EDGE AI Academy

Summer School

7-8 July 2025, Pisa, Italy

Edge AI Academy Summer School

“Deploying, benchmarking and profiling of large vision models at the edge.”

Carmelo Scribano, University of Modena and Reggio Emilia, Italy

7-8 July 2025 Pisa, Italy

Outline

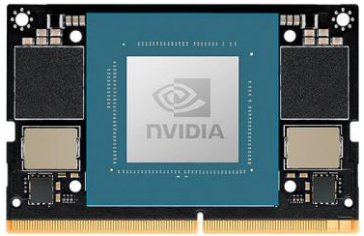
- This 45’ lecture will cover the following topics.
- Introduction: overview of the NVIDIA Inference Ecosystem.
- Part 1 - Deployment:
  - Exporting of large vision model to tensorRT - the experience in dAIEdge.
  - Limitations and handling of special cases.
  - Details of the execution model of TensorRT.
- Part 2 - Benchmarking:
  - Suggested workflow for benchmarking in tensorRT on edge platforms.
  - Detailed analysis advanced profiling tools.
- Part 3 - Performance Optimization:
  - Out-of-the box performance optimization with Nvidia modelopt.

Copyright © 2025

Background: NVIDIA Jetson Family

- The NVIDIA Jetson platform was designed to bring **GPU-accelerated** computing to edge devices.
  - Jetson System on Module (SoM) are equipped with ARM Cortex-A CPUs and CUDA Capable GPUS.
  - Jetson run Linux4Tegra (L4T), a fork of the Linux kernel. The Jetpack SDK include L4T-based ubuntu distro and all additional software (CUDA, TensorRT etc..).

Jetson ORIN NANO Developer Kit



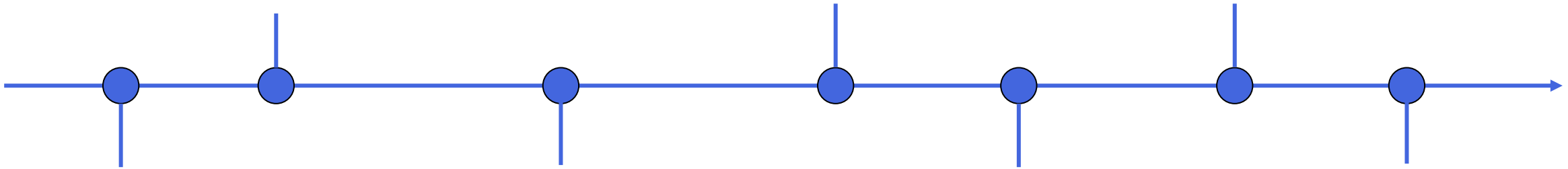
Jetson orin NANO SoM

Industrial PC Powered by ORIN SoM

Copyright © 2025

Background: NVIDIA Jetson Family

- Launched in 2014, the NVIDIA Jetson platform was designed to bring **GPU-accelerated** computing to edge devices.



Jetson TK1 (2014)
- First embedded CUDA-capable board (192 Kepler Cores)

Jetson TX1 (2015)
- Deep Learning Capable
- Pascal Cores

Jetson TX2 (2017)

Jetson AGX Xavier (2018)

Jetson Nano (2019)
- Affordable AI Edge Platform.

Jetson Orin Series (2022)
- up to 275 TOPS.
- Scalable from Nano to AGX.

Next: Jetson Thor Family
- Launching later 2025

Copyright © 2025

Background: NVIDIA Jetson Family

- Comparison of currently available jetson boards

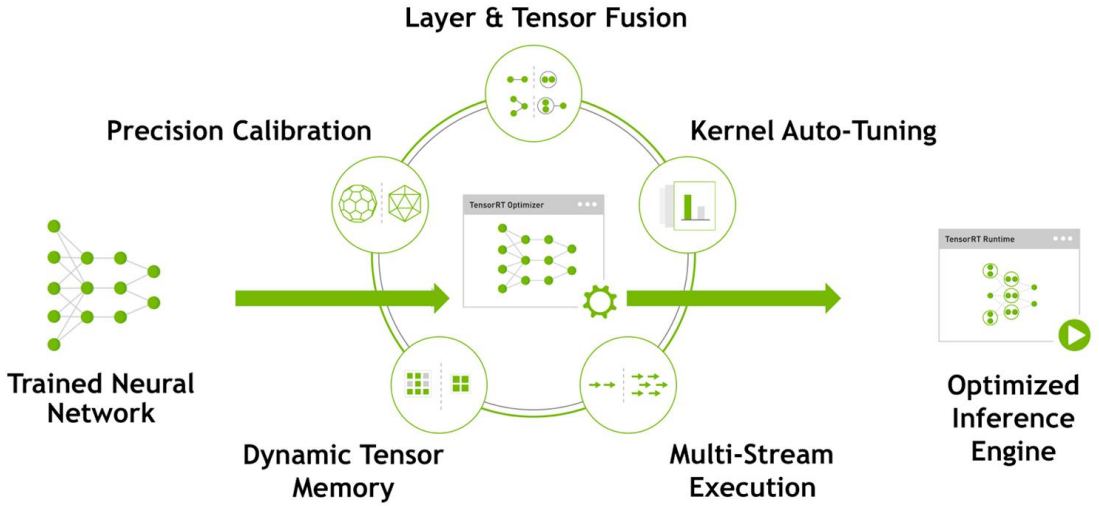
	Jetson NANO	Jetson ORIN NANO	Jetson AGX Orin
INT8 AI OPS	472 GFLOPS (no INT8!)	67 INT8 TOPS	275 TOPS
GPU	128 Maxwell Cores	1024 Ampere cores + 32 Tensor Cores	2024 Ampere cores + 64 Tensor Cores
GPU	Cortex®-A57 4-core	Cortex®-A78AE 6-core	Cortex®-A78AE 12-core
Memory *	4 GB 64-bit LPDDR4	8GB 128-bit LPDDR5	64 GB 256-bit LPDDR5
Storage	eMMC 5.1 16 GB	MicroSD and NVMe	eMMC 5.1 64 GB
Power	5W-10W	7W–25W	15 W - 60 W
Price (developer Kit)	100$	250$	>2000$

