Pursuing Energy Efficient Solutions for Object Detection in Small Underwater Robots

Rafael Vieira
ISR, Electrical and Computer Eng.
University of Coimbra
Coimbra, Portugal
rafael.vieira@isr.uc.pt

Jorge Lobo
ISR, Electrical and Computer Eng.
University of Coimbra
Coimbra, Portugal
jlobo@isr.uc.pt

***Abstract*—Small underwater robots can provide valuable environmental and marine life data, and play a vital role given the horizon of climate change, by helping to control the strain on our planet's natural resources. In this work we pursue edge computing energy efficient solutions for marine animal identification by UAVs (Underwater Autonomous Vehicles). We rely on a well known CNN for object detection, YOLO (You Only Look Once), and target reconfigurable logic systems (FPGA) that enable a dynamic and multipurpose usage of the computing resources onboard the UAV. We are using the simpler YOLO v3 Tiny and explore trade-offs between power consumption, latency, frames per second, and detection metrics. We also evaluate how filtering pre-processing can be helpful on both training and detection. Several frameworks are explored, namely PYNQ, FINN and Vitis-AI, to efficiently implement the detector on the FPGA, and some insights on the relative benefits of each are drawn. Results are presented for a total of ## implementations across ## baseline and target systems. The accelerated version with reconfigurable logic (Zynq UltraScale+ MPSoC ZCU104 FPGA) consumes 56.364% less energy than the GPU version, utilizes less hardware resources, achieves an mAP of 82.6%, and has a performance of around 70.05 FPS.**

***Index Terms*—FPGA, Yolo, Object Detection, Edge Computing, Underwater Image.**

I. Introduction

A. Motivation

To a large extent, the oceans of our planet still remain unexplored. Exploring such a wide environment would be easier and more efficient using remote devices, or robots. In order to use small robotic devices for object detection, one method is to rely on the remote device to gather data, and then process it inside a cloud computing server. However, in situations where real-time data processing is required, cloud computing is not the best alternative, due to a lot of latency, bandwidth shortage, and energy consumption when sending and receiving data. For these kinds of situations edge computing is a better option, since it can obtain and process data simultaneously. Object detection algorithms are typically run in Graphics Processing Units (GPUs), and using one in a device with limited power can shorten the autonomy by a considerable amount. However this might not be the case for low powered GPUs. The comparison in energy efficiency is made with an RTX-3060 GPU (simulating the cloud computing approach) and a Zynq UltraScale+ MPSoC ZCU104 FPGA (simulating the edge computing approach), since we intend to provide an energy efficient solution for object detection in small underwater robots, and didn't explore extensively other possible HW (Hardware) platforms in edge devices.

Underwater images are usually hindered by radiance propagation. The captured images in underwater environments are affected by a blue, or some times, more greenish tint. There are many different types of filtering algorithms that make underwater images more pleasant for human perception, and because humans may help in the classification of marine species, the studied CNN, in this case YOLOv3-Tiny, may benefit from a pre-processing filtering of the input data.

B. Related Work

*1) **Underwater object detection**:* The use of CNNs in FPGAs for underwater environments is addressed in [1], but uses CNNs only for obstacle detection, like rocks, or plants, and not for marine life monitoring as we intend.

The work described in [2] accelerates a ZYNQ processor for fish detection in underwater images, where it wasn't possible to achieve better energy efficiency than a GPU, but it was still better than a CPU. Nevertheless, the acceleration of a ZYNQ processor to detect marine species is close to our goal, but it uses MobileNet for detection and UNet for underwater image enhancement, hence, the tools and methods used differ from our work. Using an encoder/decoder like UNet as a pre-processing method is the same as using extra layers that focus purely on image enhancement. However, to train UNet we need a dataset with mask annotations. In our case, to use UNet we would need a dataset with both YOLO and mask annotations (to train YOLO), or a dataset with mask annotations and a high variety of samples, to create a UNet model that would work on any marine species dataset.

