Darkroom for CMU Smart-headlight

Analysis of the darkroom language as a platform for the CMU SmartHeadlight pipeline

Vinay Palakkode
vpalakko@andrew.cmu.edu

ACKNOWLEDGEMENT

I'm extremely grateful to James Hegarty who proactively and consistently supported me during the entire course of the project execution. I'd also like to thank Jonathan Ragan-Kelley for chipping in with insightful pointers when I was in a confused state after seeing the darkroom's performance figures which I had come up with. The reference Implementation of the CMU SmartHeadlight is an Intellectual Property of the Illumination and Imaging lab lead by prof. Srinivasa Narasimhan. Last, but not the least, I'm thankful to prof. Kayvon Fatahlian for the project idea, timely advices and for the Jetson Tegra-k1 hardware.

SUMMARY

I ported the first 4 stages of the CMU smart-headlight pipeline, namely background subtraction, thresholding, morphological dilation and inversion, to Darkroom language and analyzed the performance of the darkroom implementation compared to a hand-tuned implementation. I figured out that for a naively written C version of the same optimized by clang at O3 is outperformed by darkroom by a factor of **2X** with an exception of the inversion stage on both x86-64 Haswell Processors and ARMV7-Tegra K1 SoC. But nonetheless the darkroom generated code is about **an order of magnitude** slower than the hand-tuned implementation on intel Haswell processors. I also identified the primary reasons for the performance difference.

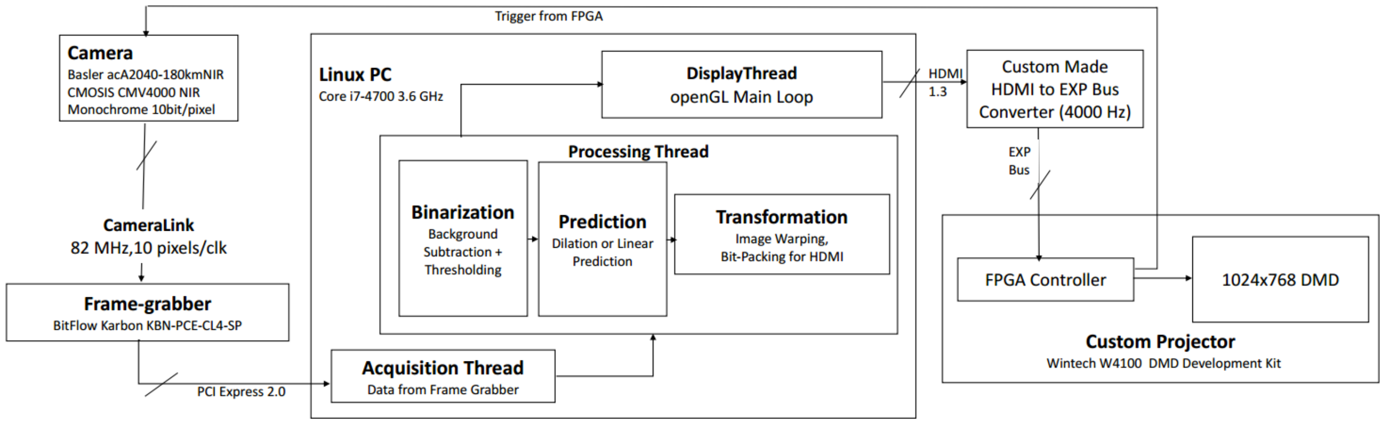
BACKGROUND

CMU Smart Automotive headlight is a reactive visual system. A detailed description about the system is available from the following links

