Hardware-based/acceleration of Frameworks for Media Authentication or Forgery Detection in Media

Jesus Villalvazo Fausto
University of California, San Diego
La Jolla, USA
jvfausto@ucsd.edu

Kwok Hung Ho
University of California, San Diego
La Jolla, USA
khh019@ucsd.edu

ABSTRACT

With the increasing use of social media and the advent of technologies that enable manipulation and generation of multi-media content, the world is reaching a point where distinguishing real from fake media is increasingly difficult. While this technology is useful in fields such as film production, virtual reality, etc, it poses security threats regarding media authentication. Such threats include manipulating public opinion, discrediting, blackmailing [13], etc. Technologies such as Photoshop and DeepFake, are allowing adversaries, without much difficulty to generate very convincing fakes of media to the human eye. This is why reliable media authentication and forgery detection systems are a very sought after topic. This paper aims to use hardware to accelerate existing frameworks that achieve forgery of media.

KEYWORDS

Forgery Detection, Media Authentication, Hardware Acceleration, FPGA, Convolutional Neural Networks

ACM Reference Format:
Jesus Villalvazo Fausto and Kwok Hung Ho. 2022. Hardware-based/acceleration of Frameworks for Media Authentication or Forgery Detection in Media. In *Proceedings of Final report (ECE 268 Wi '22)*. University of California, San Diego, CA, USA, 7 pages.

1 INTRODUCTION

While the generation of believable visual multimedia has become made possible through technology, fake multimedia has become a side effect which poses threats to society in the form of swaying public opinion, discrediting journalism, and blackmailing. As a result, the problem has come full circle back to technology, and whether it can develop fast and reliable tools to detect fake multimedia to prevent these social threats.

Existing methods for creating fake media are very widespread, but the most common include inserting an object from a different image (splicing), or from the same image (copy-move), and deleting an object or background (inpainting), followed by some post-processing such as resizing, rotation and color adjusting. Such methods are referred to as "cheapfakes". Meanwhile more advanced techniques that utilize advanced computer graphics (CG) and deep learning can generate entirely new content from generative adversarial networks (GAN) and autoencoders [13].

To detect this wide array of forgery techniques in multimedia content, popular techniques include Blind Methods, (where no prior knowledge is known), Sensor Based Methods (pertaining to knowledge of the camera/sensor), and machine learning techniques. This paper aims to 1. accelerate an existing machine learning technique for detecting image splicing via Verilog and 2. accelerate an existing blind method for detecting splicing via GPU code.

2 PRELIMINARY

2.1 FPGAs, GPUs, ASICs, CPUs

Field programmable gate arrays (FPGAs) are integrated circuits with a programmable hardware fabric. Unlike Graphics processing units (GPUs), the circuitry is not hard-etched and can be reprogrammed as needed. GPUs became widely popular in machine learning because they were originally designed to render video and graphics, thus excelling at parallel processing.

GPUs however, aren't as performant as Application-Specific Integrated Circuits (ASICs), since they can be specifically designed for machine learning workloads. [6] The FPGA can be customized to behave similar to the ASIC and GPU, whilst offering a number of other benefits. FPGAs can be more power efficient due to being able to program a specific part of the FPGA to do a task rather than the whole chip, more cost effective by saving board space, and more efficient in speed due to allowing the designer to build the neural network from the ground up and structure the FPGA to best suit the structure. In terms of speed for ML applications, the FPGA overcomes I/O bottlenecks (accelerating data ingestion) [6]. For example, the benefit of being close to hardware enables sensor fusion, which helps I/O speed and loss in applications such as robotics and autonomous vehicles. FPGAs can also accelerate high performance computing (HPC) clusters and inference time in production environments.

2.2 Forgery Generation and Detection

Forgery generation is getting easier and easier with advances in hardware, machine learning and high performance image and video processing applications such as Photoshop. There are tutorials now that teach one how to create DeepFakes in less than 5 minutes [11]. Post processing is also done to make the forgery more believable and usually comes in the form of resizing, rotating, and color/saturation adjustment [13].

ECE 268 Wi '22, March 2022, San Diego, CA, USA

ACM ISBN . . . $

For forgery detection, the most conventional methods include blind methods, sensor based methods, and machine learning methods [13].