To train a CNN using other than the encoder/decoder approach to improve the visibility of images, it is necessary to have ideal images, i.e. images just like the original input but with good visibility, possibly filtered by image processing algorithms.

Therefore we shouldn't have much problems relying on traditional image enhancement algorithms, since they tend to be approximated by CNNs focused on image enhancement.

A reduced version of YOLOv3 for underwater environments is presented in [3]. It is a good architecture to use on FPGAs and other devices alike, as it greatly reduces the number of FLOPs with little reduction in other metrics. However, this architecture was not used but kept in mind for future work, since there is more documentation available for vanilla YOLO architectures.

The studies conducted in [4] and [5] compare the performance of some versions of the YOLO architecture on the same dataset that we use. They also propose their version of Tiny YOLO (based on version 4), so the work described is used as a comparison to our results when it comes to model accuracy and speed.

*2) **Framework related comparisons**:* There are several different frameworks for Intelecutal Property (IP, or block of logic data) generation in FPGAs. These frameworks are often used to generate accelerators. Sometimes, the accelerator is already given, and needs to be programmed, as it is the case for VitisAI's Deep Learning Processing Unit used in [6], or the NVIDIA Deep Learning Accelerator (NVDLA) used in [7].

The most recently used tools for this matter are VitisAI and FINN, developed by Xilinx. FINN operates on top of PYNQ, a framework to program the Zynq7000 processor present in most of Xilinx boards, with Python language. A comparison between these frameworks is discussed in [6], where it is stated that VitisAI performs slightly better than FINN, but making custom accelerators outperforms the use of frameworks (at least for the custom accelerator used in [6]). There are several other works including the FINN framework [8] [9], but the same can't be said about VitisAI since it's more recent than FINN. It is stated in [6] that VitisAI provides a more sequential approach than FINN which is a more fine-grained oriented accelerator in spite of being outperformed. This is because FINN may be more suited for low powered FPGAs since the HW resources used depend on the model being accelerated, and quantizations lower than 8 bits are supported.

The work discussed in [10] explains trade-offs between using High Level Synthesis (HLS) and Overlays where it was found that HLS provides lower power consumption in comparison with Overlays, but using Overlays provides much faster processing. Using HLS would suit better our objectives, which stand for prioritizing lower power consumption. However, since relying only on HLS is still a very extensive job, a better approach lies on using tools like FINN and VitisAI.

C. Contribution

The practical contributions are included in the topics below for the benefit of the interested public or those who wish to further pursue our work.

1) Problems were discovered in the brakish dataset [11], which is commonly utilized in state-of-the-art for detection algorithms in underwater environments. These issues stemmed from a lack of variation in the inputs and were resolved by using data augmentations.
2) The data augmentations improved detection results and allowed our model to compete with other state-of-the-art models that employ more sophisticated architectures. Currently, the model with the best mAP in the state of the art [4] for this dataset is 93.56% with YOLOv4, while 92.40% mAP was reached in our work with YOLOv3-Tiny.
3) The comparisons between FINN and Vitis-AI are given as a consequence of the knowledge collected throughout this project.
4) In Vitis-AI, an alternate workflow is encouraged, which avoids the issue where Darknet is not supported in the Pytorch environment for Vitis-AI.
5) Finally, results for the implementation using the *Zynq UltraScale+ MPSoC ZCU104* FPGA [12] are provided. The accelerated version with reconfigurable logic consumes 56.364% less energy than the GPU, utilizes less hardware resources, maintains an acceptable mAP of 82.6%, and has a performance around 70.05 FPS.

D. Paper Overview

The paper is structured as follows: The description of the implementation and methods used are in Section II, and the achieved results are presented in Section III. Finally, conclusions are drawn in Section IV.

II. Methods and Implementation

A. Filtering

We applied different filtering algorithms on the dataset, producing some interesting results discussed in Section III. The code for RoWS [13], CLAHE [14] and GC [15] filtering algorithms was adapted from [16]. For CLAHE, the OpenCV library already has a function to produce a CLAHE image by giving the tile grid size and clip limit as input parameters. GC on the other hand, was done by applying directly **equation 1** in each pixel. As for RoWS and RGHS, they are more complex and have a mix between OpenCV and the equations described in the original articles [13] and [17] respectively.

