limitaba considerablemente el desarrollo de algoritmos complejos. A la par del creciente uso de GPUs en este tipo de aplicaciones, los principales fabricantes del momento (ATI y NVIDIA) comenzaron a desarrollar entornos de programación general (es decir, no orientado a gráficos) que pudieran explotar las características del hardware. NVIDIA, por su parte, desarrolló CUDA[1] (Compute Unified Device Architecture), que comprende un driver especializado y un compilador para un lenguaje ad-hoc derivado de C. Por otro lado, ATI se basó en un framework preexistente de programación paralela llamado Brook, adaptándolo para poder utilizarlo con su línea de procesadores gráficos programables. En los ultimos años surgió en ambos fabricantes el interés en desarrollar soporte para un lenguaje orientado al cómputo intensivo conocido como OpenCL[4].

De aquí en adelante los detalles de arquitectura y del modelo de programación se refieren a la plataforma CUDA ya que es la principal herramienta para el desarrollo de funcionalidades sobre gpu que utiliza OpenCV en el módulo gpu.

1.2.1. Modelo de programación

Componentes

CUDA comprende principalmente tres componentes:

1. Un driver especializado, capaz de exponer los aspectos necesarios del hardware
2. Una biblioteca o runtime que se comunica con este driver para presentar una interfaz de alto nivel al usuario
3. Un compilador especializado (nvcc) que genera tanto el código máquina para el GPU, como el código para CPU necesario para hacer de nexo entre ambos procesadores.

El código a ejecutar en GPU se define siempre dentro de un método o kernel que puede ser invocado desde código C convencional. Estos métodos se escriben en un lenguaje C con extensiones sintácticas propias de CUDA. El compilador nvcc, al encontrar una invocación a un método de GPU (con la sintaxis apropiada), traduce este al código máquina correspondiente.

Descripción

El modelo de programación asociado a estos procesadores es de tipo SIMD: Single Instruction, Multiple Data. Es decir, el procesamiento consiste en aplicar una misma operación a múltiples datos. De hecho, NVIDIA define este modelo como SIMT: Single Instruction, Multiple Threads, dado que en realidad una misma instrucción será ejecutada en paralelo por varios threads distintos (generalmente sobre datos también distintos). Desde el punto de vista del

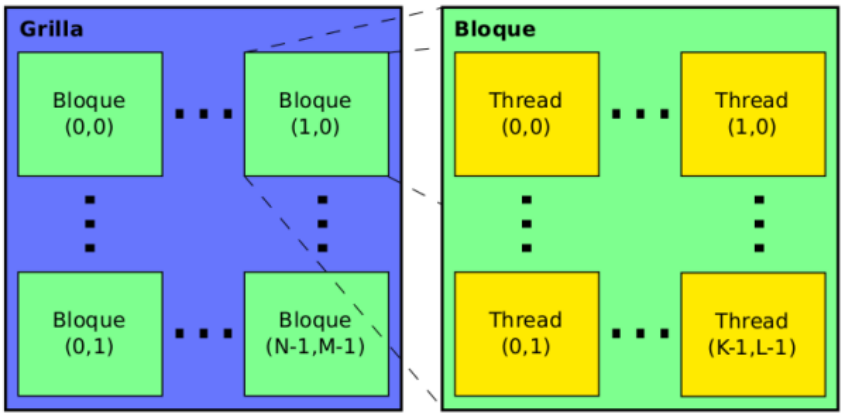

Figura 1.1: Agrupación de threads en bloques y grid

usuario, los kernels para GPU son ejecutados por múltiples threads, donde el paralelismo a nivel instrucción es resuelto por el procesador mismo. Para aprovechar las ca- racterísticas del hardware (que se detallarán más adelante), es necesario definir los threads en forma de grilla (uni o bi-dimensional), y agrupados en conjuntos de igual tamaño, llamados bloques. De este mo- do, cada thread tendrá asociado un identificador blockIdx, threadIdx, donde blockIdx corresponde al índice del bloque al cual pertenece el thread, y threadIdx al número de thread dentro del bloque mismo (figura 1.1).

