ECE532 Final Report - HDMI Upscaler

Group 2
Friday, April 14, 2023
Ben Cheng
Jay Mohile
Leo Han
Yong Da Li

https://github.com/jamohile/G2_VideoUpscaler

Overview

Motivation

Our project is motivated by the consumer demand for high-quality video experiences whether it's consuming video content like TV shows, YouTube videos, or movies or video gaming. For existing video content, poor source quality, which is often the case in old digital videos, can limit the quality of the viewing experience due to low resolution and frame rate.

On the front of video gaming, consumers spend exorbitant amounts of money (upwards of 3000 CAD for a top-of-the-line GPU) to be able to game at higher frame rates and resolutions.

We sought to create an FPGA-based device that can target both of the aforementioned use cases by upscaling and increasing the frame rate of an input video stream.

Goals

Initial Goals

Our initial goals were to implement both frame interpolation (for generating additional frames and increasing video frame rate) and resolution upscaling. The target input resolution and frame rate was 24-bit colour 720p (1280 px by 720 px) at 30 frames-per-second (FPS) and the target output resolution was 24-bit colour 1080p (1920 px by 1080 px) at 60 FPS.

Re-scoped Goals

As we got close to a final implementation, we hit DDR bandwidth limitations that made frame interpolation infeasible. Although our hand-calculations showed sufficient bandwidth, we hadn't accounted for non-idealities due to contention, timing requirements, etc.

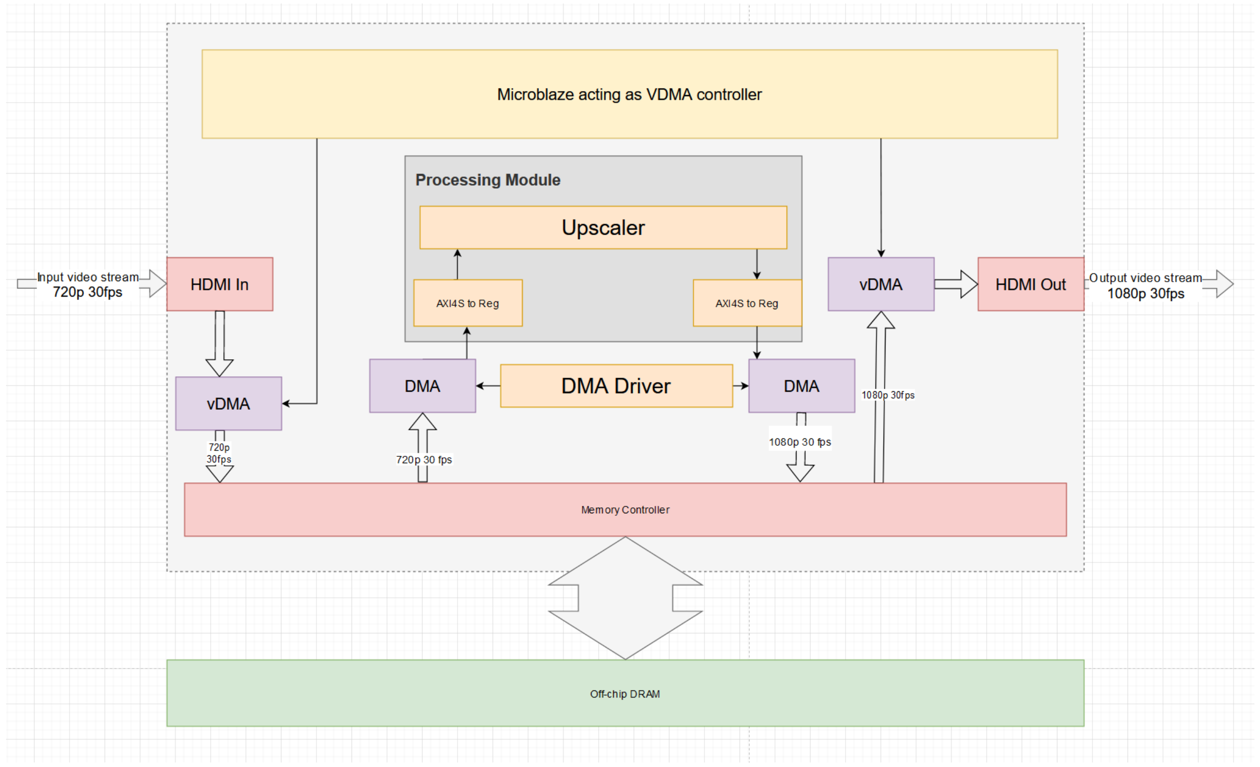
This caused us to rescope and drop frame-interpolation from the final system (though much of the required infrastructure had already been built).

