High Speed Object Detection and Inference Optimization: Project Report

Group 2 : Jesus-Sebastian Lamas, Antoine Besson, Mohamed Yassine Aouidet,
Pacome Kengali Fegue, Vansh Nath, Charles Kayssieh
Télécom Paris DSAI

June 2026



1 Introduction, Hypotheses and Motivations

In most of the object detection model benchmarkings on the COCO dataset, the main metrics analyzed are accuracy (which is mAP) and speed (measured in FLOPs or FPS). Howerver, we noticed from the beginning of our project that none of these two numbers really explain why a specific detector is *actually* slower than another one. For example, FLOPs are not a very good way to predict latency : this is because FLOPs do not take into account things like memory bandwidth, the fixed cost to launch every GPU kernel or operations that depend on data (like non-maximum supression - NMS). Therefore, we did not want to produce another ranking or bencharmk of object detectors. Instead, we aimed to measure where the inference time goes, and diving into deep details through micro architecture so we could find concrete optimizations.

Our working hypothesis is that when using a batch size of 1, the latency depends more on which operators are running, if they are fused together, how many kernels are launched, and the amount of data moving around, rather than just the raw arithmetic operations. We chose batch size 1 on purpose because, beyond being the industry standart when it comes to benchmarking, this is what matters most for deployments where latency is critical. Also, it is the most difficult case because we cannot hide the overhead of each kernel with batching, meaning that the cost per image is directly visible.

To test our hypothesis, we benchmarked 24 different detectors taken from five main families. We included two-stage CNNs (like Faster R-CNN and Mask R-CNN), the YOLO models (like YOLOX, DAMO-YOLO, YOLOv7, and also the smaller versions from YOLOv5 to YOLOv11), DETR and its variants RT-DETR, and finally some dense one-stage models like RetinaNet, FCOS and EfficientDet. We ran every model using the same protol on a NVIDIA Tesla T4 GPU from Google Colab. For every model, we measured the end-to-end latency. Then, we attributed this latency to the individual operators: convolution, normalization, activation, attention, pooling, and post-processing.

2 Implementation Process

2.1 Experimental protocol

For the input, every model received 640x640 RGB image one by one from the COCO val2017 dataset. We always used a batch size of 1. We chose to ran at first the baseline in the standart FP32 eager mode. We did not include any optimization like torch.compile, TensorRT, or quantization. We decided to keep the FP32 precision for the entire baseline to be sure of the rigor of our comparisons. This also makes it easier to see the impact and the improvement of mixed precision after.
To measure the inference time correctly, our script first runs 50 warm-up iterations, that are discared. It allow to settle the cuDNN autotuner and the caching allocator. After that, we run 1000 times forward passes. For the measure of the time itself, we placed torch.cuda.Event pairs around each pass, and we used torch.cuda.synchronize to ensure that the GOU had his job fnished. We then report the median latency and calculate the FPS from it (which is just 1000 divided by the latency in miliseconds).

For the accuracy, we used the standart COCO mAP at 50:95. We computed it with a pycocotools evaluation on the val2017 set. We also fixed all the random seed to 42, so that every model is tested on the exact same images and receives the same proposal counts.

Finally, we used the Google Colab Tesla T4 GPU as hardware. It is based on the Turing architecture and has 16 GB of GDDR6 memory. It can do about 8 TFLOPS in FP32, compared to 65 TFLOPS in FP16. It also features 320 Tensor Cores, but because our baseline is in FP32, they are left completely idle.

2.2 Reducing models to their operators

Instead of working on broad sections of the model like backbone, encoder, or head, we decided to take a closer look at performance bottlenecks. Therefore, we broke the model down into its basic building blocks, which are the elementary operators, so that we could understand what each part contributes to the speedup, its cost, and whether it is easy or difficult to accelerate.

