Engineering of a High-throughput Multi-Camera Scale-invariant Collision Avoidance Pipeline

Alexander Klein, Aaron Lye, Jannis Stoppe
Institute for the Protection of Maritime Infrastructures
German Aerospace Center (DLR)
Bremerhaven, Germany
Alexander.Klein@dlr.de, Aaron.Lye@dlr.de, Jannis.Stoppe@dlr.de

***Abstract*—Modern object detection algorithms often work on fixed image sizes for their input and usually rely on scaling the input data to match the required measurements, resulting in issues with input data that significantly differs from these fixed dimensions.**

This work illustrates a real-time visual pipeline that processes and leverages SAHI-based patch extraction with batched YOLO inference for scale-invariant object detection for a multi camera system with a centralized computing architecture. The developed solution contains optimization steps which enable high data throughput and is evaluated with regard to the complex dependencies between throughput and input delay.

***Index Terms*—Multi-camera detection, scale-invariant detection, collision avoidance, detection throughput optimization, real-time maritime perception**

I. Introduction

Autonomous maritime surface vessels have to integrate with maritime traffic. However, to navigate in the maritime environment and follow maritime rules and regulations, this requires reliable, low-latency perception of surrounding objects ranging from tiny buoys and swimmers to large vessels. Not only the object size but also the distance matters yielding very different scales for objects of the same class, e.g. distant vessels may be represented by only a handful of pixels.

Conventional object detectors in the visual domain, such as the single-stage YOLO family, operate on fixed-size inputs and consequently struggle to detect small-scale targets that occupy only a few pixels in high-resolution imagery. This may result in serious navigational issues as a late detection of such objects might be the cause for critical evasive maneuvers when early small course corrections would have been enough.

To address this issue, in this paper we present a real-time visual pipeline that processes and leverages SAHI-based patch extraction with batched YOLO inference for scale-invariant object detection illustrated in Figure 1 for a multi camera system with a centralized computing architecture. Given our developed solution we show optimization steps which enable high throughput, as well as evaluate complex dependencies between throughput and input delay.

II. Background

A. Object Detection

Object detection is a sub-discipline of image processing with the goal of identifying and localizing objects in a given image. Unlike pure classification, which looks for the presence of certain objects in the full image, an object detector infers the location, area, and type of a set of objects and returns the bounding boxes of the most likely candidates. This makes the method essential in many tasks requiring scene understanding, e.g., medical imaging, autonomous driving, robotics, or surveillance. While early methods used hand-crafted feature detectors, deep learning—and especially the introduction of convolutional neural networks (CNNs)—enabled much higher accuracy. One of the first proposed methods that showed results was region-based convolutional neural networks (R-CNN) [1], [2], which solved a two-stage problem by first localizing image patches that may contain an object (region proposal) and then using a secondary classification network for identification. To improve real-time performance, models such as YOLO [3] and SSD [4] solve the problem in a single inference step, producing a large number of possible bounding boxes; a non-maximum suppression step is then used as post-processing to filter boxes belonging to the same object.

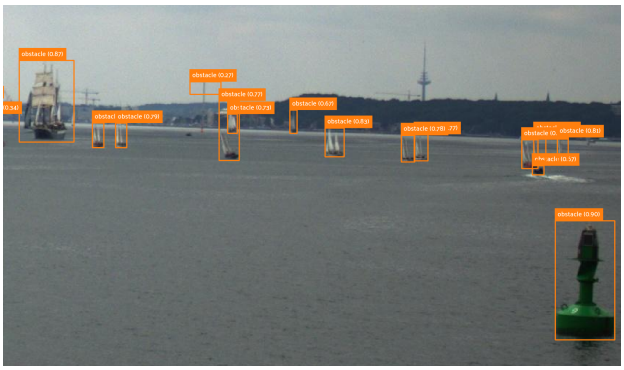

Fig. 1: Sliced detection results for an image snippet taken from a high resolution 19.6 MPix image showing detected object of varying scales.