Design and Architecture

Summary

Interfacing with video input and output peripherals on the board is handled by Digilent IPs (see "HDMI In" and "HDMI Out" blocks below). The input video stream is buffered to DDR memory via a VMDA (video DMA) which is initiated by the Microblaze. A custom-logic DMA driver controls Xilinx DMA modules to read chunks of multiple 2x2 pixel tiles which are upscaled to the same number of 3x3 pixel tiles and written back to DDR memory. The upscaled frame is then read by a VDMA and output as an DVI-over-HDMI video stream.

Block Diagram



IPs Used

Custom (47 % by area utilization)	Pre-Canned
Combinational Upscaler	dvi2rgb and rgb2dvi
Low-Latency DMA Driver (HLS)	Video-In to AXI4-Stream
Stream Buffer (Input / Output)	AXI4-Stream to Video-Out
Transposer / Inverse Transposer	AXI4-Stream to Video-Out
	Video Timing Controller
	AXI VDMA
	MicroBlaze soft-processor - MicroBlaze Debug Module - Processor System Reset - MicroBlaze Local Memory
	MIG7 Memory Interface Generator
	AXI DMA
	AXI Interconnect
	System ILA and Native ILA (debugging)
	AXI Uartlite
	AXI GPIO

Outcome

Feature Changes

The key feature change from the original design specifications was the removal of frame interpolation. The removal of this feature was necessitated by the fact that we were unable to meet the memory read/write bandwidth requirements required. With frame interpolation for a 720p 30FPS input and 1080p 60FPS output, we would need to read one 720p frame (1280 px by 720 px by 3 bytes = 2.8 MB) and write one 1080p frame (1920 px by 1080 px by 3 bytes = 6.2 MB) every 1/60 th of a second for a bandwidth requirement of 0.54 GB/s. These accesses are also in contention with VDMA input and output which writes one 720p frame and reads two 1080p frames every 1/30th of a second for a bandwidth requirement of 0.46 GB/s.

Performance

Our final prototype can perform upscaling from an HDMI input stream of 720p (1280 px by 720 px, 24-bit colour) to 1080p (1920 px by 1080 px, 24-bit colour) with good qualitative visual quality as observed on a computer monitor.

However, due to bugs interfacing between our AXI buffer custom logic module at Xilinx DMA modules, the output video exhibits a pattern of visual artifacts that is noticeable but does not significantly obfuscate the visual content.

Moreover, the resultant frame rate and latency were far worse than we had originally anticipated. The prototype can only support an output frame rate of around 5 FPS due to DDR bandwidth limitations associated with reading and writing to frame buffers.

Potential Improvements/Future Work

Since we do not need frame interpolation, we only need to store enough data to perform upscaling. Since an input tile is 2x2, we only need to store 1 row at most in memory from the input video stream (1280 px by 1 row by 3 bytes = 4 kB). For the output video stream, we need to store at most 2 rows (1920 px by 3 rows by 3 bytes = 17 kB). Combined, we would only need to store 21 kB of data at a time which is far below the 13 Mb or 1.6 MB of BRAM available on the Nexys Video board. This means that we can keep on data on chip and implement the upscaling as a stream processor that feeds in directly from the HDMI input stream and feeds directly to the HDMI output stream.