Four specific operators dominate the processing time:

- **Convolution:** It is the heaviest operator, which actually makes it the easiest to accelerate by switching its processing to FP16 or INT8 formats on Tensor Cores. We obtained a speed increase of 2x to 4x. Additionally, grouping it together with normalization and activation through Conv-BN-activation fusion eliminates unnecessary data movement around the operation.
- **Normalizations:** At inference time, BatchNorm and FrozenBatchNorm are affine transformations, so we folded them directly into the previous convolution for free. However, we need to make sure they are not mistakenly launched as separate kernels. LayerNorm is trickier; since its statistics depend on the input data, we cannot fold it.
- **Activation:** ReLU, SiLU, and GELU are limited by memory bandwidth, not by computing power. Fortunately, they become virtually free to run once they are fused directly into the convolution step before them.
- **Attention:** Attention processes scale quadratically based on the number of tokens, which is why they are computationally expensive. Deformable attention is designed to be cheaper in theory, but it requires a specialized, custom CUDA kernel to achieve those faster speeds in practice.

2.3 Profiling and optimisation tools

To figure out exactly where the models were lagging, we deployed two distinct profiling tools:

- `torch.profiler`: This tool was used to track real-world wall time and attribute it to specific modules and operations.
- NVIDIA Nsight Systems: This profiler read the physical kernel names and the CUDA-API timeline directly, it separated the actual GPU computation time from the hidden overheads of launching kernels and transferring data.

After knowing this we approached optimization in two distinct rounds:

- **Round 1:** we started with the easiest, most universally compatible tweaks across all model families. This involved folding Conv-BN operations, implementing FP16 autocast (AMP), using `torch.compile`, and manually applying CUDA graphs.
- **Round 2:** Then we introduced more complex, deployment levers. This meant exporting to TensorRT and ONNX Runtime, utilizing INT8 precision, and bringing in a compiled kernel for deformable attention.

Throughout the entire process, the idea was that we didn't just to report that a model ran faster but to explicitly explain why it ran faster at the fundamental operator level.

3 Findings

3.1 Results of the submodules and operations latency benchmarks

After running our benchmarking protocols, we grouped our results and came to a unanimous realization: the backbone was the sub module that dominated all around all models. Which made sense because it is where the most of the base level operations take place.
As we said earlier, this first step wasn't enough. The goal of our project was to pin point exactlty which operations took more time than others to then optimize said targets.
We then produced another protocol and got a clear understanding of which operations took the most time to compute and we found that convolution governs the backbone of most models.

Table 1: Per-image latency (ms, batch 1), FPS, and mAP$^{50:95}$ on COCO val2017, all 24 models.

Model	Latency	FPS	mAP
Two-stage CNN			
Faster R-CNN R50	49.0	20.4	0.131
Faster R-CNN R50 V2	98.6	10.1	0.174
Mask R-CNN R50	70.7	14.2	0.134
DETR family			
DETR R50	77.8	12.9	0.424
Conditional DETR	82.7	12.1	0.385
DAB-DETR	85.1	11.7	0.398
Deformable DETR	263.3	3.8	0.410
RT-DETR			
RT-DETR R18	24.5	40.8	0.470
RT-DETR R50	57.4	17.4	0.534
RT-DETR R101	90.1	11.1	0.548
YOLO			
YOLOX-m	21.8	45.9	0.475
DAMO-YOLO-M	25.3	39.5	0.510
YOLOv7	33.9	29.5	0.519
YOLOX-l	41.8	23.9	0.505
YOLOv7-X	60.7	16.5	0.537
YOLOX-x	75.6	13.2	0.523
YOLO-S			
yolov5s	13.1	76.3	0.425
yolov8s	14.4	69.5	0.444
yolov10s	15.8	63.1	0.456
yolov11s	16.6	60.3	0.464
yolov9s	26.0	38.4	0.460
Dense one-stage			
RetinaNet R50	49.8	20.1	0.378
FCOS R50	50.8	19.7	0.336
EfficientDet D4	109.2	9.2	0.448

Firstly, table 1 shows that all types of Conv (3x3, 1x1, 2x2) took around 56% to 90% of the compute time of the backbone. Second, we confirmed that some models did lose a lot of compute time to normalization. The DETR family in particular loses an impressive amout of time because of the normalization : 28% to 32% of its backbone is lost to a FrozenBatchNorm block that is never folded with the convolutions (because of a custom module issue that made it unrecognizable to the standard fusion function used to fold operations). Lastly, the Deformable DETR comes out as the slowest model in the set at 263 ms. The reason being its deformable-attention does not have a compiled kernel

built in and in pratice operations fall back to a Python loop. It could have been faster had we created a specific kernel for it which we only understood later.