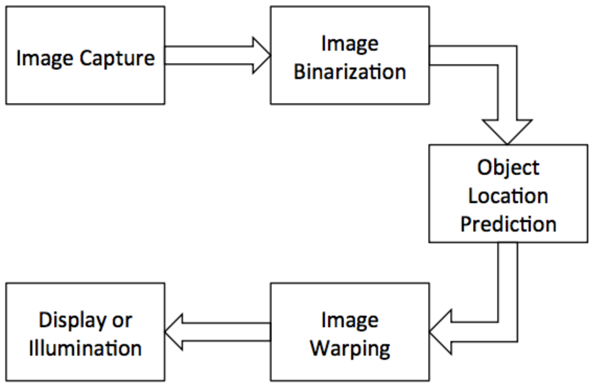
- **http://www.cs.cmu.edu/~ILIM/publications/PDFs/TNCCRKN-ECCV14.pdf**
- **http://www.andrew.cmu.edu/user/vpalakko/darkroom-headlight/**

The complete pipeline description is available here:

- **http://www.andrew.cmu.edu/user/vpalakko/darkroom-headlight/pipe.html**



In a nut shell the following block diagram summarized the various stages in the pipeline which I tried porting using the darkroom language. It is worth mentioning that, except the morphological dilation all other stages are memory bound.



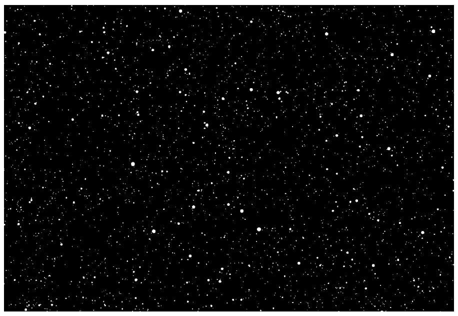
The motto of this project is to port the different stages of this pipeline on to darkroom language, which is supposed to be handy in generating, optimized X86 and ARM CPU code as well FPGA code from the darkroom pipeline description. Presently, for the hand tuned code base

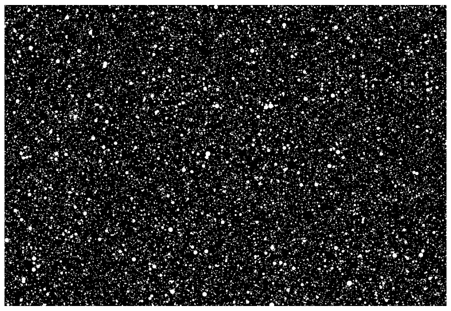
- background subtraction and thresholding (Image Binarization) of the 960 x 680 input frames on Haswell (Core i7 4790 @ 3.6 GHz) runs at about **30**us.
- And the OpenCV 11x3 dilation runs at **250**us.
- The image warping runs at **170**us.

APPROACH

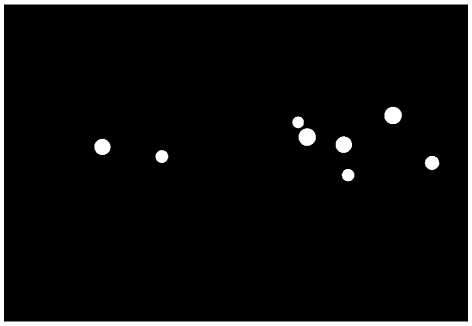
- I wrote stand alone darkroom code for each of the pipeline to test the correctness of the output. This set of terra files run with the aid of the Lua JIT compiler.
- In order to quantitatively measure how well Darkroom generated code performs, I wrote naive C code for each stage of the pipeline, so that I can measure the performance of clang's generated code of each stage Vs the darkroom generated code for the same.
- And after this I created a naive "fused" pipeline in C and implemented a darkroom fused pipeline to measure their performances so that I can see by what factor the darkroom code performs better than the naive C implementation.
- The test dataset was a set of monochrome grayscale images of simulated rain and snow particles and another set of monochrome images of white ping pong ball balls with dark background with bit depth 8bits per pixel and resolution **1024 x 768**.
- The performance numbers were generated on both x86-64 and ARMV7 based hardwares. The reported figures are average values of time taken by each stage for the images in the test data set.

