OTX: Testing the ONNX Runtime Optimizer

Siddharth Narsipur
University of Rochester
Rochester, NY
snarsipu@u.rochester.edu

Abstract

The ONNX Runtime Optimizer is a widely-used tool that performs graph-level optimizations on ONNX models. Previous work [15] suggests that similar ONNX graph optimizers could crash during inference or generate invalid models, but this has not been tested on the Runtime Optimizer with NVIDIA GPUs. OTX is a Python-based tool for benchmarking any ONNX model with any dataset by iteratively testing the base model against optimized variants. Our testing finds that enabling aggressive optimization can lead to 1–4x slowdowns in inference performance for certain image classification and object detection models.

1 Introduction

Deep Learning models are widely used for tasks such as text generation, image classification, object detection, and more [10]. As the popularity of these models grow, they are being deployed across a diverse range of hardware, from small mobile phones to large data centers.

To effectively harness the computational power of these devices, a variety of AI compilers and frameworks are being developed, such as the XLA project by Google [36] and the MLIR framework [9]. These tools translate model descriptions, usually expressed in Python, into an intermediate format and then apply a series of optimization passes to improve utilization of the target hardware.

ONNX is an open-source format that represents machine learning models as computation graphs [22]. In an ONNX graph, nodes represent operations on data (e.g., convolution, ReLU, addition) and edges define how this data flows between nodes. A graph is only translated into device-specific functions (kernels) during inference, giving it the advantage of being inter-operable across device types and manufacturers.

ONNX Runtime [1] is a popular cross-platform inference engine for ONNX models that includes an optimization tool to improve performance through graph-level transformations of the model graph. Ideally, such tools should improve inference speeds and not alter the semantics of the underlying model. However, our analysis using OTX revealed several models for which enabling the default optimization level resulted in a substantial decrease in inference speed, as well as two graph optimizations that caused the model to crash during inference.

Contributions. This work makes the following contributions:

(1) Introduces OTX, a Python-based tool to test and profile ONNX models against optimized variants using the ONNX Runtime Optimizer.
(2) Analyzes why performance degradation may occur during inference with the data collected by OTX.

2 Background

2.1 Graph Optimizations

Individual operations in deep learning models can be optimized to improve the computational efficiency of the underlying model [10]. This includes techniques such as Operator Fusion (as shown in Figure 1), where multiple tensor operations are combined into a single fused operation. Other methods include constant folding, dead code elimination, and converting operations to hardware-specific variants. Each type of transformation is applied as an optimization pass over the model graph. Tools such as the ONNX Runtime execute sequences of such passes during the optimization phase that modifies the graph and produces a new optimized variant.

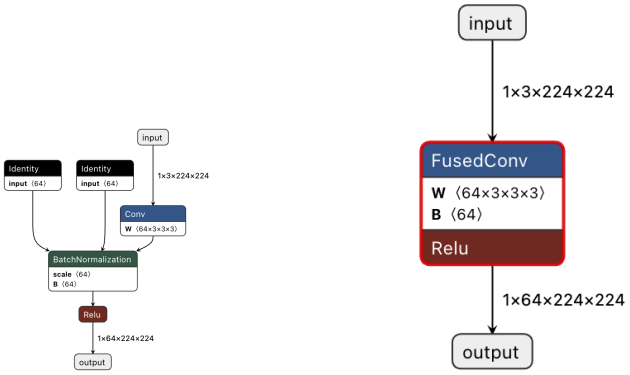

Figure 1: An example of an ONNX graph that undergoes node fusion. In this example, the convolution, batch normalization, and ReLU nodes are fused together during the optimization pass into a single fused node.

2.2 ONNX Runtime & Optimizer

ONNX Runtime (ORT) is an open-source, high performance engine for creating and running machine learning models in the ONNX format [1]. It is cross-platform and works with CPUs, GPUs, and other specialized accelerators. Each hardware provider (NVIDIA, AMD, Qualcomm, etc.) has a corresponding Execution Provider (EP) that implements the logic required for executing the model graph using device-specific libraries.