The per-family comparison is consistent with these results. Ours results show that the two-stage models are roughly half backbone and half head, with the head cost driven by RoIAlign and NMS instead of it being the convolutions (i.e. feature extraction).
As for The DETR models, since they are backbone-bound the convolution was predominant.
A notable result is that RT-DETR holds the best accuracy-per-millisecond trade-off in the set about 0.534 mAP and 57 ms. The reason behind such a low latency is because its backbone is a fully folded convolution and its hybrid encoder keeps the token count low.
The small YOLO models range between 13 and 26 ms at 0.42 to 0.46 mAP. These models conatin fused Conv-BN-SiLU backbones. Another notable result, the NMS-free YOLOv10s is both fast and has the best baseline mAP. As for EfficientDet, despite its efficiency-oriented design, is thit e second slowest model at 109 ms, and that is due to its remarkably heavy backbone and BiFPN neck run that is not fused in FP32.

3.2 What the optimisations gave

3.2.1 First Optimisation Experiments

Our first optimisation experiments already started to produce some interesting results. We present here the results from it and by showing the global effect of the optimisations of the best optimised configuration obtained for each model during our first experiments. Table 2 summarises the baseline latency, the best optimised latency, the corresponding speed-up, the configuration that produced this result, and the COCO mAP after optimisation.

Table 2: Overall optimisation results from first experiments

Model	Base	Opt	×	Config	mAP
Two-stage CNN					
Faster R-CNN R50	48.5	24.7	1.97	comp+AMP	0.131
Faster R-CNN R50 V2	89.2	44.3	2.01	comp+AMP	0.174
Mask R-CNN R50	68.9	55.4	1.24	comp+AMP	0.134
DETR family (AMP mAP pending)					
DETR R50	88.1	37.7	2.34	AMP+fold	0.424
Conditional DETR	94.0	40.7	2.31	AMP+fold	0.385
DAB-DETR	96.0	45.1	2.13	AMP+fold	0.398
Deformable DETR	291.3	202.7	1.44	AMP+fold	0.410
RT-DETR					
RT-DETR R18	24.5	19.8	1.24	compile	0.470
RT-DETR R50	57.4	44.9	1.28	compile	0.534
RT-DETR R101	90.1	76.9	1.17	AMP	0.548
YOLO					
YOLOX-m	21.8	18.8	1.16	AMP	0.475
YOLOX-l	41.8	22.0	1.90	AMP	0.505
YOLOX-x	75.6	33.4	2.26	AMP	0.523
YOLOv7	33.9	17.9	1.90	AMP	0.519
YOLOv7-X	60.7	26.6	2.29	AMP	0.537
DAMO-YOLO-M	25.3	15.0	1.69	comp+AMP	0.510
YOLO-S					
yolov5s	13.0	9.2	1.42	fold+FP16	0.434
yolov8s	14.0	9.8	1.43	fold+FP16	0.452
yolov9s	25.1	22.1	1.13	fold+FP16	0.469
yolov10s	15.5	11.3	1.37	fold+FP16	0.469
yolov11s	17.3	11.4	1.52	fold+FP16	0.469
Dense one-stage					
RetinaNet R50	49.8	21.7	2.29	comp+AMP	0.377
FCOS R50	50.8	23.9	2.13	comp+AMP	0.334
EfficientDet D4	109.2	30.5	3.57	TS+AMP	0.448

Overall, the optimisations reduce latency for every model, but the magnitude of the improvement varies strongly across families. We noticed that the largest gains were obtained when most of the runtime is spent in operators that

map well to the GPU acceleration mechanisms we used, especially convolutions and GEMMs. This is why models such as DETR R50, Conditional DETR, YOLOX-x, YOLOv7-X, RetinaNet R50, FCOS R50, and EfficientDet D4 show speed-ups above 2×.

