Implementation of Median Filter on Z -Board and AWS

Anushka Swarup, Medini J. Aradhya, Liming Xu

I .Introduction

The Median Filter is a nonlinear digital filtering technique for removal of noise from images. It works particularly well for certain types of random noises like the salt and pepper noise. This method is often used as a preprocessing step for edge detection algorithms. It is advantageous as this method better preserves the edges in certain instances.
Implementation of this filter using the AWS F1 FPGA Acceleration Platform. The Amazon web services cloud platform is compatible with existing FPGA algorithms hence makes it a versatile open source platform for accelerated computing.

The Median filters come under the category of Order-Statistic filters. These filters are nonlinear in nature and their response is based on ordering the pixels contained in the region encompassed by the filter. One application of median filters is in the smoothing of images. It is achieved by replacing the value of the center pixel by the median of the intensity values of the neighborhood of that pixel. Median filters are efficient in noise reduction for salt-and-pepper noise.

This application is amenable to FPGA-based reconfigurable computing since it is majorly used in digital image processing. It is preferred since it preserves edges while getting rid of noise by taking the median within the window selected and sliding the window over the entire image/signal. Since these processes are repetitive and simple to carry out implementation on an FPGA would provide appreciable speedup. Adding to this the parallelism functionality of the FPGA would allow for efficient resource utilization. The datapath would define the speed of the system as discussed subsequently. This when compared to the software implementation if the same should provide considerable improvement. Lastly, the reconfigurability of the system allows for varied window types to be implemented within the same hardware platform not taking away from the speed-up.

II. Algorithm

1. **Software:**
   The basic algorithm follows that the median, j, of a set of values, is such that half the values in the set are less than or equal to j and the other half are greater than or equal to j. For example, in a 3x3 neighborhood, the median is the 5th largest value. Thus, the main aim of median filters is to force pixels to be more like their neighbors. In order to perform median filtering a window of arbitrary size is made to move over the entire image. At every iteration, we sort the values of the pixels inside the window, determine their median, and assign the median value to the pixel at the center of the window in the filtered image.

   Flowchart:
   - Input image of size 128x128 was used
   - A 3x3 window was obtained at each pixel
   - Insertion sort algorithm was used to arrange the window in ascending order

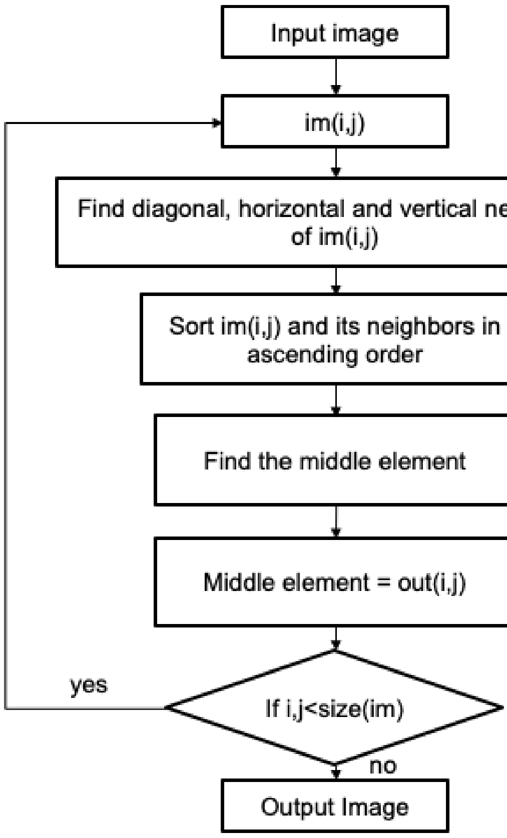

Input image
im(i,j)
Find diagonal, horizontal and vertical neighbors of im(i,j)
Sort im(i,j) and its neighbors in ascending order
Find the middle element
Middle element = out(i,j)
yes
If i,j<size(im)
no
Output Image

Fig 1: Flow Chart - software algorithm

2. **Hardware:**
   The hardware algorithm differs from the software one. The software algorithm utilizes sorting to find the median in the array. It cannot be parallelized. In theory, the sorting algorithm can still be implemented in hardware. It will have high latency, but since the datapath is pipelined, the resulting efficiency will still be similar. This will take up a lot of hardware space, though. That is the reason the hardware uses a different algorithm, proposed in *FPGA Implementation of a Median Filter* by G. L. Bates et al.

   As is shown in Fig 2, the hardware algorithm consists of three stages. The data are arranged in a 3*3 grid. In the first stage, the columns are sorted with triple sorters, in the second stage the rows are sorted with triple sorters. Finally, the diagonal is sorted and the number in the middle is the median.

   After the first two steps, the numbers are less than (or equal to) the numbers to their right on the same column. Each number on the lower rows must also have one number on each row above them that correspond to their position when they were in stage one. Take the number on location 1 in stage two, for example. It is less than the number on location 2. Meanwhile, the number on stage two must have two numbers above it that belonged to the same column at stage one. The same goes for the number at location 1. This shows that it must be less than at least four numbers above it, and it must be greater than the number to its right. The same argument can be applied to show that all numbers are less than (or equal to) the number to it’s right and above, and all numbers are greater than (or equal to) the number to it’s left and below.

Based on the arguments above, location 0 must be the smallest number, and location 1 and 3 must be greater than at least 5 of the 9 numbers in the grid, thus cannot be the median. Similarly, numbers at 5, 7, 8 cannot be the median, either. Thus the median must be on the diagonal of 2, 4, 6.

Assume the median is the smallest number of the three. Since the number at 4 is less than the three numbers to its upper right, the minimum of the three numbers on the diagonal would be less than at least 5 (2+3) numbers, thus cannot be the median. Similarly, the largest of the three is not the median, either. The median of all nine numbers must be the median of the three numbers on the diagonal. Thus the final triple sorter would find the median of all the numbers in the grid at position 4.

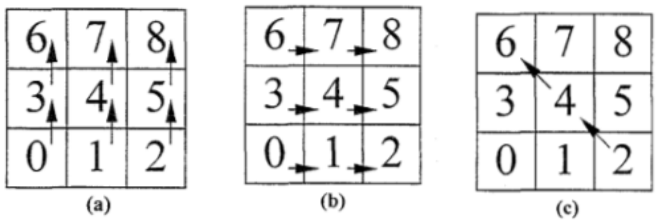


Fig 2: The three stages of the median algorithm: (a) vertical sort, (b) horizontal sort, (c) diagonal sort

The triple sorter is implemented with three dual sorters with three registers (Fig 3). Functionally it resembles the bubble sort algorithm. The latency in one triple sorter is three clock cycles. The dual sorter is implemented with a comparator and multiplexer (Fig 4).