RGHS code is also available at [16], but it contains many mistakes in the color restoration and relative global histogram stretching steps. So, the RGHS algorithm was fixed by taking advantage of the setup in [16], and re-coding in Python the equations in [17]. Note that the code in [16] was not developed by the authors of the works refered previously ([15], [14], [13] and [17]), but by other interested individuals in [16].

$$I_{out} = cI_{in}^{\gamma} \tag{1}$$

B. Training, testing, and detection

The training of the model was issued with 500 epochs and 16 images per batch. The set of hyper paremeters used was the same as the one which comes with the original cfg file (obtainable in the YOLOv3 author's website [18]), with the exception of the batch size and the number of epochs. The confidence and non-maximum-suppression thresholds used were 0.5, so when we refer to mAP, it is also known as $mAP_{0.5}$. The pre-processing applied during training consists in applying transformations to the loaded batch images. Such as:

1) Padding to a square/resizing to a shape varying between 288x288 and 448x448 (when running testing and detection algorithms we use images with size 416x416 only). The shape is chosen randomly between the ones that allow separation in grid cells.

2) Flipping the images horizontally, therefore creating a symmetric image (the images to be flipped are chosen randomly).

C. Inference on FPGA

1) PYNQ: The PYNQ framework provides a user friendly platform to program the Zynq processor in Python, present in many Xilinx development boards. The Python programs can be accelerated with the FPGA, but the configured IP needs to be compiled and designed previously inside Vivado. Therefore, PYNQ was only used to measure power consumption on sequential detection and filtering algorithms, to compare with the accelerated Vitis-AI version.

2) VitisAI: Vitis-AI creates an overlay for the Zynq processor, working as an accelerator for inference in CNNs. This accelerator is called Deep Learning Processing Unit (DPU). The DPU can be reconfigured to use different hardware resources inside Vivado, but as soon as the FPGA is booted, it becomes programmed with these IP cores, the hardware utilization stays fixed. We used the DPU architecture DPUCZDX8G-B4096, and 8 bit post training quantization. The DPU does not support max pooling layers of stride 1 therefore this layer was removed before training the model. Vitis AI supports PyTorch and TensorFlow, however, during quantization some errors occur inside YOLO layers in the PyTorch environment, and the only successfully implementations with YOLOv3-Tiny were achieved inside the TensorFlow 1.15 environment. Since the CUDA and CUDNN versions compatible with TensorFlow 1.15 are not supported in recent GPUs like the one we used, training in TensorFlow 1.15 was not possible. Hence, the model was trained in PyTorch, and saved in Darknet format, and then converted into a TensorFlow frozen graph model inside a TensorFlow 1.15 Python environment.

3) FINN: In spite of not achieving a successful implementation with FINN, we consider that it is a valuable tool for low powered FPGAs like the PYNQ-Z1 and PYNQ-Z2 boards, as it allows in depth customization for quantization and parallelization, an important feature when creating accelerators that need to fit into the available HW with precision. Therefore, we provide some comparisons in **Table** I between VitisAI and FINN for better understanding pros and cons between them.