The Runtime works by loading a model, applying an optimization pass (described below), and then partitioning the execution graph. The partitioning step splits the graph into subgraphs such that each subgraph runs on the most efficient execution provider (if there are multiple available). After partitioning, the nodes are sent to the appropriate EP for execution.

The Runtime Optimizer (referred to as the "Optimizer" henceforth) is the component of the inference engine that executes the optimization step [18]. The Optimizer can be run during inference (as "online" mode) or separately on its own ("offline" mode). There are four levels of optimization, and choosing a level enables all the optimizations of that level and all previous levels.

`DISABLE_ALL` disables all optimizations, `ENABLE_BASIC` applies basic graph optimizations like constant folding, redundant node elimination, and semantic-preserving node fusions, `ENABLE_EXTENDED` adds more complex node fusions that are device-specific, and `ENABLE_ALL` adds data layout optimizations. `ENABLE_ALL` is applied by default and is therefore examined in greater detail in this paper. Since the optimizer can be executed during inference, it has the advantage of knowing what hardware environment it is being executed in, and can perform hardware-aware optimizations and layout transformations that take advantage of the execution graph partitioning.

2.3 Deep Learning Models

Deep neural networks include many different architectures, each tailored to perform specific tasks such as image recognition, sequence modeling, and structured prediction. In this paper, we focus on three common categories of networks:

2.3.1 Image Classification. Image classification is the task of assigning a label or category to an image. These models use Convolutional Neural Networks (CNNs) that extract and learn features from images [7]. They are evaluated by a series of industry-standard metrics like accuracy, precision, and F1-score.

2.3.2 Object Detection. Object detection models are designed to identify instances of objects within an image and to draw bounding boxes around each detected object. They output a class label (what the object is) and a set of coordinates for the bounding box (where the object is) [27]. They are measured by metrics such as IoU (overlap between predicted and ground truth boxes), precision, and recall.

2.3.3 Text Comprehension. These models handle text-related tasks such as summarization and text recognition. In this paper, we focused on question-answering models. A common metric is BLEU, which evaluates the quality of the generated text by comparing it to one or more references [24, 33].

2.4 Related Work

This work is built on extending and complementing the work done by OODTE [15]. OODTE developed a differential testing engine for the ONNX Optimizer (a tool similar to the Runtime Optimizer) that evaluated 130 well-known models from the official ONNX Model Hub. They found that 9.2% of model instances either caused the optimizer to crash or led to the generation of invalid models. However, their tool was tested only on Intel CPUs and did not consider inference times in comparing the original and optimized models. Our work uses the Runtime Optimizer on Nvidia GPUs and uses GPU profiling to calculate and evaluate inference timing.

WhiteFox [37] is a tool that generates high-quality test programs for deep learning compilers, and has found bugs in PyTorch Inductor, TensorFlow-XLA, and TensorFlow Lite. Similarly, NNSmith [13] utilizes a fuzz testing approach to detect bugs in deep-learning compilers.

3 OTX Architecture

This section describes the architecture and methodology behind OTX (shown in Figure 2). At its core, OTX is a Python-based tool that loads models, optimizes them, and then compares both the original and optimized variants across a provided dataset. OTX consists of three primary modules: (1) Orchestrator Module, (2) Inference Module, and (3) Metrics Comparison Module. This module-based architecture is also inspired by OODTE, though their source code has not been released [15].

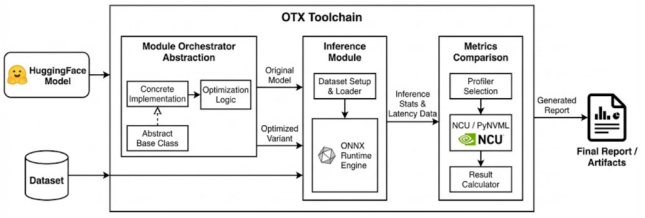

Figure 2: Architecture of OTX based on the three primary modules.

3.1 Orchestrator Module

The orchestrator module defines a base `Model` class that is then sub-classed by each specific model. The base class defines the logic for loading the model, either from disk or from Hugging Face, and then generating a new model based on the desired optimization level.