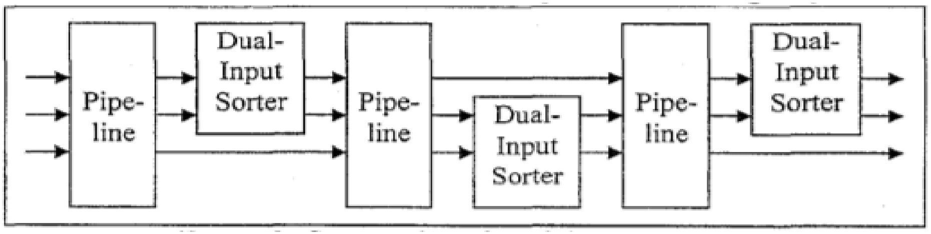


Fig 3: implementation of the triple sorter

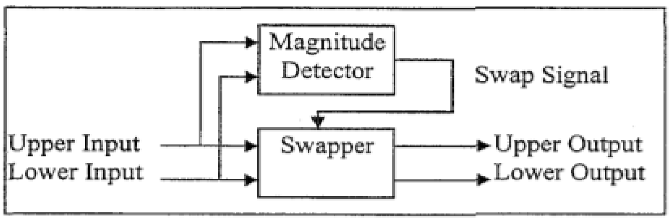


Fig 4: implementation of the dual sorter

3. **Amazon Web Services (AWS):**
   <u>Introduction :</u>

- The Amazon web services cloud platform is compatible with existing FPGA algorithms hence makes it a versatile open source platform for accelerated computing.

Implementation of Median Filter on Z -Board and AWS

Anushka Swarup, Medini J. Aradhya, Liming Xu

- Xilinx FPGAs are available in the Amazon Elastic Compute Cloud (Amazon EC2) F1 instances.F1 instances are designed to accelerate data center workloads.
- With F1 instances you can:
  1)Quickly deploy custom hardware acceleration
  2)Enjoy predictable performances with dedicated FPGAs
  3)Use your existing FPGA algorithms
- AWS has Xilinx SDAccel - Development Environment allows to run the RTL designs in the F1 instance. It also automates the acceleration of code written in C, C++ by building application-specific accelerators (OpenCL) on the F1 instances.

Process Outline :

1) Prerequisites
2) Create, Configure, Test the EC2- F1 Instance.
3) Install and run SDAccel GUI on the RDP Client.
4) Create and Run a project on SD Accel (2 Methods)
5) Generate .xclbin file
6) Run the .xclbin file on the FPGA (on F1)

III. Design and Implementation

• Datapath

As is discussed in hardware implementation, the datapath consists of three stages. The first stage consists of three triple sorters, the second stage consists of three triple sorters, and the last stage consists of one triple sorter, as is shown in Fig 5.

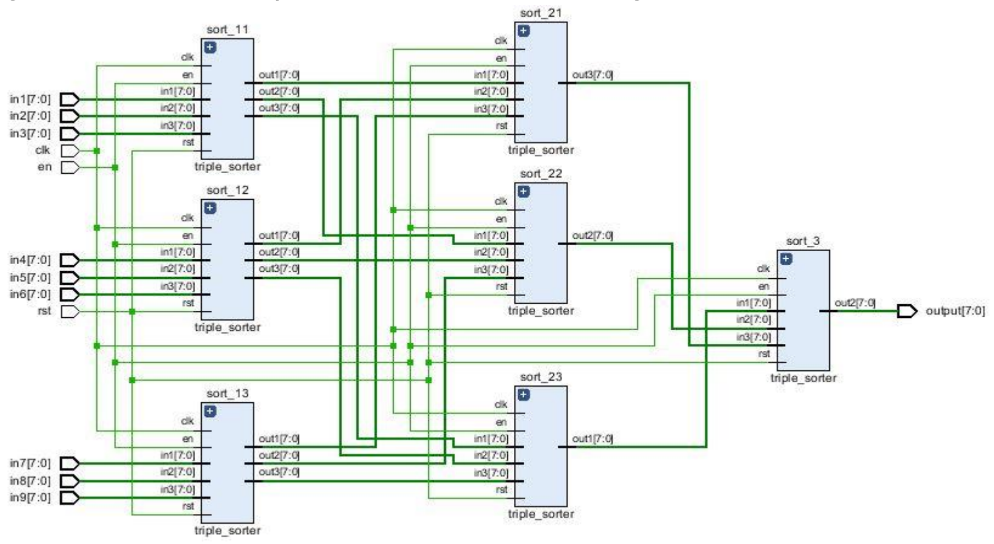


Fig 5: The datapath consists of three stages, with triple sorters running in each stage.

Each one of the triple sorters is constructed with three double sorters and three registers, this gives each of them a latency of 3 clock cycles, As is shown in Fig 6.

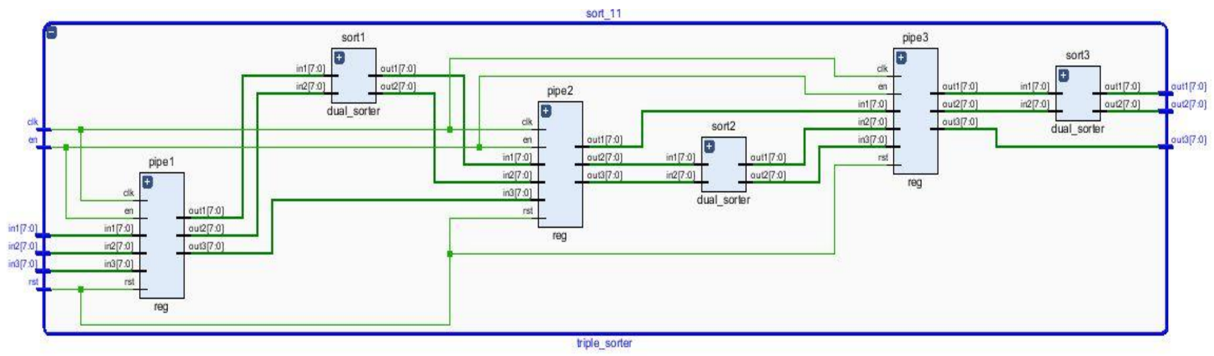


Fig 6: Implementation of the triple sorter resembles the bubble sort. It has a latency of three clock cycles

This datapath has a latency of 9 clock cycles in total, with 3 cycles in each stage. Since it is pipelined, it will output data on every clock cycle. It is unrolled 4 times in the user app so that 4 pixels can be processed at a time. A wrapper is made around it to take in an en signal to enable the datapath, as well as output a valid signal to signify the first output and start the output address generator and the output BRAM.

• **Block diagram of system architecture**

The block diagram of the median filter has three smart buffers reading from the BRAM. The input address generator will generate addresses from the previous row, current row and next row. The datapath consists of four datapaths, processing four pixels at a time. When the datapaths have the first outputs ready, the output address generator will generate addresses for the result to be written into the output RAM. The concept diagram and the schematic are shown in Fig 7 and Fig 8.