CUDA extiende el lenguaje C permitiendo al programador definir funciones especiales, llamadas kernels, que, cuando se invocan, son ejecutadas N veces en paralelo sobre N threads CUDA diferentes. Un kernel se define usando el especificador __global__ en la definición y el número de threads(junto con la agrupación en la grilla) se especifican para cada invocación del kernel usando una nueva sintaxis de configuracion de la ejecución («¡...»¿) .

En el código .... se muestra un ejemplo de definición e invocación de un kernel, el cual suma los elementos de dos vectores A y B, almancenando el resultado en un vector C.

Código 1.1: imgproc/src/opencl/cvtcolor.cl

```
1  // Kernel definition
2  __global__ void VecAdd(float* A, float* B, float* C)
3  {
4    int i = threadIdx.x;
5    C[i] = A[i] + B[i];
6  }
7
8
9  int main()
10 { ...
11   // Kernel invocation with N threads
12   VecAdd<<<1, N>>>(A, B, C);
13   ...
14 }
```

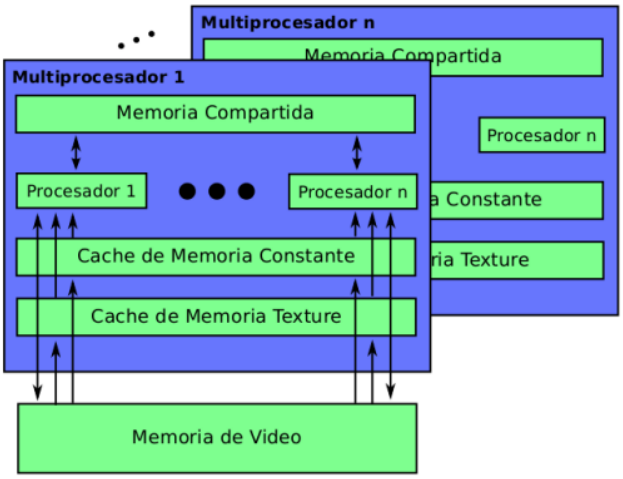

Figura 1.2: Esquema de la arquitecura de una GPU

1.2.2. Arquitectura

El GPU comprende una serie de multiprocesadores (o stream processors), con capacidad de direc- cionar a un espacio de memoria global (la memoria de video o device memory). Cada multiprocesador tiene su propia memoria compartida (shared memory), que puede ser utilizada para intercomunicar los procesadores que lo conforman. La velocidad de la memoria compartida es comparable a la de un registro de procesador, mientras que los accesos a memoria global involucran cientos de ciclos. Dado que el mismo programa se aplica numerosas veces sobre distintos datos, no hay necesidad de un control de flujo optimizado (predicción de saltos, etc). No existe tampoco una caché asociada a la memoria global, por lo que se favorece el código con gran intensidad aritmética (en contraste con la cantidad de operaciones de memoria). En realidad, la memoria de video se encuentra segmentada. Además de la memoria global, existen otros segmentos tales como la memoria constante y la de textura (espacios solo de lectura, con cachés asociadas). Sin embargo, estos segmentos tienen restricciones de tamaño y direccionamiento con lo que no es posible prescindir de la memoria global para la mayoría de las transferencias.

Cada thread será ejecutado por un procesador, con su propio instruction pointer y estado de registros. La asignación entre threads y procesadores está dada por la configuración de la llamada al método. Los bloques ejecutan siempre sobre un mismo multiprocesador, de modo de poder permitir la intercomu- nicación de los threads contenidos en él, a través del uso de la memoria compartida. Cada bloque es dividido por el scheduler en subconjuntos de dieciséis threads llamados warps (que, a su vez, se divide en dos half-warps). Cuando un warp comienza a ejecutar, el puntero a instrucción coincide para todos los threads del mismo. Mientras los threads no diverjan en su comportamiento (por ejemplo, condi- cionando la ejecución de una operación según el número de thread), el puntero avanza para todos los threads por igual. Los threads divergentes deben ser serializados, reduciendo así su performance. Al momento de volver a converger en su ejecución, el puntero vuelve a avanzar para todos los threads simultáneamente.