Blind methods are methods that do not require any prior knowledge of the image, and rely exclusively on the media asset under analysis. Examples of blind methods include looking for manipulations in Color Filter Array Artifacts (CFA), Noise level and pattern, Lens Distortion, Compression Artifacts, and Editing Artifacts. For more information on these methods, reference [13].

Sensor based methods are methods that pertain to the camera/sensor, and hence require prior knowledge of the observation model. Lastly, machine learning methods uses data to train a model to make predictions on unseen forged media and typically employ convolutions via a neural network [13].

2.3 Hardware Acceleration: Examples

One of the main goals of hardware accelerator development is to offload any computationally significant operation of a software from a CPU to FPGA [5]. Like the GPU, FPGAs provide parallelism, pipelining and bitlevel processing facility, and has great potential to reduce and/or remove performance bottlenecks in forgery detection methods that involve heavy image processing tasks.

FPGA-based acceleration methods for Convolutional Neural Networks (CNN) typically involve multiplier-accumulator (MAC) arrays [14], which as a result demand availability from on-chip DSP blocks. This however leaves many of the FPGA's memory and resources underutilized. Meanwhile using other functional units other than the MAC can yield lower results in performance. A method introduced in [14], yielded improvements in throughput, DSP resource reduction, raw arithmetic performance and DSP compared to state-of-the-art sparse convolution-based accelerators by introducing a ABM-SpConv scheme and designing specific hardware such as a convolutional unit (CU) to suit the scheme.

Another method discusses the hardware acceleration of Video Image Reconstruction [15]. Which is a method that could be utilized for fake content generation. By realizing the frame image data buffer and the image reconstruction after complex algorithm processing in an FPGA, native and AXI operations were converted and video caching times were greatly reduced, leading to greatly improved speeds.

Hardware acceleration for a task specific to media forgery detection is rare in research literature, but the sub-tasks in different methods used within forgery detection can be approached. Leveraging research in other applications of hardware acceleration, this paper aims to accelerate methods used in forgery detection.

3 IMPLEMENTATION

For the implementation, 2 forgery detection frameworks will be considered for acceleration. The first is a blind method which uses CFAs and Noise Levels for image splicing detection. The second is a machine learning framework that uses a Semi-Global Network (SGN).

3.1 ML Framework: Overview

The second technique proposed, follows the work of a machine learning framework [2], and aims to accelerate the operations within the convolutional neural network. The framework [2], uses a "semi-global network" (SGN), which utilizes the fact that the spliced region should not only relate to local features (e.g. spliced edges), but also global ones (e.g. illumination, color).

To be specific, the network contains 4 sections, the local LSTM auto-encoder network, the global ResNet 16 network, the classification network, and the Segmentation step [2]. Due to having multiple networks, it is desired to find the main bottleneck in speed and the amount of convolution operations happening in each section.

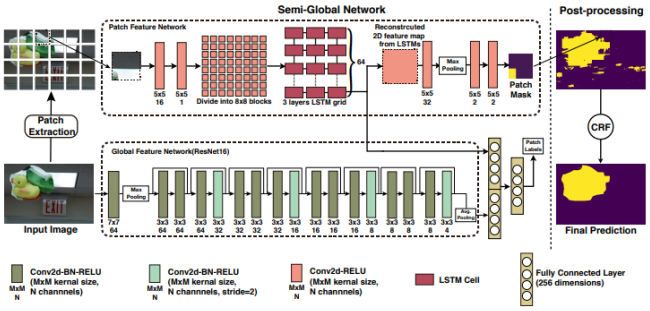

Figure 1: SGN Framework [2]

Although the framework can utilize fast and efficient modern GPUs, there is a lot of space for optimization via hardware. For example, an FPGA designer can translate the entire inference step into hardware. As this paper is more concerned about accelerating inference rather than training, a PyTorch model is first trained with regular GPUs via the Columbia Dataset [10], which is an Image Splicing Detection Evaluation Dataset. After training, inference time is measured and recorded before and after optimization.