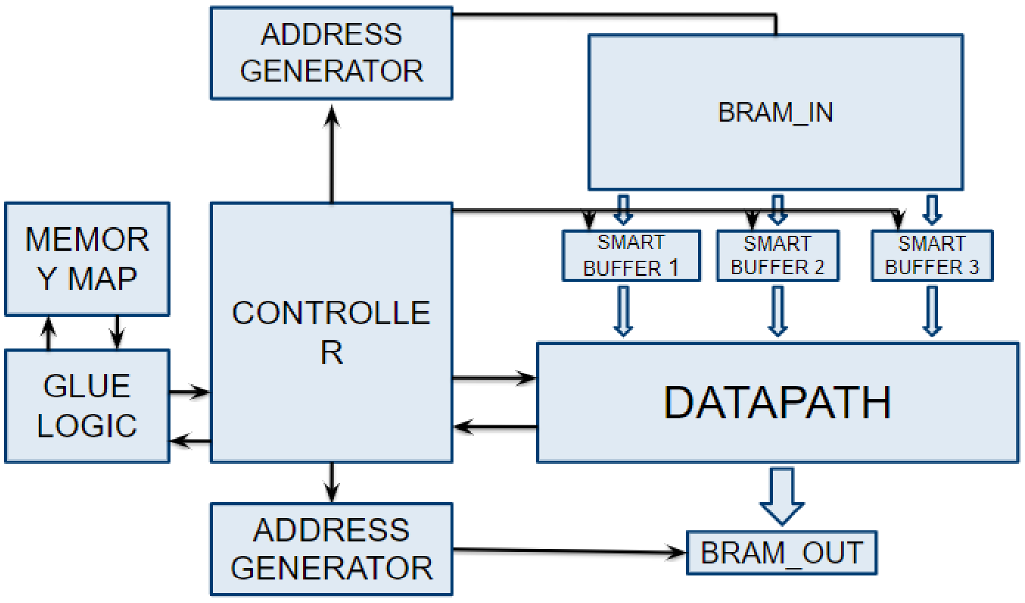


Fig 7: Concept of the block diagram with three smart buffers loading data into datapath.

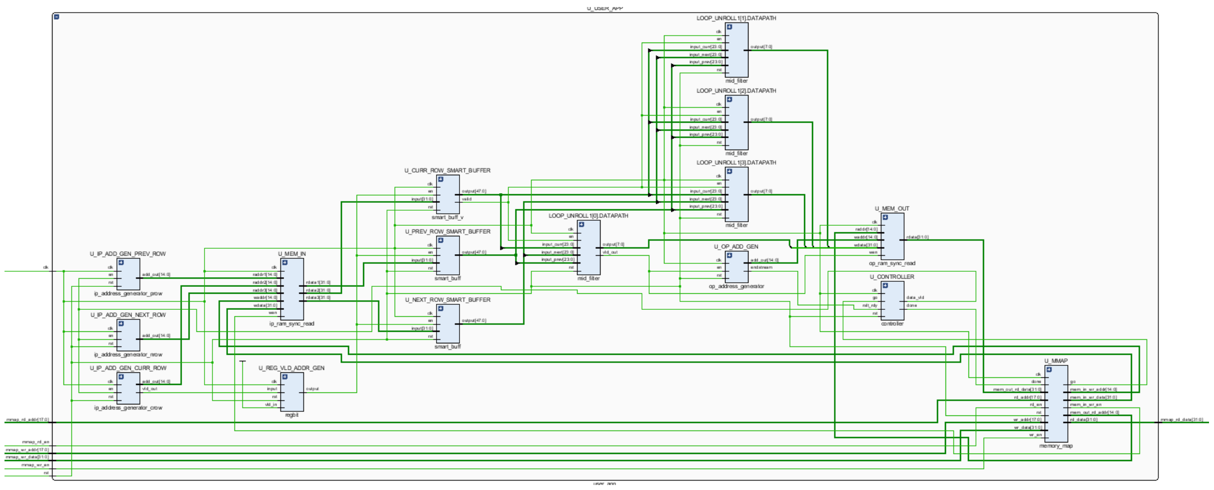


Fig 8: The schematic of user_app. The datapath has been unrolled four times.

• Data arrangement and buffering

The data are loaded into the RAM one word at a time, which consists of 4 pixels. Once the go signal is set, the input address generator will start generating addresses to read from the BRAM. The addresses from the previous row start from 0, the addresses from the current row starts from the number of words in a row (number of pixels/4), and the next row address starts from the number of words in a row times two.

The smart buffer loads 4 pixels (one word) at a time and pushed 6 pixels into the datapath. This allows four datapaths to run in parallel. The smart buffer is shown in Fig 9.

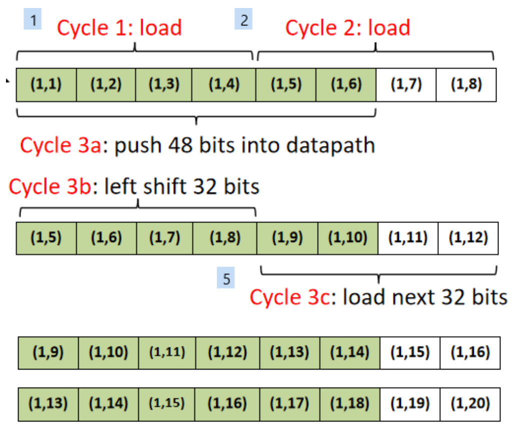


Fig 9: The smart buffer loads 4 pixels at a time and pushes 6 into the datapath.

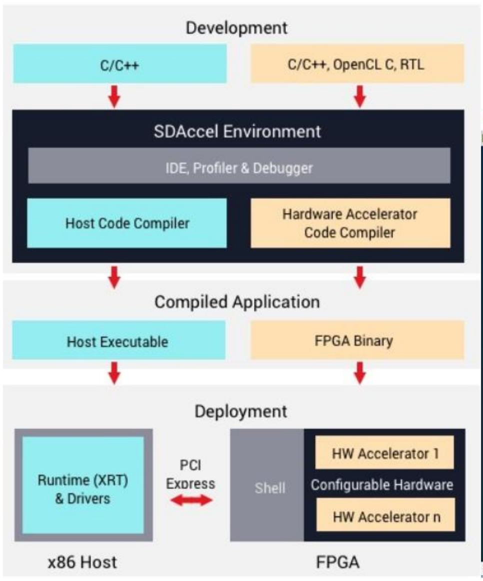
The way the addresses are generated will result in garbage data at the edge of the image. This happens when, say, the filter tries to find the median of pixels around (5,1). Since it does not have pixels to its left, the datapath will take the pixel values from the previous row at the other edge of the image instead. This cannot be avoided in the hardware, since the way the datapath is unrolled demands sharing of data, thus should be dealt with in software. Also, due to the way data are loaded, the number of pixels in a row must be multiples of 4. This is also unavoidable since the BRAM loads a word at a time, not a byte. It can also be dealt with using software by simply adding padding at the edge of the image so that the column number is a multiple of 4.

