UNIVERSITY OF ARKANSAS

Accelerating DICe via FPGAs

Honeywell FM&T Sponsored Research

Presenter: Keaten N. Stokke

Advisor: Dr. David Andrews

September 6th, 2019

UNIVERSITY OF ARKANSAS

Thank You Honeywell!

- Provided funding for 1 Ph.D. and 1 Master’s student for 3 semesters
- Learned a tremendous amount of engineering skills
- Submitted our first ever paper to the ReConFig conference
- Granted me the ability and inspiration to pursue graduate school

Special shoutout to Mr. Dennis Stanley!!!

Overview

- **Objective**
- Results
- Challenges
- Additional & Future Works
- Conclusion
- Project Decomposition
  - Phase 1: Understanding DICe
  - Phase 2: Identifying Key Functions
  - Phase 3: Architecture Design
  - Phase 4: Client [PC]
  - Phase 5: Server [FPGA - PS]
  - Phase 6: Application [FPGA - PL]
  - Phase 7: Testing & Improvements

Objective

- Investigate hardware acceleration of the Digital Image Correlation engine (DICe) software from Sandia National Laboratories
- The use of FPGAs gave us a few possible approaches to this problem

1) Vivado HLS - “C to Gates”
2) Convert DICe to Verilog
3) Port to the HMC

We chose option two and opted to convert key functions in DICe to Verilog!

Objective

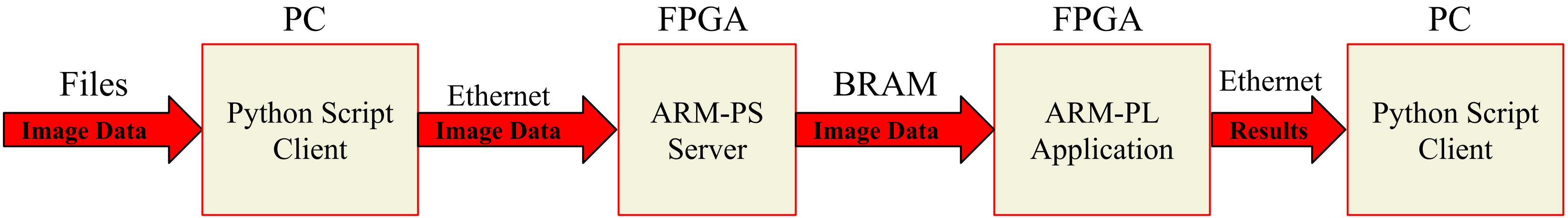
- We took the route of hardcoding key DICe functions from C++ to Verilog because it gave us more options
  - Allowed us to target FPGAs
  - Allowed for the potential use of the HMC by Micron
- The Hybrid Memory Cube (HMC) would be a viable option for the future to eliminate Ethernet data transfer
- DICe source code required our full attention

Overview

- Objective
- **Results**
- Challenges
- Additional & Future Works
- Conclusion
- Project Decomposition
  - Phase 1: Understanding DICe
  - Phase 2: Identifying Key Functions
  - Phase 3: Architecture Design
  - Phase 4: Client [PC]
  - Phase 5: Server [FPGA - PS]
  - Phase 6: Application [FPGA - PL]
  - Phase 7: Testing & Improvements

Results

- Created a working end-to-end DICe hardware accelerator prototype
  - A Python script is run on the PC to transmit available frames to FPGA
  - The FPGAs LWIP server receives the frame, converts to IEEE 754
  - The processed frame is then written into BRAM, a start signal is sent
  - Correlation on the PL starts, server requests new frame if needed
  - Results are saved to BRAM, read from the server, and sent to the client
  - Client receives results, converts and formats, writes to result .txt files