B. SAHI: Slicing Aided Hyper Inference

Slicing Aided Hyper Inference (SAHI) [5] enables multi-scale inference without modifying the underlying detector expanding upon its' limited receptive field. The following steps are performed:

1) **Slicing:** Given a high-resolution query image I of size (h_I, w_I), SAHI generates a set of overlapping patches $\{P_k^I\}_{k=1}^n$ by sliding a window of fixed size (h_P, w_P) across the image with a fixed stride.
2) **Detection:** Along with the original query image (to include large object detection) the patches get resized and potential objects are inferred using a given detector.
3) **Non-maximum suppression** (NMS) is applied on the prediction to avoid generating multi-detections for the same object.
4) **Merging:** Finally the prediction results are aligned given the origin patches offset and scaling. Given that all predictions are transformed back into a unified image space, they are then merged using a custom non-maximum merging (NMM) which filters and merge predictions based on a similarity metric which takes into account class, confidence and their intersection over union (IoU).

Note that step 2 and 3 are part of the machine learning model and have been separated for performance reasons.

The method is fully model-agnostic; any object detector can be wrapped by SAHI's pre- and post-processing pipeline. Furthermore, SAHI can be applied to the training data itself. By slicing annotated images into the same patch grid, a dataset is produced that emphasizes small-object instances. Detectors fine-tuned on this subset exhibit markedly improved average precision on small targets compared to models trained on the raw images alone. Consequently, SAHI offers a practical, detector-agnostic solution that enhances small-object detection performance in the context of maritime collision avoidance.

III. Implementation

For maritime surface operations we developed a parallel end-to-end object detection pipeline [6] based on a centralized computing architecture, enabling real time collision avoidance for a multi-camera system. The detection system features 3x high-resolution (19.6 MPix) cameras spanning a 120 degrees horizontal field of view (FoV) connected to the central computing unit via Ethernet. The central server runs the software pipeline enabling real-time detections of multi-scale objects on the full resolution (≈ 60 Mpix) using SAHI + YOLOv11n detector [7] at it's core. The software pipeline is designed as push-based feed-forward architecture of connected computation modules, where each module is an independent processing unit that can consist of one or more nodes, solving a single task. Each module is designed to work as independent process that takes in some input x and generates an output y, exceptions are modules that have to interact with elements outside the pipeline including the camera system which gets the input from the camera sensor or producer modules that provide the output to some external API.

Consequently each module is only dependent on a single upstream module. This allows the implementation of each module as worker process given a startup configuration and an input and output buffer, where for non-source modules the input buffer is the output buffer of a previous module in the processing graph. For the data transportation between modules a single buffer based on shared memory is implemented, the buffer employs locks to secure data access as well as a field documenting last write operations to the buffer. By synchronizing the timestamp of the write access, each subsequent module can react to changes in the input data, avoiding the use of more complex signaling.

A watchdog creates and connects the pipeline modules holding the final reference to the shared memory and then observes the pipeline during operations. Due this design even a crash of a worker isn't critical to the pipeline itself as the modules have no cross dependencies and the final buffer reference is always with the watchdog. Following this design also allows to easily restart a worker processes without restarting the complete pipeline. In our case this behavior is mostly useful in case the network becomes unstable making the camera sensors unavailable, allowing the camera to crash safely and restart without delay.

The implementation of the modules and the pipeline graph are shown in Figure 2. The cameras system, receives the the interleaved frames of the 3 high resolution cameras, preprocesses them and writes the data in an unified buffer, allowing subsequent modules access to the raw image data. Afterwards the pipeline splits in the detection path and the visualization path. The visualization is done using a panorama stitching module that combines the camera frames into a single view based on their physical orientation and then stores the results in an ring buffer for external API access. For the detection of multi scale object, we employ a custom SAHI implementation separated into slicing, detection and merging modules. While technical each camera is independent the slicing and detection only consist of a single worker node in comparison to the merge module. This allows the application of more optimization techniques. The detection modules uses the bare bones core of ultralytics YOLOv11n [7] detection model, without any post processing steps, for inference and writes the results directly to the output buffer.