This would allow us to remove much complexity in the way of the DMAs and the DMA driver. Moreover, we would even be able to remove the VDMAs and directly stream the output of the DVI2rgb module into our stream processor and then stream the output back into an rgb2DVI module which outputs to an HDMI stream. By eliminating all off-chip memory accesses, we should be able to comfortably meet throughput requirements for an output of 1080p at 60 fps and potentially even higher resolutions and frame rates. Moreover, the input-output latency should also be much lower, allowing for greater usability in latency-critical applications such as video gaming.

Furthermore, the input-output relationship is deterministic between how many pixels streamed and how many streamed out and since we would no longer need to write to a frame buffer, the system would no longer have to track the state of which row within a frame is being processed.

The stream processor architecture would also be much more conducive to using HLS for development as HLS provides great support for datastream architectures via primitives such as hls::stream.

Project Schedule

	Original Milestone	Actual Milestone
1	System design and development environment - Complete top-level diagram - List of all external IP - Figure out memory/communication between all modules - Estimation of resources required - MicroBlaze pseudo-code - Integrate Vivado with git	- Realized we don't have enough BRAM to store two frames - Architectural level discussion about off-chip buffer, size of buffer, pipelining, and amount of parallel processing - Integrated Vivado with git We were on track for the first milestone. But in our architectural discussion, we realized we couldn't assess different designs without a working implementation because the timing, space, and bandwidth constraints are unknown.
2	Video Processing and Algorithm Research - develop IP to handle video into frame buffer - Develop IP to handle video out from frame buffer - Research upscaling and frame interpolation approaches (Python mockup)	Started work on HDMI input and output by following the Digilient tutorial. We tried programming the FPGA with the pre-packaged bitstream and recreating the block diagram design in a new project. The bitstream failed to detect HDMI input and the block diagram ran into timing failures in the DDR3 MIG. Researched image resolution upscaling and frame interpolation techniques, from basic to advanced. Further brainstormed architecture with draft block diagrams We had planned to get the HDMI input and output independently working by this milestone, but that didn't happen. At this point the HDMI

		video part was officially behind. The processing and algorithm part was still on track.
3	Video Passthrough and Algorithm Implementation - Integrate HDMI input and output to achieve passthrough using a frame buffer - Implement chosen algorithms in Verilog and verify correctness with testbench	HDMI input was partially verified. Data from a laptop’s HDMI output was DMA’ed into DDR3. It was verified by the memory debugged in the SDK and the one frame was reconstructed by trickling the data back to the laptop via MicroBlaze Uartlite, and using a Python script to create the image. HDMI output was far behind schedule. Thanks to a tip from a friend, we realized only the bitstream in the Vivado 2016.4 release of the Digilient tutorial worked. We were able to run the demo to control test patterns on an external monitor. This helped psychologically since we saw that it could work. However the block diagram fails to synthesize in 2016.4 due to Vivado mysteriously failing to recognize the MIG IP, even though it was listed in the project’s IP catalog. The HDMI output portion of the block diagram was manually recreated in Vivado 2018.3 but it was failing to compile due to numerous synthesis and timing errors. We started reading the Xilinx IP documentation for all the blocks used to really understand how the system worked. Previously we were blindly following the Diglient tutorial. We were able to use a pre-canned HLS IP Video Test Pattern Generator to output test patterns to an external monitor. The algorithm side produced a processing pipeline with well-defined memory interfaces and dataflows. A resolution upscaler Verilog module was created and simulated to produce the correct upscaled image. Algorithms still on track.
4	Frame Interpolation - Integrate HDMI passthrough with frame interpolation	HDMI input work was paused because HDMI output was lagging. We built from the know-good Video Test Pattern Generator HDMI output block diagram and ILA’ed all upstream stages until gradually resolved all issues. We achieved HDMI output from a DRAM buffer. MicroBlaze code was used to directly write to