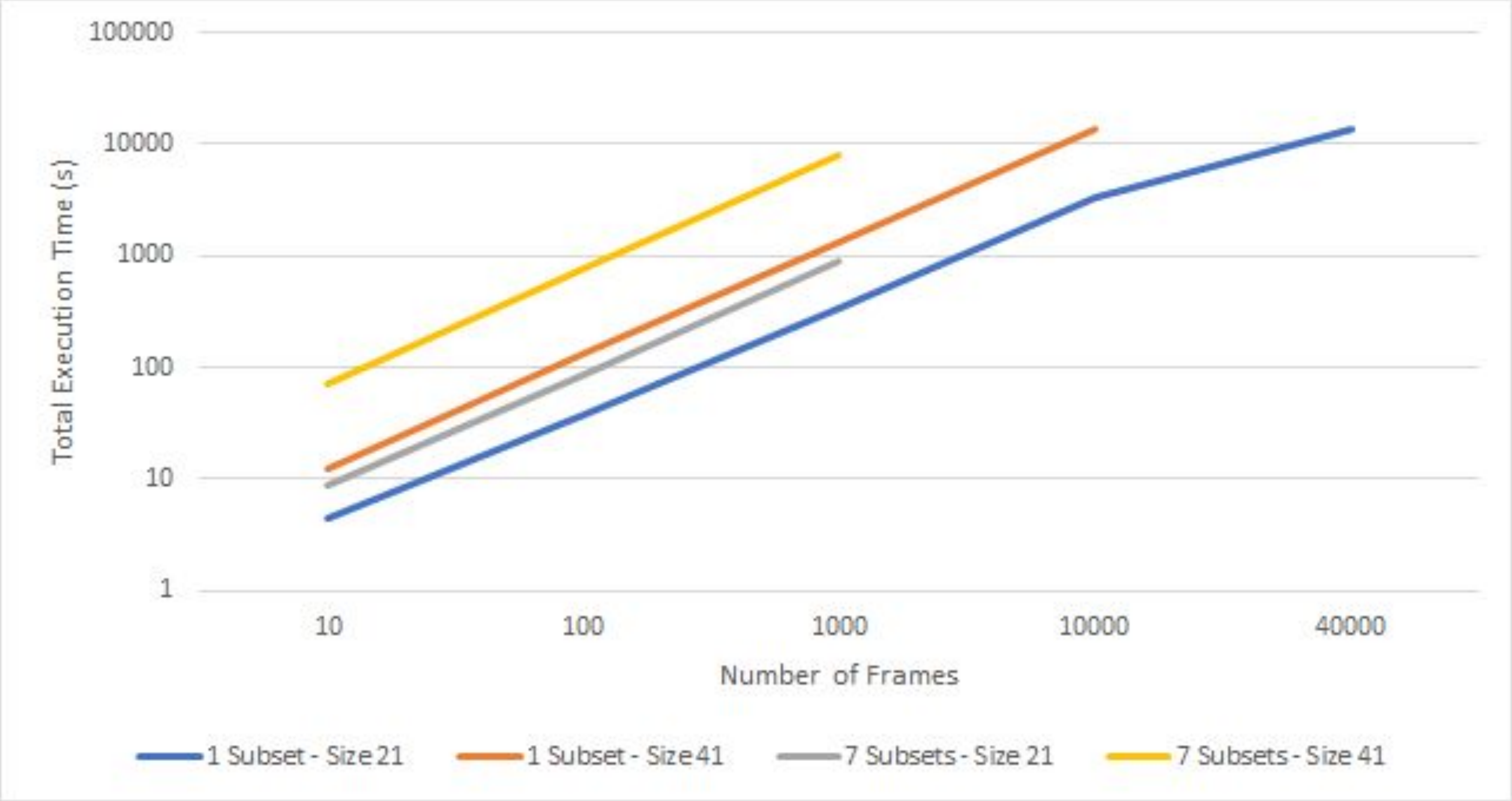


UNIVERSITY OF ARKANSAS

Results

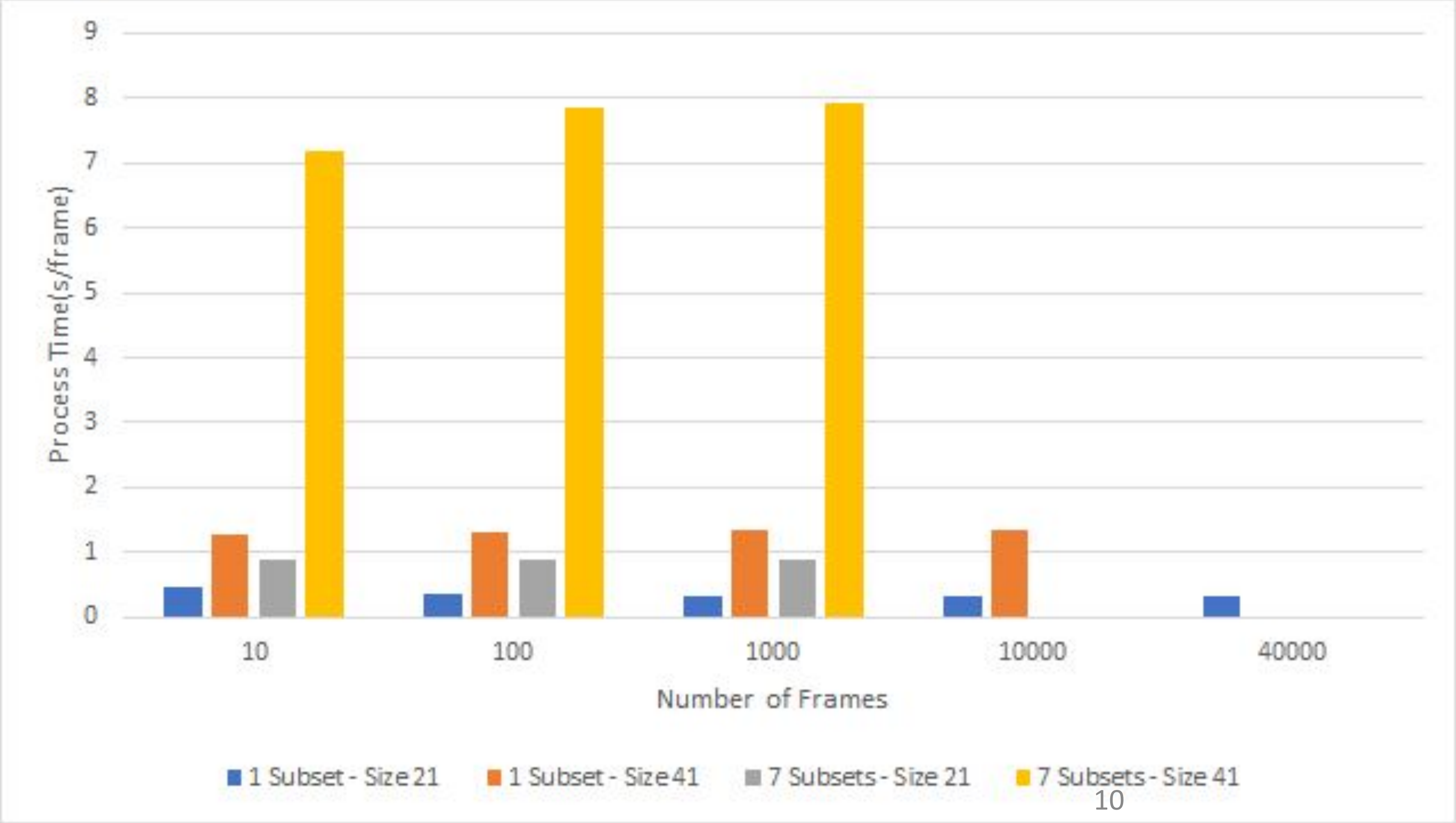
- Ran successful tests of this system with real images
  - Test with 1 subset
    - 40,000 frames, 21x21 subset (control)
    - 10,000 frames, 41x41 subset (quadrupled)
  - Test with 7 subsets
    - 1,000 frames, 21x21 subset (control)
    - 1,000 frames, 41x41 subset (quadrupled)

Results

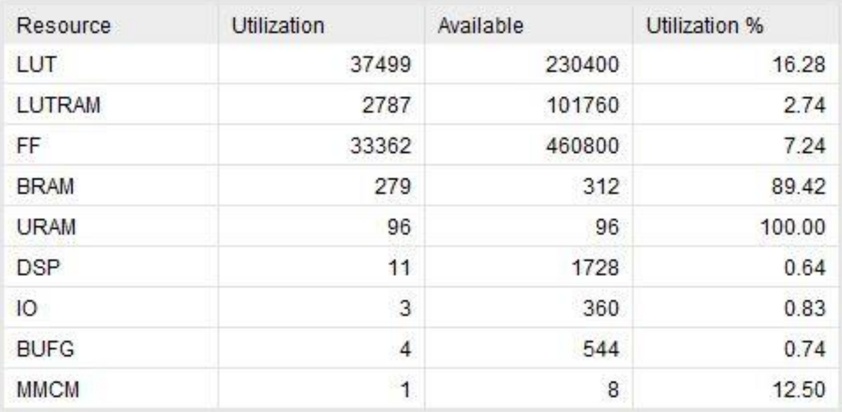


100000
10000
1000
100
10
1
Total Execution Time (s)
10
100
1000
10000
40000
Number of Frames
1 Subset - Size 21
1 Subset - Size 41
7 Subsets - Size 21
7 Subsets - Size 41

Results

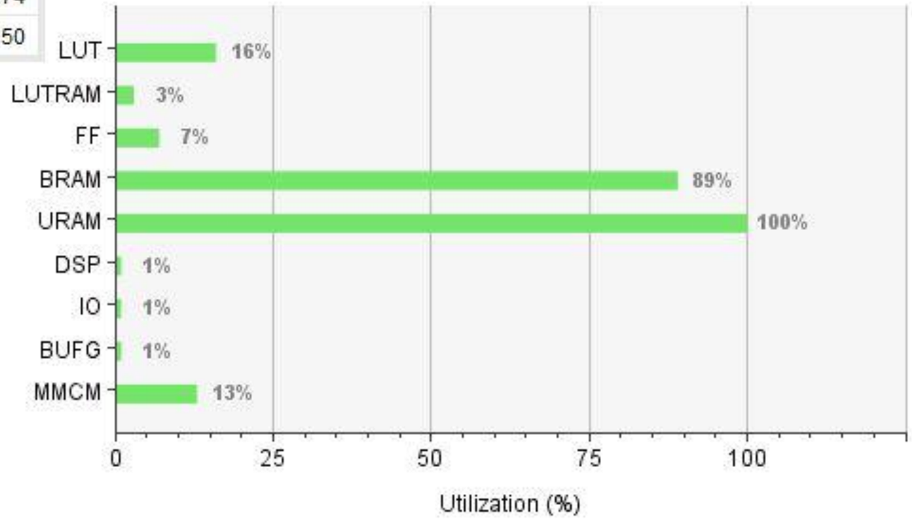

Process Time(s/frame)
9
8
7
6
5
4
3
2
1
0
10
100
1000
10000
40000
Number of Frames
1 Subset - Size 21
1 Subset - Size 41
7 Subsets - Size 21
7 Subsets - Size 41

Results



Resource	Utilization	Available	Utilization %
LUT	37499	230400	16.28
LUTRAM	2787	101760	2.74
FF	33362	460800	7.24
BRAM	279	312	89.42
URAM	96	96	100.00
DSP	11	1728	0.64
IO	3	360	0.83
BUFG	4	544	0.74
MMCM	1	8	12.50

ZCU104 Resource Utilization



Results

Test Case: 2 Frames (448x232), 1 Subset (21x21)					
Test Run	**DICe GUI**	**Simulation (Current Design)**	**Simulation (Pipelined Design)**	**Demo (Total)**	**Demo (Ethernet)**
Execution Time:	**116 ms**	**138 ms**	**55 ms**	**947 ms**	**933 ms**

- Our current demo is 8.16x slower than the DICe GUI
  - Correlation: 14%, Client: 1.5%, **Ethernet: 84.5%**
  - Our simulations are only 1.19x slower than the DICe GUI
- With a pipelined design, it could be 2.1x faster than the DICe GUI
  - We would need more BRAM resources to save the gradients

Results

Ethernet Capability

FPGA	Processor	Ethernet IP	Speed (Mbps)	Actual Speed (Mbps)	Tested Speed (Mbps)
Virtex-7 [VC707]	MicroBlaze	AXI Ethernet Subsystem (SGMII)	1000	X	X
Kintex-7 [KC705]	MicroBlaze	AXI Ethernet Lite (MII)	1000	56	8
UltraScale+ MPSoC [ZCU104]	**Zynq**	**PHY (RGMII)**	**1000**	**950**	**+/- 50 (*500)**

- After many attempts, Virtex-7 failed to work and Kintex-7 was slow

Zynq is the way to go!

