Computer Vision Acceleration Using GPUs

Shehrzad Qureshi
Medallion Solutions
President
shehrzad@medallionsolution.com
AGENDA

- Introduction to Computer Vision
- Typical Computer Vision Processing Pipeline
- Lens Calibration Problem
- Lens Correction on the GPU (OpenCL implementation)
- Other Applications (Stereo Vision)
- Q&A
Introduction to Computer Vision
INTRODUCTION TO COMPUTER VISION | What is it?

- Oftentimes entangled with the closely allied field of Image Processing.
- Succinct definition = “Systems that see” (as opposed to the mere processing of digital picture data)
- Inverse of Computer Graphics
  - Ascertain underlying 3D geometry comprising a 2D scene (e.g. depth)
  - Versus accurate rendering of 3D geometry on to a 2D canvas
- Characteristic algorithms
  - Feature extraction
  - Depth perception
  - Structure from motion (optical flow)
  - All computationally expensive!
INTRODUCTION TO COMPUTER VISION | Why is it important?

- Driving force behind many emerging applications.
  - Gesture recognition (advanced user interfaces)
  - Face detection/recognition (biometrics)
  - Robotics (medical, defense)
  - Autonomous vehicles or assisted driving (e.g. lane detection)

- Much of computer vision well suited to GPU acceleration!

- Lots of activity in mobile arena – GPUs uniquely positioned to accelerate vision in these platforms
  - Vision traditionally implemented in hardware (FPGAs, ASICs)
  - All modern mobile devices contain GPUs
  - Perfect platform for vision acceleration (Google Goggles, gestures)
INTRODUCTION TO COMPUTER VISION | Real-World examples leveraging GPU

- Stereoscopy in robotic surgery
  - Restoration Robotics, Inc: hair transplants performed by a robot, without intervention by a surgeon!
  - Real-time 30 FPS 1024x768 stereo rectification accelerated on the GPU
- Semiconductor optical wafer inspection
  - Classic industrial vision application (optical defect recognition)
  - GPU performed lens correction (multiple cameras)
- Bomb detection
  - Android or Meego tablet connected to X-Ray scanner via USB
  - Huge (10-90 megapixel!) frames streamed into tablet where GPU assists in various vision and image processing tasks
Computer Vision Pipeline
**Computer Vision Processing Pipeline**

1. **Image Acquisition**: color or monochrome frame streamed from camera
2. **Color Conversion (optional)**: RGB-to-grayscale downconversion
3. **Lens Correction**: focus of this presentation, compensate for lens aberrations
4. **Preprocessing**: filtering, contrast enhancement, compensate for non-uniform illumination, optical flow
5. **Segmentation**: separate frame into foreground and background channels
6. **Object Analysis/Feature Extraction**: begin to make sense of the scene
7. **Heuristics/Expert System**: application dependent, make decisions or measurements based on what system sees. For example,
   1. Servo motor based on where object of interest moved (robotics)
   2. Update UI (gesture recognition)
Certain blocks are easily amenable to efficient GPU implementations

- Depending on how the object analysis and/or feature extraction is performed, it may be straightforward to implement on the GPU
  - Blob contours, or connected components, very tricky to parallelize!
  - In contrast, feature extraction based on edge detection can be done efficiently on the GPU in a straightforward manner

- Luckily for the designer, the “GPU blocks” in above diagram are contiguous
- This means we keep the pixels on the GPU which has major implications on efficiency
Lens Distortion
Lens Distortion | Problem Statement

- Serious problem that thwarts vision algorithms if not addressed
  - “Fish-eye” effect
  - Prevalent with inexpensive optics, like the cameras you’ll find on consumer grade mobile devices
- Lens projects straight lines into curves
  - Radial distortion, increases in magnitude as we move farther away from the image center
  - Barrel distortion bulges outward
  - Pincushion distortion bends inward
- Rectilinear correction can compensate for these lens aberrations
**Lens Distortion Correction | Equations**

- **Model:**

\[
\begin{pmatrix}
  x^* - x_0 \\
  y^* - y_0
\end{pmatrix}
= L(r)
\begin{pmatrix}
  x - x_0 \\
  y - y_0
\end{pmatrix}
\]

- **Where:**
  - \((x, y)\) are the original (distorted) point coordinates
  - \((x_0, y_0)\) is the image center
  - \((x^*, y^*)\) are the undistorted (corrected) point coordinates

- Then we need to measure the distortion model \(L(r)\) where:

\[
L(r) = k_0 + k_1 r + k_2 r^2 + ... \]
\[
r = \sqrt{(x - x_0)^2 + (y - y_0)^2}
\]
Lens Distortion Correction | Calibration Procedure

- One time calibration procedure
  1. Image known test pattern
  2. Extract corners
  3. Enumerate curves known to be straight
  4. Feed points into optimizer
  5. Coefficients of L(r) final output

- Can be a time consuming process but it is performed off-line
Lens Correction | Warping
Lens Distortion Correction | GPU Implementation Strategies