		the VDMA registers to control this, rather than using the Xilinx provided VDMA wrapper library. The image processing timeline was integrated with both frame interpolation and upscaling. An AXI4-stream interface was chosen to communicate between the custom logic and the DDR memory. Work started to create AXI4-stream master and slave interfaces to integrate with the HDMI video.
5	Video Upscaling - Integrate HDMI passthrough with frame interpolation and frame upscaling	We combined the HDMI input and output designs to achieve passthrough. We also wrote MicroBlaze code to control all this, which a large chunk of went unmodified into the final project demo. We were 2-milestones behind on the HDMI part. We had to craft a fake EDID and hack the rgb2dvi Digilent IP to use this EDID to force the FPGA to broadcast itself as a 1280x720 @ 30Hz monitor. AXI4-Stream slave interface was verified to be correct. Single DMA scatter/gather was implemented using the MicroBlaze as a control. Work was expanded on this to support multiple DMA pipeline which is needed for the final version. We were still 2 milestones behind
6	Buffer Time and Bonus - Integrate simple computer-based control and telemetry to control upscaler settings	Modified the HDMI pass-through block diagram to support multiple frame buffers and generate a MicroBlaze interrupt on receiving an input frame. Integration work between the HDMI and processing pipeline buffers. We switched from scatter-gather DMA to simple DMA as we were running out of time. We were still 2 milestones behind, but managed to simplify our design and continued working on it throughout the week to have a valid prototype during demo.

Learning from this experience, there are three areas in which our original plan could have been improved. Some unrealistic planning led to significantly increased work later in the project.

First, our original plan assumed that video passthrough would be achievable in only 2 weeks. In reality, we significantly underestimated the difficulty of working with the HDMI specification. While significant pre-canned IP was utilized, there were significant tooling, integration, and configuration challenges that caused schedule slip. Building on this, streaming video generates extremely high bandwidth traffic — making it very difficult to probe and debug at the hardware level. In our original plan, this passthrough functionality was on the critical path; in the future, we'll make sure to prevent significant blocking issues like this.

Second, our original plan left large portions of system integration as a "later" problem. While high-level interfaces were agreed upon (e.g N-sized buffers, pixel-wise memory regions), this proved to be insufficient detail. In reality, minute factors such as the bit-width of buffer read transactions had significant knock-on effects during integration. This required us to build large amounts of "glue" that may have otherwise been avoided, and reduced the accuracy of some earlier planning (e.g bandwidth estimation). The takeaway from this is to spend significant time designing the interfaces, before splitting off to do independent component-level work.

Finally, we underestimated how slow the hardware design cycle can be. Whereas software projects can often iterate at a seconds-minutes cadence, seeing the result of a minute hardware change could require up to an hour. This led to tedious debugging and significant idle time that may have been better spent through more efficient planning. The key takeaway from this stage is that hardware operates on a very different clock than software, and requires an intentional, premeditated design process to be productive.

Description of the Blocks

Pre-Existing IP

The HDMI video input and output IPs were used normally, as was seen in the Digilent HDMI tutorial. The flow of input image data is:

1. HDMI input to dvi2rgb IP. Output data as 24-bit parallel color data with 3 synchronization signals (pixel clock, horizontal sync, vertical sync)
2. Dvi2rgb to Video-in to AXI4-stream. Converted 24-bit parallel color data into AXI4-stream
3. Axi4-stream to VDMA.
4. VDMA to AXI interconnect
5. AXI interconnect to DMA (controlled by MicroBlaze)
6. DMA to MIG
7. MIG to DDR3 RAM

Then the flow of output image data is exactly the same, but in reverse:

1. DDR3 RAM to MIG
2. MIG to DMA (controlled by MicroBlaze)
3. DMA to AXI interconnect
4. AXI Interconnect to AXI4-stream to video out. Converts the AXI4-stream data into 24-bit parallel color data.
    a. Paired with Video Timing Controller to generate the 3 required synchronization signals (pixel clock, horizontal sync, vertical sync)
5. Video Out + timing signals to rgb2dvi IP
6. Rgb2dvi to HDMI output

One “hack” we used is to connect a GPIO block’s input pins to the slave-to-memory-mapped (s2mm) pointer out pins of the VDMA block. These pins toggle when new image data is received. The GPIO was configured to generate an interrupt on any change of input pins. So we used this GPIO block to generate an interrupt on receiving an incoming frame. This was used by the MicroBlaze and custom DMA logic to synchronize when we feed video data into our custom processing block.