3.2 ML Framework Acceleration: Approach

The main bottleneck of this method is the computational intensity of the convolutional layers. Due to this, only convolution will be considered in optimization although other operators such as ReLU, MaxPool, and LogSoftmax were also present in the framework. The part of the method that utilizes convolution the most is the Global Feature Network, which has a ResNet 16 Backbone consisting of 16 convolutional layers. The method also uses LSTM as a local feature network, which runs convolution three times on the input prior to feeding it to the LSTM. In one forward step, LSTM is followed by the Global Network, then Classification and then Segmentation. All these sections feature convolution except classification. It can be seen that the convolution operation is an integral part of this method. In fact, for most CNN's they are responsible for 90% of the execution time [4].

To accelerate the ML framework, the convolution operation needed to be optimized into hardware. To achieve this, its implementation was designed in a FPGA simulation environment via Intel's Quartus Prime with Verilog code and simulated in ModelSim.

The reason why convolution is such an integral part of the CNN process is that it allows you to pick out features from the image, such as sharpness, edge detection, color differences amongst other through a simple process. In the convolution process you have three

matrices you care about: the Image, the Kernel and the Output,The Image is the input that is going to change, whereas the Kernel is what helps you extract the desired feature [7]. This is done through multiplying and adding the image in a certain form.

What this means in code is to multiply each value of the kernel by each value in it's corresponding space in the image and then adding them up. This needs to be repeated a certain amount of times, represented by the following equation.

$O_w = \frac{I_w - K_w}{S} + 1$ [7]

$O_h = \frac{I_h - K_h}{S} + 1$

$iterations = O_w * O_h$

Where I_w and I_h is the width and height of the image, S is the step size, and K_w and K_h are the width and height of the kernel. In a CPU this process is fairly linear and it can take a lot of time by doing. This impedance is worked around the GPU by parallelizing each kernel addition and multiplication, but it can be a lot faster if done in an FPGA.

The convolution algorithm implemented was a sliding window method, in which both the pipelined and parallelized methods where implemented. For both methods we created a module that did all the multiplications of an individual kernel at the same time and then returned the addition of the multiplications in the next clock cycle.

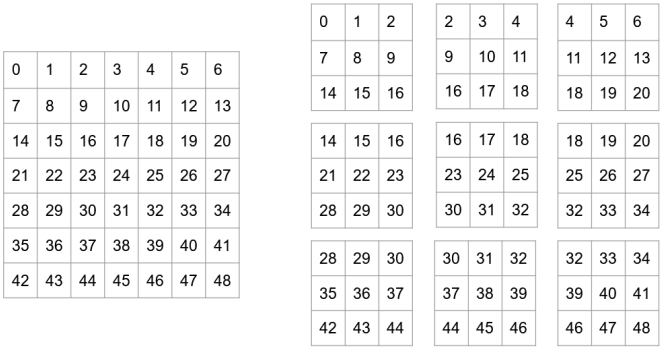

Figure 2: Example of splitting the image for parallelism

The difference between the pipelined and parallelized versions is the amount of work done in parallel. In the pipelined version, there is a sliding window that navigates thought the image section by section. In the parallelized version there is a module created for each section of the image, and all the kernel multiplications and additions happen at the same time.

3.3 Blind Framework: Overview

The two blind methods in this framework include CFA and Noise levels. As they are both blind, they use statistical methods to determine any abnormalities. Both codes we tested we grabbed from an online repository to improve[16].

The Color Filter Array Artifacts(CFA) is how digital cameras interpret images. When a camera takes a picture, it represents it's RGB values by interpolating the pixels in a certain pattern. What the algorithm looks for is inconsistencies between the patterns. For example, in the algorithm we tried to accelerate, the researchers take an image, split it up into sections of sixteen by sixteen and then attempt to use statistical methods to see if the transitions from one block to the next make sense given the context of the image. They give the block a score based on the deviation from the expectation, and if the score falls outside of a given standard deviation, then the section is flagged as likely spliced [3].

Noise level theory takes the estimated noise that an image has within each section and compares to the noise of other blocks. If the noise levels are too high or too low, the section is flagged as likely forged. The code that we tried to optimize does this by passing a high pass filter through a wavelet function on blocks that do not overlap each other, and when the statistical analysis is done they merge the blocks together to flack the combined sections as real or forged [8].