Results

KC705 vs. ZCU104

Total Simulated Execution Time vs. DICe GUI

- ZCU104: 12x faster than KC705
    - ○ Ethernet speed can theoretically be 19x faster [~50Mbps → ~950Mbps]
- Note that the simulated FPGA design is 6.4x faster than the GUI (when correlating 2 frames)

Test Case: 64x48 Design	Total Transfer Time	Total FPGA Correlation Time	Total GUI Correlation Time	Total GUI Execution Time	FPGA vs. GUI
KC705	1.25 s	0.0025 s	0.016 s	0.058 s	25x slower
ZCU104	0.104 s	0.0025 s	0.016 s	0.058 s	5.6x slower

Results

Verification

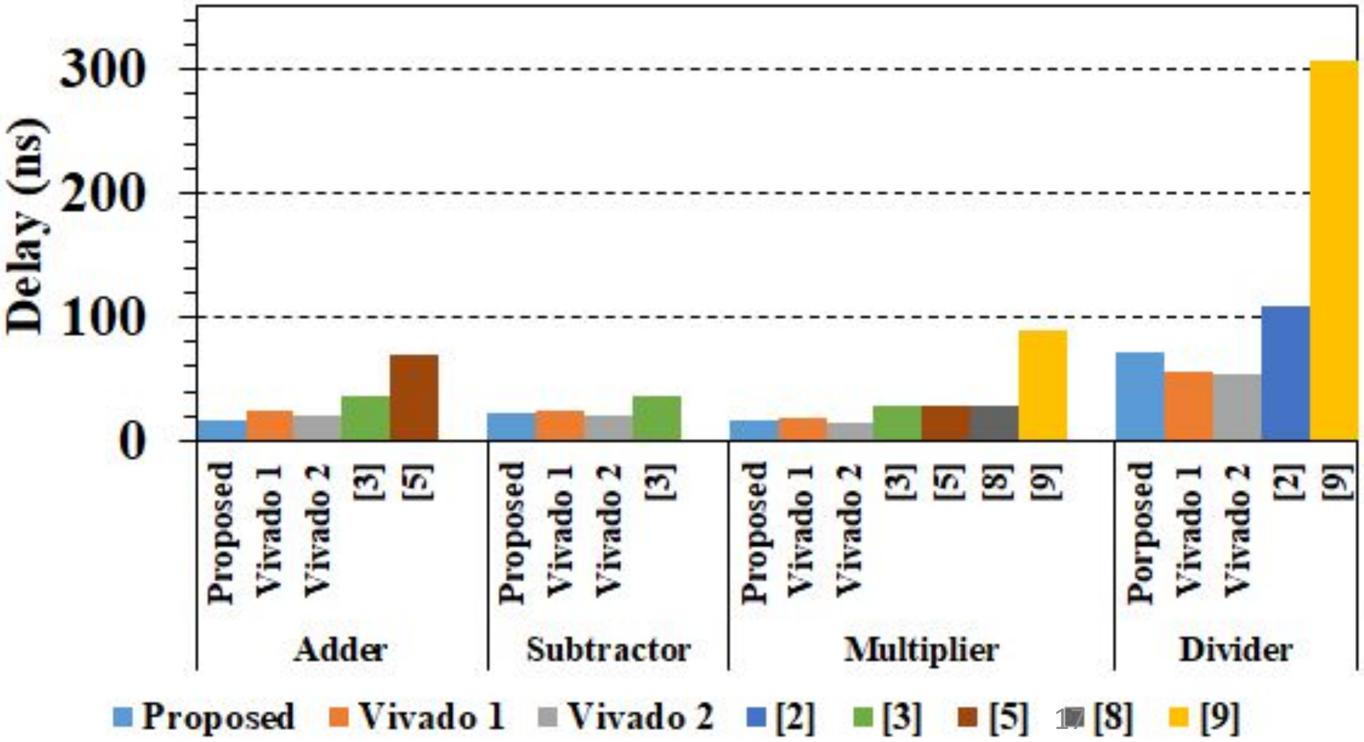
- Two major drawbacks exist with the current state of our application
  1) Results are unverified
  2) Some of the results haven't been implemented (beta, sigma)
- The results we do have are not correct when compared to the outputs of the DICe GUI, we have been working to correct this
- Currently, we produce the results Displacement X, Displacement Y, and Rotation Z.

Results

- Created a library of floating-point operations that resulted in a paper
  - Submitted to the ReConFig conference
  - Functions: addition, subtraction, multiplication, division, sine, cosine, **arcsine, arccosine**
- Aimed at reducing latency for each operation
  - Previous works emphasize on pipelining which wasn’t useful
- Novel work is based on Finite-State Machines (FSMs)
  - Operations can be locally embedded in custom IP
  - Runs at 150 MHz, improved functions work at 200 MHz