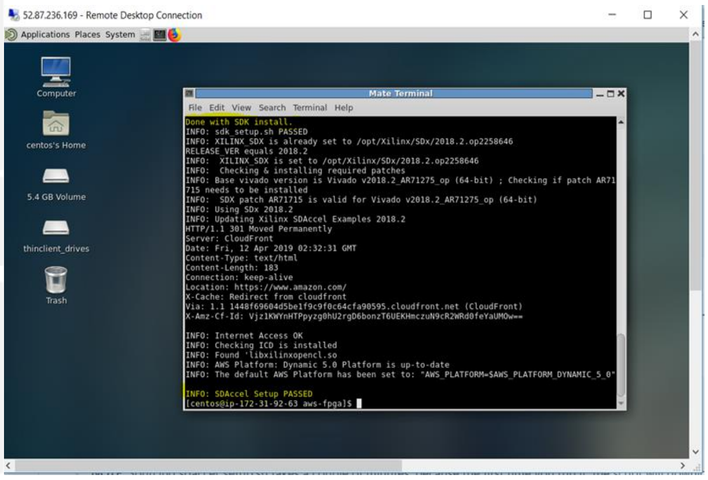
AWS Details :

- Steps 1 through 3 set up and run the SDAccel platform on Instance.
- We started by creating an account on AWS and configuring it to (N.Virginia) US East 1b region to access the F1 instance. We then created an S3 bucket to store data needed during the compilation of the program on the FPGA. Subsequent to that we created a user and allocated its permissions to use the FPGA F1 instance correctly.
- We then Launched an f1.2xlarge instance (this instance has access to 2 FPGAs) by creating a key pair as prompted during the launch. Note: since we used a windows system we had to convert the ".ppm" file to ".ppk" file using PuTTYGen such that it is in the appropriate format for access by the remote machine.
- Then using command prompt of our machine we launched PuTTY to connect to the EC2 F1 instance we launched using the following command: ***putty -i <AWS key pairs.ppk> -ssh centos@<public IP address of EC2 instance> 22*** where the AWS Key pair.ppk was the full path to the file we downloaded in the previous step and converted to .ppk format. And the Public IP address is mentioned once the F1 instance is launched.
- We then successfully accessed the FPGA on the F1 instance. (But this was accessed as a terminal window without a GUI)
- To access the Centos GUI that we are connected to we enabled Remote desktop connection with the following
  source <(curl -s https://s3.amazonaws.com/aws-fpga-developer-ami/1.5.0/Scripts/setup_gui.sh)
- We noted down the password generated by this line of command. we then launched the RDP client with the IPv4 Public IP details mentioned on the AWS F1 instance Page. (Note: This IP changes with each Instance created). This opens up the Centos environment (username: centos & Password: generated from above command)
- We were able to log in. which means that we were connected to the instance running] on Centos 7 and the FPGA Developer AMI.
- We then set up the SD Accel Environment with the following command:
  $ aws configure
  - AWS Access Key ID [None]: <your access key>
  - AWS Secret Access Key [None]: <your secret key>
  - Default region name [None]: <your AWS region >
  - Default output format [None]: json
- followed by:
  git clone https://github.com/aws/aws-fpga.git $AWS_FPGA_REPO_DIR
  cd $AWS_FPGA_REPO_DIR
  source sdaccel_setup.sh

Implementation of Median Filter on Z -Board and AWS
Anushka Swarup, Medini J. Aradhya, Liming Xu

- This set up the SD Accel Environment on the Centos 7, Running on AWS EC2 F1.




52.87.236.169 - Remote Desktop Connection
Applications Places System
Computer
centos's Home
5.4 GB Volume
thinclient_drives
Trash
Mate Terminal
File Edit View Search Terminal Help

```
Done with SDK install.
INFO: sdk_setup.sh PASSED
INFO: XILINX_SDX is already set to /opt/Xilinx/SDx/2018.2.op2258646
RELEASE_VER equals 2018.2
INFO:  XILINX_SDX is set to /opt/Xilinx/SDx/2018.2.op2258646
INFO:  Checking & installing required patches
INFO: Base vivado version is Vivado v2018.2_AR71275_op (64-bit) ; Checking if patch AR71
715 needs to be installed
INFO:  SDX patch AR71715 is valid for Vivado v2018.2_AR71275_op (64-bit)
INFO: Using SDx 2018.2
INFO: Updating Xilinx SDAccel Examples 2018.2
HTTP/1.1 301 Moved Permanently
Server: CloudFront
Date: Fri, 12 Apr 2019 02:32:31 GMT
Content-Type: text/html
Content-Length: 183
Connection: keep-alive
Location: https://www.amazon.com/
X-Cache: Redirect from cloudfront
Via: 1.1 1448f69604d5be1f9c9f0c64cfa90595.cloudfront.net (CloudFront)
X-Amz-Cf-Id: Vjz1KWYnHTPpyzg0hU2rgD6bonzT6UEKHmczuN9cR2WRd0feYaUMOw==

INFO: Internet Access OK
INFO: Checking ICD is installed
INFO: Found 'libxilinxopencl.so
INFO: AWS Platform: Dynamic 5.0 Platform is up-to-date
INFO: The default AWS Platform has been set to: "AWS_PLATFORM=$AWS_PLATFORM_DYNAMIC_5_0"

INFO: SDAccel Setup PASSED
[centos@ip-172-31-92-63 aws-fpga]$
```

Fig 10 : Structure of the SD Accel Environment Fig 11: Installed environment on Centos7 RDP Client (AWS EC2 Instance Running)

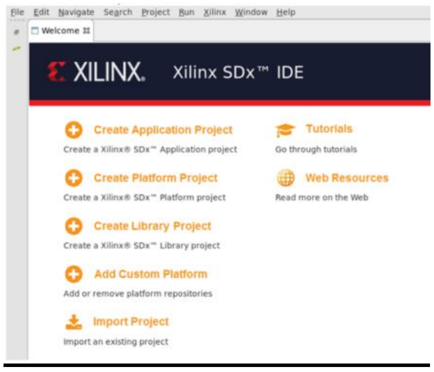
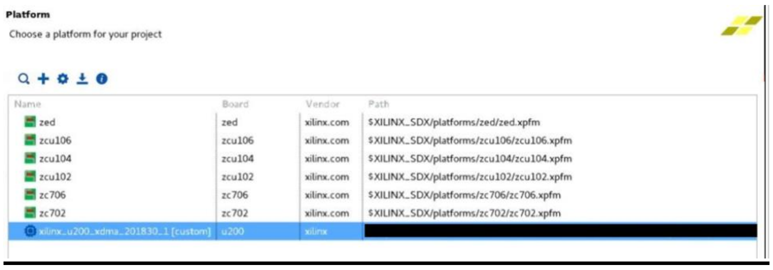

File Edit Navigate Search Project Run Xilinx Window Help
Welcome
XILINX. Xilinx SDx™ IDE
Create Application Project
Create a Xilinx® SDx™ Application project
Tutorials
Go through tutorials
Create Platform Project
Create a Xilinx® SDx™ Platform project
Web Resources
Read more on the Web
Create Library Project
Create a Xilinx® SDx™ Library project
Add Custom Platform
Add or remove platform repositories
Import Project
Import an existing project

Fig12 : SD Accel Development Environment.

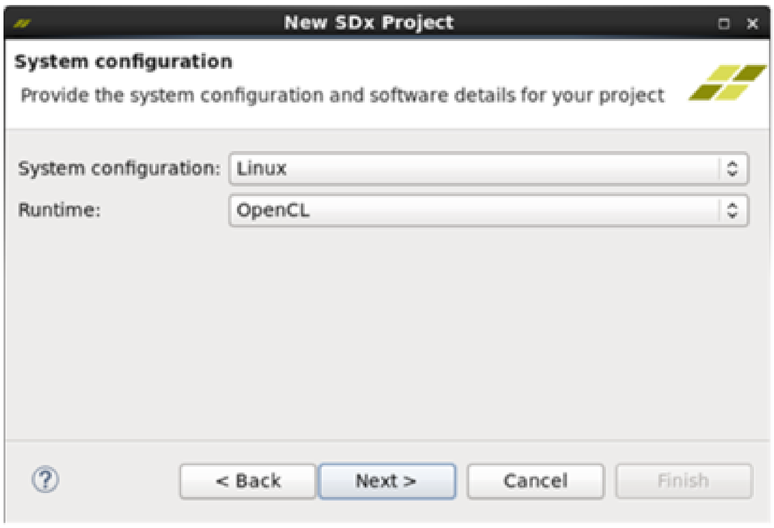
- We then downloaded the Xilinx Runtime and Deployment and Development Shell to be able to interface with AWS F1.
- Then follow the Getting Started Guide. https://www.xilinx.com/products/boards-and-kits/alveo/u200.html#gettingStarted .This will make the U200 platform available on the Local SDx suite.