Para arbitrar el acceso a la memoria compartida, se implementa un sistema de sincroniza- ción por barriers. Cuando un procesador ejecuta una determinada instrucción de sincronización

(__syncthreads()) para un thread dado, este último no continuará ejecutando hasta que todos los otros threads del bloque hayan pasado por esta misma instrucción. La sincronización solo es posible a nivel bloque. Para lanzar una ejecución de código GPU, los datos sobre los cuales se operarán deben estar disponibles en la memoria de video (el procesador no puede operar sobre la memoria principal, o memoria del host). La transferencia entre memoria del host y la memoria de video puede hacerse mediante DMA (el usuario solo debe ejecutar un método similar a memcpy).

1.2.3. Aspectos de performance

Existen varios aspectos que deben ser tenidos en cuenta para lograr un código eficiente. No es trivial tener en cuenta todos los aspectos simultáneamente debido a que estos se afectan mutuamente. En general, solo es posible intentar por prueba y error distintas variaciones en la implementación que prioricen un aspecto u otro por vez. Además, puede ocurrir que no sea posible extraer tanto paralelismo como podría ser aprovechado por la arquitectura, por un límite intrínseco del problema a resolver. Algunos de los aspectos mas relevantes son:

Accesos a memoria

Dado que los accesos a memoria son considerablemente más costosos que las operaciones aritméticas, es necesario minimizar su utilización y, a la vez, hacerlo de forma eficiente. Para lograr esta efi- ciencia se pueden agrupar un conjunto de accesos a memoria en una sola transacción, esto ocurre automáticamente siempre que se cumplan ciertas caracterísicas en el patrón de acceso a memoria En general, los accesos deben ser a posiciones contiguas de la memoria pero los requisitos puntuales dependen de la versión de la arquitectura(compute capability).

Más allá del patrón de acceso a memoria, es posible esconder la latencia asociada a la memoria global si existen suficientes operaciones aritméticas independientes que pueden ser ejecutadas mientras se espera que se complete el acceso mismo.

Control de flujo

Además de las operaciones aritméticas y de memoria, hay que tener en cuenta a las instrucciones de control de flujo (es decir, los ifs, while, etc.). En esta arquitectura, un salto o branch puede ser parti- cularmente costoso en el caso en donde distintos threads de un mismo warp tomen caminos distintos (divergent branches, según la documentación). En estos casos el compilador debe serializar las instruc- ciones, lo cual puede tener un impacto importante en la performance. Para el caso de los ciclos for, el compilador es capaz de evaluar la posibilidad (o incluso, se lo puede forzar mediante una directiva especial) de hacer loop unrolling o desenrollamiento de ciclos. Al eliminar o reducir el impacto de la condición de fin de ciclo, es posible mejorar el rendimiento considerablemente. En cuanto a los if s, puede ser beneficioso reemplazar la ejecución condicional de ciertas operaciones aritméticas con una ejecución incondicional, de modo que la contribución de los casos indeseados al resultado final sea neutra.

Tamaño de bloques

Existen varios factores que determinan cuántos threads por bloque conviene utilizar. Cómo mínimo, es necesario tener tantos bloques como multiprocesadores. En caso contrario, habría multiprocesadores ociosos. Sin embargo, es deseable que haya al menos dos bloques activos por multiprocesador. Esto permite que las latencias asociadas a la sincronización por barriers y a los

accesos a memoria de video puedan ser escondidas mediante la ejecución de otro bloque. Para lograr esto es necesario no solo que haya el doble de bloques que de multiprocesadores, sino que la cantidad de memoria compartida y de registros necesarios por bloque sea lo suficientemente baja. Además, el tamaño de los bloques debería ser elegido de modo que sea múltiplo del tamaño de un warp, para evitar warps incompletos. Para bloques de al menos 64 threads, el compilador es capaz de ma- nejar eficientemente los bancos de memoria de registros de modo de esconder las latencias asociadas a dependencias lectura-escritura. En general, más threads por bloque permite un manejo más eficiente de los tiempos asignados a cada warp. El problema es que, a mayor tamaño de bloque habrá mas recursos utilizados por thread. El balance de estos factores en general puede ser obtenido por experimentación con distintos tamaños. Sabiendo la cantidad de registros y de memoria compartida por thread que un determinado kernel requiere, es posible determinar el tamaño de bloque que maximice la cantidad de warps activos por multiprocesador en relación a la cantidad máxima de warps activos en general (que estará dada por la cantidad de multiprocesadores, registros y memoria disponible, entre otros factores). Esta medida se denomina occupancy. Sin embargo, no necesariamente una mayor occupancy implica mejor rendimiento, dado que siempre se estará sujeto a las particularidades del código en cuestión.