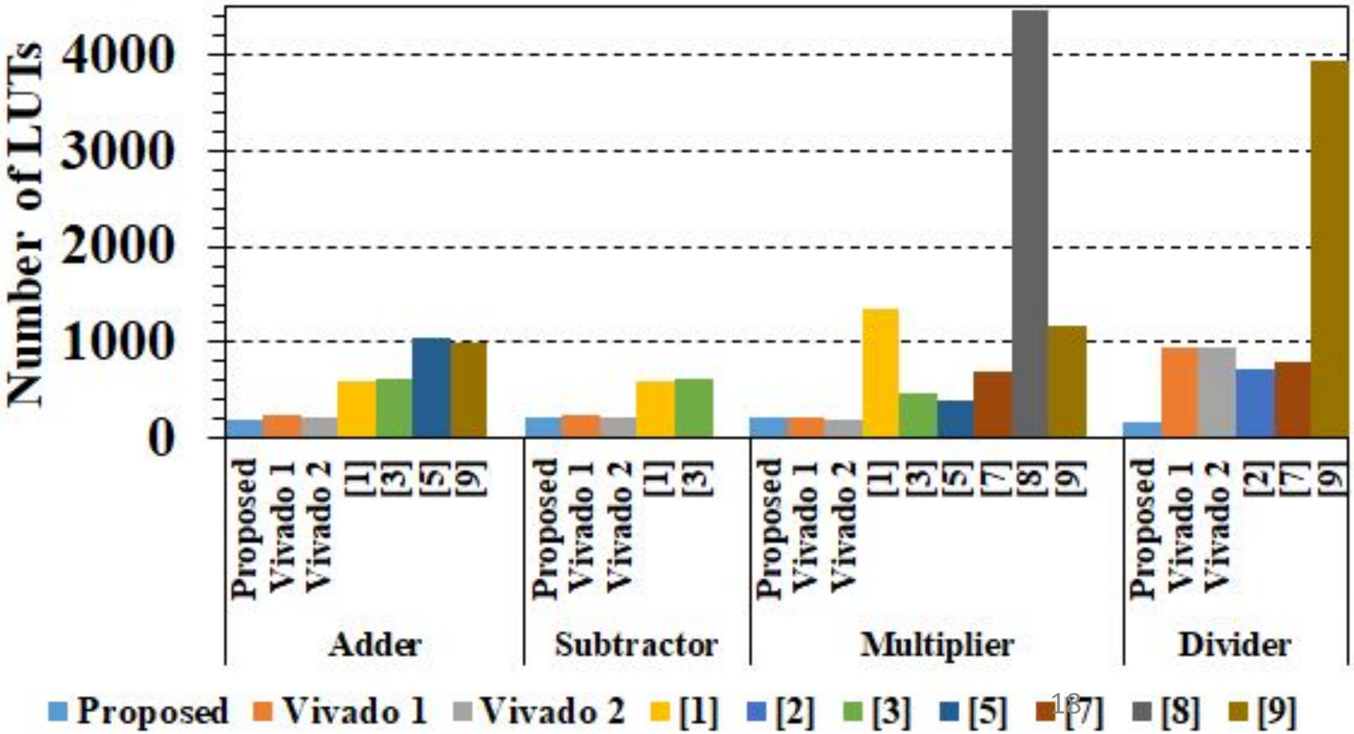
Results

Floating-Point Library: Delay Comparison



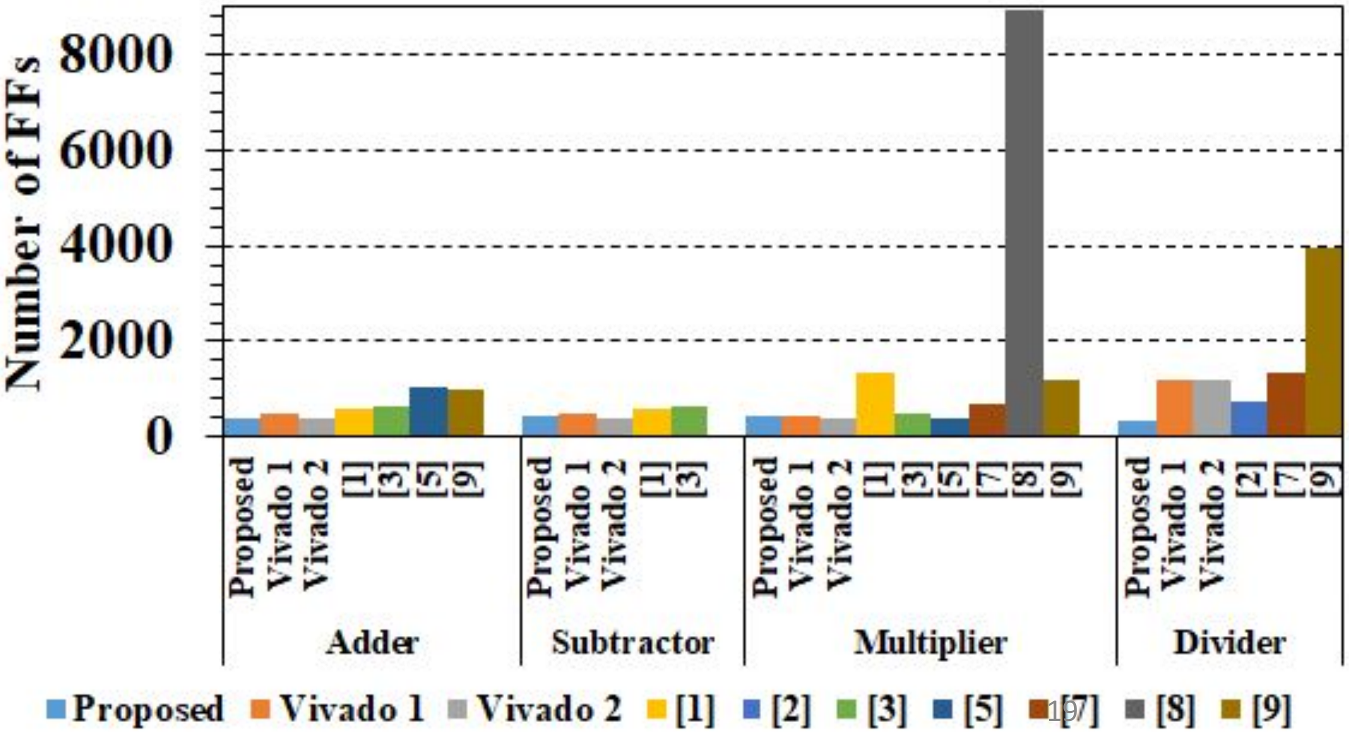
Results

Floating-Point Library: LUTs Comparison



Results

Floating-Point Library: FFs Comparison



Results

Floating-Point Arithmetic Functions

Function	# Clks	Freq (MHz)	Delay (ns)	DSP	# Slices	Power (W)	(F) / (D*S*M)
Adder	**4**	**150**	**23.1**	**-**	**40**	**0.37**	**129.8**
Subtractor	**5**	**150**	**29.7**	**-**	**44**	**0.37**	**84.1**
Multiplier	**5**	**150**	**29.7**	✓	**56**	**0.37**	**72.1**
Divider	**30**	**150**	**194.7**	**-**	**39**	**0.37**	**15.4**

Results

Floating-Point Trigonometric Functions

Function	# Clks	Freq (MHz)	Delay (ns)	DSP	# Slices	Power (W)	(F) / (D*S*M)
Sine	**92**	**150**	**603.9**	✓	**351**	**0.39**	**0.53**
Cosine	**86**	**150**	**564.3**	✓	**328**	**0.39**	**0.61**
Arcsine	**97**	**150**	**639.6**	✓	**365**	**0.39**	**0.48**
Arccosine	**105**	**150**	**689.7**	✓	**339**	**0.39**	**0.48**

Results

What comes in the box...

- Documentation for all IP
  - Each IP comes with its own documentation that provides a description of what its function is
- FPGA security report
  - Additionally, a report is provided on FPGA security that highlights the capabilities of the ZCU104
- Everything is available on GitHub
  - github.com/keatenstokke/DICE

Results

Knowledge Gained

- How to use the Vivado tools (2015.4 and 2018.3)
- Working with series-7 FPGAs (Kintex-7 and Virtex-7)
- Working with Zynq FPGAs (Zybo and UltraScale+ MPSoC)
  - How to target multiple cores simultaneously for processing
  - How to transmit data from PS → PL
- How to generate processor interrupts
- How to create and design custom IPs
- How to interface with memories (BRAM, DRAM, FLASH)
- Learned to setup Ethernet data transmission
- Learned how to interface with the LWIP
- Debugging HDL with VIO

… And the list goes on!

Overview

- Objective
- Results
- **Challenges**
- Additional & Future Works
- Conclusion
- Project Decomposition
  - Phase 1: Understanding DICe
  - Phase 2: Identifying Key Functions
  - Phase 3: Architecture Design
  - Phase 4: Client [PC]
  - Phase 5: Server [FPGA - PS]
  - Phase 6: Application [FPGA - PL]
  - Phase 7: Testing & Improvements

Challenges

- Assembling the design's infrastructure
  - When images should be sent, when IPs should start/end/reset
  - Establishing communication PC → FPGA-PS → FPGA-PL
- Ethernet was difficult to implement on the KC705/VC707
  - A lot of time was spent on this before switching to the ZCU104
- LWIP echo server requires a lot of networking knowledge
  - ZCU104 is capable of 1 Gbps transmission speeds
  - We hit a max of 500 Mbps transmission speeds
  - On average, our application has +/- 50 Mbps transmission speeds

Challenges

- BRAM limitations constrained us
  - Currently working with ¼ sized frame (448x232)
- Image preprocessing was tricky
  - Convert each pixel to IEEE 754 floating-point format
  - Started as a Python script, migrated to the FPGAs ARM core
- Floating-point functions took time
  - We created a library of floating-point arithmetic operations
- Each analysis mode in DICe has a lot of user-defined parameters and features

Challenges

- Validation has been very difficult
    - Build source code from scratch with required libraries
    - Trace through dozens of functions and loops and manually cross-check numbers to results on FPGA
- How to accurately place a subset within the image
    - Current design requires manually listed pixel numbers for squares and circles
- The DICe source code is **extremely** dense
    - 98 total source code files that are being updated every week

Overview

- Objective
- Results
- Challenges
- **Additional & Future Works**
- Conclusion
- Project Decomposition
  - Phase 1: Understanding DICe
  - Phase 2: Identifying Key Functions
  - Phase 3: Architecture Design
  - Phase 4: Client [PC]
  - Phase 5: Server [FPGA - PS]
  - Phase 6: Application [FPGA - PL]
  - Phase 7: Testing & Improvements

Additional Work

- Additional work that is needed to make the project fully functional
    - Achieve at least 1 Gbps Ethernet transmission speeds
    - Add support for the missing results (beta, gamma, sigma)
    - Add support for the additional optimization/correlation method (simplex/robust)
    - Improved user-defined parameter support
    - Validation

Future Work

- Addition of all analysis modes (subset-based full field, global)
- Image obstructions
- Full sized frame support (896x464)
- Faster Ethernet (10Gb+), or port to HMC
- Direct link between camera and FPGA (video → frames)
- Subset improvements (greater size, greater quantity, extra shapes, improved selection) [current design meets requirements]
- Add parrelization and general optimizations within code

Overview

- Objective
- Results
- Challenges
- Additional & Future Works
- **Conclusion**
- Project Decomposition
  - Phase 1: Understanding DICe
  - Phase 2: Identifying Key Functions
  - Phase 3: Architecture Design
  - Phase 4: Client [PC]
  - Phase 5: Server [FPGA - PS]
  - Phase 6: Application [FPGA - PL]
  - Phase 7: Testing & Improvements

Conclusion

- Implemented an end-to-end hardware accelerator system for DICe
    - Performed correlation on up to 1,000 frames with multiple subsets, 40,000 frames with a single subset
- 100% BRAM utilization (38 Mb total)
- Approximately 6,500 lines of Verilog written
- Complete design, code, and documentation is available on GitHub
    - github.com/keatenstokke/DICE
- Development on this project will continue
    - Verify & Optimize

UNIVERSITY OF ARKANSAS

Thank You Again, Honeywell

Questions?