Algorithm 1 Convolution Pipeline

Require: $O_w = \frac{I_w - K_w}{S} + 1$ ▷ Right dimensions
Require: $O_h = \frac{I_h - K_h}{S} + 1$
$K_o[K_w][K_h]$ ▷ output of kernel multiplication
start ← False
done ← True ▷ To notify when work is done
while start ≠ True **do**
receive image and kernel pixels ▷ Wait to start
end while
done ← False
x_pix ← 0
y_pix ← 0
i ← 0
j ← 0
Move to next clock cycle
while x_pix ≤ K_w **do**
while y_pix ≤ K_h **do**
while i ≤ K_w **do** ▷ Multiplies sections of Kernel
while j ≤ K_h **do**
j ← j+1
K_o[i][j] ← K[i][j]*I[i+x_pix][j+y_pix];
end while
i ← i+1
end while
Next Clock Cycle
while i ≤ K_w **do** ▷ Adds kernel outputs
while j ≤ K_h **do**
j ← j+1
O[$\frac{x\_pix}{S}$][$\frac{y\_pix}{S}$] += K_o[i][j]
end while
i ← i+1
end while
y_pix += s
Next Clock Cycle
end while
y_pix ← 0
x_pix += s
end while
done ← True

Algorithm 2 Convolution Parallel

Require: $O_w = \frac{I_w - K_w}{S} + 1$ ▷ Right dimensions
Require: $O_h = \frac{I_h - K_h}{S} + 1$
$K_o[K_w][K_h][O_w][O_h]$
$Temp_o[O_w][O_h]$ ▷ output of kernel multiplication
start ← False
done ← True ▷ To notify when work is done
x_pix ← 0
y_pix ← 0
while x_pix ≤ K_w **do** ▷ Sets up parallel modules
  while y_pix ≤ K_h **do**
    Generate module ▷ Creates module at given xy position
    mod_start ← start
    while start ≠ True **do**
      Wait ▷ Idle until to start
    end while
    while i ≤ K_w **do** ▷ Multiplies sections of Kernel
      while j ≤ K_h **do**
        j ← j+1
        K_o[i][j][x_pix][y_pix] ← K[i][j]*I[i+x_pix][j+y_pix];
      end while
      j ← 0
      i ← i+1
    end while
    Next Clock Cycle
    while i ≤ K_w **do** ▷ Adds kernel outputs
      while j ≤ K_h **do**
        j ← j+1
        $Temp_o[\frac{x\_pix}{S}][\frac{y\_pix}{S}]$ += K_o[i][j]
      end while
      j ← 0
      i ← i+1
    end while
    y_pix += s
  end while
  x_pix += s
end while
while start ≠ True **do** ▷ Main logic starts
  receive image and kernel pixels ▷ Wait to start
end while
done ← False
  Next clock cycle
  Delay 1 clock cycle
i ← 0
j ← 0
while $i \leq O_w$ **do** ▷ Transfer from module output
  while $j \leq O_h$ **do** ▷ To actual output O[i][j] = $Temp_o$[i][j]
  end while
end while
done ← True

Figure 3: Color Filter Array (original, spliced, segmentation)

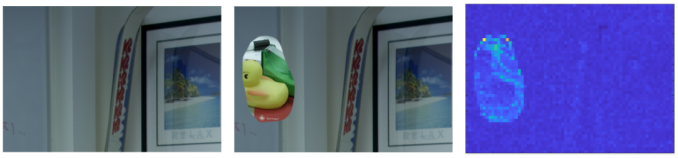

Figure 4: Noise Levels (original, spliced, segmentation)

3.4 Blind Framework Acceleration: Approach

The main bottleneck for speed was the fact that the code ran in MATLAB. In production, MATLAB code is usually compiled into a standalone program (.lib or .dll files), or in C/C++. Stepping from MATLAB to C can be achieved through the MATLAB Coder Addon.

For even faster inference, the code can be implemented as CUDA, which is similar to C in syntax but is a parallel computing platform and programming model created by NVIDIA [1]. Running code in the GPU can allow for a massive speed boost by having access to a broad class of intensive data parallel computation tasks [12].