Memoria local y registros

La memoria local corresponde al espacio asignado para las variables estáticas declaradas directa- mente en el código de un kernel. Dicho espacio es en realidad un segmento de la memoria global, con lo que la latencia asociada a su acceso es la misma. Sin embargo, dado que una variable declarada den- tro de un kernel (y que no será asignada a un registro, sino a memoria local) resulta en una dirección de memoria distinta para cada thread, los requerimientos de acceso eficiente a memoria se cumplen automáticamente.

Si bien en general, el acceso a registros no implica un costo extra en ciclos de procesador, en casos de dependencias lectura-escritura que el compilador no pueda resolver, o que no puedan ser escondidas mediante la ejecución de otros threads, esto puede no ser así. Sin embargo, si existen al menos 192 threads activos por multiprocesador, estas latencias pueden ser ignoradas. La cantidad total de registros disponibles por multiprocesador es un recurso que se divide por cada thread, y se asigna durante la compilación del código correspondiente. Como fue mencionado, redu- ciendo el tamaño de los bloques es posible disminuir la cantidad de registros necesarios para la eje- cución de cierto kernel, en caso de que se esté excediendo el límite físico disponible por el hardware utilizado. Sin embargo, la eficiencia en el uso de registros estará sujeta a las decisiones realizadas por el compilador. Por esta razón, puede resultar beneficioso reescribir el kernel de alguna forma equivalente, con el objetivo de disminuir la cantidad de registros requeridos por thread y posiblemente incrementar el índice de occupancy. En general, mediante una directiva al compilador, es posible limitar explícita- mente la cantidad de registros que un kernel puede usar por thread. Sin embargo, el efecto de este límite será el de alojar variables locales a la memoria local por lo que, en la mayoría de los casos, esto tendrá un impacto negativo en el rendimiento.

Explotación del paralelismo

En general, lo más importante es exponer correctamente el paralelismo del problema a resolver. Es posible que existan distintos algoritmos paralelizados, con distintas ventajas y desventajas. Compartir datos comunes a través de la memoria compartida suele ser deseable. Sin embargo, hay que tener en cuenta que en ciertas situaciones, debido al inmenso poder de cómputo de estos procesadores, recompu- tar datos en cada thread en vez de cargarlos

de memoria puede dar mejores resultados de rendimiento. El paralelismo también puede ser explotado a nivel host, haciendo uso de streams. Un stream co- rresponde a un pipeline de kernels que se aplican en serie a un flujo de datos de entrada. Si se tienen varios de estos pipelines, es posible hacer un uso más eficiente de los tiempos muertos intercalando la ejecución de los distintos kernels. Este esquema corresponde justamente a una arquitectura clásica tipo pipe & filter. Por último, si se hacen uso de threads a nivel sistema, podría ser posible utilizar el CPU mientras el GPU se encuentra computando. Sin embargo, muchos kernels suelen correr muy rápidamente, y hacer uso del CPU en esos casos podría introducir latencias adicionales.

