Compositing High Dynamic Range Images Using Low Dynamic Range Image Sets on an FPGA using HLS

Sean Nijjar Qianfeng (Clark) Shen

1.0 High Dynamic Range Images Using Comparametric Camera Response Function

High dynamic range (HDR) imaging allows an image to better represent reality by increasing the colorspace of the image as well as by providing a more linear response with brightness.

In traditional low dynamic range (LDR) images that are captured by typical cameras, brightness and color response is non-linear. Instead, pixel values have a non-linear relationship with brightness as shown in figure 1. Therefore, it follows that an LDR image is a distorted representation of reality; details can be lost in saturated dark or light parts of the scenes (seen at the ends of the response curve)

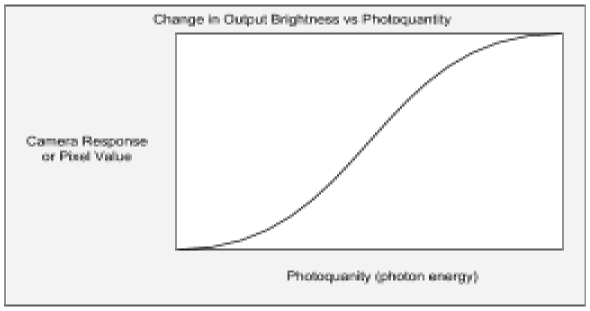


Figure 1. The response curve of a typical LDR sensor as brightness in a captured scene changes

To retain image details in bright and dark image areas, LDR images can be combined to produce an HDR image which represents a less distorted picture of the scene. The comparametric camera response function (CCRF) lookup table (LUT) method is one way to combine LDR images to produce and HDR images. Additionally, it is also a very high performance method since minimal computation is required.

Figure 2 outlines the process of the CCRF LUT algorithm where the input LDR images comprise the top row of images and the output HDR image is on the bottom right.

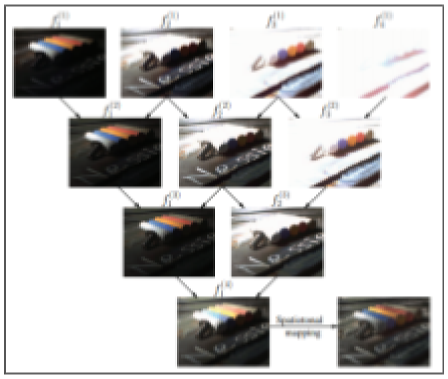


Figure 2 Transformation process of 4 LDR images into 1 tonemapped HDR image

1.1 Constructing CCRF and CRF LUTs.

HDR reconstruction is a process of estimating photoquantity q from the a set of exposures $f_i = f(k_i q)$, where f is called camera response function (CRF) [1]. There are quite a few ways of reconstructing the HDR image from LDR images. Comparametric camera response function (CCRF) [2] is the method used in this paper. Compare to other methods, CCRF is computationally simple and easily to be directly implemented on FPGA for real-time system [3].

Figure 3 shows the process of how LDR pixels are composited to HDR pixels by using CCRF. In this paper, CCRF LUT and Inverse CRF LUT are implemented on FPGA as dual port ROMs. The LUTs are offline precomputed and dumped to COE files for ROMs initialization.

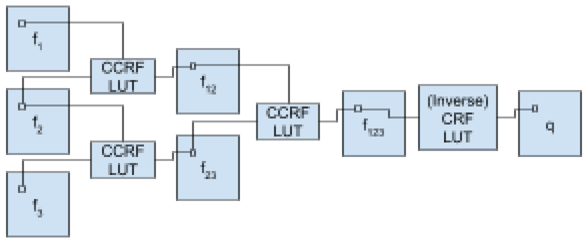


Figure 3 The composition of an tonemapped HDR image from 3 LDR images

In this section, how Inverse CRF LUT and CCRF LUT are generated from LDR images will be introduced.

1.2 Test image set

The test LDR image data set used in this paper is downloaded from [4]. It contains a set of aligned RAW and JPEG image stacks of multi-exposure image sequences that could be used for testing HDR reconstruction. The exposure difference of this image set is $\Delta_{Ev} = 2$. There are 5 LDR images in every image stack. The resolution of each LDR image is 1080P (1920*1080 pixels)

1.3 Generating CCRF LUTs from image set

$$\widehat{q} = argmin_q \left[\frac{(f_1 - f(q))^2}{\sigma_{x_1}^2} + \frac{(f_1 - f(kq))^2}{\sigma_{x_2}^2} \right] \quad (1)$$

Equation 1 [2] is the key for constructing CCRF LUT. where $\widehat{q}$ is the photoquantity value; $f_1 \in [0, 255]$ and $f_2 \in [0, 255]$ are the two entries of the 2D CCRF LUT, which are from the pixel grey value of two input LDR images. f is the camera response function; $\sigma_{x_1}^2$ and $\sigma_{x_2}^2$ are the variances of the rows and columns of the Comparagram (a joint histogram construct from LDR images), which can be estimated from the interquartile range (IQR) along each column and row of the Comparagram.

Equation 1 is executed for all combination of f_1 and f_2, every yield $f(\widehat{q})$ is stored in the CCRF LUT. Therefore, each CCRF would contains 65536 of 8-bits values. According to equation 1, to construct a CCRF LUT, the ingredients required would be the CRF f and the Comparagram (to get the value of $\sigma_{x_1}^2$ and $\sigma_{x_2}^2$).

1.2.1 Creating CRF

To create CRF, OpenCV 3.4 and Python is used on the linux platform on MPSOC. The CRF is generated by using the hdr library function of OpenCV 3.4.