In our approach, the blind methods scripts are implemented as MEX and CUDA, which according to related works, that have done a similar approach for image compression methods [12], yielded increase in speed performance as shown in the table below. To be able to implement the code into MEX and CUDA, we had to get rid of the dynamically allocated memory, ensuring that the code would be able to compile without memory allocation issues.

	Runtime	Speed-up
MATLAB Implementation	0.949422	N/A
MEX Implementation	0.786238	1.2×
CUDA Implementation	0.09680	1.7×

Figure 5: Image Compression Acceleration Results from [12]

For the wavelet transform function that was implemented in [12], the function took 524288 clock cycles in CPU environment's MATLAB implementation and the GPU CUDA implementation took 20480 clock cycles. Despite the 10× runtime as seen in the runtime above, time-consuming data transfers between host PC and the GPU card can limit the overall speedup as shown.

Nonetheless, the approach delivers a promising GPU focused acceleration method that will be implemented for the two blind methods.

4 RESULTS

4.1 ML Framework Results

The ML framework implements these networks via PyTorch's nn module, which allows very efficient speeds via GPU. On a Windows 10 PC with a NVIDIA RTX 3060, the results of the accuracies are as follows (After training for 30 epochs):

Validation Set Accuracies (55 Samples)	
Loss Label	0.22245
Accuracy Label	0.8268
Loss Mask	0.09680
Accuracy Mask	0.7848

Figure 6: Inference on Validation Set (Without Acceleration)

Where the accuracies were evaluated according to Percentage of Correct Key-points (PCK) [9], defined as (distance between prediction and target) $< somethreshold$.

The different internal networks were also measured for speeds to find the main bottle-neck. As mentioned before, the entire network consists of four parts. The local LSTM auto-encoder network, the global ResNet 16 network, the classification network, and the Segmentation step. The below speeds are averages of each section after running on the validation set. The speeds were measured using Python's $time.time()$ function. The results from the figure show

Validation Set Inference Speeds (seconds)	
Local Patch Network	0.1845
Global ResNet Network	0.065
Classification	0.0005
Segmentation	0.1225
Total SGN Network	0.3725

Figure 7: Inference on Validation Set (Without Acceleration)

that the local LSTM patch network was a significant bottleneck. According to [2], the entire Local Patch Network consists of 2 convolutional layers followed by LSTM. Despite this, it was decided to accelerate the convolution operation rather than the LSTM Recurrent neural network (RNN) because convolution operations still took 53% of inference time. Checking the layers for Local, Global and segmentation sections revealed 16 convolutional layers in the global network, 3 convolutions in the segmentation layer and 2 in the local network.

By accelerating the convolution operation by roughly 1000× the predicted speed performance increase is calculated as follows:

$$C_{SGN} = \frac{C_{lstm} + C_{global} + C_{segmentation}}{1000} \tag{1}$$

Where C_{SGN} is the new total time for all convolutional operations in the SGN network and C_{lstm}, C_{global}, and $C_{segmentation}$ are total time for convolution in each respective section in the SGN before acceleration. Measuring the ratio of convolution operations to the rest of the network, the hypothesized speed increase is as follows:

$$SpeedIncrease = \frac{C_{SGNnew} + N}{C_{SGNold} + N} \tag{2}$$

4.2 Blind vs CNN

We also ran a validation comparison on the CFA, Noise levels and CNN frameworks. We ran the codes in the Columbia data set and compared the mask acurasies result.

Validation Inference Accuracies	
Method	Mask Accuracy
CFA	63.9%
NOI	38.3%
CFA	78.48%

Figure 8: Blind and CNN: Accuracy Results (ROI)

As seen above, the CNN framework does significantly better than both the blind methods, being 14 percent higher than the CFA method and 40 percent more accurate than the noise levels method. While the training takes significant time, after it is done validating each image does not take much time, so it is a superior method to the other ones if only running one methodology.