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 más adelante)

Módulo imgproc
: Contiene algoritmos para aplicar a imágenes (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  #include "opencv2/core/core.hpp"
2  cv::Mat img;    //Estructura de datos para imagenes
```

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

```
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 ocurría, las aplicaciones se aceleraban automáticamente sin necesidad de modificar el código. A medida que aumentaba la densidad de transistores también aumentó la pérdida de energía en forma de calor y esto puso un límite en el número de transistores que pueden integrarse eficientemente. La eficiencia energética puso un límite 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 actualmente dos formas comunes de traducir esta mejora en una implementación aprovechable

En primer lugar está 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 aplicación, que desarrollan continuamente aplicaciones con requerimientos intensivos de cómputo, tienen la necesidad de obtener constantemente mayores capacidades por parte del hardware, lo que lleva a combinar estas dos ideas. Es decir, se utilizan 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 para utilizar eficientemente 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 cómputo por parte del hardware. Además, muchos de los algoritmos pertenecientes a este campo se caracterízan por ser vergonzosamente paralelos (traducido literalmente de embarrassingly parallel) y, por lo tanto, son candidatos usuales para hacer uso del cómputo paralelo heterogéneo.

Como se explicó en el capitulo 1, la función inicial de la gpu era renderizar imágenes 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 visión por computadoras 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 características intrínsecamente paralelas de algunos de sus algoritmos son explotadas utilizando dos plataformas muy conocidas para paralelismo y cómputo heterogéneo:

Por un lado utiliza el framework OpenCL, un estándar abierto que permite escribir programas destinados a ejecutarse sobre plataformas heterogéneas, conteniendo unidades de CPUs y otro tipo de procesadores. Entre estos se encuentran distintas unidades utilizadas como aceleradores de cálculo, tales como graphics processing units (GPUs), digital signal processors (DSPs) y field-programmable gate arrays (FPGAs).

Otras implementaciones utilizan la plataforma CUDA que, como se vió, es un modelo de programación creado por NVIDIA y solo implementado en las GPUs de su línea. Si bien parece un tanto restrictivo, esta plataforma tuvo un gran éxito y ha dominado el área de programación heterogénea en los ultimos años.

Además de este conjunto de funciones altamente paralelizables, muchas otras tareas asociadas al campo de la visión por computadora son dificiles de adaptar a arquitecturas de este tipo 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 ésta y suelen ser mas fáciles de implementar (incluso obteniendo mejor performance) sobre CPUs.

Muchos algoritmos de alto nivel están compuestos por diversas subtareas, algunas de las cuales son fácilmente paralelizables (y por lo tanto pueden ser aceleradas mediante GPUs), y otras tienen características naturalmente secuenciales(y por lo tanto corren mas eficientemente sobre CPUs) La combinación 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 imágenes, lo cual involucra mover grandes cantidades de datos. Estos aspectos deben ser tenidos en cuenta cuando se intenta acelerar una aplicación mediante programación heterogénea, ejecutando tareas sobre CPU y GPU en un mismo procedimiento.

2.2.1. El módulo gpu

Como se mencionó en la sección anterior, CUDA es la principal plataforma para programación heterogénea, fundamentalmente por el dominio de NVIDIA en el mercado de GPUs. Es por esto que, a fines de 2010 cuando se comenzó 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 fácil de adaptar el código para utilizar el nuevo módulo.

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 módulo gpu define la clase gpu::GpuMat que provee la base para manejar la memoria de la GPU. Es similar a interface Mat descrita previamente pero tiene algunas limitaciones: principlamente se diferencian porque no permite estructuras de dimiensiones arbitrarias(solo en 2 dimensiones) y no permite funciones que devuelvan referencias a datos, ya que las referencias a memoria GPU no son válidas en CPU.

Todas las funciones implementadas sobre GPU reciben parámetros de entrada y salida que son instancias de GpuMat. De esta forma, se puede invocar varias funciones sobre GPU secuencialmente sin necesidad de transferir datos, reduciendo así el overhead de transferencia. Además de esto, la API para el manejo de funciones sobre GPU se mantuvo lo mas similar posible a la existente para CPU, de manera que la programación se hace muy simple para

quienes ya conocen la libería.

Por último, el contenedor GpuMat almacena los datos de la imagen en la memoria GPU pero no provee acceso directo a estos datos. Para pdoer modificar los datos de los pixeles en el programa central, es necesario descargar primero la imagen a la memoria principal.