Platform
Choose a platform for your project

Name	Board	Vendor	Path
zed	zed	xilinx.com	$XILINX_SDX/platforms/zed/zed.xpfm
zcu106	zcu106	xilinx.com	$XILINX_SDX/platforms/zcu106/zcu106.xpfm
zcu104	zcu104	xilinx.com	$XILINX_SDX/platforms/zcu104/zcu104.xpfm
zcu102	zcu102	xilinx.com	$XILINX_SDX/platforms/zcu102/zcu102.xpfm
zc706	zc706	xilinx.com	$XILINX_SDX/platforms/zc706/zc706.xpfm
zc702	zc702	xilinx.com	$XILINX_SDX/platforms/zc702/zc702.xpfm
xilinx_u200_xdma_201830_1 [custom]	u200	xilinx	[redacted]

Fig13 : Alveo U200 Platform on the SDx Environment.

- Create the SDx project like before and when prompted set the system configuration as follows :



New SDx Project

System configuration

Provide the system configuration and software details for your project

System configuration: Linux

Runtime: OpenCL

Fig 14: SDx System Configureation

- Then select the Empty Project, this created a blank project.
- 2 methods can be followed to implement the project :
  1) To integrate the Existing IP from Vivado into our SD Accel environment using RTL Kernel Wizard (tool part of SD Accel). (Details in PPT)

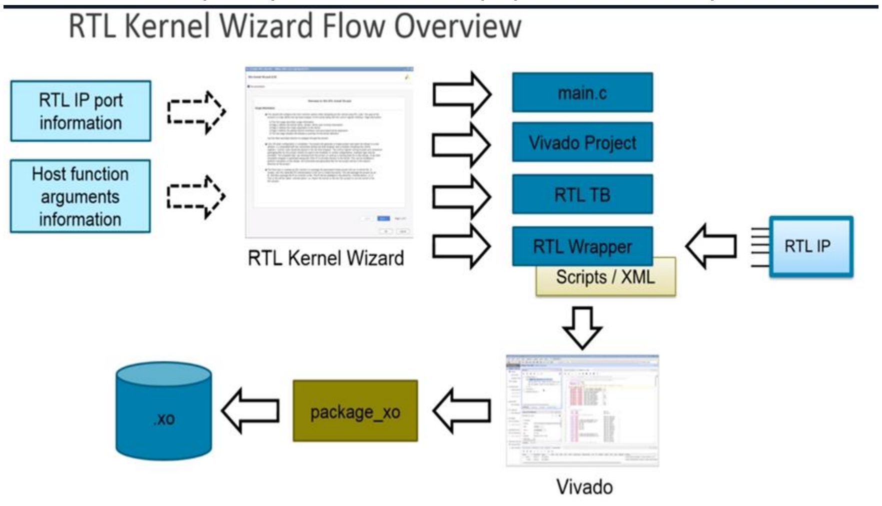


Fig 15: Block diagram showing process flow to generate .xclbin file

2) *<u>To run an Open CL code consisting of HW & SW emulations directly from SD Accel. To generate the .xclbin file required to run the project on the FPGA.</u>*

- *Once the project SW Emulation and HW emulation are carried out, the .xcl bin bile was created in the Project directory. This file was then executed via the terminal giving us the output of the project created with timing details.*

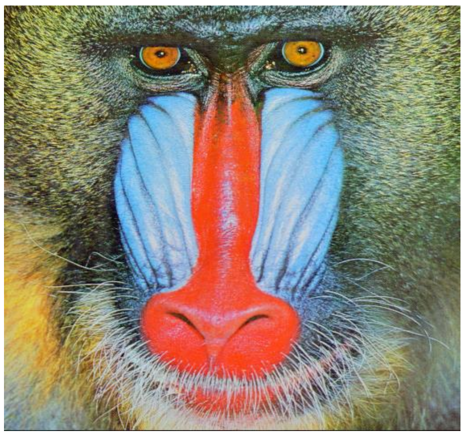
<u>IV. Testing and demonstration</u>

1. **Test/demo setup**
   a. A set of 4 popular test images were used to compare the results of the MATLAB function and the C++ code with the hardware output.

b. MATLAB R2017b was used for Median Filtering using the function “medfilt2”.
c. A C++ implementation of Median Filtering was developed and was implemented on the reconfig.hcs.ufl.edu server using putty.
d. The hardware implementation of the Median Filter was carried out using the Xilinx Vivado 2018.3 software.
e. The results from these three different implementations were obtained and then analyzed for their visual accuracy and speedup.

2. Dataset used

a. The images shown in Table 1 were converted into grayscale and then used as inputs to the MATLAB, C++ and Hardware Implementations of the Median Filter.

Fig 16: Test Image 1	Fig 17: Test Image 2
 Fig 18: Test Image 3	Fig 19: Test Image 4

3. The baseline for comparison (e.g., c program running on Zynq ARM processor)

a. Results from the MATLAB function are used for visual and pixel by pixel comparison with the hardware results.
b. Whereas, results from the C++ code are used to compare the speedup with the hardware implementation.

4. **AWS Testing.**
   1. SD Accel Project was created on the centos 7 RDP Client of the EC2 instance.
   2. A few glitches were faced while packaging the IP via the RTL Kernel wizard.
   3. Thus the Open CL approach was used to implement the Median Filter on the EC2 F1 instance.
   4. Test image1 in the above table was used with salt and pepper noise added to it.
   5. SW and HW Emulation was carried out with this image.
   6. ".xclbin" file was created and executed on the FPGA.
   6. The results of the Emulation time between software and Hardware were compared and approximate speedup calculated.

V. Test results

a. Fig 22 shows the input grayscale image of the test image.

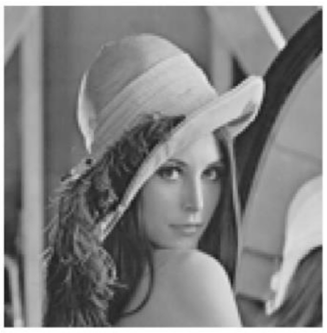


Fig 20: Input Image

b. Fig 23 & 24 are the corresponding MATLAB and Hardware output of the Median Filter implementations.