4.3 CNN Hardware Acceleration Experiments

As the convolution takes 53% of the time in the framework we are testing, we could theoretically decrease the efficiency significantly through the FPGA. To measure how much we could increase the speed of the system, we measured the speed of the FPGA frameworks we have, and compared it to a convolution in torchnn with the same Image, Kernel, Step size and output. In our experiments we took a 7x7 image with a kernel size 3x3 and a step size of 2. The following are the image, kernel, and expected output.

We calculated the speed of the torchnn code by running it 1000 times, and through the built in time library we took an estimate time of how long an individual convolution would take. To measure the time it takes for our algorithm to work, we take the clock cycles and multiply it by the speed of the FPGA(Fmax). We also added one clock cycle for loading the image in and one for taking the image out, as those are not part of the algorithm but it will make a comparison closer to the GPU implementation.

Time to do example convolution(seconds)	
pytorch nn	$1.15 * 10^{-5}$
FPGA pip	$1.28 * 10^{-7}$
FPGA parallel	$7.5 * 10^{-9}$

Figure 9: Time to run the convolution

Our FPGA implementation is a lot faster than the torch nn algorithm by several orders of magnitude. Even taking a conservative approach and saying that the overhead of the CPU partitioning the image for the FPGA and stitching the outputs together slow down the algorithm tenfold, the FPGA would still be over 100× faster than the traditional GPU implementation.

For the total hardware acceleration we calculated the total convolution time and assumed our acceleration could be implemented on top of it. For the Global ResNet Network we took the convolution, max pooling, and ReLU sections and estimated the convolution

time by doing the assumption they are 90% of the total time in that section, which takes us .059 seconds. In the Patch Network we manually measured all the convolutions and calculated a total of .137 seconds , making a total of .0196 seconds for the total network. Giving the ratio that we have between the pythorch implementation and the parallel one in the FPGA, we have a 1533 times increase. Even with a conservative estimation of the FPGA implementation slowed down tenfold by the CPU doing segmentation towards the FPGA, we still end up with a 153 times increase. That would reduce the total time to .0013 seconds. Which would make the network $\frac{.3725}{.3725-(.196-.0013)}$ times faster than before. Which is 2.10× as fast without the hardware acceleration.

4.4 Blind Acceleration Experiments

For the MATLAB approach we ran the respective codes with their image set so that the comparison amongst the methods was a fair one.

	CFA Runtime	NOI Runtime
MATLAB	0.700325	0.157405
MEX	0.635581	0.071557
CUDA	7.37	0.390944

Figure 10: Speed comparison of blind networks with and without acceleration

Some of the hypotheses held true when comparing the methods outlined in the experiment. The compiled C++ code was significantly faster than MATLAB, as MATLAB is an interpretation language, and we made modifications to the code so memory allocation would not be dynamic for the C++ compilation, allowing the memory to be more efficient. What was not expected was the GPU implementation being significantly slower than the MATLAB code. This may have been for the algorithms and code to not have been developed for parallelism, so sending the code to the GPU would not have had any advantage. It also may have been that the CPU was doing all the tasks the GPU was supposed to have.

5 MOVING FORWARD

5.1 Machine Learning Framework

An issue that we ran during resting is the limit of I/O pins in the Arias II board we where simulating. testing we got around the problem by loading the image and the kernel directly into the module, but that is not feasible during real world convolutions. To move around the problem we would need to move to a newer board such as ARIAS 10 to get access to more pins, or to optimize the algorithm around the pipeline method, feeding in only the section that you are convolving at the time, as this method was still proven to be faster than through pytorch. When one is able to get around the I/O pin issue, then scalability of the project would not be an issue, as for an image of 7x7 only used 15 ALUTs. Assuming that the amount of LUTs would increase in a quadratic quadratic as it is a two dimensional array one could still do 60x60 sections before one starts getting close to the logic limit without having any hit to time performance.

Another issue is that the image sizes typically change, so that could give more load to the CPU to segment the image into proper sections to feed it into the FPGA. That issue could be mitigated by having different image sizes available, but that would be space inefficient.