De esta forma, el módulo GPU esta diseñado como una extensión 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 una pequeña porción de código adicional para ejecutar una funcionalidad 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 la última con una imagen almacenada, leída 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 había 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. Transformación 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 operación 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 líneas 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 ningún pixel sobre el cual trabajar, el thread hace entonces 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 la grilla 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  __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)
2  {
3    int x = get_global_id(0);
4    int y = get_global_id(1) * PIX_PER_WI_Y;
5
6    if (x < cols)
7    {
8        int src_index = mad24(y, src_step, mad24(x, scnbytes, src_offset));
9        int dst_index = mad24(y, dst_step, mad24(x, dcnbytes, dst_offset));
10
11       #pragma unroll
12       for (int cy = 0; cy < PIX_PER_WI_Y; ++cy)
13       {
14           if (y < rows)
```

```
15          {
16            __global const DATA_TYPE* src = (__global const DATA_TYPE*)(srcptr + src_index);
17            __global DATA_TYPE* dst = (__global DATA_TYPE*)(dstptr + dst_index);
18            DATA_TYPE_4 src_pix = vload4(0, src);
19  #ifdef DEPTH_5
20            dst[0] = fma(src_pix.B_COMP, 0.114f, fma(src_pix.G_COMP, 0.587f, src_pix.R_COMP * 0.299f)
                  );
21  #else
22            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);
23  #endif
24            ++y;
25            src_index += src_step;
26            dst_index += dst_step;
27          }
28        }
29    }
30  }
```

No se va a hacer una descripción detallada del código ya que el trabajo está centrado en la plataforma CUDA pero, de forma general, el código se llama 1 vez por cada columna de la matriz y, dentro de esta función, el segundo *for* está paralelizado usando la directiva #pragma unroll (línea 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 paralelización se puede utilizar para operaciones que tiene como entrada pixeles correspondientes a distintas imágenes. El proceso es muy similar al caso previo (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 función (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. Fútbol robot

El trabajo se centra en un sistema de visión por computadora para el reconocimiento de objetos en un partido de fútbol de robots.

El fútbol 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 cada robot dentro de un equipo. Además, ningún robot puede tener el color caracterísico de la pelota(naranja) ni el del equipo contrario en su camiseta. Todo el procesamiento se realiza desde un sistema central y no se permite la intervención de humanos a menos que el juego este detenido. Este sistema dispone de las imágenes tomadas por una única cámara situada sobre el campo de juego.

El procesamiento por parte del sistema de control se puede dividir en 3 pasos:

1. 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.

2. Planificación de las acciones de los robots: Se determina las acciones a tomar con el objetivo de lograr el objetivo de trasladar la pelota al objetivo.

3. Control de los robots: Se usa un sistema de comunicación inalámbrico para mover los robots de acuerdo a la estrategia definida.

3.1.1. Detección en tiempo real

Dentro de las etapas definidas para el sistema de control hay diversos procesos cuyo tiempo es necesario considerar: latencia de la cámara(desde que se toma la imagen hasta que se comienza a procesar), latencia de la detección de los objetos, latencia de definición de estrategia y latencia de comunicación.

El algoritmo de control debería correr sobre un sistema de tiempo real duro: el procesamiento sería válido solo si se completan todas las etapas dentro de una cierta ventana de tiempo, normalmente hasta que se haga una nueva captura del campo, la cual invalida el estado descrito por la imagen anterior. El objetivo de esto es que todo el procesamiento se realice

en un tiempo donde la imagen sobre la cual se estan tomando las decisiones sea representativa del estado actual del campo, en caso contrario la decisión puede ser incorrecta.

La realidad continuamente cambiante del campo implica que, aún corriendo sobre un sistema de tiempo real duro, el tiempo que transcurre entre las distintas capturas puede llevar a tomar decisiones incorrectas. Por lo tanto se utilizan dos aproximaciones para adaptar el procesamiento lo mas posible a la realidad:

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

En este trabajo nos centraremos en esta segunda aproximación.

3.1.2. La libreria bottracker

El trabajo esta centrado en el primer paso del procesamiento que realiza el sistema central de control para fútbol 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 ésta se deben detectar las posiciones de los robots de cada equipo y de la pelota.