Honeywell

NSC
National Security Campus

UNIVERSITY OF ARKANSAS

Overview

- Objective
- Results
- Challenges
- Additional & Future Works
- Conclusion
- **Project Decomposition**
    - **Phase 1: Understanding DICe**
    - Phase 2: Identifying Key Functions
    - Phase 3: Architecture Design
    - Phase 4: Client [PC]
    - Phase 5: Server [FPGA - PS]
    - Phase 6: Application [FPGA - PL]
    - Phase 7: Testing & Improvements

Phase 1: Understanding DICe

- Traced through code to identify key functions and how they operate
  - Started with main.cpp and worked our way down
  - Hard to identify which functions belonged to the “tracking” analysis mode
- Code is **very dense**
  - Thousands of lines of code spread across 98 files
  - Lack of comments in the source code
  - No descriptions for functions or algorithms
- Building from the source code files was a hassle
  - Required the Trilinos library - contains 57 separate packages
  - Build instructions were not detailed, extra software required

Overview

- Objective
- Results
- Challenges
- Additional & Future Works
- Conclusion
- **Project Decomposition**
  - Phase 1: Understanding DICe
  - **Phase 2: Identifying Key Functions**
  - Phase 3: Architecture Design
  - Phase 4: Client [PC]
  - Phase 5: Server [FPGA - PS]
  - Phase 6: Application [FPGA - PL]
  - Phase 7: Testing & Improvements

Phase 2: Identifying Key Functions

Main Functions

main() → execute_correlation() → generic_correlation_routine() → computeUpdateFast() → initial_guess() → initialize_guess_4() → initialize() → interpolate_bilinear() → interpolate_grad_x_bilianer() & interpolate_grad_y_bilianer() → gamma_() → mean() → residuals_aff() → map_to_u_v_theta_aff() → map_aff() → test_for_convergence_aff() → save_fileds()

Phase 2: Identifying Key Functions

Main Functions

- **computeUpdateFast()**: Gradient-based optimization algorithm is done using a loop over the whole image to find the subset in the deformed image.
- **initial_guess()**: Called to start the optimization with a good first guess.
- **initialize_guess_4()**: Initializes the subset in the vicinity of the previous guess.
- **initialize()**: Initializes the work variables.
- **interpolate_bilinear()**: The easiest of ways to interpolate the pixel intensity values for non-pixel locations. Bilinear interpolation uses values of only the 4 nearest pixels, located in diagonal directions from a given pixel, in order to find the appropriate color intensity values of that pixel and tries to achieve a best approximation of a pixel's intensity based on the values at surrounding pixels.

Phase 2: Identifying Key Functions

Main functions

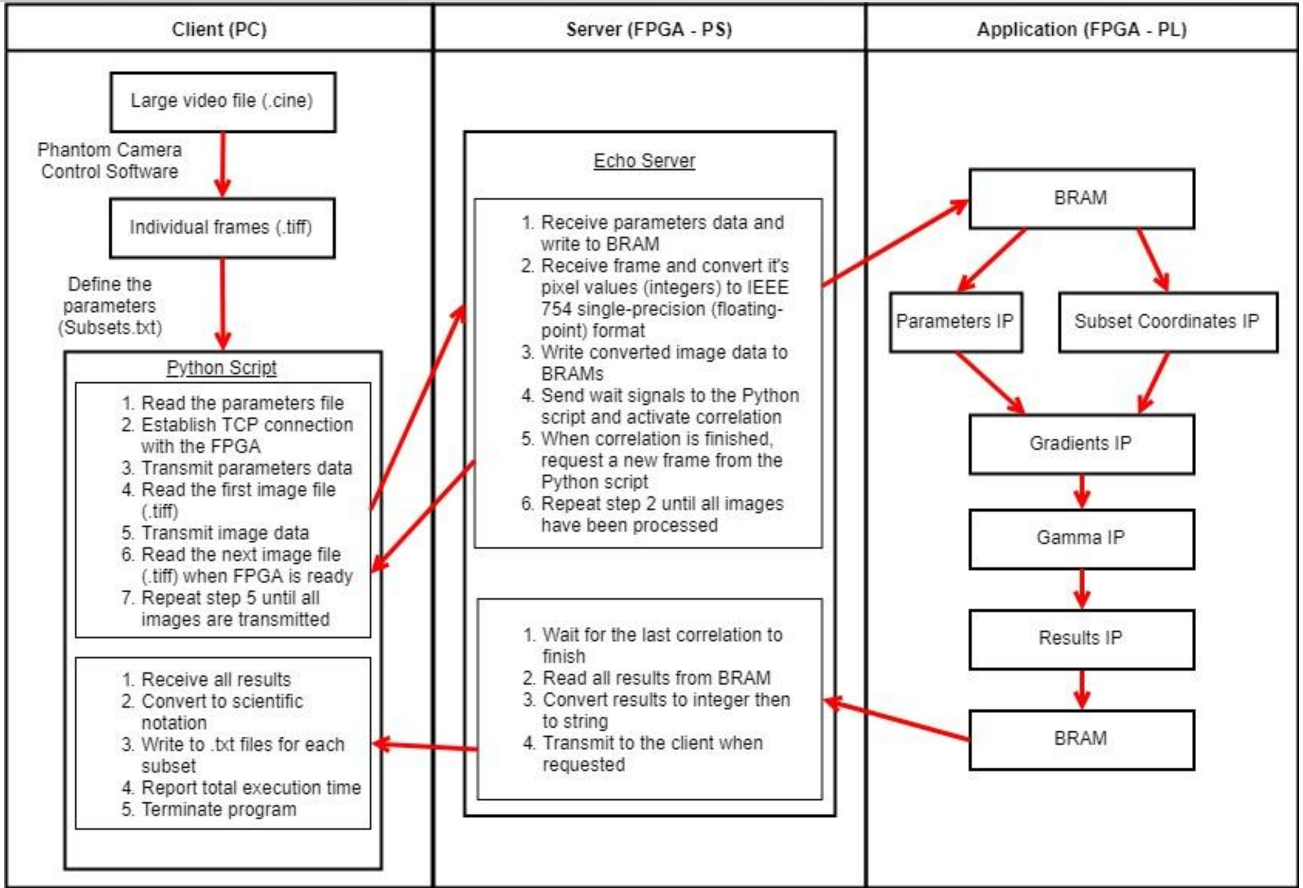
- **interpolate_grad_x_bilianer() & interpolate_grad_y_bilianer()**: The gradients also need to be interpolated when the values are requested at non-integer locations.
- **gamma_()**: Returns the gamma correlation value between the reference and deformed subsets. It has a for-loop over the number of pixels per subset and computes the difference between the intensities of the pixels and the mean value of the intensities for that subset.
- **mean()**: Computes the mean intensity value for the reference or deformed subset.
- **residuals_aff()**: Computes the residuals for the shape function (affine). Residuals are a measure of mismatch.
- **map_to_u_v_theta_aff()**: Converts the current map parameters to u, v, and theta.
- **map_aff()**: Maps the input coordinates to the output coordinates.
- **test_for_convergence_aff()**: Returns true if the solution is converged.
- **save_fileds()**: Saves off the parameters to the correct fields.

UNIVERSITY OF ARKANSAS

Overview

- Objective
- Results
- Challenges
- Additional & Future Works
- Conclusion
- **Project Decomposition**
    - Phase 1: Understanding DICe
    - Phase 2: Identifying Key Functions
    - **Phase 3: Architecture Design**
    - Phase 4: Client [PC]
    - Phase 5: Server [FPGA - PS]
    - Phase 6: Application [FPGA - PL]
    - Phase 7: Testing & Improvements

Phase 3: Architecture Design



Client (PC)

Large video file (.cine)

Phantom Camera Control Software

Individual frames (.tiff)

Define the parameters (Subsets.txt)

Python Script

1. Read the parameters file
2. Establish TCP connection with the FPGA
3. Transmit parameters data
4. Read the first image file (.tiff)
5. Transmit image data
6. Read the next image file (.tiff) when FPGA is ready
7. Repeat step 5 until all images are transmitted

1. Receive all results
2. Convert to scientific notation
3. Write to .txt files for each subset
4. Report total execution time
5. Terminate program

Server (FPGA - PS)

Echo Server

1. Receive parameters data and write to BRAM
2. Receive frame and convert it's pixel values (integers) to IEEE 754 single-precision (floating-point) format
3. Write converted image data to BRAMs
4. Send wait signals to the Python script and activate correlation
5. When correlation is finished, request a new frame from the Python script
6. Repeat step 2 until all images have been processed

1. Wait for the last correlation to finish
2. Read all results from BRAM
3. Convert results to integer then to string
4. Transmit to the client when requested

Application (FPGA - PL)