1.2.2 Creating Comparagram

According to [5]:

"The comparagram between two images is a matrix of size M by N, where M is the number of gray levels in the first image and N is the number of gray levels in the second image. The comparagram, which is assumed to be taken over differently exposed pictures of the same subject matter, is a generalization of the concept of a histogram to a joint histogram bin count of corresponding pixels in each of the two images. The convention is to have each pixel value from the first image plotted on the first (i.e., "x") axis, against the second corresponding pixel (e.g., at the same coordinates) of the second image being plotted on the second axis (i.e., the "y" axis). Since the number of gray levels in both images is usually the same (i.e., 256), the

comparagram is usually a square matrix (i.e., of dimensions 256 by 256)." [5].

The pseudo code for creating comparagram is as following:

```
Comparagram_b = zeros(256,256) as uint8
Comparagram_g = zeros(256,256) as uint8
Comparagram_r = zeros(256,256) as uint8
image = zeros(img_num_per_stack,1920,1080,3)
For i in range(0, img_num_per_stack):
    Load image grey values to matrix image[i]

For i in range(0, img_num_per_stack-1):
    For x in range(0, 1920):
        For y in range(0, 1080):

Comparagram_b[image[i][x][y][0]][image[i+1][x]
[y][0]]]++

Comparagram_g[image[i][x][y][1]][image[i+1][x]
[y][1]]]++

Comparagram_r[image[i][x][y][2]][image[i+1][x]
[y][2]]]++
```

Once Comparagram is generated, the variances $\sigma^2_{x_1}$ and $\sigma^2_{x_2}$ can be estimated using the method mentioned in this section previously. Then all the ingredients required for constructing CCRF LUT are ready. The CCRF LUT could be generated by repeat equation 1 for all combination of f_1 and f_2.

2 Initial Design

The initial design was created to support image/video stream HDR processing. The reason to support stream processing vs offload processing is so the hardware pipeline can be used in an embedded platform, connected to a video camera. The anticipated use case would be an HDR pipeline for autonomous cars.

To accomodate a streaming design, the hardware pipeline needed to be able be self-reliant with respect to job scheduling, execution, and completion. For that reason, the software component of the design was only implemented as a testbench and usability interface for future adoption in other application scenarios.

2.1 Original Specifications

The original design objectives have not changed and are listed below:

Be able to composite HDR images from image sets of three, full HD (1920x1080 pixels) LDR images each at a frame rate of 60 frames per second.

2.2 Initial Design

The original system design is shown in figure 4. The original components are described as follows - components marked with '*' do not exist in the final design:

Control Logic: The control log module was responsible for loading parallel jobs into

Low level scheduler: HDR image composition was scheduled by the low-level scheduler. The low level scheduler would take a job request and decompose it into its CCRF lookup subtasks - the subtasks would be ordered for execution.

CCRF engine: The CCRF engine processes job subtasks, one at a time. It has a connection to system memory and stores the result of the computation there.

**CRF estimator*: The original design was ambitious in that it would be able to perform camera response function estimation for each set of images. However, this process is time consuming and unnecessary. CRF estimation only needs to be performed once per camera. In other words, for an autonomous car, this would only need to run during manufacturing - it could be done offline.

**Interrupt generator*: The interrupt generator sends job status messages to the software application. For example, if a job is accepted or rejected and when a job is finished. This is removed and the AXIDMA module satisfies these needs in the final design.

Driver: The driver runs asynchronously to the user application. It takes a job request from the user application and forwards it to the hardware. If the hardware is busy, the driver will receive a NAck. Otherwise, the hardware will receive an Ack. When the job is finished, a “done” packet will be received. The user application can wait for these jobs to finish or it can run asynchronously. Along with the job request, the input LDR images are transferred to the PL/FPGA DDR memory.

*High level scheduler: The high-level scheduler is responsible for submitting jobs to the driver. It has been absorbed into general user code in the final design.

Image Reader/Writer:The image reader/writer is implemented with OpenCV3 routines that allow the application to read different image file encodings and write them back to file. OpenCV3 allows the application to read images like JPEGs, which are found in the test dataset.

*Video Loop Generator: The video loop generator emulates a video stream by replay LDR image sets to the driver in a loop. This functionality was absorbed into general user code in the final design/

Several changes have been made during the project - including several major changes:

1. The CRF estimator was removed because it was determined to be unneeded
    a. The CRF estimator is reflected through the CCRF lookup table and only needs to be computed when a “new camera” is connected. The CRF and the lookup table can be computed offline and then stored in the FPGA at programming time.
2. The interrupt generate was not created
    a. Instead, the AXIDMA module and library were used for communication in both directions (PS to PL and PL to PS).
3. The hardware design was originally managed by a central controller with one job able to execute at a time
    a. One supported job at a time seemed reasonable at the time because if one job could be finished in the desired frame time, then it would be unnecessary to support concurrent jobs

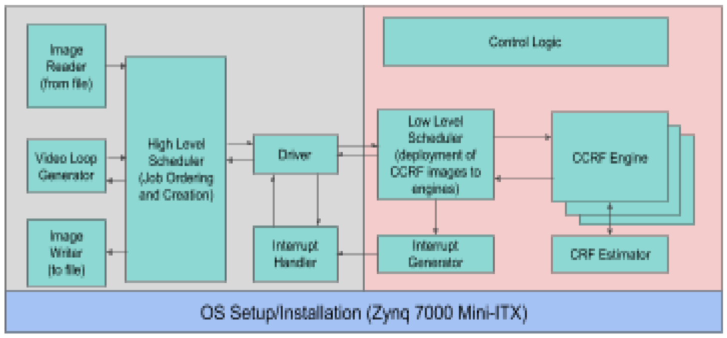


Image Reader (from file)
Video Loop Generator
Image Writer (to file)
High Level Scheduler (Job Ordering and Creation)
Driver
Interrupt Handler
Control Logic
Low Level Scheduler (deployment of CCRF images to engines)
CCRF Engine
Interrupt Generator
CRF Estimator
OS Setup/Installation (Zynq 7000 Mini-ITX)

Figure 4 Original block level design plan

The above design was derived before an understanding of what constituted good HLS practices. For example, we identified the hls::stream<> interface after this design. His stream interface leads to streaming oriented designs. After incorporating the hls::stream, the hardware implementation change from having a central controller to having a pipelined data flow design. This had the added benefit of enabling multiple concurrent jobs to be supported, for “free”.