Para realizar esto disponemos de una librería [2] que permite realizar el procesamiento de cuadros capturados por la cámara de video del campo de juego. La función central de la libreria recibe por parámetro una imagen correspondiente al cuadro actual del estado del campo y detecta a partir de este los elementos de interés (robots y pelota) devolviendo la información relevante.

La clase central de la libreria es *bot_tracker*. Al ser instanciada, esta clase debe configurarse con los parámetros necesarios(imagen de background, colores, etc) para poder realizar todo el proceso de detección.

La función *process* es donde se realiza todo el procesamiento para la detección de los objetos, recibiendo por parámetro un cuadro a la vez. Este proceso no se va a explicar nuevamente pero se puede encontrar en detalle en Ref. [2, capitulo 5]

En este trabajo en particular nos centraremos en la primera etapa del procesamiento, la detección de blobs, la cual implica tres pasos:

1. Convertir a escala de grises el cuadro capturado.
2. Construir una imagen nueva mediante la diferencia absoluta entre el cuadro y el background (ambos en escala de grises)
3. Aplicar un valor de threshold(umbral) a la imagen resultante del paso anterior, resultando en una matriz binaria.

3.2. Implementaciones sobre GPU

Como se explicó previamente, una de las aproximaciones para que el procesamiento de imágenes se adapte de la mejor forma posible a la realidad cambiante del juego de fútbol consiste en disminuir la latencia asociada a los pasos involucrados en este algoritmo. La aceleración del proceso de detección de blobs mediante el uso de GPU tiene este objetivo.

Si la optimización lograda mediante el uso de la GPU no supera el overhead asociado a transferir los datos (hacia/desde la memoria GPU), entonces se reducirá el tiempo total

requerido para esta etapa y las decisiones que se tomarán a continuación estarán basadas en información mas actual.

En esta sección se evaluará la posibilidad de aplicar esta aproximación sobre la función *process*(librería bottracker), implementada actualmente usando solo funciones sobre CPU.

El algoritmo sobre el cual trabajaremos permite evaluar objetivamente las ventajas de la optimización mediante gpu ya que es suficientemente simple para optimizar pero a la vez tiene la complejidad que brinda un código real, en el cual parte de éste tiene la posibilidad de ser acelerado sobre la gpu y otra parte del código debe mantenerse en la CPU(porque no aporta una ventaja y/o no esta implementado), de esta forma existen distintas combinaciones de gpu/cpu que deberan ser evaluadas en cuanto a performance.

3.2.1. Algoritmos

Hay tres operaciones que se utilizan en el algoritmo de detección de blobs y que ya se encuentran implementadas en el módulo gpu de OpenCV. A continuación se describen brevemente estas funciones y como se modifica el algoritmo para utilizar las implementaciones sobre GPU.

Conversión a escala de grises

El primer paso en el algoritmo de detección de blobs es transformar la imagen capturada del campo (almacenada en RGB) en una escala de grises.

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. La librería OpenCV optimiza la conversión a escala de grises utilizando el template para transformaciones unaria de imagenes explicado en la sección 2.2.2

Para utilizar esta aceleración, el paso de conversión se realiza de la siguiente forma:

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

Donde d_frame es la imagen del estado actual del campo de juego(en espacio de colores RGB) almacenada sobre la gpu, d_gray0 es una variable declarada gpu::GpuMat donde se almacenará el resultado(imagen en escala de grises), y CV_BGR2GRAY es un valor definido por la librería que indica que tipo de conversión queremos hacer.

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 cálculo aritmético simple entre valores para cada pixel y es totalmente independiente uno de otro, por lo que es fácilmente paralelizable. OpenCV implementa esta función mediante una transformación binaria como se describió en la sección 2.2.2.

La función implementada sobre GPU se invoca mediante:

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

Donde d_gray representa el frame en escala de grises

Conversión a valores binarios

Este paso también involucra una operación 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 función que hay que calcular es:

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

Para ejecutar esta funcionalidad sobre GPU se debe invocar a la siguiente función definida en el módulo gpu:

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

Para modificar la implementación existente sobre CPU se debe tener en cuenta que, además de reemplazar las operaciones por su versión en GPU, se deberán agregar explicitamente las declaraciones de variables y transferencias de datos necesarias entre las memorias. Por ej. si se quieren implementar los tres pasos sobre GPU quedaría:

Código 3.1: Ejemplo de método process sobre GPU