Each sub-class needs to define the following functions: `setup_dataset` for loading a dataset from disk and preparing it for inference, `score_output` to define the way an output can be scored, `compare_output` to define how different outputs can be compared, and `prepare_input_feed` which defines how the dataset should be passed to the Runtime engine. In essence, the Orchestrator module handles all the model-specific functionality so that the `Inference` and `Metrics_Comparison` modules can focus on model-agnostic logic.

3.2 Inference Module

The `Inference` module is responsible for performing the actual inference using the ONNX Runtime engine and for collecting the metrics used by the `Metrics_Comparison` module.

The module uses the CUDAExecutionProvider [17] and verifies that inference is being performed on the GPU. Inference takes place on a single GPU based on the provided device_id. To ensure that all metrics captured are fair, a warm up session with 10-20 samples is conducted before each inference run.

GPU profiling is done in one of two modes: PYNVML and NCU. PYNVML mode uses the `nvidia-ml-py` library [21] which provides Python bindings to NVIDIA's GPU management and monitoring functions. This mode captures data on GPU utilization, memory usage, execution time, power usage and temperature. PYNVML mode runs as a background thread sampling the GPU state every 10ms. The final result is then computed as an average over all the

samples. NCU mode uses the NVIDIA Nsight Compute CLI [19] that captures detailed performance metrics such as SM throughput & usage, block & grid sizes, memory throughput, and DRAM usage on a per-kernel level. Since the overhead of launching NCU is significant, NCU mode is only run on 5-10 randomly selected samples from the larger dataset.

3.3 Metrics Comparison Module

The `Metrics_Comparison` module analyzes the collected data and scores each model's output using the `score_output` function defined earlier. The `compare_output` function is used to compare the base model against any of its optimized variants. Finally, the results are saved as a JSON report or text file.

4 Experiment Setup

4.1 Experiment Process

For each model, inference is performed on each of the four variants (base, basic, extended, all) ten times each, and the reports are averaged to calculate the final results. If any run was unreasonably slow or fast (differs by more than 3 standard deviations), it was excluded from the results and re-computed.

4.2 Models Used

For our experiment, we selected 15 models from the ONNX Model Zoo on HuggingFace [23]. Eight Image Classification models: ResNet-50 [5], AlexNet [8], ShuffleNet-V2 [16], GoogleNet-12 [34], MobileNet-V2 [30], SqueezeNet1.1 [6], EfficientNet [35] and VGG [32]. Five Object Detection Models: SSD [14], YOLO-V2 [28], Faster R-CNN [29], UltraFace [12], Mask R-CNN [4]. Two Text Generation models: GPT2 [25] and BERT-Squad [2, 26]. We utilized models of various opsets based on availability, ranging from 7 (the lowest compatible with the latest ONNX Runtime version) up to 12.

4.3 Computational Resources

All tests were run on a NVIDIA RTX A5000 with 24GB of memory running CUDA Version 13.1 and Driver version 590.44.01. OTX should be generalizable to other NVIDIA hardware, although this has not yet been tested. The `Orchestrator` module is provider-agnostic and should work with other hardware providers, while the other two modules contain NVIDIA-specific logic.

4.4 Datasets

Standard industry datasets where selected for each model type. For Image Classification, we used 973 images from the ImageNet-256 dataset on Kaggle [3]. For Object Detection, we used two datasets: 750 images from YOLOv3 Lyft Dataset [31] and 593 images from the COCO 2017 Dataset [11]. For question answering, we used 502 questions from the SQuAD2.0 dataset [26].

To accurately compare models of the same type, we employed the same comparison methods. For image recognition, we used Top-1 accuracy and Top-5 accuracy. For object detection, we calculated F1-score, IoU, and precision. For text generation, we calculated BLEU score [24].