BRAM

Parameters IP

Subset Coordinates IP

Gradients IP

Gamma IP

Results IP

BRAM

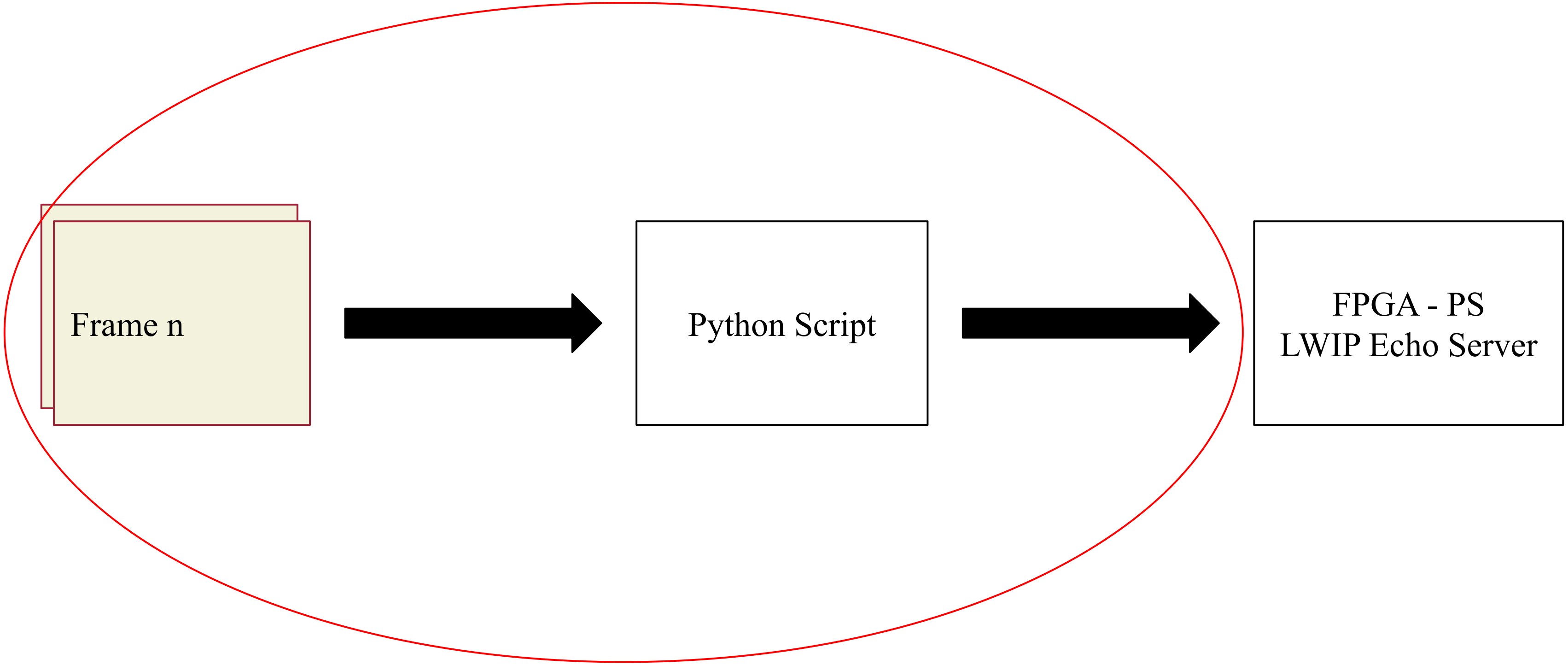
Phase 3: Architecture Design

- Client used packets to send signals to communicate with the server
  - Server reciprocated with the proper response
  - End of frame, wait, send new frame, process done, results
- Server performed read and writes to memory within the PL
  - IPs wrote to slave registers to communicate with the server
  - AXI slave registers, BRAM controllers
- Application IPs communicated directly with each other via wires
  - Block diagrams used to connect IPs together to send data

Overview

- Objective
- Results
- Challenges
- Additional & Future Works
- Conclusion
- **Project Decomposition**
  - Phase 1: Understanding DICe
  - Phase 2: Identifying Key Functions
  - Phase 3: Architecture Design
  - **Phase 4: Client [PC]**
  - Phase 5: Server [FPGA - PS]
  - Phase 6: Application [FPGA - PL]
  - Phase 7: Testing & Improvements

Phase 4: Client [PC]



Phase 4: Client [PC]

- A PC-based script was required to send the image data to the FPGA
- Python 2.7.15 was the language used
  - Great for prototyping - do more with less code
  - Versatile, easy to use, fast to develop
  - Lots of libraries available (Numpy, PIL, Bits, Codecs)
- The Python script acts as a TCP client
  - TCP is slower than UDP, but more reliable
  - We can’t afford the loss of even 1 pixel

Phase 4: Client [PC]

1) User runs script from terminal
   a) All frames are in a specified directory
   b) User-defined parameters are set (# of subsets, size, shape)
2) TCP connection is established with the FPGA
3) All parameters data is transmitted, begin looping through images
   a) Packet size is 1,344 bytes, 3 bytes pixel, 448 pixel per packet
   b) A total of 232 packets are sent for one 448x232 frame (¼ size)
4) Transmit each frame, wait until the server requests a new frame
5) Receive all results at the end of correlation, format and write to a file

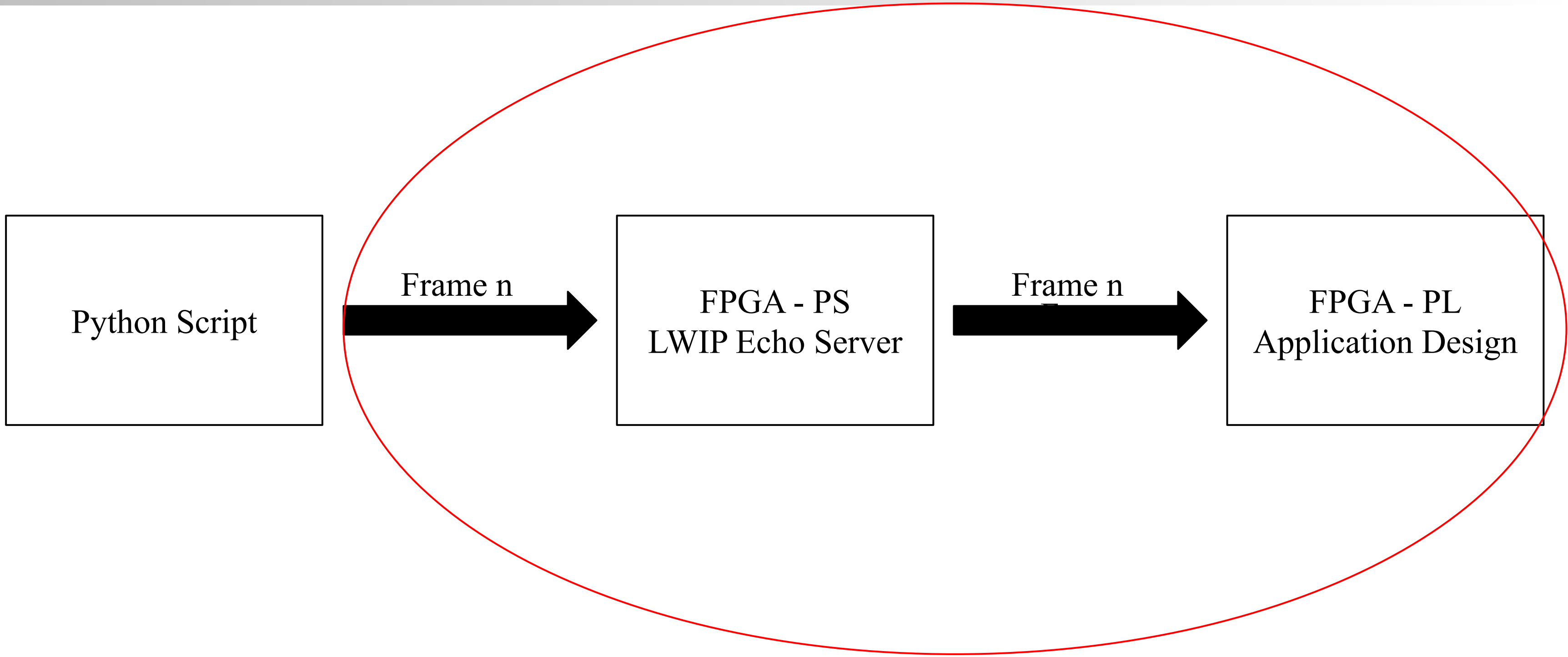
Phase 4: Client [PC]