We found that the dense one-stage models showed particularly strong gains. EfficientDet D4 improves from 109.2 ms to 30.5 ms, corresponding to a 3.57× speed-up. This is the largest improvement in the table and suggests that it contains many operations that can be simplified or fused by TorchScript and AMP. RetinaNet R50 and FCOS R50 also improve by more than 2×, which confirms that convolution-heavy detectors are good candidates for precision and graph-level acceleration.

In terms of accuracy, the optimisations preserved the accuracy as the reported mAP values remain effectively unchanged after optimisation, which was a good sign.

3.2.2 TensorRT FP16 Optimisation Results

After the first optimisation round, we focused on TensorRT as advised by our supervisor because it applies several of the useful optimisations at the same time. Globally, TensorRT gave us the best optimisations for the models.

Table 3: Main TensorRT FP16 results.

Family / model	TensorRT setting	Measured gain
RT-DETR R18	ORT+TensorRT FP16	2.36×
RT-DETR R50	ORT+TensorRT FP16	5.06×
RT-DETR R101	ORT+TensorRT FP16	5.57×
YOLOv5s	TensorRT FP16	1.78×
YOLOv8s	TensorRT FP16	1.86×
YOLOv9s	TensorRT FP16	2.88×
YOLOv11s	TensorRT FP16	2.29×
YOLOX-m	TensorRT FP16	1.91× conv. gain
YOLOX-x	TensorRT FP16	3.15× conv. gain
YOLOv7	TensorRT FP16	5.00× conv. gain p
YOLOv7-X	TensorRT FP16	4.88× SiLU gain n
EfficientDet D4	Full TensorRT FP16	2.32× wall-clock
FCOS R50 static zone	TensorRT attempt	1.93× static-zone gain
RetinaNet R50 static zone	TensorRT attempt	1.66× static-zone gain

We obtained the strongest TensorRT result is obtained on RT-DETR. For RT-DETR R50 and R101, TensorRT FP16 reduces latency by 5.06× and 5.57× respectively. This is much larger than the gain from AMP alone, because TensorRT does not only change precision. It also fuses convolution, GEMM, residual, reformat, and activation work into a smaller number of engine kernels.

For the YOLOX, DAMO-YOLO, and YOLOv7 group, the TensorRT result is most visible at the operator level. TensorRT accelerates total convolution time significantly, especially for the larger models. YOLOv7 reaches a 5.00× convolution gain, while YOLOv7-X reaches a 4.88× gain on SiLU because the activation is fused with the convolution.

Overall, we discovered that the TensorRT results show that the best acceleration comes when the model is made of regular and fuseable operators. For example, Convolution, BatchNorm, and activation are good TensorRT targets.

3.3 Remaining Bottlenecks after Optimisation

After optimisation, we found that the main cost no longer comes from regular convolution, BatchNorm, activation, or GEMM layers, since these are accelerated well by FP16, fusion, and TensorRT. Instead, the remaining latency is concentrated in operators that are dynamic or irregular.

The main examples are NMS, top-k selection, gather operations, RoIAlign, and deformable attention. These operations depend on the number of detections, proposals, or sampling points, so they are harder to fuse in the TensorRT engine. This is why some models still have latency floor even after their convolutional backbone are accelerated.

Overall, we found that the optimisation shifts the bottleneck. Before optimisation, latency is dominated by dense compute and many small kernels. After optimisation, it is dominated by post-processing, dynamic shapes, and unsupported custom operators.

4 Alignment with the Proposed Plan

4.1 Did we do everything we set out to do?

To be honest no, not entirely. We finished roughly 70% of our original plan. We did manage to complete the profiling across all models and mesured many optimizations, both cheap and heavy ones. What we unfortunately did not achieve is our most ambitious objective. After analysing the optimisations operations per operations and identifying the bottlenecks in the existing models, we had has a goal to design our own detector that uses those findings to be fast by construction. It would be assembled from the fastest and most fusable blocks. The theory behind it was clear, testing on state of the art model which operation is best optimized on a T4, then taking those fast and very well acccerelable parts to make the best model possible on a T4. We ran out of time before we could build and measure our optimized model. In short, we were very ambitious for the time (especialy given the exams at the end of the project) we had and had to drop the last part.