5 Results

5.1 No Significant Differences

During testing, the following models displayed no significant difference in result accuracy or inference metrics across any model variants: AlexNet, EfficientNet-Lite4, UltraFace, Mask R-CNN, GPT-2, and BERT-Squad. A further analysis of these models showed that only one or two nodes were changed per model between the base and fully optimized variants. This suggests that the uploaded models had already been optimized, or that the structure of these models doesn't lend to any further optimization. To maintain brevity in the discussion and clarity in the graphs, these models are excluded from the rest of the results section.

5.2 Node Transformations

This section analyzes how different optimization levels affect the number and type of nodes in the computation graph on the models studied. Faster R-CNN is excluded from this section due to its unique model architecture that relies on operations none of the other models use.

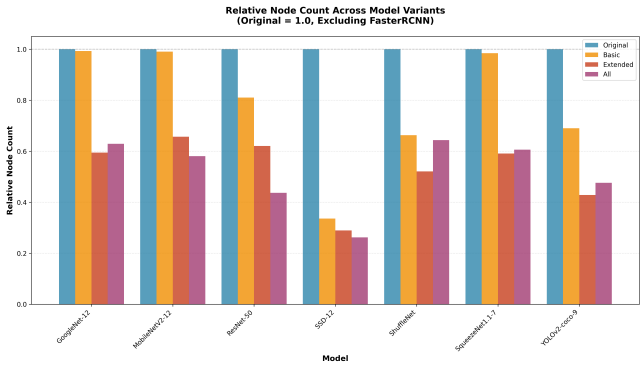

Figure 3: Change in total node count (relative to the original model) across the four model variants.

Figure 3 shows how the total number of nodes change for each model variant relative to the original model. On average, compared to the original model, there was a 22% reduction in the basic variants, 47% reduction in the extended variants, and a 45% reduction in the optimized variants.

Figure 4 shows the average node distribution of the top 5 node types across the image models. Most notably, Convolution nodes jumped from 38% of nodes in the original models to 72% of all nodes in the optimized models. This can be partly explained by the disappearance of ReLU (18% to 0%) and BatchNormalization (11% to 0.2%) nodes, suggesting that the `ENABLE_ALL` pass prefers to aggressively fuse convolution nodes with activation functions and operations, and decompose large convolutions into smaller ones.

5.3 Accuracy

All 15 models tested by OTX reported no difference in accuracy between the original model and any of its optimized variants, implying that the Optimizer did not perform any optimizations that affected the semantics of the underlying model.

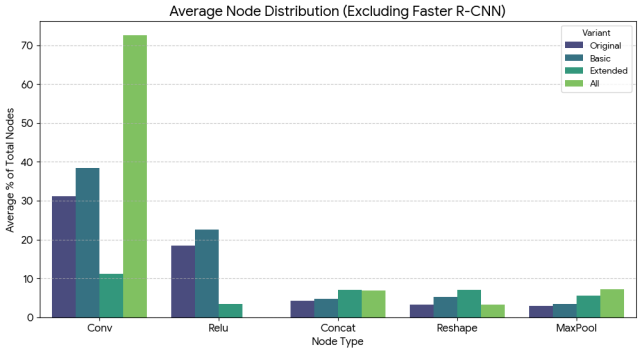

Figure 4: Average node distribution (Top 5 Node Types) across the four model variants.

However, inference on the extended variants of the MobileNetV2 and YOLOv2 models failed, producing the following errors: `INVALID_ARGUMENT`: unsupported convolution activation mode Clip and `INVALID_ARGUMENT`: unsupported convolution activation mode LeakyRelu.

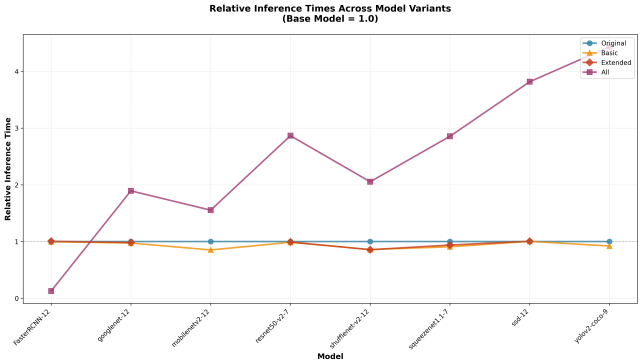

Figure 5: Change in inference performance across the four model variants.