TABLE I: Pros and Cons of using FINN and Vitis AI

	FINN	**Vitis AI**
Concept	Builds an accelerator for a given model. Hardware resources vary from model to model.	The accelerator is already built, and needs to be configured with an xmodel.
Deployment	Generates a bitfile to be loaded as a PYNQ overlay	Generates a xmodel file that configures the DPU to run a certain model
Supported Platforms	Any Zynq devices, including low power FPGAs like PYNQ-Z1 and PYNQ-Z2	Supports limited platforms, usually FPGAs with more HW resources like Ultrascale+ series or Alveo boards.
Energy Efficiency	May provide superior energy efficiency since it supports low powered FPGAs and builds accelerators for specific case scenarios.	Due to restricted accelerator architectures, configurations, and supported platforms, may provide worse solutions than FINN when it comes to energy effiency.
Quantization	Widely used with Quantization Aware training	Widely used with both Quantization Aware Training and Post Training Quantization
Ease of use	Requires an engineer with experience/expertise to, needs user interaction in all stages of the workflow defining when and how parallelization/ quantization occur, and how to tune all the steps to refine the model until it generates a bitfile.	The customization of parallelization/quantization methods is optional, there is a default configuration to use as example. Diving in extensive low level flow is not required. Easier to use since it requires less expertise.
Documentation, Support, and Community	Released in 2017, the community is no longer as active as it was, and devs dont show much activity. Documentation usually only provides an overview of the project. Some tutorials and examples no longer operate correctly.	Released in 2020, with recent dev and community activity, many problem-solving threads in github issues forums. Widely documented. Ease of access to low level flow, All examples/tutorials are operational.
Frameworks for model training	Pytorch and Brevitas.	Pytorch, Tensorflow 1.15 and Tensorflow 2.0. Works better with Tensorflow 1.15.

III. Results

A. Filtering

Since the images from the brakish dataset are captured with a flashlight, they have increased brightness in the center, and become dark near the corners. GC was used only to slightly darken the image, using $\gamma = 1.2$.

Contrast correction with CLAHE was the most efficient method, using a 8x8 tile grid size, and a clip limit of 4. But, by using CLAHE to enhance sharpness we are sill lacking in color correction, as it's still affected mainly by green radiation.

The green tint didn't disappear in RoWS, and RGHS was stained by a blue color instead. RoWS and RGHS were designed to enhance the quality of underwater images that were affected by light scattering of sunlight, but these images are illuminated by a flashlight, the science behind light scattering is not exactly the same.

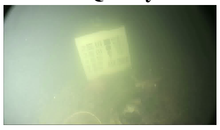
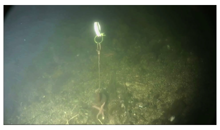
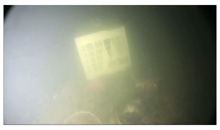
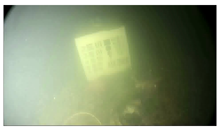
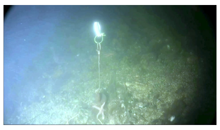
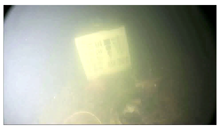
As for Color Restored GC_CLAHE, it possesses the original θ_g described in [17] divided by a factor of 1.275 (experimental value) in order to compensate the excess presence of green color in the image, and θ_b was left as it is. This is a custom made filtering algortihm using CLAHE, GC and Color Restoration from RGHS, obtained by experimenting CLAHE and GC with RGHS.

B. Metrics and Augmentation results for YOLOv3- Tiny

Table IV shows the obtained metrics after training YOLOv3-Tiny with the brackish dataset.

While using the brakish dataset, over-training can't be detected through the provided testing set, since the images lack variety from the training set. This can be proven by very sligtly changing the values of the pixels, using GC filtering in the testing set. **Table III** has the results obtained by evaluating the model with the GC filtered testing set with $\gamma = 1.2$, which caused an mAP drop of 14.1%. Close to zero mAP was achieved when doing the same procedure in CLAHE filtered testing sets, which further changes the pixel's values. This behaviour of the model exposes the lack of input variation provided by the dataset, since this problem

TABLE II: Table of filtered and unfiltered samples from the brakish dataset.

Filter	Better Quality	Low Quality
No Filter		
GC		
CLAHE		
GC_CLAHE		
RoWS		
RGHS		
Color Restored GC_CLAHE		

was undetectable with the original testing set. In order to achieve better mAP results and partially solve this problem, the dataset was expanded 6 times with 6 different filtering algorithms depicted in **Table II**. This data augmentation process on top of the ones issued during training (discussed in Section II) provided better mAP results for the original and filtered testing sets. The detection metrics for each testing set are in **Table V**.