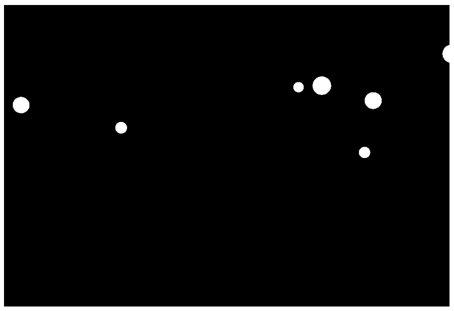
A couple of simulation images of the simulated snow at different snow fall rates.





A couple of simulation images of the simulated ping pong ball explosion experiment





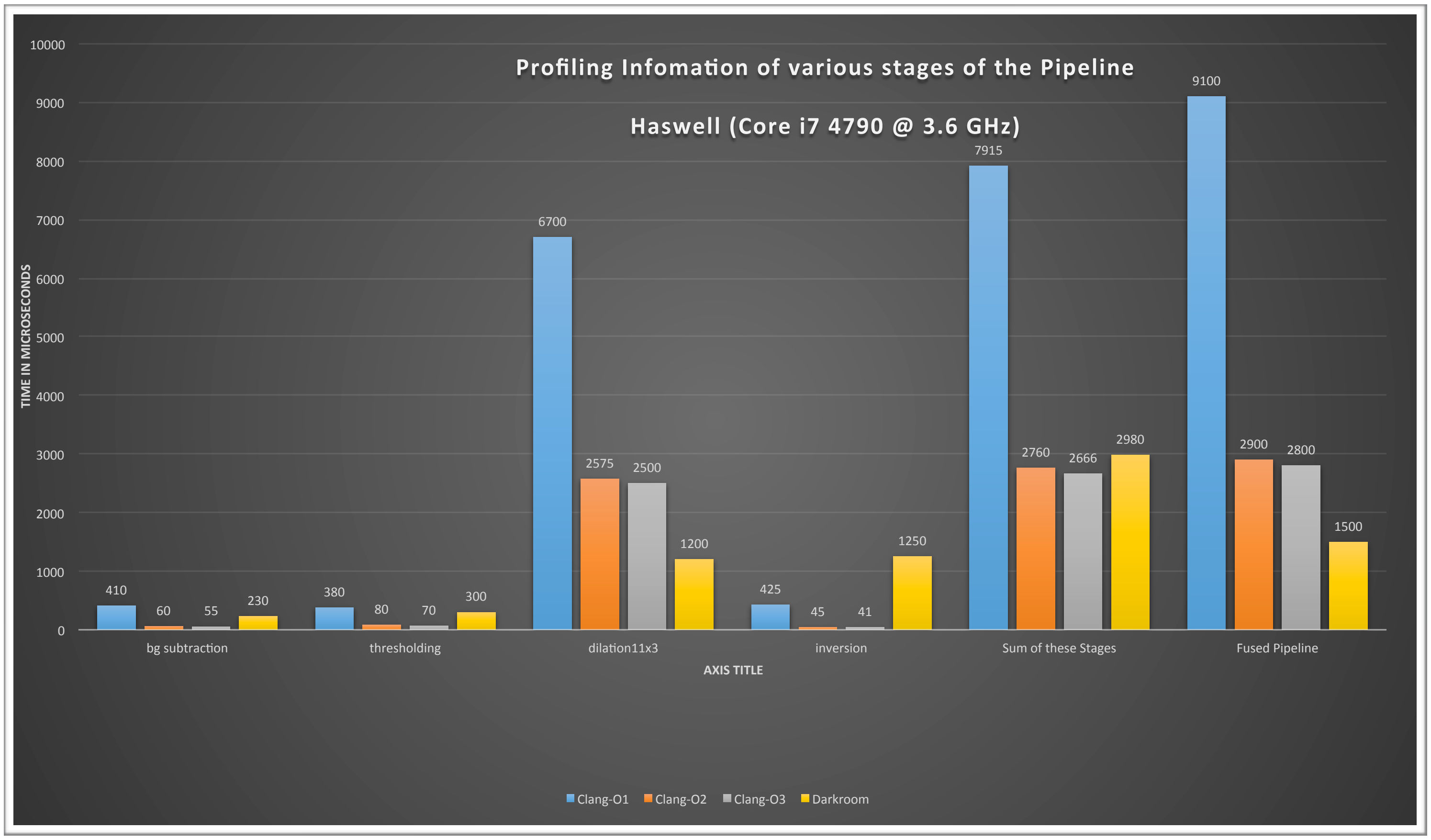
RESULTS