The merging modules include the post processing done by the model (NMS), as well the the NMM step which completes encapsulation by SAHI. Both post-processing methods involve numerous sequential operations, so we divided the computation across three nodes that can run independently in parallel. As each worker thread might operate at varying loads, we enforce lockstep operation of each using a synchronization lock. This lock acts as an information barrier that coordinates read and write operations on both the shared input and output buffers. This architecture allows us to split the module into multiple worker nodes without of desynchronizing the data stream.

The post processing module contains the multi camera merging, object tracking and distance estimation, which are operated in a sequential fashion partially due to shared setting, reducing overhead and complexity when providing the results to the API. The actual functionality is of not much interest for the paper. We refer the reader to [6].

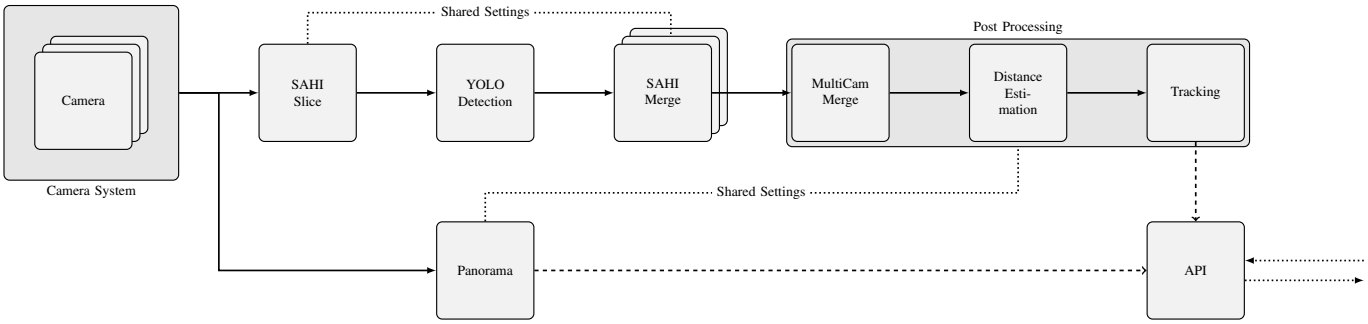

Fig. 2: Multi camera software pipeline implemented on the central compute unit for the real-time visual collision avoidance. The pipeline is split in two path, the (top) detection path which

IV. Optimization

A. Parallelization

The naive approach to object detection would simply execute all processing steps sequentially. Splitting this execution in parallel running modules as shown in our pipeline, we can improve the utilization of the hardware resources available to us and process the incoming camera frames in a non-blocking manner.

To minimize synchronization overhead, we enforce a strict "single-input" rule for each module. This eliminates the need for complex synchronization mechanisms required by multi-input systems.

Given an buffer is updated, a downstream module can take in new data for processing at their own pace, making the module always work at maximum throughput on the newest data. Because each module operates independently, the throughput of a branch is dictated by its highest-order bottleneck. In our case this effect enables high visual frame rates: the panorama module runs faster than the camera sensors and is not constrained by the largest bottleneck, which is the detection module.

B. GPU and compute accelerations

Graphics Processing Units (GPUs) are specialized for parallel processing, leveraging thousands of lightweight cores to execute the same instruction on multiple data elements simultaneously. This architecture makes them particularly effective for workloads exhibiting data-level parallelism, such as matrix operations, convolutions, and vectorized arithmetic. Modern deep learning models rely heavily on tensor operations which map efficiently to GPU execution. By offloading these computing-intensive stages of our pipeline to a GPU, we can achieve order-of-magnitude speedups compared to CPU-only execution, reducing inference latency per frame significantly.

Another significant gain came from vectorizing the selection loop in the post-processing of the model output. NMM operates as an iterative divide-and-conquer procedure: given a set S of n bounding boxes, we aim to partition it into j subsets $S_1, S_2, \ldots, S_j$ that groups boxes according to a distance metric. The algorithm starts with a seed box, computes its IoU with every other box, forms a subset, and removes those boxes from S. A new seed is then selected from the remaining boxes, and the process repeats until the set is empty. To accelerate this, we pre-compute all pairwise IoU values using GPU tensor multiplications, reducing the selection step to a single comparison per potential subset.