Another “hack” we used was to modify the EDID file that the dvi2rgb Digilient IP block uses. When you plug an HDMI cable from a source (your laptop) into a monitor (in this case, the FPGA is acting like a monitor), there is an I2C exchange of information that tells the laptop what screen resolution and frame rate are valid. Since we wanted to demonstrate 720p to 1080p upscaling, we had to force the laptop to output 720p @ 30Hz video to the FPGA. We crafted our own EDID magic string and put that in source file that the dvi2rgb IP uses. Thus we tricked our laptop to think that we’re connecting to a 720x1280 @ 30Hz monitor when we plug into the HDMI input of the FPGA.

Custom IP

The following custom IP was developed for this project. For simplicity, the block-diagram aliases some of these, ignoring macro-architecturally insignificant utility modules. Each of the modules presented here has an accompanying test-bench that exercises a variety of expected and edge cases. Modules were developed using a mix of SystemVerilog/Verilog, while testbenches exclusively used SystemVerilog.

Block Diagram Module	Name	Function
Upscaler	**CellUpscaler**	Converts a 2x2 section of an image (input) to a 3x3 output using Bilinear Interpolation. This can be used as part of any 1.5x upscaling scheme (e.g 720 to 1080).
Upscaler	**ChunkUpscaler**	Upscales many cells (2x2) in parallel, by extracting from a correctly formatted pixel buffer.
Upscaler	**ChunkInterpolator**	Interpolates between two sets of input cells, to generate an equally sized set of intermediate pixels.
Upscaler	**PixelwiseAverage**	Individually averages all color channels of an input pixel.
Upscaler	**Tranposer**	Converts data from pixel-wise format (used by HDMI in/out) to cell-wise, required by video processing modules.
Upscaler	**InverseTransposer**	Inverse of Transpose.
Upscaler	**Processor**	Combines upscale and interpolation blocks, to perform both in a highly parallel way.
AXI4S-to-Reg	**StreamInputBuffer**	An AXIS slave that combines several data-beats across several transactions (each fixed bit width) into a single register of arbitrary bit width, on a fully parameterized basis.
AXI4S-to-Reg	**StreamOutputBuffe r**	The inverse of InputBuffer, generates an AXIS master to output an arbitrary width buffer over several AXI transactions.
DMA Controller	**DMA Controller**	Sequentially initiates several AXI DMA operations to achieve a form of scatter gather. Microblaze-orchestrated DMAs were too slow, so a hardware approach was required.

Description of Design Tree

https://github.com/jamohile/G2_VideoUpscaler

Complete Verilog source code is in src/vivado. Libraries (ex. board files) are in src/vivado-library Since we Vivado HLS, the files for that are in the src/dma_driver_hls. The src/integrated_upscaling contains the code used for the final project. The other folders in src/ are intermediate snapshots of progress

```
.
├── docs
├── images
├── misc
├── src
│   ├── dma_driver_hls
│   ├── dma_passthrough
│   ├── hdmi_inout
│   ├── integrated_upscaling
│   ├── vivado
│   └── vivado-library
└── utils
    └── image
```

Tips and Tricks

- Leave margins when calculating bandwidth. Consider all potential bottlenecks when designing data pipeline. E.g. in reading and writing via DMA, we encountered limitations with Microblaze software control latency, MIG/DDR bandwidth limitations, AXI bus contention, etc.
- Read the Xilinx IP docs if you're stuck, don't blindly follow the tutorials because they might not always work on your Vivado version
- Use Generous Factors of Safety When Estimating Bandwidth
- Double-check for Bottlenecks at Every Stage