The learning curve at the start was harsh. This was our first real project in this area. Most members had had little experience with profiling tools or with the techniques for optimising models. Furthermore, we took a bit of time to truly understand how the models and operations operated at the GPU level, something we were not familiar with. Closing that gap took us time, we had to go through trial and errors. Our supervisor was slighty disapointed in us at the start, as our results showed we did not understand fully the scope of the subject. Fortunately, he was very understanding and gave us detailled feedback as well as advices on how to study the models. Lastly, coordinating a team of six around everyone's schedules was not always easy, so seeing each others outside of the weekly reunions on friday had to be planned in advance.

4.2 What we would do differently

We have two main takeaways from this project. The first would be to agree on a shared protocol from the very start. We lost quite some time (about one to two weeks) rewriting and reorganising code because we each began with slightly different measurement setups and only later converged on a common harness. The most basic exemple we can give is the image size. We did not all agree on a common entry image size and while most had taken the default approach of 640x640, that was not the case for all at the begining. This was unfortunate as the results were not comparable (bigger images = more compute = slower models). Fixing the protocol on day one (same warm-up count, same timing method, same seed, same COCO subset, same everything) would have saved that time and maybe let us create our own model.

The second lesson is the difficulty linked to getting used to Colab servers. Limited allocated GPU time cost us a lot of hours when running benchmarks that each took close to an hour. Moreover, in order to stay connected to Colab servers we could not close our computers nor go anywhere which sometimes proved to be a challenge. We discovered Onyxia far too late (the last week) which for free would have given us higher limit and the ability to close our computers once the script started.

Contribution Section

How we divided the work

We split the study by detector family. Each member took one family, profiled its models under the shared protocol, ran the same set of optimisations on them, and then we regrouped to compare, summarise and interpret the results across families. This way of working was kept through the entierety of the project, letting us work in parallel and compare results at the end. Indeed, because everyone followed the same harness and the same metrics, the per-family results were comparable, which is what made the cross-family ranking in the final part of the study possible.

Who did what

- **Antoine Besson** worked on the DETR family (DETR R50, Conditional DETR, DAB-DETR and Deformable DETR) and wrote some of the shared tooling the rest of the team used, namely : the latency harness, the COCO loader and the evaluation path. In general, he was the one that suggested the common protocol (FP32, 640x640, etc). On his own family found intersting results around its operators, for exemple the unfused FrozenBatchNorm block in the DETR backbone. He checked both with `torch.profiler` traces and an Nsight Systems pass to have

accurate results. He also handled several of the trickier problems along the way: - versions problems - git merges and git structure

For the code, everyone did the same thing, so Antoine was no exception. He first worked on establishing a baseline, then on the first general optimizations, then on analyzing the models down to the operation level, to first establish a baseline for the longest and most costly operations, then on the optimizations of those. One thing worth mentionning is that he managed to implement tensor RT on all his models, as well as implement some other optimizations not quoted in this report since they could not be compared to other models that did not have it correctly implemented by the end of the project due to lack of time linked to the exams. He believes that one of his most noteworthy contribution sits outside the code. He organized the team meetings, contacted members through the week to be sure the workload was split roughly even across members and to make sure everyone stayed up to date on the project. This made him realize the important of organization in such a team project and that whilst individual code contribution is great, organizing tasks and helping outside the code is just as important.

- **Jesus Sebastian Lamas** Worked on the three YOLOX (m, l, x), DAMO-YOLO, the YOLOv7 and the YOLOv7-X and had many changes to make to his architecture to ensure that all his results were in the standardized format, so that it allowed for fair comparison in the article.

  His first step (as well as for every other student) was to establish the baseline benchmark for his models using torch.profiler and nsight to get the low level operations breakdown. Antoine and him had agreed on a common protocol, so he fully adapted and revised the skeleton of the code to match and be compatible with the models that he was in charge of.