3 System Design

The application is composed of three high-level parts: the hardware, the software, and the platform.

The software contributes in a non-essential fashion and is primarily for testing and ease of use of the hardware pipeline, so software details will only be covered lightly. The hardware portion of the design does not have any dependencies on the software portion. Additionally, the software requires a host OS to be present on the system. The three parts are built in a way to support a data-streaming flow for HDR image compositing. A high-level diagram of the system is shown in figure 5. Figure 6 shows a more detailed view of the CCRF IP block.

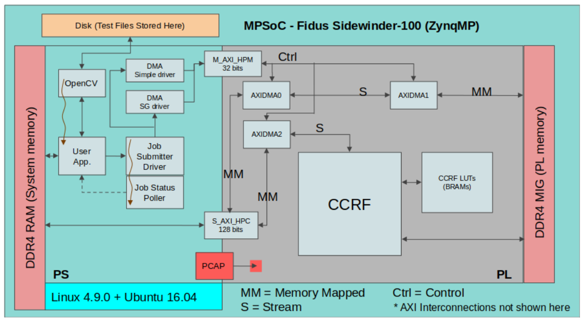


Figure 5 Final design block level diagram

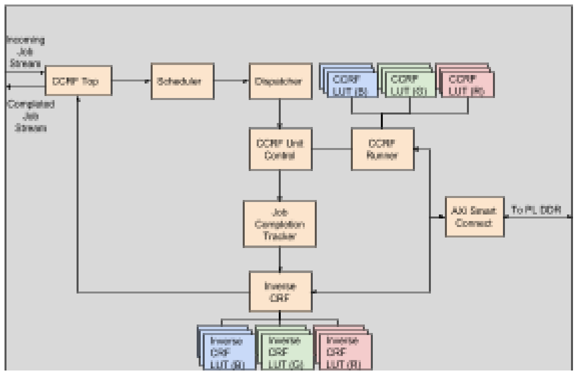


Figure 6 CCRF IP block level diagram

3.2 System Platform

The CCRF HDR application was developed and tested on a Xilinx MPSoC platform - specifically the Fidus Sidewinder-100 (ZynqMP) that has an SoC with an ARM64 CPU, a GPU, and Zynq Ultrascale+ FPGA on chip. Ubuntu 16.04 with Linux kernel 4.9.0 was installed on platform.

3.3 Application Software

The application software is responsible for generating HDR jobs to send to the hardware. It acts primarily as a testbench, although it also provides and API for generating jobs and sending them to the FPGA. The control flow for the software portion of the CCRF application is shown in figure 7.

To make a streaming design easier to implement, the software is run across two threads: one for "user" code and one for "driver" code. The driver thread will run in the background, managing job state and completion, as well as automatic resubmission in the event that a job was previously not allowed. Job submissions may be rejected if the FPGA is too busy at the time of job submission.

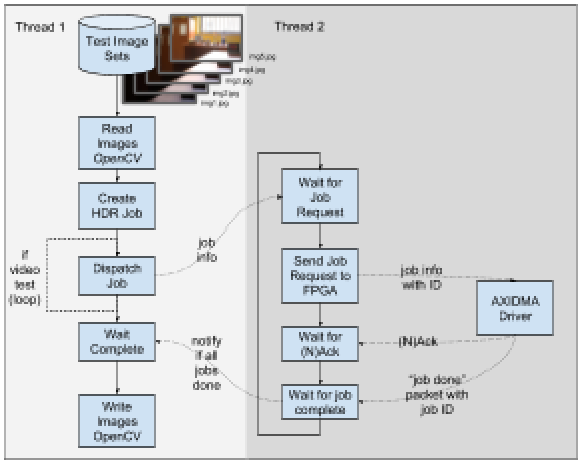


Figure 7. CCRF application software-side control flow diagram

Prior to dispatching jobs to the FPGA, the software driver will allocate FPGA accessible scratchpad memory and send the scratchpad information, namely start address and length, to the FPGA.

3.3.1 Dispatching Jobs

The software driver communicates with the FPGA through the use of an AXIDMA controller. The AXIDMA controller is provided by Xilinx. Transactions are performed with [6], which is an API built on-top of Xilinx' AXIDMA programming interface that allows a user to perform writes to and reads from the AXIDMA controller through a file I/O interface. The will manage transaction bursting and state.

A job is dispatched by sending a job package to the FPGA through the AXIDMA interface. The job package contains information such as the number of LDR images, input image starting addresses, output image starting address, image size, and job ID. The job package is defined below, in figure 8:

```
struct JobPackage
{
      JobDescriptor
job_descriptor;
  JOB_ID_T job_ID;

      //    member
functions excluded
};
```

```
struct
JobDescriptor
{
          uintptr_t
OUTPUT_IMAGE_LOCATI
ON;
          uintptr_t
INPUT_IMAGES[6];
           uint16_t
IMAGE_WIDTH;
           uint16_t
IMAGE_HEIGHT;
            uint8_t
LDR_IMAGE_COUNT;

        //    member
functions excluded
};
```

Figure 8. JobPackage and JobDescriptor definitions

One limitation with the definition above is that LDR image sets with more than 6 input images are not supported. However, the number can be changed, and the system recompiled. For all LDR image counts less than the maximum, an insignificant number of bytes will be wasted for each job request (~0.0001% of total memory reads)

After job requests are sent, the driver will poll the AXIDMA for response from the hardware. These responses are one of three messages:

- Ack + job ID: the FPGA has accepted job the ID provided
- NAck + job ID: the FPGA could not accept the job request
- Done + job ID: the job with the included ID is completed and the results are stored in the output location specified in the job package

If a job is NAck'ed, then the driver will try a resubmission after it receives its next "done" message.