5.4 Perfomance

Figure 5 shows how inference performance changed across the model variants. The basic variant showed a minor but discernible speedup in 4 of the 7 models. The extended variant performed similarly to the original model across all model types. An exception is Faster R-CNN, whose optimized variant showed a near 8x speedup, compared to the significant slowdowns (1.8x to 4x slower) for the other six models.

Figure 6 shows the change in GPU utilization between the original and optimized variants. We can see that GPU utilization dramatically drops for all the models after being fully optimized. This is true even for the Faster R-CNN model that had improved its inference speeds.

For the models that show degraded inference after optimization, two main reasons were discovered: (1) node placement and (2) GPU kernel assignment.

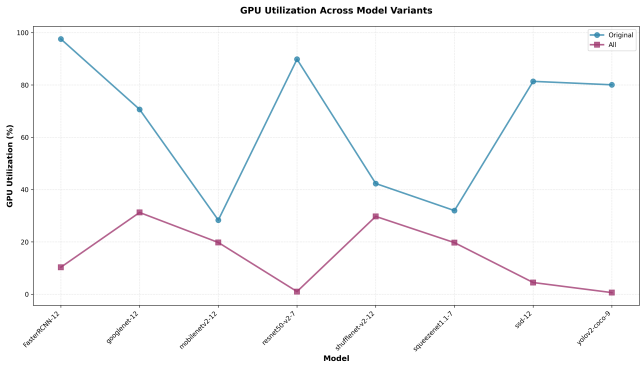

Figure 6: Change in GPU utilization between the original and fully optimized model.

5.4.1 Node Placement. For the ResNet50, SSD, and YoloV2 models, GPU utilization dropped to less than 5%, as most compute nodes began being assigned to the CPU Execution Provider after optimization despite the availability of a GPU (shown in table 1). Upon probing the ONNX Runtime, we observed that these nodes were characterized by the suffix `_NCHWC`. The runtime also explicitly indicated that no GPU Execution Provider was available for nodes with this specific configuration.

Table 1: Percentage of Nodes Assigned to GPU by Optimization Level

	% Nodes Assigned to GPU			
Model	**Original**	**Basic**	**Extended**	**Optimized**
ResNet50	100.0	100.0	100.0	6.4
YoloV2	100.0	100.0	N/A	25.6
SSD	95.3	95.3	95.3	51.7
Faster R-CNN	90.9	90.9	90.8	87.0

5.4.2 Kernel Assignment. The performance of the others models whose inference time worsened after optimization was explained by examining ShuffleNet. Utilizing NCU mode during inference, the results highlight two primary performance characteristics of the original model:

- **High-performance FFT-based convolutions:** The profiler indicates high memory throughput for Fast Fourier Transform convolutions.
  - `fft2d_r2c_32x32`: 60.37% SM throughput, 90.89% memory throughput.
  - `fft2d_c2r_32x32`: 70.16% SM throughput, 92.58% memory throughput.
- **Large** 128×128 **Tile GEMM:** General Matrix Multiply operations showed high Compute (SM) utilization.
  - `ampere_sgemm_128x128_nn`: 93% SM throughput, 67.40% memory throughput.

Analysis of the optimized model revealed that a portion of the convolution nodes began using matrix multiplication kernels characterized by small grid sizes which limit hardware occupancy.

- **Matrix multiplication kernels with low grid size:**
  - `gemmk1_kernel`, 1.80% SM Throughput, 1.63% Memory Throughput, Grid Size: 32 thread blocks

This leads to a counter-intuitive performance loss where operator fusion results in a suboptimal graph (shown in table 2). The transformation shifts execution from optimized kernels to fragmented operations. The original model leverages the cuDNN library [20], which employs highly optimized algorithms for convolution. These operations utilize Matrix-Matrix ($M \times M$) multiplication, allowing for high throughput and utilization.