One thing to note is that the implementation of this ML hardware acceleration method only speeds up the convolution operation. However in actual deployment, data transfer between the rest of the network (i.e. CPU) and the FPGA can potentially be a main bottleneck. The less data transfers, the better the implementation will be. Implementing ReLU as well into the FPGA is of high priority because in between convolution lies a lot of ReLU as well as pooling layers. Implementing blocks of these operations together onto the FPGA will be very beneficial to speed itself and the data transfer bottleneck.

5.2 Blind Framework

For the blind methods the main goal would be to optimize the code to run in CUDA where it could take advantage of the parallelization going on in the GPU. This could be done for the CFA algorithm, which shows more promise in giving close results to the CNN and does take more time to run compared to the noise levels algorithm.

Due to the two implementations being converted entirely into CUDA and MEX, the entirety of the inference steps were holistically accelerated. Moving forward, other blind methods, such as Compression Artifacts and Block Artifact Grid could also be tested.

REFERENCES

[1] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *International Conference on Compiler Construction*. Springer, 244–263.

[2] Xiaodong Cun and Chi-Man Pun. 2018. Image splicing localization via semi-global network and fully connected conditional random fields. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*. 0–0.

[3] Ahmet Emir Dirik and Nasir Memon. 2009. Image tamper detection based on demosaicing artifacts. In *2009 16th IEEE International Conference on Image Processing (ICIP)*. IEEE, 1497–1500.

[4] Li Fei Fei. 2021. Convolutional Neural Networks (CNNs / ConvNets). https://cs231n.github.io/convolutional-networks/

[5] Adeel Hashmi. 2011. *Hardware Acceleration of Network Intrusion Detection System Using FPGA*. Ph. D. Dissertation. Manchester Metropolitan University.

[6] Griffin Lacey, Graham W Taylor, and Shawki Areibi. 2016. Deep learning on fpgas: Past, present, and future. *arXiv preprint arXiv:1602.04283* (2016).

[7] Fei-Fei Li, Ranjay Krishna, and Danfei Xu. 2021. Convolutional Neural Networs(CNNs/ConvNets). https://cs231n.github.io/convolutional-networks/

[8] Babak Mahdian and Stanislav Saic. 2009. Using noise inconsistencies for blind image forensics. *Image and Vision Computing* 27, 10 (2009), 1497–1503.

[9] Tewodros Legesse Munea, Yalew Zelalem Jembre, Halefom Tekle Weldegebriel, Longbiao Chen, Chenxi Huang, and Chenhui Yang. 2020. The progress of human pose estimation: a survey and taxonomy of models applied in 2D human pose estimation. *IEEE Access* 8 (2020), 133330–133348.

[10] Tian-Tsong Ng, Jessie Hsu, and Shih-Fu Chang. 2009. Columbia image splicing detection evaluation dataset. *DVMM lab. Columbia Univ CalPhotos Digit Libr* (2009).

[11] Dimitris Poulopoulos. 2021. How to produce a deepfake video in 5 minutes. https://towardsdatascience.com/how-to-produce-a-deepfake-video-in-5-minutes-513984fd24b6

[12] Vaclav Simek and Ram Rakesh Asn. 2008. Gpu acceleration of 2d-dwt image compression in matlab with cuda. In *2008 Second UKSIM European Symposium on Computer Modeling and Simulation*. IEEE, 274–277.

[13] Luisa Verdoliva. 2020. Media forensics and deepfakes: an overview. *IEEE Journal of Selected Topics in Signal Processing* 14, 5 (2020), 910–932.

[14] Dong Wang, Ke Xu, Jingning Guo, and Soheil Ghiasi. 2020. DSP-efficient hardware acceleration of convolutional neural network inference on FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4867–4880.

[15] Fei Yan, Zhuangzhuang Zhang, Zhaodong Ding, Huaxi Zhang, Yinping Liu, and Jia Liu. 2021. Design of Hardware Acceleration System for Video Image Reconstruction Based on AXI Bus. In *2021 IEEE 15th International Conference on Electronic Measurement & Instruments (ICEMI)*. IEEE, 162–167.

[16] Markos Zampoglou, Symeon Papadopoulos, and Yiannis Kompatsiaris. 2017. Large-scale evaluation of splicing localization algorithms for web images. *Multimedia Tools and Applications* 76, 4 (2017), 4801–4834.