All the figures below are based on the experiments conducted on systems with the following configuration

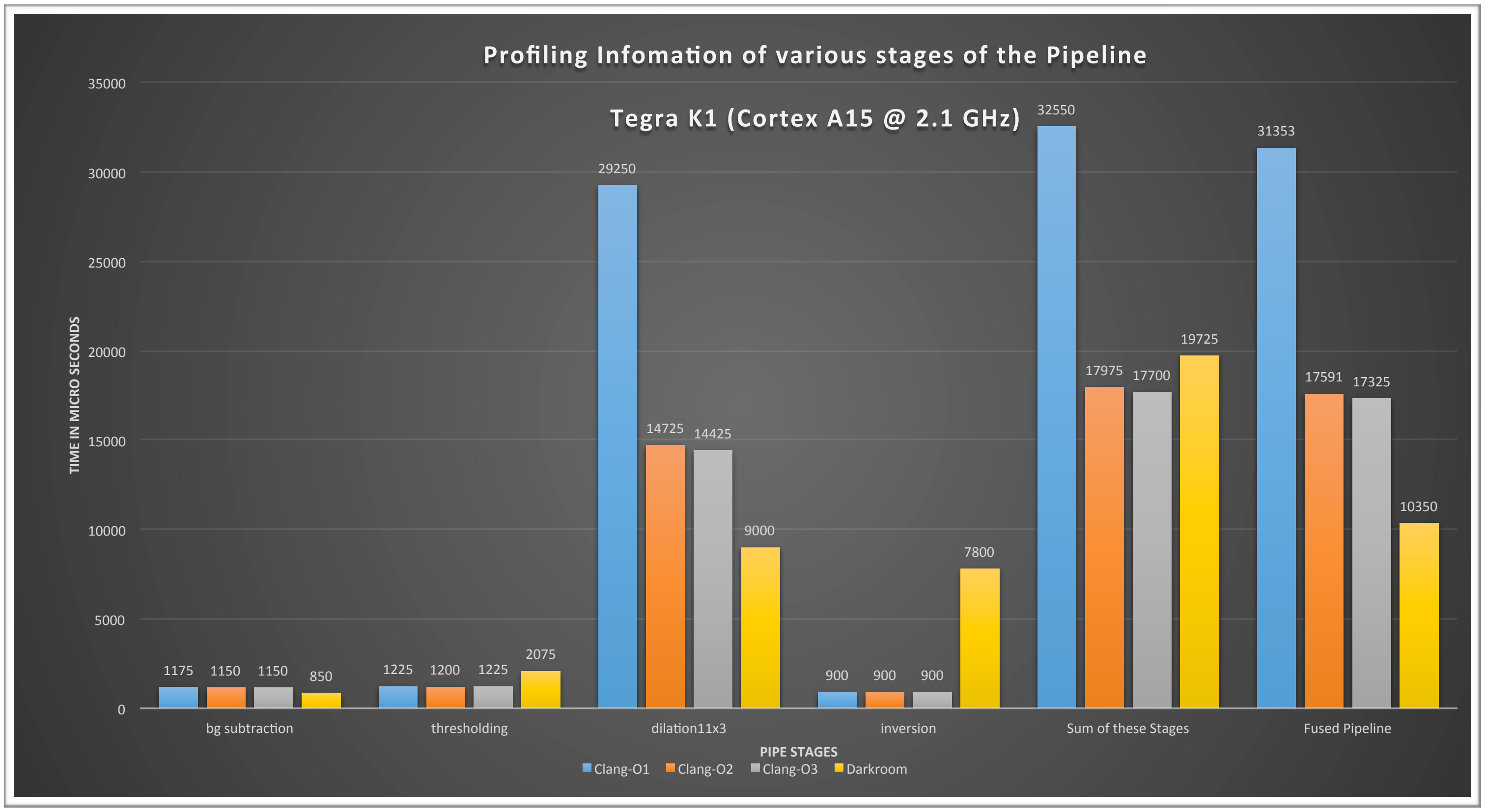
X86-64 : Haswell (Core i7 4790 @ 3.6 GHz, 16 GB DDR3)
ARM-V7 : Jetson (Tegra K1: Cortex A15 @ 2.1 GHz, 2GB DDR)