Table 2: Kernels Launched: Original vs. Optimized Variants of ShuffleNet

Kernel	Block Size	Grid Size	SM Util.	SM Thru.	Memory Thru.
`fft2d_r2c_32x32`	512	256	~ 86%	60.7%	90.89%
`ampere_sgemm_128x128_nn`	256	900	~ 93%	70.16%	93%
GEMV (Optimized Variant)	256	32	~ 21%	1.8%	1.6%
BN (Optimized)	512	3	~ 4%	2.19%	2.2%

The optimized graph introduced nodes dependent on small, generic General Matrix-Vector (GEMV) kernels. This shifted the computation to Matrix-Vector ($M \times V$) operations that require a high volume of sequential launches for the convolution kernels that ShuffleNet relies on. Consequently, the system incurs significant overhead from kernel launching while suffering from low GPU utilization during the execution of these lightweight kernels.

5.4.3 Faster R-CNN Performance. Contrary to the other models, Faster R-CNN demonstrated a unique performance improvement after optimization. The optimized model delivered a significant 7.79× speedup alongside a 91% reduction in peak memory usage (shown in table 3).

Table 3: Faster R-CNN Performance Metrics: Original vs. Optimized

Metric	Original	Optimized
Inference Time (50 images)	83.91s	10.77s
Throughput	0.60 samples/s	4.64 samples/s
GPU Utilization	97.30%	10.30%
Memory Usage (Peak)	18.23 GB	1.67 GB
Precision	0.59	0.60

Profiling of the optimized Faster R-CNN model identified the "backbone" as the primary performance bottleneck. The backbone is the main feature extractor at the beginning of the inference pipeline [29]. The analysis revealed three distinct inefficiency patterns that contributed to high inference latency:

- **High-Latency Grouped Convolutions:** The backbone execution is dominated by `conv2d_grouped_direct_kernel`. While this kernel achieves relatively high hardware saturation (73.91% SM throughput and 73.87% memory throughput), a single invocation necessitates the launch of over 15.56 million threads (15, 200 blocks ×1024 threads), resulting in a standalone execution time of 5.21 ms.
- **Inefficient Frequent Kernels (FFT):** The `fft2d_c2r_32x32` kernel is invoked 151 times per inference pass. However, due to its small grid size (only 4 blocks), it fails to saturate the GPU, yielding negligible utilization (2.70% SM throughput and 4.25% memory throughput).
- **Unfused Activation Overhead:** The profiling detected activation kernels such as `ampere_scudnn_128x64_relu` that can be fused into the preceding convolution kernels.

Significant performance improvements in the optimized model were achieved through a combination of memory and graph optimizations. Data layout adjustments eliminated 84 inefficient transpose operations, which directly reduced memory bandwidth usage. The execution pipeline was further optimized by offloading specific kernels—including `GatherKernel`, `BinaryElementWiseSimple`, `SliceKernel`, `Clip`, `ExpandKernel2D`, and `NMSKernel`—to the GPU, which previously ran on the CPU in the basic model. Finally, all 56 ReLU nodes were fused into the smaller, faster convolution kernels that replaced the monolithic kernel.

6 Limitations

OTX was only tested on 15 models and three model variants, while there are over 2,900 models on the Hugging Face ONNX Model Zoo. Different model architectures may break OTX or provide insights into the performance of the optimizer that weren't covered here.

All experiments were conducted on a GPU with the Ampere architecture, which is two generations old. The CUDAExecutionProvider may perform better on newer architectures.

NVML mode is run as a background polling thread which may add overhead to inference metrics and influence the metrics evaluation.

As mentioned above, some nodes were assigned to the CPU execution provider during inference. These statistics were not monitored or considered by the `MetricsComparison` mode.

7 Future Work

Directions of future work include:

- This project has emphasized the importance of the Execution Provider in assigning the appropriate kernels to nodes and the Inference Engine in assigning nodes to execution providers. Future work could profile this relationship across different execution providers.
- Extending this work to other optimizers, like the ONNX Optimizer and experimenting with the ordering of passes like in OODTE.
- Testing OTX on the TensorRT and TensorRT for RTX Execution Providers.