3.4 Application Hardware

The hardware pipeline is a streaming design that can be used independently of the software driver. For example, the hardware pipeline could be connected to a camera feed with minor changes. The following hardware modules implement the CCRF streaming functionality and are displayed in Figure 6:

- Job tracker (I/O and Job (N)Ack)
  - Notifies requestor (software driver) if a job request is accepted, rejected, or done
- Job scheduler
  - Takes a job package and decomposes it into its CCRF subtasks. The subtasks are streamed to the dispatcher
- Job subtask dispatcher
  - The job dispatcher takes the input subtask stream and dispatches subtasks to available CCRF units if all dependencies of the task have been completed
- CCRF unit
  - Processes a job subtask: it takes input images and outputs them to the memory locations specified in the job package
  - The CCRF unit forwards the image output address of the subtask it is processing to the job result notifier
  - Image data is connected through the S_AXI_HP ports
- CCRF LUT(s)
  - Implemented as BRAM ROMs, the CCRF LUT is computed offline and is programmed with a COE file
  - Multiple CCRF LUTs are connected to each CCRF unit for improved throughput through parallel reads
- Job result notifier
  - The job completion notifier tracks completed subtasks from the CCRF unit and forwards a completion message to the job tracker

- ○ The result notifier receives packets from the scheduler that specifies the output address to look for to denote the completion of a job

In the current iteration of the design, there is only one CCRF unit, although the original design called for multiple CCRF units. However, only one CCRF unit is required to meet the performance objectives for the design. Additionally, it is possible to saturate DDR memory bandwidth with one CCRF unit, so there is no need for additional units.

4 Methodology

Throughout this project, we followed Xilinx' development and verification flow for HLS designs. Early in the design phase, though, an alternative approach was attempted but later abandoned because it did not apply correctly to HLS design. A development plan was put in place but it was not followed as initially planned, due to competing commitments from outside this project (i.e. other courses). For that reason, given the size of the team (2 people) a more dynamic development approach was followed where tasks were picked up and completed as team members were available to complete them. In addition to competing commitments, numerous tool and to a much lesser extent methodology issues forced the initial development plan to be changed. Other aspects of the development methodology remained the same:

- Platform
- Source control
- Testing
- Tools

4.1 Platform

Development of the CCRF HLS application was targeted for a Fidus Sidewinder-100 board. The development board is a PCIe (for power only) attached board that contains a ZynqMP SoC. The SoC contains an ARM64 multi-core CPU, an ARM GPU, and a Xilinx ZynqUltrascale+ FPGA. Detailed specifications are provided below:

Board: *Fidus Sidewinder-100*
SoC: *Zynq Ultrascale+*
FPGA Part: *xczu19eg-ffvc1760-2-i*
Development Environment: *Vivado 2017.4*
Operating System: *Ubuntu 16.04*
Kernel: *Linux 4.9.0*

The software platform consisted of an Ubuntu 16.04 installation with Linux kernel 4.9.0. OpenCV3 (an open source computer vision library) was used for reading of test .jpg images and an AXIDMA software library [6] was used as an easier programmer target to perform AXIDMA transfers from the software side.

4.2 Partitioning

The design was partitioned into hardware and software in a way that the hardware component can independently schedule, run, and complete HDR compositing jobs without dependence on a software driver. The software driver only contributes initial job information. In an embedded deployment, this information could be specified through configuration, since image information will be known ahead of time (LDR image count, image resolution).

On the hardware side, the design was partitioned into modules that are separated that communicate with standardized AXI and BRAM interfaces. This allowed each module to be implemented independently.

By following standardized interfaces, it is low-effort to parallelize or partition the hardware development, as long as the data payloads are specified ahead of time. By using HLS, we were able to define the payloads of each interface through the use of structs or static classes in

C++. Using standardized interfaces allowed the team to complete modules as they were needed and removed intermodule development dependencies (except for end-to-end system testing) - it reduced the need for coordination between team members.

Due to a bug in the tool, we did have to break this practice for the CCRF module. The CCRF module was split into separate control and compute modules because Vivado would not express a memory port as an m_axi interface when the CCRF module was a single module.

4.3 Simulation, Verification, Testing

The design was partitioned into hardware and software in a way that the hardware component can independently schedule, run, and complete HDR compositing jobs without dependence on a software driver. The software driver only contributes initial job information. In an embedded deployment, this information could be specified through configuration, since image information will be known ahead of time (LDR image count, image resolution).

On the hardware side, the design was partitioned into modules that are separated that communicate with standardized AXI and BRAM interfaces. This allowed each module to be implemented independently.

By following standardized interfaces, it is low-effort to parallelize or partition the hardware development, as long as the data payloads are specified ahead of time. By using HLS, we were able to define the payloads of each interface through the use of structs or static classes in

C++. Using standardized interfaces allowed the team to complete modules as they were needed and removed intermodule development dependencies (except for end-to-end system testing) - it reduced the need for coordination between team members.

Due to a bug in the tool, we did have to break this practice for the CCRF module. The CCRF module was split into separate control and compute modules because Vivado would not express a memory port as an m_axi interface when the CCRF module was a single module.o be made. Namely, HLS directives would have to be understood and a cycle by cycle execution model would have to be supported. For the most part, these capabilities are already in Vivado HLS and the added features would be of marginal utility (i.e. the user would not need to write a wrapper in the test harness that connects all the user modules involved in the test).