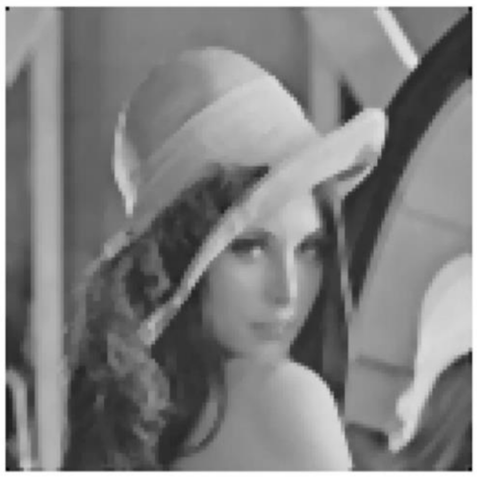
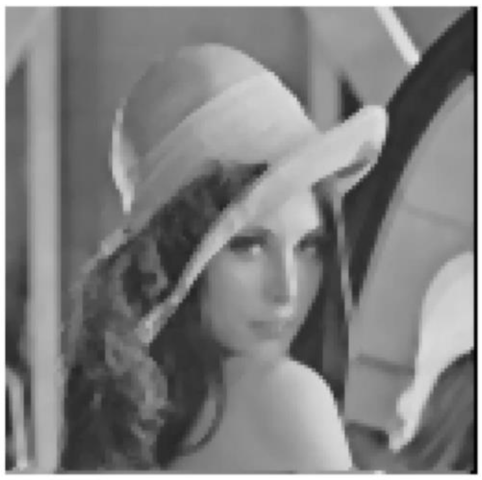
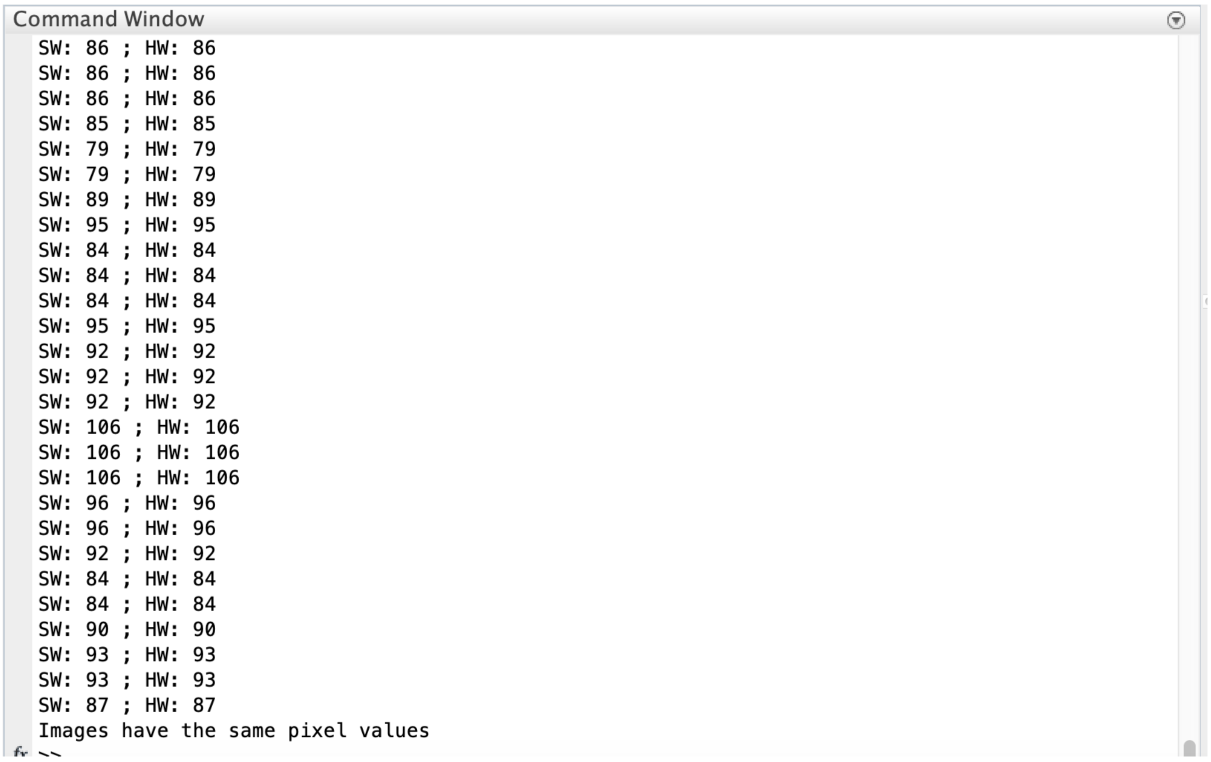
 Fig 23: MATLAB result of Median Filtering	 Fig 24: Hardware result of Median Filtering

Fig 25 shows the pixel by pixel comparison of the MATLAB and Hardware result. They are exactly the same.


Command Window

```
SW: 86 ; HW: 86
SW: 86 ; HW: 86
SW: 86 ; HW: 86
SW: 85 ; HW: 85
SW: 79 ; HW: 79
SW: 79 ; HW: 79
SW: 89 ; HW: 89
SW: 95 ; HW: 95
SW: 84 ; HW: 84
SW: 84 ; HW: 84
SW: 84 ; HW: 84
SW: 95 ; HW: 95
SW: 92 ; HW: 92
SW: 92 ; HW: 92
SW: 92 ; HW: 92
SW: 106 ; HW: 106
SW: 106 ; HW: 106
SW: 106 ; HW: 106
SW: 96 ; HW: 96
SW: 96 ; HW: 96
SW: 92 ; HW: 92
SW: 84 ; HW: 84
SW: 84 ; HW: 84
SW: 90 ; HW: 90
SW: 93 ; HW: 93
SW: 93 ; HW: 93
SW: 87 ; HW: 87
Images have the same pixel values
>>
```

Fig 25: Comparison between a section of the MATLAB filter output image & Hardware output

c. On adding salt and pepper noise and applying Median Filtering using the hardware we were able to obtain the result shown in Fig. 26

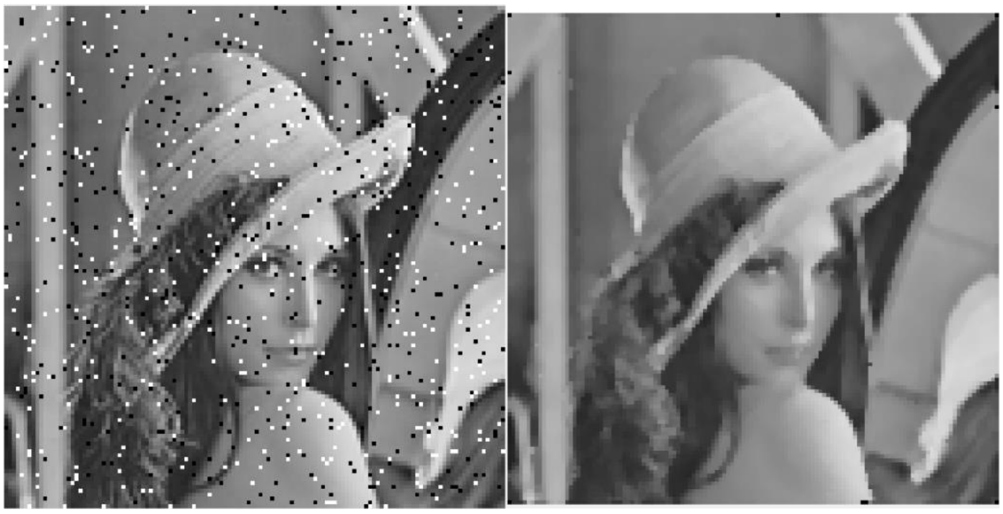


Fig 26: (a) Image with Salt & Pepper Noise ; (b) Image (a) after Median FIltering