The bar graphs show the time taken by background subtraction, thresholding, 11x3 dilation stages and inversion. The next cluster of bars are basically the cumulative values of all these stages. And the final one shows the time taken by a fused C implementation and the corresponding fused darkroom kernel for all the 4 stages.

Profiling figures on X86-64 architecture



Profiling figures on ARMV7 architecture



1. There is a *minimum ramp up time of **200**us* for any piece of darkroom code. This can typically be attributed to the fact that the line buffering is implemented on CPU code by the image, spawning threads and pinning them to the available CPU cores. (by default it spawns number of threads = number of cores.). Also the fact that for stencil operations, to avoid data sharing across the threads, the values on the stencil boundaries are recomputed. But surprisingly, by setting the number of cores = 1, at compiled time didn't seem to reduce this overhead.

2. But, *a single stage which is compute bound, when implemented using darkroom shows better performance than a naive clang optimized one*. But darkroom in this case wins only by a factor of 2, as I could not see a flurry of AVX2 instructions in the disassembly. The LLVM IR code is not seem to be optimized for AVX despite setting {V=8} while compiling there pipeline.

3. The bitwise not when implemented as a standalone stage was behaving abnormally. It was slower by an order of magnitude. This needs further investigation

4. Though with Memory bound operations, individual stages perform poorly, when the kernel is fused, we see that darkroom runs 2X faster than a naively fused C-pipeline. For this pipeline from background subtraction to inversion, except morphological dilation, rest of the stages are memory bound. Since the line buffering with increased producer consumer locality, the increase in performance is intuitive. But the fact that the SIMD capability of the architecture (both x86 and ARM) is not leveraged by LLVM as efficiently as an expert programmer could manually do, seems to be the reason for darkroom not beating the hand tuned code.

5. The current implementation of Smart Headlight uses OpenCV dilation. And on inspecting the source code of the same, I realized they the source consist of heavy usage of SSE4/AVX2 intrinsics which gets compiled-in based on the host architecture. OpenCV is not heavy relying on icl/ gcc compilers to do the trick. And OpenCV 11x3 dilation for a 1024 x 768 frame runs at ~300 us, which is about 5 times faster than the darkroom version.

6. Warping the image from the camera plane to the projector plan is implemented as an inverse lookup operation in our current implementation. Essentially we have a X-LUT and Y-LUT corresponding to every output pixel. Essentially,

   Projected_Image(x,y) = **Processed_I(u, v);**
   where **u = X-LUT(x,y) and v = Y-LUT(x,y)**

   Darkroom doesn't support this sort of operations directly. There are two ways of implementing it.

   1. precompute u,v and load them as image functions. Since input image is smaller than the output image, we need to padd input with zeros to match the resolution of output. (darkroom requires all the image functions of the same resolution). With this approach we increase the number of initial DDR access by 2X as we are loading 3 images instead of 1. Also we are reading more pixels than required (because of padding). This is a very bad solution.