After abandoning the above approach, verification and test reverted to the recommended Xilinx flow, which consists of HLS CoSim, HDL simulation, and live hardware debug (Internal Logic Analyzer for Xilinx). Simulation tests were only written if they catered to a specific problem that was not identified or covered in HLS tests. For example, a simple system-level simulation was created to debug several problems that were not identified by simulation. Unsurprisingly, the majority of these issues revolved around interface issues. Interface issues between modules won’t usually be exposed by module level unit-tests or Cosim tests that don’t have to deal with things like data width conversion or memory bursting. Additionally, these cosim tests won’t verify that implicitly generated signals - such as the tlast signal on an axi streaming interface.

Ideally, with more time, test coverage would significantly improve and more thorough tests would be developed at the HDL and cosim level.

4.3.1 Test Data

Publicly available LDR image datasets, like HDREye[4] were used for testing. These image sets contain 5 LDR images per set, with 12 image sets.

5 Results

The CCRF project currently does not meet functional targets due to a misconfiguration of the inverse CRF of CCRF LUT coefficient files. For scenes with over-exposure it does not meet performance targets as currently implemented. An optimized version has been developed that far exceeds performance needs, but there is currently an additional defect that causes it to produce slightly incorrect output, so those results are not shared here. For more information regarding performance of the optimized version, see Appendix C: In Progress Optimization.

5.1 Functional Output

An example input LDR image sets with its corresponding output HDR image from the CCRF engine are shown in figure 9.

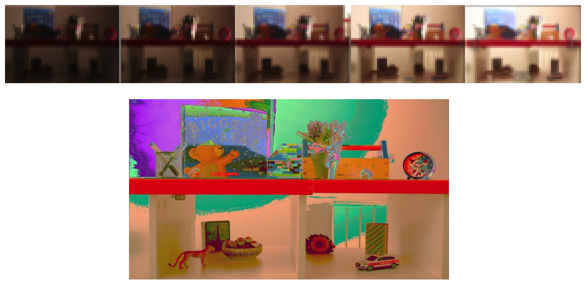


Figure 9 Output HDR image sample

5.2 Performance Results

Performance objectives were to be able to process a “FullHD” images (1920x1080 pixels) with 5 LDR input images at a frame rate of at least 60fps. Unfortunately, due to a functional correctness bug, these targets were not met. With an optimized version, measured performance is an average of 156.9fps and 226.4fps for 5 full HD LDR inputs and 3 full HD LDR inputs respectively (see Appendix C.)

Performance was measured by calculating the frametime for individual HDR frames. The HDR images were produces from LDR imagesets found in the HDREye dataset[4]. Frame time was used to derive frame rate in the following manner:
Frame rate = 1 / (frame time)

Frametime was calculated in two ways, see figure 10:

1) The time needed, with cycle accuracy, for an HDR image to pass only through the hardware pipeline.
   - This measurement more accurately reflects performance in an embedded scenario where image data would already be dumped in the PL DDR memory by a device such as a camera
2) The time, with microsecond accuracy, for a job to complete from the perspective of the PS
   - This measures the amount of time that elapses from when a job is submitted by the PS (i.e. image data transfer start to PL DDR from PS DDR) to when the PS DDR is finished with the copy-back of the output HDR image from PL DDR to PS DDR
   - This measurement is not useful for the intended use case, but is provided to provide expectations for other use cases

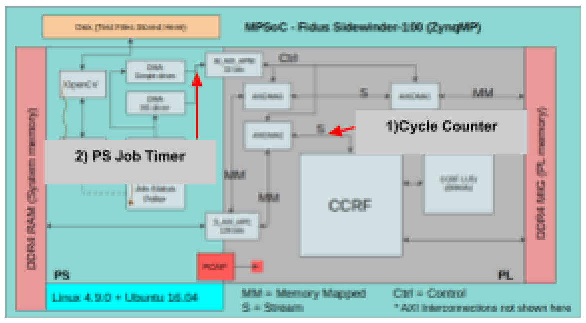


Figure 10 Start and end capture points for two job timers

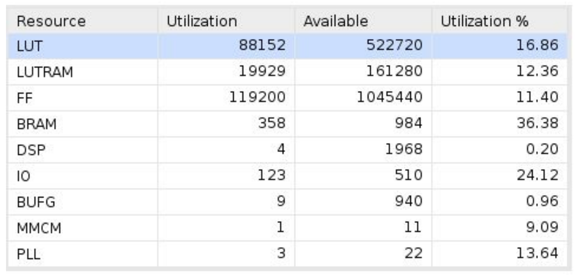
To better show performance trends, the dataset was manipulated to create additional data points. LDR images from the 5 LDR images were removed to also simulate 4 LDR and 3 LDR images. Additionally, LDR images were processed at multiple resolutions to demonstrate how resolution affects performance.

	1920x1080 (2.007MPixel)
3 LDR	10.28fps
4 LDR	7.51fpsl
5 LDR	5.53fps

Table 2

6 Design Utilization

Utilization numbers for the design are reported in figure 11



Resource	Utilization	Available	Utilization %
LUT	88152	522720	16.86
LUTRAM	19929	161280	12.36
FF	119200	1045440	11.40
BRAM	358	984	36.38
DSP	4	1968	0.20
IO	123	510	24.12
BUFG	9	940	0.96
MMCM	1	11	9.09
PLL	3	22	13.64

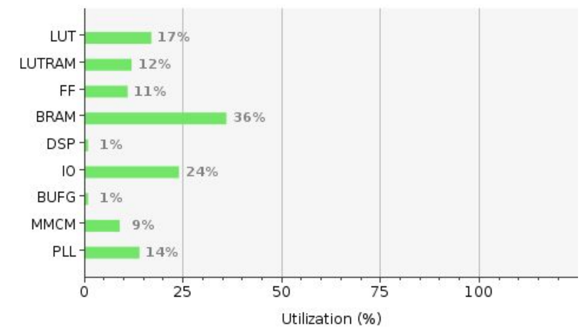


Figure 11 Design hardware utilization

6 Future Work and Design Improvements

The design presented in this work can still be improved considerably in throughput terms. This section outlines several design changes that can be made to achieve this improvement. Even if there is no need for improved throughput, these changes will also increase the overall efficiency of the design - in some cases, reducing latency almost for free and in other cases, allowing the user to run a slower clock because the design would provide much higher throughput per cycle.