TABLE III: Testing results on GC filtered dataset, model trained on original brackish dataset.

Class	AP	Precision	Recall	F1
fish	0.874	0.667	0.919	0.773
small_fish	0.545	0.510	0.783	0.618
crab	0.845	0.842	0.899	0.870
shrimp	0.842	0.739	0.894	0.809
jelly_fish	0.476	0.603	0.612	0.608
star_fish	0.889	0.834	0.918	0.874
		mAP: 0.745		

C. Energy Consumption

The results obtained through Vitis-AI provide better energy efficiency than using a GPU, as it is shown in **table VII**. Using the FPGA results in a performance drop of 71.629% in FPS, energy efficiency raise of 56.364% in Joules per frame, and a mAP drop of 9.8%. These values are considered a good outcome since the frame rate and mAP are still in a good value range of 70.05 FPS and 82.6% respectively.

TABLE IV: Testing results with original brackish dataset (mAP values are not in percentage, therefore ranged between 0 and 1).

Class	AP	Precision	Recall	F1
fish	0.977	0.946	0.984	0.964
small_fish	0.606	0.662	0.801	0.725
crab	0.966	0.951	0.988	0.969
shrimp	0.943	0.888	0.9824	0.933
jelly_fish	0.837	0.780	0.919	0.844
star_fish	0.991	0.981	0.994	0.987
		mAP: 0.886		

A matter that should not be overlooked is the energy consumption in idle time. For idle times, while using PYNQ, the instant power supply was around 12.3W, and in Vitis-AI around 17.3W. PYNQ can load and unload bitfiles which can be accelerators developed by FINN or other frameworks, as long as they have the correct connections to the Zynq processor being used. This means that using PYNQ overlays would save an even greater amount of power if we take idle time power consumption into account.

Table VI shows the power measurements and energy consumed for filtering algorithms studied in this work. To execute filtering algorithms the processor has to run operations for each pixel of the image. For more complex algorithms iterating through every pixel of the image several times. When comparing this process with model inference, the image only needs one pass through the model to obtain a result, and the model in question ”only looks once”, therefore being fast enough so that the higher instant power being consumed justifies for the much shorter processing time. The most energy efficient filtering algorithm is CLAHE which spends nearly two times (in joules per frame) the energy spent by the model inference.

IV. Conclusions

A. Choice of architecture

There are many techniques emerging that are increasingly sophisticated to increase the speed and accuracy of CNNs. People that are interested in the subject tend to incorporate new discoveries into their work. This is not necessarily a good thing, since we must always remember to select a CNN architecture that is appropriate for the dataset the network will train on. If the objective is to detect objects from the COCO dataset, a basic three-layer CNN will not yield decent results, but it will already be a suitable design for smaller datasets like MNIST or CIFAR10. The dataset’s complexity should match the complexity of the architecture.

The YOLOv3 Tiny architecture can only reach 33% mAP on the COCO dataset, which has a wide range of inputs and 80 classes, but the YOLOv4 architecture can achieve 65.7%, nearly double that of the YOLOv3-Tiny. In other words, with the COCO dataset, more sophisticated architectures are required to get acceptable metrics. This is not the case for the Brakish dataset, which has a limited variety of inputs and just six classes. It is typical for simpler architectures to get high mAP levels when properly configured and used in the right scenario.

YOLOv4 and further versions still perform well in almost all situations. Yet, the use of older versions should be more

TABLE V: Testing results on filtered and unfiltered testing sets, model trained on augmented brackish dataset.