  Moving into the optimization phase, he implemented Conv+BN fusion and AMP FP16. However issues started during this part as torch.compile and CUDA Graphs failed repeatedly due to dynamic dimensions (YOLOX) and flat nn.Sequential structures (YOLOv7). To solve these main conflicts, he built a unified, compilation structure in the code with automatic fallbacks.

  The biggest architectural changes happened during TensorRT integration. Critical environment conflicts broke torchvision. The solution being letting go of torch-tensorrt for an ONNX-to-standalone-TensorRT approach. Furthermore, the complex output structures of models like DAMO and YOLOv7 did not survive ONNX export, which required adapting the evaluation protocol since post-optimization mAP could no longer be computed for those specific engines.

- **Mohamed Yassine Aouidet** His role was inference-optimization and hardware-profiling pipeline for the YOLO-s family, YOLOv5s, YOLOv6s, YOLOv8s, YOLOv9s, YOLOv10s, and YOLOv11s.

  His profiling showed that most of the execution time, between 61% and 82%, are from CPU and Python launch overhead rather than actual GPU computation. This overhead and synchronization during NMS post-processing are the reason that increased latency and pushed the baseline p99 latency up to 601 ms.

  Further analysis made on YOLOv9s showed that it was slowed down by its RepNCSPELAN4 neck, while YOLOv10s had lower jitter because it's NMS free.

  To reduce PyTorch launch overhead, he developed a custom `brick_profiler` to optimize each operation separately using Conv-BN fusion and selective FP16 Tensor-Core casting. The analysis also showed that FP16 did not help all models the same way. For example, it reduced performance on YOLOv5s C3 blocks at batch size 1, but gave a 2.13× speedup on the YOLOv6s backbone.

  This optimized PyTorch version achieved 1.13× to 1.52× median speedups and reduced the p99 latency tail by up to 33× without reducing mAP.

  As a final deployment, Yassine built a TensorRT FP16 pipeline. He exported the models to ONNX opset 17 and converted them into FP16 TensorRT engines. This removed Python overhead and achieved 1.78× to 2.88× median speedups compared to FP32 baselines.

  TensorRT version also reduced latency variation, lowering the p99 latency tail by 44× to 80× and keeping the standard deviation below 0.11 ms.

  Finally, he wrapped the TensorRT engines with CUDA graphs and defined the next steps for INT8 Post-Training Quantization and EfficientNMS plugin integration.

- **Charles Kayssieh** worked on the two-stage R-CNN family with Faster R-CNN R50, Faster R-CNN R50 V2, and Mask R-CNN R50. He initially wanted to use a ResNet-101 backbone, but torchvision didn't have pre-trained

weights for it, so he chose the ResNet-50 V2 instead.
To profile the models, he built a pipeline with CUDA events and NVTX tags. He separated the latency measurements from the mAP evaluation, in order to avoid PCIe bottlenecks. For the optimizations, he added AMP FP16 and wrote a recursive function to fold the Batch Normalization layers. He also evaluated torch.compile, which gave a 1.97x speedup on the static V1 model but caused slowdowns and crashes on the V2 and Mask R-CNN due to the dynamic shapes of the RPN.
The native implemention of TensorRT with PyTorch (Torch-TensorRT) failed on Colab. This is explained by the fact that Faster R-CNN needs custom operators like roi_align and nms, and it also uses dynamic control flow. Torch-TensorRT doesn't support these features, hence the compilation errors. Then, he deciced to export the model to ONNX so he could use ONNX Runtime with TensorRT. But he face a lot of environment issues. Even after he installed TensorRT via pip, the system's dynamic linker just refused to load it. In the end, he built a fallback using the standard CUDA Execution Provider in ONNX Runtime. However, there was a known bug in ORT version 1.26. Because of this bug, he had to disable io_binding and just rely on standard CPU memory copies. In the end, the final speed actually became slower than the basic eager PyTorch baseline.