Cumulatively, ignoring clock frequency changes, these improvements can lead to a theoretical throughput improvement of ~10.5x to ~42.5x for LDR count equal 3 and 5, respectively, and ignoring in progress, but buggy, optimizations. When considering in progress optimizations (see Appendix C), potential improvements can exceed 30x and 60x for LDR counts of 3 and 5, respectively. The improvement opportunities are listed below and detailed in Appendix B.

- Tiled streaming with BRAM caching of intermediate results
- External memory bandwidth improvements by using additional memory ports
- Support for CCRF LUTs with exposure deltas greater than 1
- More efficient pixel formats

7 Contributions

This section outlines the high-level tasks completed by each group member

7.1 Clark

- Platform Components:
    - Board setup

    - OS setup, configuration, and installation
    - Container setup and automation of startup/reset processes to enable board to work
    - ILA debug setup - partial
- Software components:
    - AXIDMA framework research
        - Tested AXIDMA framework
    - AXIDMA direct mode linux driver
    - PS DDR-PL DDR communication software support
    - Plddr read software utility
        - Enables command line dumping of PL DDR contains at specified location and size
- Hardware Components:
    - CCRF compute implementation
    - Inverse CRF LUT generation
    - CCRF LUT generation
        - COE file BRAM ROM programming
    - Initial block design with Zynq, AXIDMA, and loopback FIFO
    - PS DDR-PL DDR communication blocks implementation

7.2 Sean

- Platform components:
    - ILA debug setup - partial
- Software components:
    - OpenCV3 integration
    - AXIDMA library integration
    - Asynchronous driver
    - Job packaging
- Hardware components:
    - Job tracker
    - Job Timer
    - Scheduler
    - Dispatcher
    - CCRF memory port fix
        - CCRF module split
    - Inverse CRF
    - Job result notifier
    - Block design/IP integration of other hardware components

8 Project Problems

This section outlines development, tooling, environment, and organizational issues from the project.

8.1 Development Problems

There were no note-worty development problems, excluding the poorly informed decision to try to create a testing environment, as mentioned in section 7.3. However, several tool problem regularly delayed development, significantly. Section 11.2 outlines some of the tooling problems the team encountered.

8.2 Platform and Tooling Problems

Several tooling and platform issues arose throughout the development process that delayed progress. Several examples are included below:

- Vivado HLS failed to express a memory port as an m_axi interface
    - #pragma HLS INTERFACE m_axi port=<port> and the deprecated form #pragma HLS RESOURCE core=m_axi variable=<port> were being ignored for a specific memory port and Vivado HLS continued to express the memory port as ap_memory
    - Several days were spent debugging the problem
        - Eventually it was determined that another port had a value that propagated into a memory read and this caused confusion for Vivado HLS
        - To solve the problem, the module was partitioned into two separate modules and the two ports were isolated from each other

- When performing FPGA programming through the USB-JTAG interface using a Xilinx hw_server, in order to perform ILA debug, the card was no longer accessible
    - ssh access would hang
    - After several days of investigation, it was found that a power management defect caused the board to reset and run bare metal in a low power state
        - To bypass this issue, the linux kernel had to be rebuilt without the power management feature enabled
- During ILA debug, a hw_server could not be created or connected to from container
    - The USB-JTAG device would unexpectedly update its ID at an unknown time after reboot, which caused Vivado to be unable to find the USB-JTAG device
    - Our hypothesis was that because the USB-JTAG device was connected to the card, it would have booted later than the container and possibly taken longer to boot
        - Therefore, it would be initializing its device ID on startup, but its startup would have occurred after the agent's and container's startup.

In addition to the above problems, there appears to be a platform problem where if the FPGA is reprogrammed more than once, then AXIDMA readback to the PS from the PL stops working. This problem started to occur part-way through the project. The root issue has not been identified. To bypass this problem, the board must be restarted before each bitstream programming.

8.3 Organizational Problems

Organizational problems were mostly non-existent. The only issues worth mentioning are the obvious ones: time sharing between this and other projects.

9 Thoughts on HLS

This project was the team's first exposure to HLS design. A compiled list of feedback is provided below:

9.1 HLS (The Good)

- Interface generation/synthesis is extremely fast compared to HDL approaches
- For the most part, types can be shared between SW and HW
    - Goof for HW-SW codesign
- Implicit control state machines automatically generated
- Fairly good level of low-level control when needed
    - User can specify which resources certain variables or interfaces use
- Algorithmic and possibly other implementations and unit tests can be shared between HW and SW deployments
    - More common code and less dual-maintenance
- Initial testing for algorithmic functional testing is faster
    - But just because it passes cosim, it doesn't necessarily mean it will work in hardware
- Probably lots of interesting use cases for template[s] (metaprogramming) and constexpr
    - Especially for complex parameterization

9.2 HLS (The Bad)

- Many gripes arise from mistargeted or disorganized documentation
    - The technology itself is useful - much more productive than HDL in most cases
    - Marketing could improve here
        - this isn't exactly C/C++ compiled and synthesized into HW
    - Not so much HW design for software

programmers as it is improved productivity for HW designers
    - Please bring all your documentation online like Nvidia does!
        - Pdfs are harder to reference and are more difficult to reorganize according to emerging trends and design changes
- Type safety is lost/not possible in some scenarios (e.g. array of pointers to buffers must be stored as uintptr_t)
- Can't instantiate a compile time known numbers of non-static/stateful objects
    - Would like to be able to instantiate a "scheduler" object, for example
    - The primary purpose of an object is to carry state