	AP							
	Unfiltered	**GC**	**CLAHE**	**GC_CLAHE**	**RoWS**	**RGHS**	**Color Corr. GC_CLAHE**	**All of them together**
fish	0.961	0.970	0.967	0.971	0.954	0.958	0.979	0.965
small_fish	0.740	0.722	0.693	0.682	0.708	0.712	0.714	0.702
crab	0.974	0.971	0.975	0.974	0.981	0.972	0.979	0.972
shrimp	0.956	0.914	0.943	0.945	0.978	0.971	0.916	0.944
jelly_fish	0.915	0.891	0.908	0.892	0.918	0.915	0.905	0.900
star_fish	0.995	0.997	0.988	0.992	0.997	0.992	0.982	0.991
mAP	0.924	0.911	0.912	0.910	0.923	0.920	0.912	0.913

TABLE VI: Energy consumption for filtering algorithms in Ultrascale's Zynq+ processor

	GC	**CLAHE**	**GC_CLAHE**	**Color Corr. GC_CLAHE**	**RoWS**	**RGHS**
Joule per Frame	27.861	7.511	30.177	267.861	772.824	738.256
Average Instant Power (W)	12.521	12.534	12.577	12.358	12.356	12.360

TABLE VII: Summary of measurements for performance and energy efficiency for the different HW platforms.

	Inference Performance and Efficiency			
Hardware Platform	**FPGA**		**Computer**	
	CPU	CPU+DPU	CPU	CPU+GPU
Frames per Second	1.88	70.05	16.79	246.91
Joules per Frame	3.83	0.31	8.45	0.55

encouraged, since we achieved nearly the same mAP with YOLOv3-Tiny as other works that focused on the advantage of using more complex architectures.

B. Underwater Image Filtering, where does it stand?

There are several reasons to exclude the filtering methodology used in this work as a pre-processing algorithm:

1) Since the filtering process isn't accelerated, it would take a long time to run, resulting in high latency and increased energy consumption.
2) The filtering algorithms studied in this dissertation didn't positively influence the model as pre-processing algorithms.
3) By analysing **Table V** one can conclude that the studied filtering methods don't create an advantage when detecting species with the augmented model either.

Because of these reasons, we can look at the studied filtering methods as post-processing algorithms for better human perception of the obtained results, and of course, as dataset augmentations.

If the purpose is for better human perception, this post processing could also occur outside of the edge device, after gathering the data for a cloud device to analyse.

C. Energy efficiency, did the result match the expectations?

It was said in **section I-B** that this work would be focused on prioritizing low energy consumption. However we used Vitis-AI which provides high frame rates and focuses on performance, as it only supports FPGAs designed for high performance computing, and offers limited amount of architecture configurations for the DPU. The initial objective would be relying on FINN since it offers endless hardware configurations, for example, by simply choosing different combinations of folding parameters (PE and SIMD). However, since FINN was left behind due to technical issues and difficulties, Vitis-AI was used as an amend.

D. Future Work

Before going on to supplementary and extra tasks, there are still improvements that may be made to this project, such as:

- Explore and compare the results from implementing the custom made architectures for underwater environments at [3] and [5], since they seem to be a better choice than YOLOv3-Tiny, using less parameters and less FLOPs.
- Use an accelerator created with the FINN framework.
- Try other DPU architectural configurations and tools provided by Xilinx to support projects made with Vitis-AI, such as the Vitis-AI Optimizer and Profiler.
  It is also possible to add custom IP in the Vivado project, this raises the possibility of having dedicated IP to read images from a camera in real time.

The following interesting additions would be useful, perhaps after the project's core is more stable:

- Make use of the encoder/decoder approach to incorporate filtering into the edge device, which would use CNNs and make it simpler to parallelize the filtering procedure, since there are tools like FINN and Vitis-AI that can do this for us.
- Using two filter modes that would be selected based on the turbidity of the water (this is possible with turbidity sensors). When the water is clear, filtering algorithms such as RoWS or RGHS may produce superior results. In circumstances when the water is more turbid or visibility is reduced for other reasons, the edge device can employ the filtering developed in this work (Color Restored GC_CLAHE). This opens up the prospect of investigating the undersea habitat at night, or greater depths where the light levels are low.

References

[1] Minghao Zhao, Chengquan Hu, Fenglin Wei, Kai Wang, Chong Wang, and Yu Jiang. Real-time underwater image recognition with fpga

embedded system for convolutional neural network. *Sensors*, 19(2), 2019.