While offloading computational operations to the GPU often seems straightforward, a substantial performance gain was achieved by optimizing GPU calls and minimizing data flow from the CPU (RAM) to the GPU (VRAM). In machine learning, batching is a technique that stacks multiple inputs and infers their results in a single pass through a model, allowing maximum utilization of parallel computing units and reducing the overhead of sequential memory access. In this way it was more efficient to use a single detector module that computes the results for every sliced image patch from all cameras in a single step.

To transfer data between devices we followed the paradigm of keeping information as compressed as possible, which is why the camera module directly offloads the RAM buffer to the GPU after pre-processing the camera inputs. In the detection branch this means that the slicing module expands the data on the GPU instead of creating unnecessary large data transfers between devices. Later operations in the detection post-processing, which benefit from the CPU's sequential execution, only require the transfer of the filtered bounding boxes.

We optimized the model itself by using *float16* precision instead of *float32*, and additionally, using PyTorch's compile feature allowing graph optimization for the detection model with a fixed input size. Given our single batched inference call, this reduces computation time in the module significantly. Since the detection module is the largest bottleneck in the pipeline, this speed-up directly affects the delay and throughput of the detection branch.

V. Evaluation

For all evaluations we uses a workstation with an Intel Xeon Gold 6246R CPU with 96 GiB 3200 MHz DDR4 RAM and an Nvidia A6000 GPU with 48 GiB of VRAM. In the experiments we differentiate between two different operation modes, *base* represents the *eager* execution mode of the machine learning model whereas *opt* refers to the optimized compiled model as

Method	μ	σ
Basline	$0.7403s$	$0.0145s$
Sliced SAHI Framework	$4.6537s$	$0.0373s$
Sliced Pipeline at 1 fps	$0.2218s$	$0.0174s$
Sliced Pipeline Opt. at 1 fps	$0.1808s$	$0.0059s$

TABLE I: Runtime comparison of computations for the detection of $3 \times 19.6MPix$ camera images. The Baseline predictions is a no-slice down scaled prediction using a forward pass through the SAHI framework with an image size of width and height of $640x640$ pixel. The slicing methods use an horizontal and vertical slicing overlap of 10%.

Target fps		3	4	5	6	7
base	delay in s	0.31	0.35	0.48	0.47	0.48
	fps	3.00	4.00	3.97	3.87	3.64
opt	delay in s	0.28	0.24	0.27	0.40	0.40
	fps	3.00	4.00	5.00	5.28	4.85

TABLE II: Measured pipeline delay and frame rate versus the camera's target fps for both the baseline and optimized ML models. Delay standard deviation ranges from $0.01s$ to $0.06s$.

explained in section IV. Besides the model in the detection module the rest of the pipeline is unchanged.

A. Small Object Detection

In Table I we compare the runtime for a single prediction of the given camera system. The evaluation comprises two calls to the SAHI framework: a baseline method that predicts a single down-scaled image at the model's native 640×640 pixel resolution per camera, and a sliced prediction that uses a horizontal and vertical slicing overlap of 10 %. Our methods correspond to measurements taken from the pipeline operating with a fixed camera frame rate of 1 fps: the eager model execution, denoted *Sliced Pipeline*, and the compiled model, denoted *Sliced Pipeline Opt.*, which excludes our custom post-processing step.

The slicing process generates 64 patches per camera image, including the original frame; with three cameras this yields 195 images that must be processed in each forward pass for sliced predictions. The pipeline operates 21 times faster in eager-execution mode and 25 times faster when the model is optimized. Even though the detector must handle 195 more input data, the pipeline remains 3.3 (eager) and 4.1 (optimized) times faster than a single detection performed with the SAHI framework.