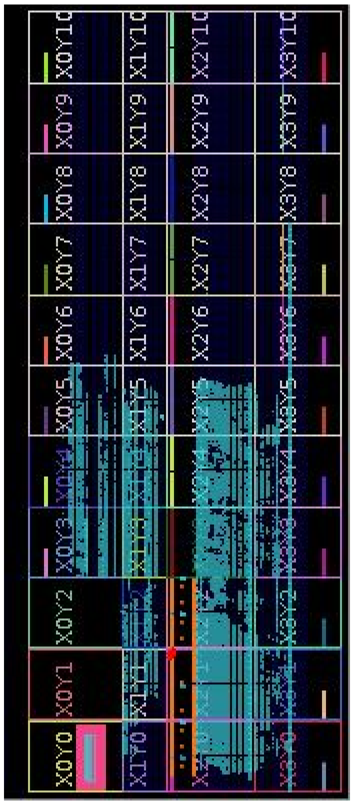
9.3 HLS Wishlist

- Allow for placeholder dangling ports (strictly follow interface definition if user specifies)
    - On occasion, we had to inject dummy code so a dangling port wouldn't be hidden from IP Integrator
- Allow for stateful object instantiation
    - If lifetime is known and instance count is statically known, a user should be able to "instantiate" a stateful object in hardware
    - Object members would be registers in hardware
    - Methods would update the registers that map to the members of the object
    - Some design work would be needed to manage "method call" interfaces
    - Possibly provide a directive for the user to tell the compiler the object is not dynamically instantiated
- Better error reporting
    - Had to infer that sometimes a failure with no message implies HLS pragma issues
        - The tools should report the error message
    - Some sample "bad" error messages:
        - What does "Key 'delay_budget' not known in dictionary" mean?
        - Why does the interface type of one port matter for the interface type of an independent port?
        - "....data pack is only applied on source(port) or destination(variable)"
            - Which port or variable? Where is it missing?

REFERENCES

[1] Mann, S., and M. A. Ali. "The Fundamental Basis of HDR: Comparametric Equations." *High Dynamic Range Video*. 2016. 1-59.
[2] Ali, Mir Adnan, and Steve Mann. "Comparametric image compositing: Computationally efficient high dynamic range imaging." *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*. IEEE, 2012.
[3] Ai, Tao. *HDRchitecture: Real-Time Quantigraphic High Dynamic Range (HDR) Imaging on Field-Programmable Gate Array (FPGA) for WearCam (Wearable Camera)*. Diss. University of Toronto (Canada), 2014.
[4] https://www.repository.cam.ac.uk/handle/1810/261766
[5] Mann, Steve, and S. Mann. *Intelligent image processing*. IEEE, 2002.
[6] Perez B, Choi J, Xilinx AXI DMA, GitHub, https://github.com/rogumag/xilinx_axidma, Git: https://github.com/rogumag/xilinx_axidma.git

Appendix Material



Appendix B:Optimization Opportunities

This appendix outlines some optimization opportunities that have been identified as well as their expected theoretical performance increase.

Tiled streaming:

A drawback of the current design is the fact that each intermediate result is stored in main memory, which adds load to the memory interface. If the design stored intermediate results in BRAM, then the memory bus bandwidth would be effectively increased. By tiling input images (and by extension, CCRF computations), tiles of the intermediate results can be stored in BRAM. By making this change, each job only requires one main memory read in of each whole input image. This allows more BRAMs to be instantiated for downstream CCRF subtasks, which reduces their completion time. Figure B1 shows what the improvement might look like:

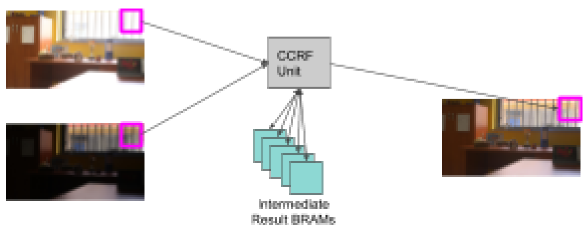


Figure B1 Tiled streaming at a high level. Images are never stored to DDR memory excluding the output

Expected theoretical improvement: Varies by LDR count and BRAM instantiation count. Greater benefit for higher LDR count: ~(LDR-1)x

<u>CCRF LUTs with exposure coeffient = 2</u>:
Add CCRF LUTs with values for double exposure delta (k=2) instead of a single exposure delta. This has the effect of pruning the CCRF subtask tree, which reduces the overall number of intermediate results to process. Refer to figure B2 for a demonstration of the CCRF graph change:

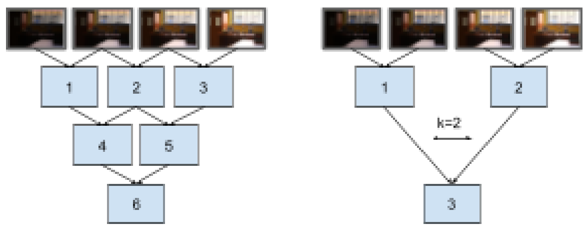


Figure B2 The resulting CCRF graph as a result of supporting more than one exposure delta CCRF LUT

Expected theoretical improvement: varies with LDR image count but no improvement for LDR=3, 2x for LDR=4 and ~2x for LDR=5, if the system is bottlenecked by main-memory reads

<u>More efficient color channel layout</u>:
Currently, an easy to use, but wasteful BGRA (Blue-Green-Red-Alpha) pixel format is used. The alpha channel is not used in HDR and all pixels will be set to opaque, therefore, the alpha channel is wasted memory.

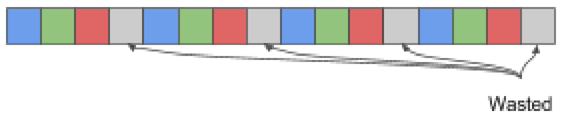


Figure B3 Each alpha channel is wasted in the pixel data.