8 Conclusion

This work introduces OTX, a Python-based tool to test and benchmark any ONNX model by iteratively testing the base model against optimized variants.

References

[1] ONNX Runtime developers. 2021. ONNX Runtime. https://onnxruntime.ai/. Version: x.y.z.
[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In

Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT). 4171–4186.

[3] dimensi0n. 2023. ImageNet 256×256 (1,000-class subset) Dataset. https://www.kaggle.com/datasets/dimensi0n/imagenet-256. Accessed: 2025-12-08.

[4] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2961–2969.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[6] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2017. SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5MB Model Size. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 1–10.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc., 1097–1105.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*. 1097–1105.

[9] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[10] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444. doi:10.1038/nature14539

[11] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2017. *COCO 2017: Common Objects in Context 2017*. https://cocodataset.org/#home

[12] Linzaer. 2019. Ultra Light Fast Generic Face Detector. https://github.com/Linzaer/Ultra-Light-Fast-Generic-Face-Detector-1MB.

[13] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23)*. ACM, 530–543. doi:10.1145/3575693.3575707

[14] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. In *European Conference on Computer Vision (ECCV)*. Springer, 21–37.

[15] Nikolaos Louloudakis and Ajitha Rajan. 2025. OODTE: A Differential Testing Engine for the ONNX Optimizer. https://arxiv.org/abs/2505.01892. Preprint — arXiv:2505.01892; accessed 08Dec2025.

[16] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 116–131.

[17] Microsoft. 2025. CUDA Execution Provider. https://onnxruntime.ai/docs/execution-providers/CUDA-ExecutionProvider.html. Accessed: 2025-12-08.

[18] Microsoft. 2025. Graph Optimizations in ONNX Runtime. https://onnxruntime.ai/docs/performance/model-optimizations/graph-optimizations.html. Accessed: 2025-12-08.

[19] NVIDIA Corporation. 2025. *Nsight Compute CLI — Nsight Compute Documentation*. NVIDIA Corporation. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html Accessed: 2025-12-08.

[20] NVIDIA Corporation. 2025. *NVIDIA cuDNN — CUDA Deep Neural Network Library Documentation*. NVIDIA Corporation. https://developer.nvidia.com/cudnn Accessed: 2025-12-08.

[21] NVIDIA Corporation. 2025. nvidia-ml-py: Python Bindings for the NVIDIA Management Library. https://pypi.org/project/nvidia-ml-py/. Version 13.590.44, Accessed: 2025-12-08.

[22] ONNX Community. 2024. Open Neural Network Exchange (ONNX). https://onnx.ai/. Accessed: 08 December 2025.

[23] onnxmodelzoo / Hugging Face. [n. d.]. onnxmodelzoo — ONNX Model Zoo on Hugging Face. https://huggingface.co/onnxmodelzoo. Accessed: 2025-12-08.

[24] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL '02)*. Association for Computational Linguistics, USA, 311–318.

[25] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI Technical Report* (2019). https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[26] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know What You Don't Know: Unanswerable Questions for SQuAD. arXiv:1806.03822 [cs.CL] https://arxiv.org/abs/1806.03822

[27] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 779–788.

[28] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: Better, Faster, Stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 7263–7271.

[29] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *Advances in Neural Information Processing Systems*, Vol. 28. 91–99.

[30] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4510–4520.

[31] Lavanya Shukla. 2025. YOLOv3 Lyft Dataset. https://www.kaggle.com/code/lavanyashukla01/yolov3-lyft-dataset. Kaggle notebook; accessed 2025-12-08.

[32] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations (ICLR)* (2015).

[33] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.), Vol. 27. Curran Associates, Inc., 3104–3112.

[34] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9.

[35] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*. 6105–6114.

[36] The OpenXLA Team. 2023. *XLA: Optimizing Compiler for Machine Learning*. https://github.com/openxla/xla Accessed: 2025-12-08.

[37] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (Oct. 2024), 709–735. doi:10.1145/3689736