- Originally the Python script was going to read in the (.cine) video file and split into individual frames (.tif)
    - The Phantom cameras have free software that already does this
- The original script converted the pixels from integer to IEEE 754
    - We moved this process to the FPGA and saw a 5.33x speedup
    - Sending 4x less data to FPGA, less time in loops processing pixel values
    - FPGA dedicates an ARM core to processing this

Overview

- Objective
- Results
- Challenges
- Additional & Future Works
- Conclusion
- **Project Decomposition**
  - Phase 1: Understanding DICe
  - Phase 2: Identifying Key Functions
  - Phase 3: Architecture Design
  - Phase 4: Client [PC]
  - **Phase 5: Server [FPGA - PS]**
  - Phase 6: Application [FPGA - PL]
  - Phase 7: Testing & Improvements

Phase 5: Server [FPGA - PS]



Python Script

Frame n

FPGA - PS
LWIP Echo Server

Frame n

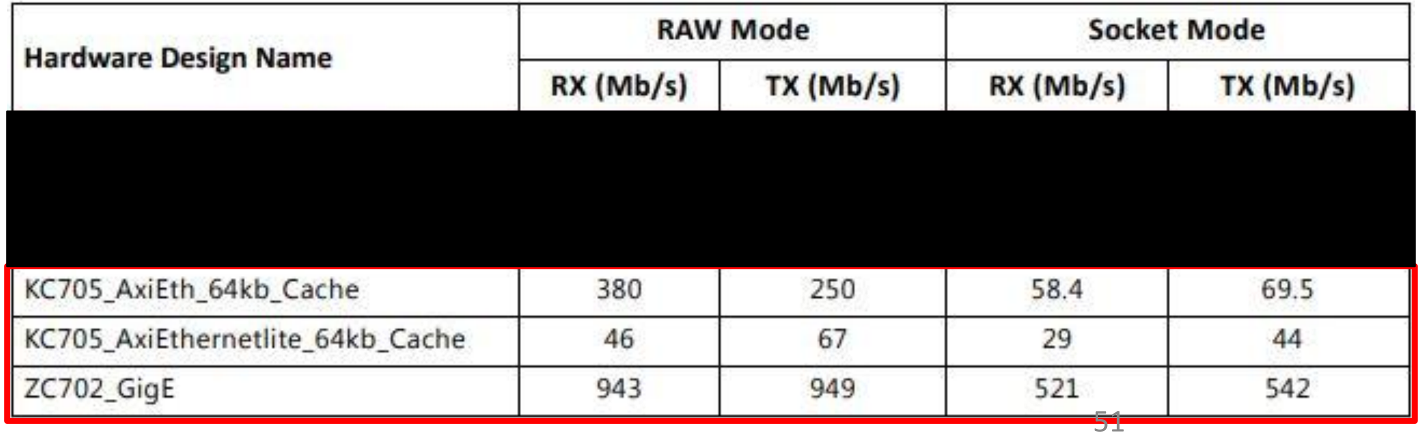
FPGA - PL
Application Design

Phase 5: Server [FPGA - PS]

- Lightweight IP (LWIP) is a full TCP/IP networking stack for embedded systems
    - Other options: PetaLinux, FreeRTOS
- The echo server provided us with a quick template to receive and transmit data between the PC and the FPGA
    - We made modifications to the “recv_callback” function to accept packets of predetermined length and transmit results
- The pixel data is received, converted to IEEE 754 floating-point format, then written into BRAM so the application can start

Phase 5: Server [FPGA - PS]

- ZCU104 Ethernet transmission speeds
  - Capable of about 950 Mbps (1 Gb) speeds
  - Max transmission speed we were able to obtain was 500 Mbps (with buffer overflow errors)
  - Current design works with about +/- 50 Mbps speeds



Hardware Design Name	RAW Mode		Socket Mode	
	RX (Mb/s)	TX (Mb/s)	RX (Mb/s)	TX (Mb/s)
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
KC705_AxiEth_64kb_Cache	380	250	58.4	69.5
KC705_AxiEthernetlite_64kb_Cache	46	67	29	44
ZC702_GigE	943	949	521	542

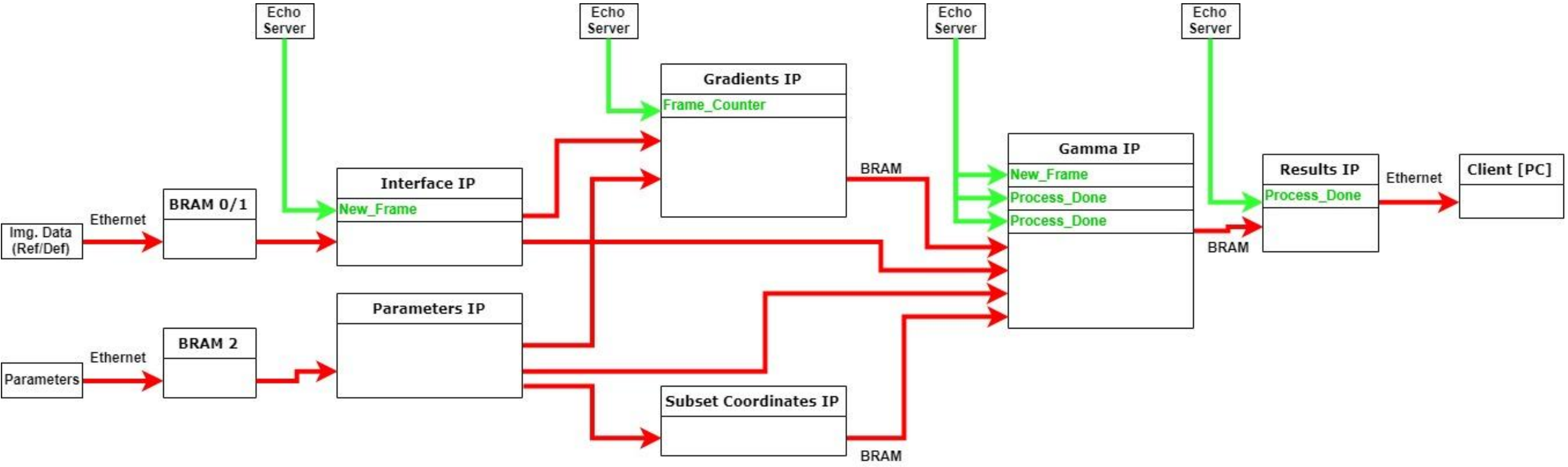
* Table from Xilinx Reference Guide

Overview

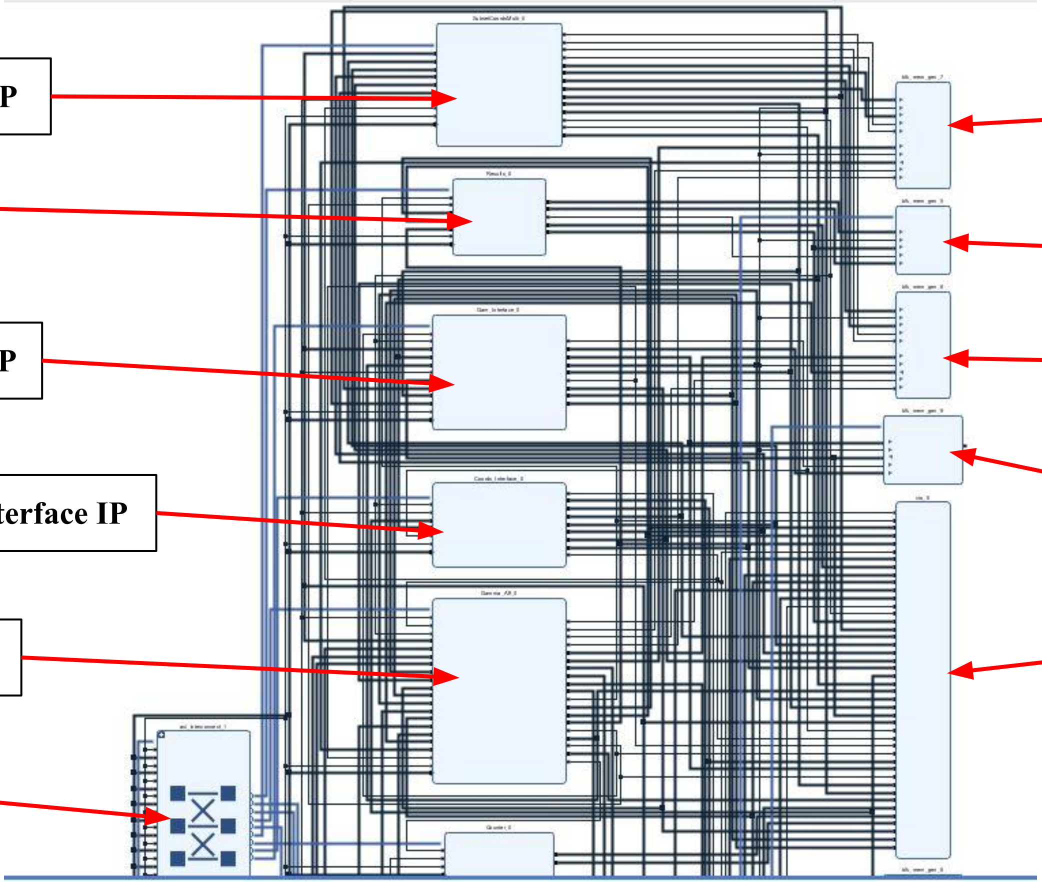
- Objective
- Results
- Challenges
- Additional & Future Works
- Conclusion
- **Project Decomposition**
  - Phase 1: Understanding DICe
  - Phase 2: Identifying Key Functions
  - Phase 3: Architecture Design
  - Phase 4: Client [PC]
  - Phase 5: Server [FPGA - PS]
  - **Phase 6: Application [FPGA - PL]**
  - Phase 7: Testing & Improvements