d. **Output with Other Images:**

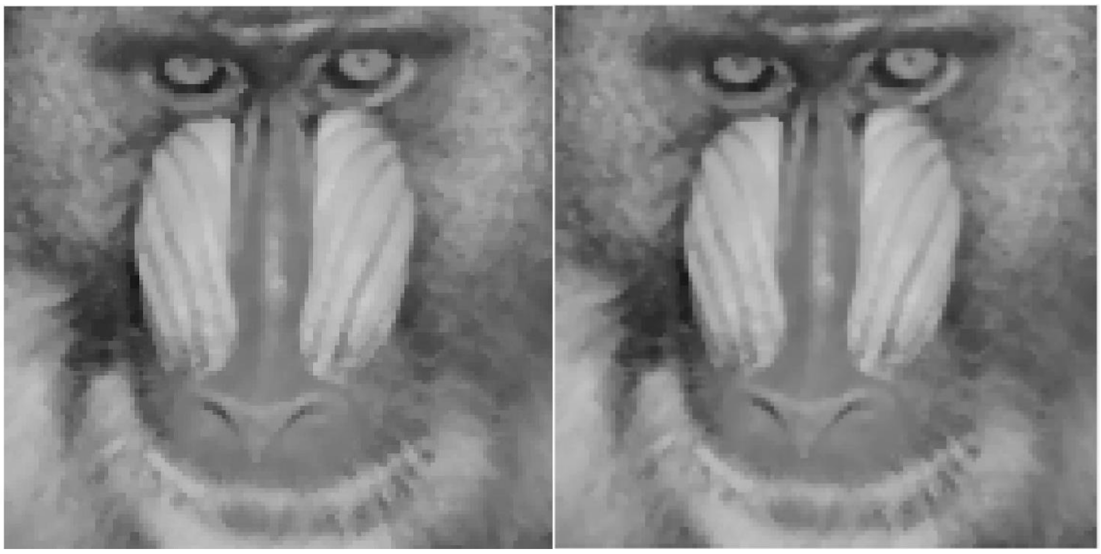


Fig 27: (a) MATLAB output for Median Filtering; (b) Hardware Output

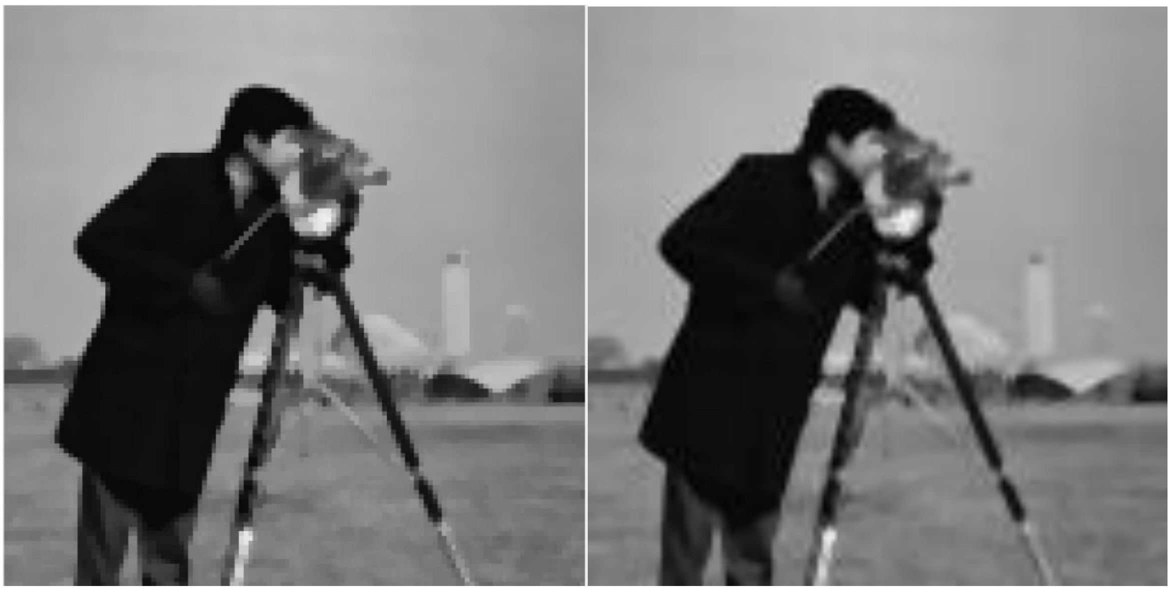


Fig 27: (a) MATLAB output for Median Filtering; (b) Hardware Output

e. **Images of Variable Size:** The application works with images of variable sizes as well. The image in Fig. 28 is a rectangular image of size 400X300.

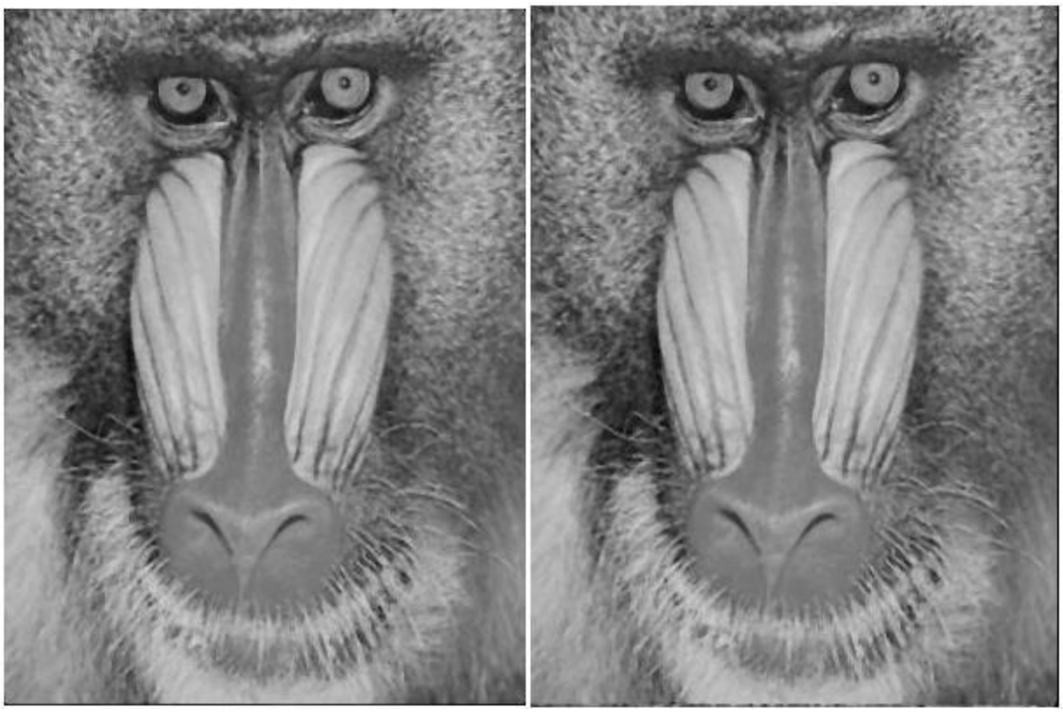
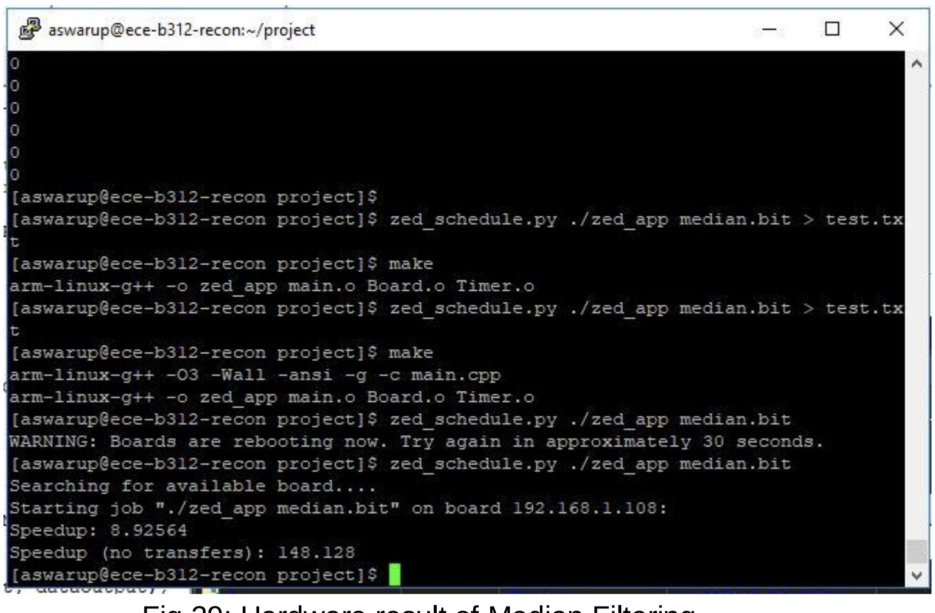