- **Pacome Kengali Fegue** worked on the dense one-stage group: RetinaNet R50 (with an attempt at R101), FCOS R50, and the EfficientDet family (D4, D5 and D6). Getting these models running was the first hurdle, since detectors like these are not centralised the way recent language models are on a single hub: the weights he found came through Detectron2, which has not been maintained for some years and was fragile across versions, so finding reliable sources took real effort. He began with the profiling phase, learning `torch.profiler` from tutorials and using it to time each model on the COCO validation set, then reading the traces in TensorBoard down to the individual operations. That is where he saw that model size does not decide speed and that the Tensor Cores sat idle on the FP32 baseline. The traces also pointed him at the levers: the FrozenBatchNorm that, at batch size 1, is only a constant affine map and folds into the preceding convolution; the SiLU activation that costs more than ReLU because of its exponential and the extra memory traffic, which can be fused into the convolution as one kernel; and the precision change needed to wake the Tensor Cores. Colab made this work painful, with crashes, memory failures and long runs that had to be restarted, so he moved small-scale runs onto his laptop GPU, which was faster and, being a newer generation, actually used the Tensor Cores; this made the hardware-generation effect concrete. For the larger sweeps he switched to Modal, writing code locally and running it remotely on a T4-equivalent machine, which let him test about forty-five configurations in a few hours instead of days. A recurring benchmarking problem was that the COCO class indexing differs between model sources (0 to 79, 1 to 80, or 1 to 90 with gaps), which broke the mAP computation when moving from one family to another; rather than patch around it he rewrote the evaluation, benchmarking and profiling code by hand on top of `pycocotools`, choosing model-specific scripts over one abstract harness so each model's quirks could be handled cleanly. His main contribution, and the part he says he learned most from, is a zone-based optimisation approach for models that do not compile end to end. EfficientDet is the clean case, compiling into a single full-model TensorRT engine that gives the exact before/after operator comparison in the report; FCOS and RetinaNet do not, because their NMS emits a variable number of boxes and the dynamic shape breaks the engine build, so he compiled only the static zone (backbone, FPN and dense head) and left NMS in eager, applying the best lever per zone (`torch.compile`, AMP, or a plain FP16 cast). With more time he would have tried a GPU-native or Triton NMS to bring the remaining host-bound part into the engine.

- **Vansh Nath** focused on the RT-DETR family of object detectors. He implemented and ran the benchmarking and profiling pipeline for the RT-DETR R18, R50 and R101 variants, using the common experimental protocol. For each model, he measured latency, FPS and mAP, he then analysed the end-to-end performance, submodule breakdown, and operator-level bottlenecks. In order to analyse the models in more detail he decomposed RT-DETR into its backbone, encoder, decoder and post-processing stages, and then further analysed the dominant operations inside these components, especially convolutions, ReLU activations, GEMM/addmm operations and residual additions.
He then carried out the first optimisation experiments for this family. He tested torch.compile in reduce-overhead mode and AMP FP16 mixed precision. He went on to investigate CUDA Graph capture (but failed to do so due to some technical difficulties.
After the first experiments, he then went on to work on an ONNX Runtime and TensorRT acceleration path for the RT-DETR models. This optimisation gave him the strongest accelerations especailly for the heavier RT-DETR

variants.
Finally, he analysed the RT-DETR optimisation results by comparing the PyTorch FP32 baseline with the optimised backends. Rather than just reporting the final speed-ups, he investigated the results at operator level to understand where the gains came from and which bottlenecks remained. He found that TensorRT and FP16 mainly accelerated dense compute operations like convolutions and GEMM/linear layers through Tensor Core usage and kernel fusion and also reduced kernel launch overhead. At the same time, he identified remaining bottlenecks such as node_select, top-k and gather-style operations, which are less suitable for TensorRT acceleration.

How we organised ourselves

We coordinated day to day through a WhatsApp group and met twice every week, either over Google Meet or in person. Some meetings included our supervisor and some did not. The supervisor meetings were weekly and used to present the previous week's results and to plan the coming week, while the internal meetings were where we wrote up the weekly report together and compared results. We shared code through GitLab and wrote the report on Overleaf, using the `overleaf.enst` instance.