```
1  //recibo frame por parametro
2  cv::gpu::GpuMat d_gray0, d_gray1, d_gray2;
3  d_frame.upload(frame);
4  //subir el frame a memoria de GPU
5  cv::gpu::cvtColor(d_frame, d_gray0, CV_BGR2GRAY);
6  cv::gpu::absdiff(d_gray0, d_backgroundGray, d_gray1);
7  cv::gpu::threshold(d_gray1, d_gray2, threshold, 255, CV_THRESH_BINARY);
8  //traer resultado a memoria de CPU
9  d_gray2.download(gray2);
10 //...PROCESAwwMIENTO SOBRE CPU A PARTIR DE gray2
```

3.2.2. Sistema de pruebas

Para realizar las pruebas se reutilizo gran parte del programa cliente descrito en [2, capitulo 4].

En términos generales, este programa recibe por parámetro un video que representa la entrada de la cámara de video central del partido de fútbol robot. Dividiendo esta entrada en frames, realiza el procesamiento de cada uno de éstos hasta finalizar el video. El programa imprime información relevante sobre el tiempo de procesamiento que es utilizada luego para analizar la ejecución.

3.2.3. Resultados

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 procesamiento del caso de prueba completo sobre las distintas implementaciones posibles.

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

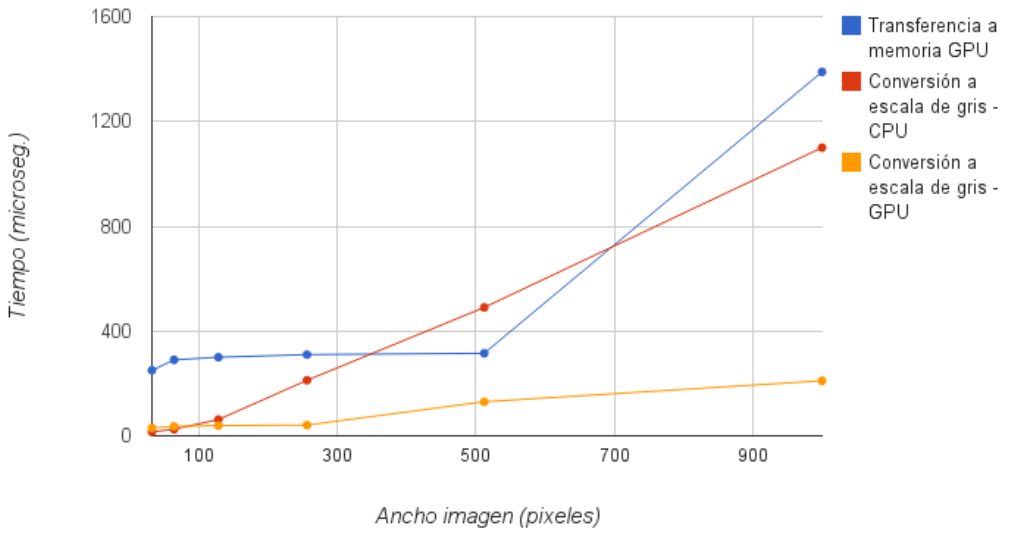

Figura 3.1: Relación entre el tiempo de transferencia y el tamaño(ancho) de la imagen

3.2.4. Hardware utilizado

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

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 [3] , 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

Desde el punto de vista aplicativo, los conocimientos adquiridos podrian aplicarse en diversos campos como el reconocimiento de patrones, seguridad, procesamiento de imagenes satelitales, etc.

Bibliografía

[1] NVIDIA CUDA ToolKit Documentation. `http://docs.nvidia.com/cuda/`. [accedido ultima vez 7-Marzo-2015].

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

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

[4] OpenCL standard. `https://www.khronos.org/opencl/`. [accedido ultima vez 7-Marzo-2015].