As seen by figure B3, 25% of all pixel data is being wasted. This means that 25% of the data being transferred during memory reads is not useful. By packing the pixel data into a BGR format, effective throughput is increased by 33% (12 useful pixels vs 16 useful pixels) - see figure B4 where for each (128bit) read, all pixels contain useful channel information.

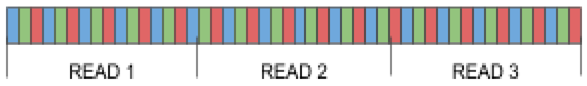


Figure B4 By removing the alpha channel, 100% of accessed data can be used

Expected theoretical improvement: ~1.33x

Appendix C: Optimizations In Progress

An optimized CCRF implementation is nearly completed. It provides significant performance improvements over the current version. Measured performance is 156.9fps for a 5 LDR image at a resolution of 1920x1080 and 226.4fps for a 3 LDR image at the same resolution. This optimized version does not include the proposed performance improvements as recommended in Appendix B: Optimization Opportunities.

Figure C1 demonstrates the current issue with the optimized implementation. Ignoring the oversaturation of the purple and teal (sea green), which has since been fixed, vertical lines are seen through the image. These lines are not distortions, rather they are lines of pixels that were stored at the wrong addresses in memory. Looking closely at the picture reveals that the pixels have been shifted to earlier locations in the picture. In particular, notice the clock on the right side of the image. Slices of it are stored in memory to the left of the clock.

Immediately before submission of this report, a possible cause of the issue was found in the scheduler. The scheduler was not updated to support wide, bursted reads (64 back to back 512bit word reads). The scheduler is responsible for setting aside scratchpad memory for the CCRF job. However, there are strict alignment requirements to successfully perform bursted reads/writes. The scheduler was not updated to account for this. Since the writing of the report, the fix as been applied and is in testing.

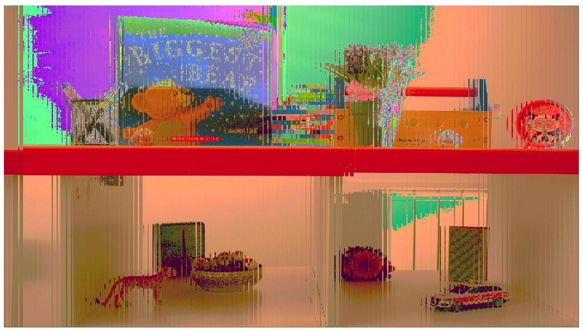


Figure C1 tonemapped image from high-performance version with bugs

README.md

This project implements an LDR (low dynamic range) to HDR (high dynamic range) image pipeline on an FPGA with an HLS (high level synthesis) implementation of the CCRF HDR image compositing algorithm as presented by Mann[__].

Contributors:
SeanNijjar
Qianfeng (Clark) Shen

Building The Project

There are 3 components to the project, each of them is built separately:

1) Software driver
    a) used for testing and development purposes
    b) Not mandatory for embedded designs
2) HLS IPs
    a) The core design components are implemented in (Vivado) HLS
3) Zynq Ultrascale+ solution
    a) Vivado project and IP interconnect

1) Software driver
From the project root, run the following make command:
```
> make driver_test_zynq
```
This will output an executable: driver_test_main
This executable is the test harness to run sample images through the FPGA pipeline

2) HLS IPs
Run the IP build tcl scripts located in the hls_ip/ directory. Each script corresponds to a separate IP. The following tcl scripts currently exist:

- ccrf_compute.tcl
- inverse_crf.tcl
- job_timer.tcl
- scheduler.tcl
- dispatcher.tcl
- job_result_notifier.tcl
- run_ccrf.tcl
- scheduler_top_level.tcl

3) Zynq Ultrascale+ solution
An archived vivado project is provided at at vivado_project/zynq_ultrascale_ccrf_bursted.xpr.zip.

Use this project to generate a bitstream compatible with the Xilinx ZynqMP Ultrascale+ platform. For convenience, a bitstream file and bitstream binary file are provided at vivado_project/zynq_ultrascale_ccrf_bursted.bit and vivado_project/zynq_ultrascale_ccrf_bursted.bit.bin, respectively.

Running The CCRF Engine

To run the ccrf engine, run the software driver, and provide 2 arguments:

1) The directory of the input image set list
2) The image set list file name

Example
./driver_test_main /

Modifications for Embedded Designs

Although this project is designed with embedded platform use in mind, it currently is not implemented as one. To modify this project for embedded designs, several simple changes and additions need to be made.

The PL design currently interfaces with the PS over axidma. In an embedded design where a device like a camera is connected directly to the PL, the PS component can be completely removed. Where confusion arises, please reference the included vivado project.

1) In a modified project, include the following IPs:
    a) CCRFSchedulerTopLevel
    b) JobTimer [optional]
    c) CCRFScheduler
    d) CCRFDispatcher
    e) RunCCRF
    f) CCRF_Compute
    g) CCRF LUT BRAMs (refer to included project)
    h) JobResultsNotifier
    i) InverseCRF
    j) Inverse CRF LUT BRAMs (refer to included project)
2) Connect the IPs
    a) Refer to the included project as reference for how this should be done
3) Connect the memory ports (m_axi) for CCRF_Compute and InverseCRF to system memory (DDR) or whichever memory will contain LDR and HDR image buffers
4) Connect a hardware driver that sends HDR image compositing ‘job’ information (see below)

Driving the CCRF HDR pipeline:

The pipeline operates on “JobPackage”s. JobPackage is a predefined class in the job_package.hpp file. Pushing job package data to an HLS stream that drive the CCRF Scheduler Top Level will allow a user to use the CCRF engine in an embedded design.

Acknowledgements

This project leverages libaxidma, found at https://github.com/rogumag/xilinx_axidma.git for communication between the Zynq PS and PL.