Use of AI Tools

As suggested by our supervisor, AI assistants were used during the project, and we want to be transparent about how.
In the code. AI assistants helped with coding support: scaffolding benchmark and profiling scripts, debugging Python, documenting commands, and improving the README. They also helped us adapt the benchmarking and profiling scripts to each model family. Beyond the code itself, we used them to help plan and organise the work, and for technical clarification of profiling terminology, for example understanding what specific outputs from `torch.profiler` and NVIDIA Nsight Systems actually mean. The interpretation of the measurements and the conclusions drawn from them were done by us, the authors. Similarly, it is important to point out that whilst we did use AI to upgrade some of the code, the base of it was written by us, we reviewed the code outputed by AI agents and the optimisation ideas were our own.
In the report. AI assistance for the report was limited to clarity, grammar and typo correction. The technical content, the results and the analysis are our own.

References

References

[1] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. arXiv:1607.06450, 2016.

[2] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko. End-to-end object detection with transformers. In ECCV, 2020.

[3] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In NeurIPS, 2022.

[4] X. Ding, X. Zhang, N. Ma, J. Han, G. Ding, and J. Sun. RepVGG: Making VGG-style convnets great again. In CVPR, 2021.

[5] Z. Ge, S. Liu, F. Wang, Z. Li, and J. Sun. YOLOX: Exceeding YOLO series in 2021. arXiv:2107.08430, 2021.

[6] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask R-CNN. In ICCV, 2017.

[7] D. Hendrycks and K. Gimpel. Gaussian error linear units (GELUs). arXiv:1606.08415, 2016.

[8] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861, 2017.

[9] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In ICML, 2015.

[10] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. In CVPR, 2016.

[11] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. In ICCV, 2017.

[12] S. Liu, F. Li, H. Zhang, X. Yang, X. Qi, H. Su, J. Zhu, and L. Zhang. DAB-DETR: Dynamic anchor boxes are better queries for DETR. In ICLR, 2022.

[13] D. Meng, X. Chen, Z. Fan, G. Zeng, H. Xu, L. Yuan, L. Sun, and J. Wang. Conditional DETR for fast training convergence. In ICCV, 2021.

[14] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. arXiv:1710.05941, 2017.

[15] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In CVPR, 2016.

[16] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In NeurIPS, 2015.

[17] M. Tan, R. Pang, and Q. V. Le. EfficientDet: Scalable and efficient object detection. In CVPR, 2020.

[18] Z. Tian, C. Shen, H. Chen, and T. He. FCOS: Fully convolutional one-stage object detection. In ICCV, 2019.

[19] Ultralytics. YOLO11 architecture (C3k2, SPPF, C2PSA). `https://docs.ultralytics.com`, 2024.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In NeurIPS, 2017.

[21] A. Wang, H. Chen, L. Liu, K. Chen, Z. Lin, J. Han, and G. Ding. YOLOv10: Real-time end-to-end object detection. In NeurIPS, 2024.

[22] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. In CVPR, 2023.

[23] C.-Y. Wang, H.-Y. M. Liao, Y.-H. Wu, P.-Y. Chen, J.-W. Hsieh, and I-H. Yeh. CSPNet: A new backbone that can enhance learning capability of CNN. In CVPRW, 2019.

[24] C.-Y. Wang, I-H. Yeh, and H.-Y. M. Liao. YOLOv9: Learning what you want to learn using programmable gradient information. In ECCV, 2024.

[25] X. Xu, Y. Jiang, W. Chen, Y. Huang, Y. Zhang, and X. Sun. DAMO-YOLO: A report on real-time object detection design. arXiv:2211.15444, 2022.

[26] Y. Zhao, W. Lv, S. Xu, J. Wei, G. Wang, Q. Dang, Y. Liu, and J. Chen. DETRs beat YOLOs on real-time object detection. In CVPR, 2024.

[27] X. Zhu, W. Su, L. Lu, B. Li, X. Wang, and J. Dai. Deformable DETR: Deformable transformers for end-to-end object detection. In ICLR, 2021.