B. Pipeline Timings

Due to the parallelization of the pipeline, the maximum throughput is limited only by its bottleneck. However, an optimal throughput does not guarantee a minimal delay to the camera frames. Table II compares the delay and throughput of the detection branch as a function of the configured camera frame rate for the two selected configurations. The throughput rises until the detector reaches its computational limit: for the *base* configuration the maximum throughput is between 4 and 5 fps, whereas for the *opt* configuration it is between 5 and 6 fps. Once the camera frame rate exceeds this limit, throughput

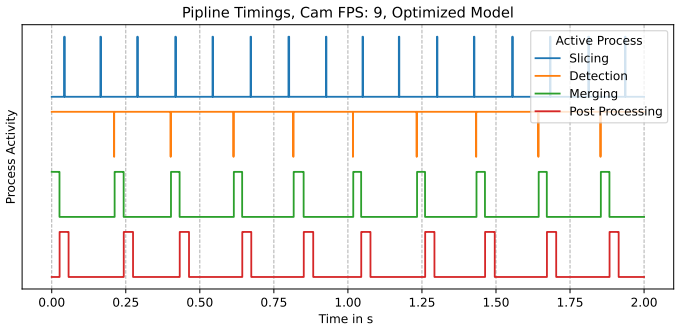

Fig. 3: Module timings for the detection pipeline given an optimized machine learning model with the camera system set to operate at 9 frames per second.

declines and the delay to the source frame increases markedly. Figure 3 visualizes this phenomenon. When a new camera frame arrives, the detector is already fully occupied, causing occasional frame drops and a delay to the last processed frame. This delay corresponds to the phase shift between camera and detector and reaches its maximum when the detector's processing time is slightly longer than the inter-frame interval.

Optimizing both latency and throughput therefore constitutes a Pareto-type optimization problem: the theoretical optimum for both metrics is just before the attainable maximum throughput, with sufficient buffering in the detector to prevent cascading delays. Over-booking the GPU further introduces secondary effects, such as increased copy and synchronisation overheads that become more pronounced at higher frame rates.

VI. Conclusion

We have presented a real-time visual pipeline that couples SAHI-based patch extraction with batched YOLO inference to enable multi-scale object detection across a multi-camera maritime system. By adopting a modular, push-based architecture and aggressively optimizing GPU utilization we reduced the detection latency by more than an order of magnitude while maintaining high recall for small objects. Empirical evaluation on a three 19.6-MPix camera, demonstrated that our implemented pipeline achieves stable results under load, but fine adjustments have to be made when maximizing detector throughput. The results confirm that careful system-level design, in addition to algorithmic advances, is essential for deploying deep-learning-based perception in safety-critical maritime collision-avoidance applications. Future work might explore improved slice selection selection to reduce the input data as well as adaptive flow control for the camera system as well as parameter tuning for the operational usage.

References

[1] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[2] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, 2015.

[3] J. Terven, D.-M. Córdova-Esparza, and J.-A. Romero-González, "A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas," *Machine learning and knowledge extraction*, vol. 5, no. 4, pp. 1680–1716, 2023.

[4] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*. Springer, 2016, pp. 21–37.

[5] F. C. Akyon, S. O. Altinuc, and A. Temizel, "Slicing aided hyper inference and fine-tuning for small object detection," in *2022 IEEE international conference on image processing (ICIP)*. IEEE, 2022, pp. 966–970.

[6] C. Rethfeldt, A. Klein, M. Riesner, M. Kurowski, J. Stoppe, S. Schubert, and T. Jeinsch, "Bridging XLUUV and MASS – technologies for autonomous multi-domain operations," *Journal of Physics: Conference Series*, vol. 3123, no. 1, p. 012019, oct 2025. [Online]. Available: https://doi.org/10.1088/1742-6596/3123/1/012019

[7] G. Jocher, J. Qiu, and A. Chaurasia, "Ultralytics YOLO," https://github.com/ultralytics/ultralytics, Jan. 2023, Version: 8.0.0, License: AGPL-3.0, Accessed: 2025-06-10.