- Given \( L(r) \) and a newly acquired image, how do we correct the image in an efficient manner?
- Obviously marching through each pixel, evaluating \( L(r) \) to get updated coordinates, etc. highly inefficient!
- Much better to use a pre-computed resampling grid and then apply this resampling to each incoming frame
- Essentially, lens distortion boils down to an image warp
- Warping also useful in other contexts (e.g. stereo rectification)
- Goal is an efficient OpenCL kernel to do the warping.
forward mapping requires pixel interpolation because mapped pixel will fall in between pixels
   - nearest neighbor is easy but inaccurate
   - bilinear interpolation usually good enough, some applications may warrant bicubic
major problem are holes in the output image which need to be fixed after the fact.
Lens Distortion Correction | GPU Implementation Strategy (reverse warping)

- avoids the holes problem, still requires interpolation
- preferred to forward mapping, but assumes that L(r) is invertible
OpenCL Implementation | Design

- Undistort 2592x1944 8-bit image
- Two resampling matrices needed
  - x mapping (mapx)
  - y mapping (mapy)
- dst(x,y) ← src( mapx(x,y), mapy(x,y) )
- mapx(x,y) and mapy(x,y) will be floating-point numbers so bilinear interpolation needed!
- Out-of-place processing, separate input and corrected images
OpenCL Implementation | Design

- Instead of implementing the bilinear interpolation manually, *leverage the texture capabilities of the hardware.*
  - Texture fetches are cached
  - Perform bilinear interpolation in hardware
  - Clamping or wrapping at image boundaries, so no need for manual checks in the kernel code.
- Leads to extremely simple OpenCL code!
**OpenCL Implementation | Code**

- Host code consists of mostly boiler-plate code pilfered from the SDK samples.
- Instantiates four 2D image buffers
  - Read-only input image: `cl_image_format = {CL_R, CL_UNSIGNED_INT8}`
  - Read-only CL_FLOAT mapx & mapy: `cl_image_format = {CL_INTENSITY, CL_FLOAT}`
  - Write-only output image: `cl_image_format = {CL_R, CL_UNSIGNED_INT8}`
- Launches and times image warping kernel
__constant sampler_t samp = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;
__constant sampler_t sampf = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_LINEAR;

__kernel void img_warp(__read_only image2d_t in, __write_only image2d_t out
__read_only image2d_t xmap, __read_only image2d_t ymap)
{
    size_t x=get_global_id(0), y=get_global_id(1); /* upper-left corner of this work-item’s tile */
    size_t nx=get_local_size(0), ny=get_local_size(1);
    /* resample each pixel in this work-item's tile */
    for (int ii=0; ii<nx; ++ii) {
        for (int jj=0; jj<ny; ++jj) {
            int2 xycoords={x+ii, y+jj};
            /* read xmap and ymap */
            float4 xmapTexel = read_imagef(xmap, samp, xycoords);
            float4 ymapTexel = read_imagef(ymap, sampf, xycoords);
            /* warping - read from input and write into output */
            float2 posf = (float2)(xmapTexel.x, ymapTexel.y);
            uint4 pixel = read_imageui(in, sampf, posf);
            write_imageui(out, xycoords, pixel);
        }
    }
}
OpenCL Implementation | Performance

- 47 ms to warp 2592x1944 UINT8 image
  - get_local_size(0)=32 and get_local_size(1)=24 for best performance.
  - Does not include data transfer time to copy over the input image.
- 65 ms to warp same image on quad-core CPU using OpenCV cvRemap() function
- OpenCV implementation highly optimized, multi-threaded, and tuned over a period of many years
- OpenCL implementation not completely optimized or tuned.
- Should use half-floats for mapx and mapy in OpenCL kernel for some speedup.
- CPU specs
  - Core2 Quad (2.4 GHz) running 32-bit XP
  - Visual Studio 2008 & OpenCV 2.1
- GPU (same host as above)
  - Radeon HD 5800
  - Catalyst 11.5 driver and OpenCL 1.1 (AMD APP SDK v2.4)
Conclusion | Other Applications (Stereo Vision)

- Image warp useful in other contexts
- Warping required for *rectification* used in stereo vision
  - Stereo vision produces depth from the subtle differences between two images
  - Image the same object from two slightly different vantage points
  - Just like your two eyes, the lateral shift between the images your eye produces is the “disparity”
  - Magnitude of disparity directly proportional to depth!

- “Epipolar constraint” in stereo vision means that prior to computing disparity, pixels corresponding to same object must lie on the same scanline
- Stereo rectification aligns images acquired from binocular vision systems such that they meet this epipolar constraint
- Can use the same OpenCL kernel presented here to perform GPU accelerated stereo rectification on the GPU!
Conclusion | Q&A

- Questions?
- shehrzad@gmail.com or shehrzad@medallionsolution.com
- www.medallionsolution.com
Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.

The contents of this presentation were provided by individual(s) and/or company listed on the title page. The information and opinions presented in this presentation may not represent AMD’s positions, strategies or opinions. Unless explicitly stated, AMD is not responsible for the content herein and no endorsements are implied.