[2] Liangwei Cai, Ceng Wang, and Yuan Xu. A real-time fpga accelerator based on winograd algorithm for underwater object detection. *Electronics*, 10:2889, 11 2021.

[3] Lin Wang, Xiufen Ye, Huiming Xing, Zhengyang Wang, and Peng Li. Yolo nano underwater: A fast and compact object detector for embedded device. In *Global Oceans 2020: Singapore – U.S. Gulf Coast*, pages 1–4, 2020.

[4] Minghua Zhang, Shubo Xu, Wei Song, Qi He, and Quanmiao Wei. Lightweight underwater object detection based on yolo v4 and multi-scale attentional feature fusion. *Remote Sensing*, 13:4706, 11 2021.

[5] Shijia Zhao, Jiachun Zheng, Shidan Sun, and Lei Zhang. An improved yolo algorithm for fast and accurate underwater object detection. *Symmetry*, 14(8), 2022.

[6] Michal Machura, Michal Danilowicz, and Tomasz Kryjak. Embedded object detection with custom littlenet, finn and vitis ai dcnn accelerators. *Journal of Low Power Electronics and Applications*, 12(2), 2022.

[7] Shenbagaraman Ramakrishnan. Implementation of a Deep Learning Inference Accelerator on the FPGA. Master's thesis, LUND UNIVERSITY, Sweden, 2020.

[8] Michaela Blott, Thomas Preusser, Nicholas Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks, 2018.

[9] Thomas B. Preuser, Giulio Gambardella, Nicholas Fraser, and Michaela Blott. Inference of quantized neural networks on heterogeneous all-programmable devices. In *2018 Design, Automation &amp Test in Europe Conference &amp Exhibition (DATE)*. IEEE, mar 2018.

[10] Colin Yu Lin, Zhenghong Jiang, Cheng Fu, Hayden Kwok-Hay So, and Haigang Yang. Fpga high-level synthesis versus overlay: Comparisons on computation kernels. *SIGARCH Comput. Archit. News*, 44:92–97, 2017.

[11] Malte Pedersen, Joakim Bruslund Haurum, Rikke Gade, Thomas B. Moeslund, and Niels Madsen. Detection of marine animals in a new underwater dataset with varying visibility. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[12] Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit Webpage. https://www.xilinx.com/products/boards-and-kits/zcu104.html. Accessed: 2022-08-13.

[13] Liu Chao and Meng Wang. Removal of water scattering. In *2010 2nd International Conference on Computer Engineering and Technology*, volume 2, pages V2–35–V2–39, 2010.

[14] Karel J. Zuiderveld. Contrast limited adaptive histogram equalization. In *Graphics Gems*, 1994.

[15] Shanto Rahman, Md Mostafijur Rahman, M. Abdullah-Al-Wadud, Golam Dastegir Al-Quaderi, and Mohammad Shoyaib. An adaptive gamma correction for image enhancement. *EURASIP Journal on Image and Video Processing*, 2016(1):35, Oct 2016.

[16] Yan Wang, Wei Song, Giancarlo Fortino, Lizhe Qi, Wenqiang Zhang, and Antonio Liotta. An experimental-based review of image enhancement and image restoration methods for underwater imaging, 2019.

[17] Dongmei Huang, Yan Wang, Wei Song, Jean Sequeira, and Sébastien Mavromatis. Shallow-water image enhancement using relative global histogram stretching based on adaptive parameter acquisition. In Klaus Schoeffmann, Thanarat H. Chalidabhongse, Chong Wah Ngo, Supavadee Aramvith, Noel E. O'Connor, Yo-Sung Ho, Moncef Gabbouj, and Ahmed Elgammal, editors, *MultiMedia Modeling*, pages 453–465, Cham, 2018. Springer International Publishing.

[18] Joseph Chet Redmon's webpage. https://pjreddie.com/. Accessed: 2022-08-08.