   2. Create a mesh-grid (X,Y) over the image resolution W x H. Store the precomputed homography matrix **H (**as **LUT-taps)** from the camera plane to projector plane. Then, generate the warp-space (u,v) using H and (X,Y) on the fly. Which means we are introducing two more stages in the pipeline. Then we will have to use **darkroom.gather** to gather pixels from the line buffer. We need to know the gather range at compile time, which is very difficult if we are generating the warp-space on the fly.

SUGGESTIONS

1. **Support for boolean Images**
   After the background subtraction and thresholding, the pixels are boolean values. So with in the pipeline if there could be a stage where the 8bits pixels can be packed into bitfields, our headlight implementation will get a huge benefit out of it. Because the bit-packed output will reduce the bandwidth required to transfer the image to the DLP projector over HDMI

2. **Enhance the support for image warping**
   Image warping fits poorly into darkroom. For warping f(x,y) to g(u,v), it is not always possible to compute u,v on the fly and use darkroom.gather. Precomputing and loading the u,v functions from DDR will result in 2X additional reads from the memory. Since warping is already a memory bound operation, this would make it worse.

3. **Ability to use LUT values as function indices**
   As of now the use of LUT values can be used only as taps. But it will be very convenient if they could be used to index functions. This would make the warping operations fit better into darkroom. The size restriction of 256 for an LUT can still be there, because in most of the cases we could compress an LUT and interpolate the values in between as and when needed.

4. **Make the stages in pipeline capable of handling image functions of different resolutions**
   The darkroom.Compile accepts an array of input Image functions, an array of output Image functions and a single width “W” and height “H”. It might make the compiler complicated, but we often encounter many algorithms where input and output images may be of different resolutions. Padding the images to to make them of the same dimension does not help us a lot, especially in cases like ours where we warp images from 960 x 340 to 1024 x 768.

5. **Implementation of bitwise not**
   I had reported that the NOT was broken and James fixed it. But the simple darkroom kernel which just negates the pixel values takes as much time as the 11x3 image dilation kernel. Although with a single stage of memory bound op, darkroom is bound to perform poorly, the 1.2 ms for inverting a 1024x768 monochrome image is anomalous.

REFERENCES

- **Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines**
  James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan
  SIGGRAPH ’14

- **http://darkroom-lang.org/**

- **http://terralang.org/**

APPENDIX

The fanciest aspect of darkroom is that it needed less than 40 lines of code to implement the 4 stages of the pipeline. I would expect not more than another 10 lines of code to attach the image warping stage to this pipeline.

```
import "darkroom"


local W = 1024
local H = 768

local M = 5 -- this is M/2 of the MxN
local N = 1 -- this is N/2 of the MxN

local thresh = 10

a   = darkroom.input(uint8)
b   = darkroom.input(uint8)
tap = darkroom.tap(uint8)


-- [0] background subtraction stage
out1   = im(x,y)  [uint16[1]](a) - [uint16[1]](b)  end
bg_out = im(x,y) [uint8[1]](out1)   end

-- [1] binarises the input image function based on the threshold "thresh"
bin = im(x,y) darkroom.vectorSelect( [uint8[1]](bg_out) >= thresh, [uint8[1]](1), [uint8[1]](0)) end

-- [2] morphological dilation with rectangular structural element. Assumes binary input vals
im wSum(x,y) [uint8[1]](
  map i=-M,M j =-N,N   reduce(sum) bin(x+i,y+j) end)
end
-- run the dilation kernel
dil = im(x,y) darkroom.vectorSelect([uint8[1]](wSum) > 0, [uint8[1]](255), [uint8[1]](0) ) end

-- [3] Inversion kernel
im output(x,y)
      not  dil(x,y)
end

compiledPipeline = darkroom.compile({a,b},{output},{},W,H,{V=8})
terralib.saveobj("../lib/fused.o",{drm_fused=compiledPipeline})
```