Phase 6: Application [FPGA - PL]

Simplified Block Diagram



Phase 6: Application [FPGA - PL]



Subset Coordinates IP

Results IP

Gamma Interface IP

Subset Coordinates Interface IP

Gamma IP

AXI Interconnect 1

BRAM 7

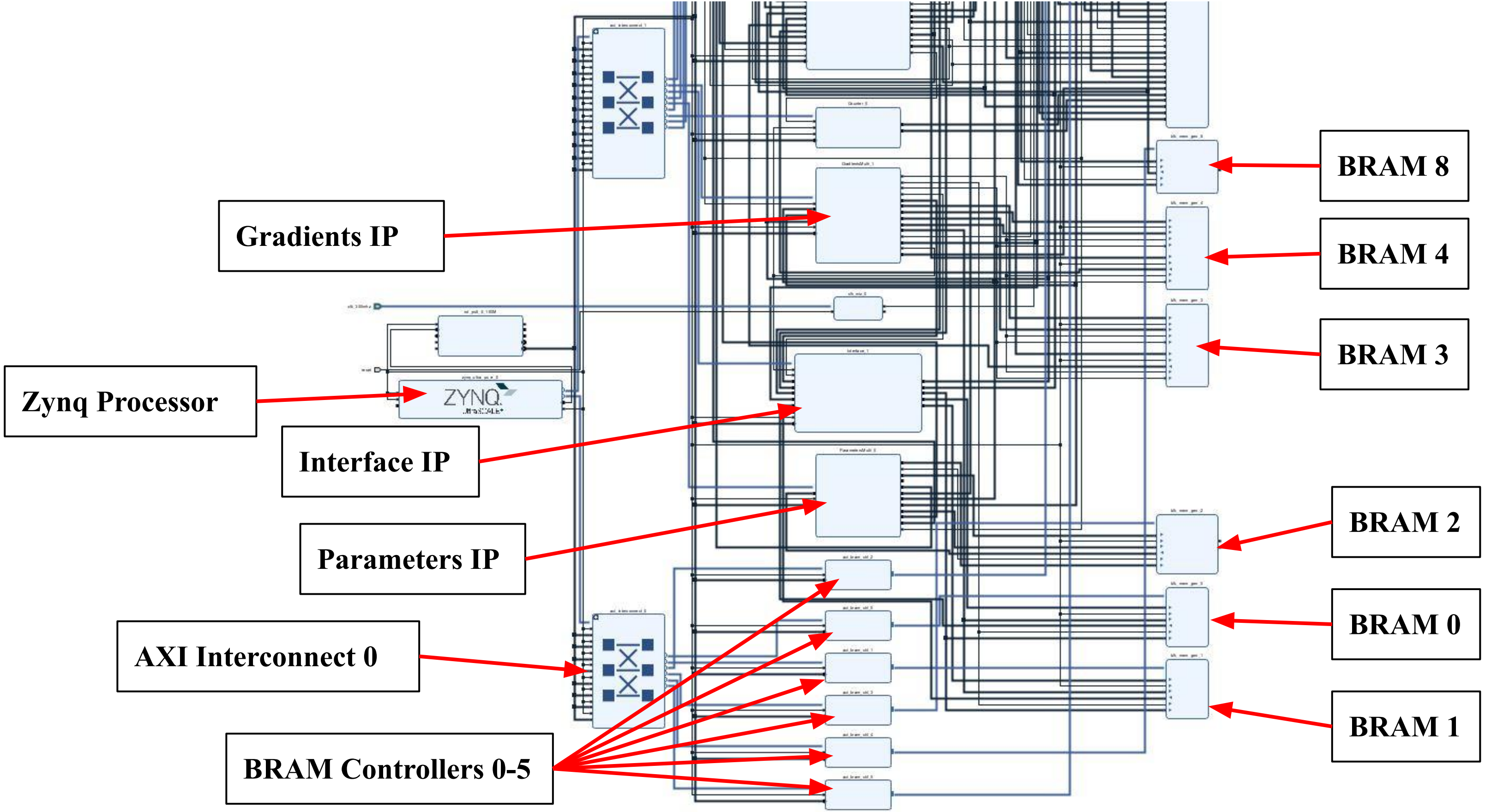
BRAM 5

BRAM 6

BRAM 9

VIO

Phase 6: Application [FPGA - PL]



Gradients IP

Zynq Processor

Interface IP

Parameters IP

AXI Interconnect 0

BRAM Controllers 0-5

BRAM 8

BRAM 4

BRAM 3

BRAM 2

BRAM 0

BRAM 1

Phase 6: Application [FPGA - PL]

- BRAMs/BRAM Controllers
  - Used for storing the local data (frames, gradients, subset coordinates, and user-defined parameters)
  - BRAM controllers interface with the echo server
- Parameters IP
  - Reads the user-defined parameters from BRAMs and sends them to the other IPs
  - Variables such as: number of subsets, shape, size, centerpoint, correlation method

Phase 6: Application [FPGA - PL]

- Subset Coordinates IP
  - Computes the coordinates of the pixels that are inside the region of interest (subset) and saves the results into BRAMs
  - Tells the correlation where the subset is within a frame
- Subset Interface IP
  - Is responsible for reading the subset information from BRAMs and sending them to the Subset Coordinates IP as needed
  - Handles receiving and sending of data for multiple subsets

Phase 6: Application [FPGA - PL]

- Interface IP
  - Handles the addressing to switch between the reference and the deformed frames in BRAM and sends the data to both the Gradients IP and Gamma IP
- Gradients IP
  - Gradients are the intensity between adjacent pixels
  - Computes the difference between intensities in both X and Y direction and saves the results into BRAMs

Phase 6: Application [FPGA - PL]

- Gamma Interface IP
  - Is responsible for reading the subset information from BRAMs and sending them to the Gamma IP as needed
- Gamma IP
  - Uses the subset coordinates and gradients to perform the image correlation algorithm and sends the displacements to the Results IP
- Results IP
  - Receives the displacements from the Gamma IP and saves them into BRAMs
  - Interfaces with the echo server to send the results to the client

Overview

- Objective
- Results
- Challenges
- Additional & Future Works
- Conclusion
- **Project Decomposition**
  - Phase 1: Understanding DICe
  - Phase 2: Identifying Key Functions
  - Phase 3: Architecture Design
  - Phase 4: Client [PC]
  - Phase 5: Server [FPGA - PS]
  - Phase 6: Application [FPGA - PL]
  - **Phase 7: Testing & Improvements**

Phase 7: Testing

- Ethernet throughput tests
  - Sent 1 GB of data to the echo server to test transmission speeds
- Converting pixels to IEEE 754
  - Originally on Python script, added multiprocessing
  - Tested 1,000 numbers for correctness
  - Moved this step to the FPGA [ARM - PS]
- Performed unit testing on each function within the IPs
  - Used the VIO to monitor signals as they changed
  - Ran large scale tests with thousands of frames

Phase 7: Improvements

- Added support for multiple frames and multiple subsets
  - First tests were between 2 frames with 1 small subset
  - Added support for square and circular subsets
- Added support for user-defined parameters
  - Users can define the number of subsets, size, and shape prior to run
- Scaled up images from 64x48 to 448x232
  - Max size with our BRAM limitations; ¼ full-frame
  - Moved from static images to real test images