Fig 28: (a) MATLAB output for Median Filtering; (b) Hardware Output

f. **Speedup vs. baseline:** As can be seen from Fig. 29 the hardware was 9.77 times faster than the software when data input and output were taken into consideration. The hardware was 176.15 time faster than the software when data transfers were not taken into consideration.



```
0
0
0
0
0
0
[aswarup@ece-b312-recon project]$
[aswarup@ece-b312-recon project]$ zed_schedule.py ./zed_app median.bit > test.tx
t
[aswarup@ece-b312-recon project]$ make
arm-linux-g++ -o zed_app main.o Board.o Timer.o
[aswarup@ece-b312-recon project]$ zed_schedule.py ./zed_app median.bit > test.tx
t
[aswarup@ece-b312-recon project]$ make
arm-linux-g++ -O3 -Wall -ansi -g -c main.cpp
arm-linux-g++ -o zed_app main.o Board.o Timer.o
[aswarup@ece-b312-recon project]$ zed_schedule.py ./zed_app median.bit
WARNING: Boards are rebooting now. Try again in approximately 30 seconds.
[aswarup@ece-b312-recon project]$ zed_schedule.py ./zed_app median.bit
Searching for available board....
Starting job "./zed_app median.bit" on board 192.168.1.108:
Speedup: 8.92564
Speedup (no transfers): 148.128
[aswarup@ece-b312-recon project]$
```

Fig 29: Hardware result of Median Filtering

AWS :

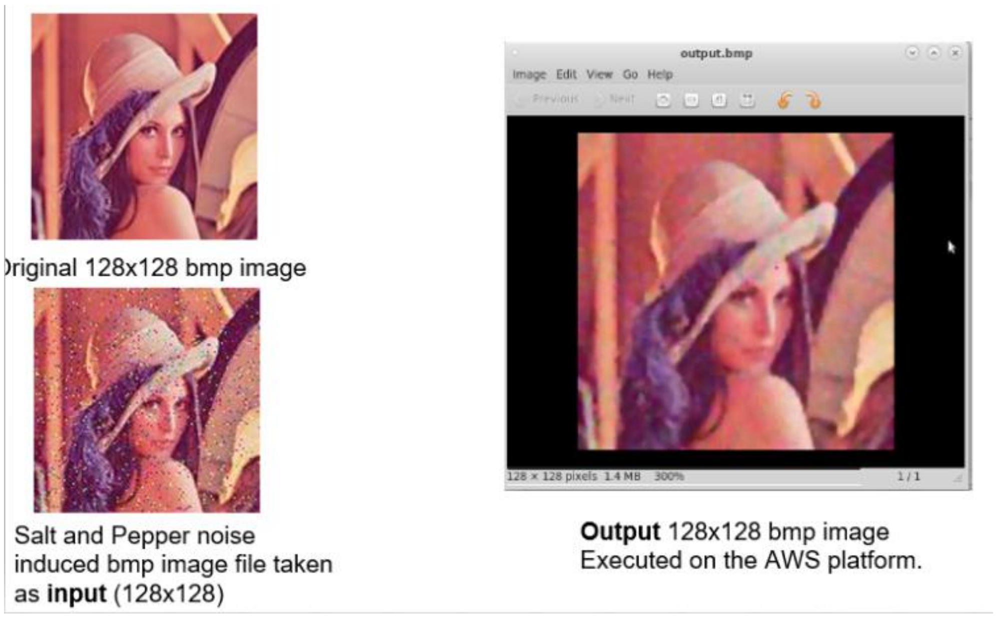
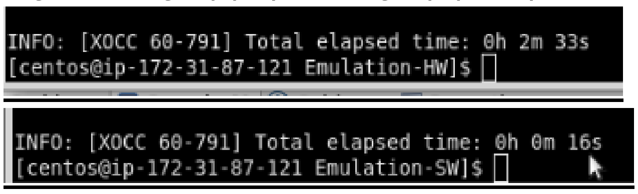

riginal 128x128 bmp image
Salt and Pepper noise induced bmp image file taken as **input** (128x128)
output.bmp
Image Edit View Go Help
Previous Next
128 × 128 pixels 1.4 MB 300%
1 / 1
Output 128x128 bmp image Executed on the AWS platform.

Fig 30: (a) Original Image (b)Input Image (c) Output Image



```
INFO: [XOCC 60-791] Total elapsed time: 0h 2m 33s
[centos@ip-172-31-87-121 Emulation-HW]$
```

```
INFO: [XOCC 60-791] Total elapsed time: 0h 0m 16s
[centos@ip-172-31-87-121 Emulation-SW]$
```

Fig 31 : Comparison between SW and HW emulation times

VI. Conclusion

- We have successfully implemented the Median Filter on the Zedboard and AWS. We have verified the speedup in comparison to C++ Software execution.
- Pixel by pixel verification was done to compare the hardware and the MATLAB outputs.
- SW and HW Emulation times were compared when implemented on AWS EC2 F1 instance.
- Based on the Results obtained we observe a 8.9x speed increase in the hardware implementation compared to the C++ Software implementation.

VII. References:

1) Leiou Wang, "A new fast median filtering algorithm based on FPGA," *2013 IEEE 10th International Conference on ASIC*, Shenzhen, 2013, pp. 1-4.
2) Gonzalez, Rafael C., and Richard E. Woods. Digital Image Processing. Pearson, 2018.
3) https://github.com/aws/aws-fpga
4) https://www.xilinx.com/products/design-tools/acceleration-zone/aws.html#gettingstarted
5) https://www.programming-techniques.com/2013/02/median-filter-using-c-and-opencv-image-processing.html