***** In Jetson Memory is unified between CPU and GPU.

Copyright © 2025

TensorRT

- TensorRT is NVIDIA high-performance inference SDK and runtime, designed to optimize, accelerate and deploy AI Models.
  - Released in 2017, TensorRT is compatible with both desktop GPUs and Jetson Platform. Included in JetPack SDK.
  - Closed-source proprietary software. Most of the Inner workings are undocumented.
  - Support both Python and C++



Copyright © 2025

Part 1: Deploying

Intro to TensorRT workflow

- Build and Execution Phases

TensorRT Default Optimizations:

- Layer Fusion, Kernel Auto-Tuning, Graph Optimization

Advanced: Custom Extensions

- Implement custom TensorRT layers (For unsupported operators or to streamline postprocessing).

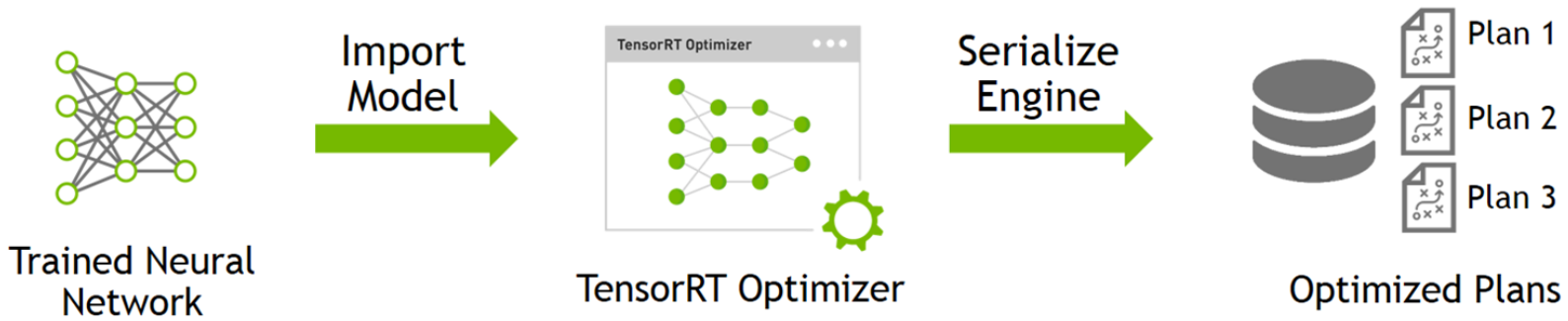
Copyright © 2025

TensorRT: Build Phase

Build Phase

- Parse model (via network definition or parsing).
- Construct and optimize the computation graph.
- Apply optimizations:
  - Layer fusion
  - Kernel auto-tuning
  - Graph simplification
- Select optimized kernels and memory layout
- Serialize the Engine Plan for future inference.

Note: This is a resource-intensive, one-time operation permodel / device / precision configuration.

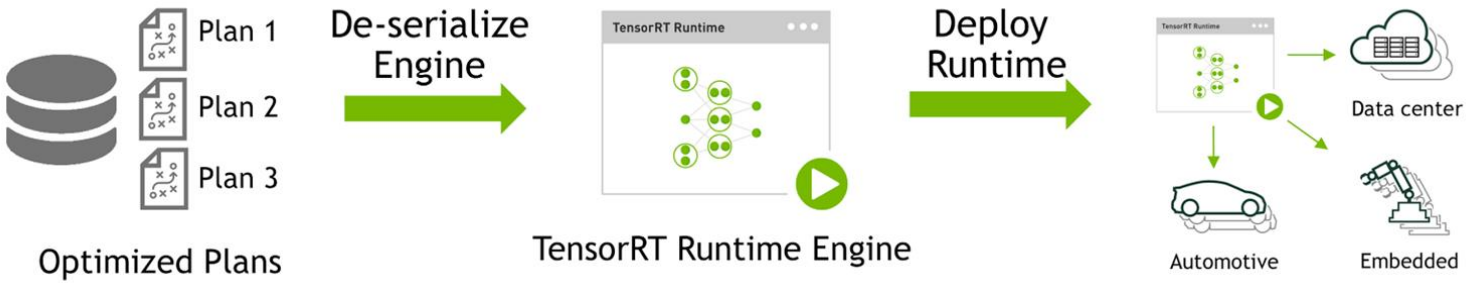


Copyright © 2025

TensorRT: Execution Phase

Execution Phase

- Load serialized Engine Plan into memory.
- Allocate device memory for inputs, outputs, and intermediate buffers.
- Run optimized inference by executing pre-selected kernels.



Copyright © 2025

Build Phase Optimizations

- TensorRT attempts to perform many different types of optimizations in a network during the build phase.

Copyright © 2025

Kernel Auto-tuning

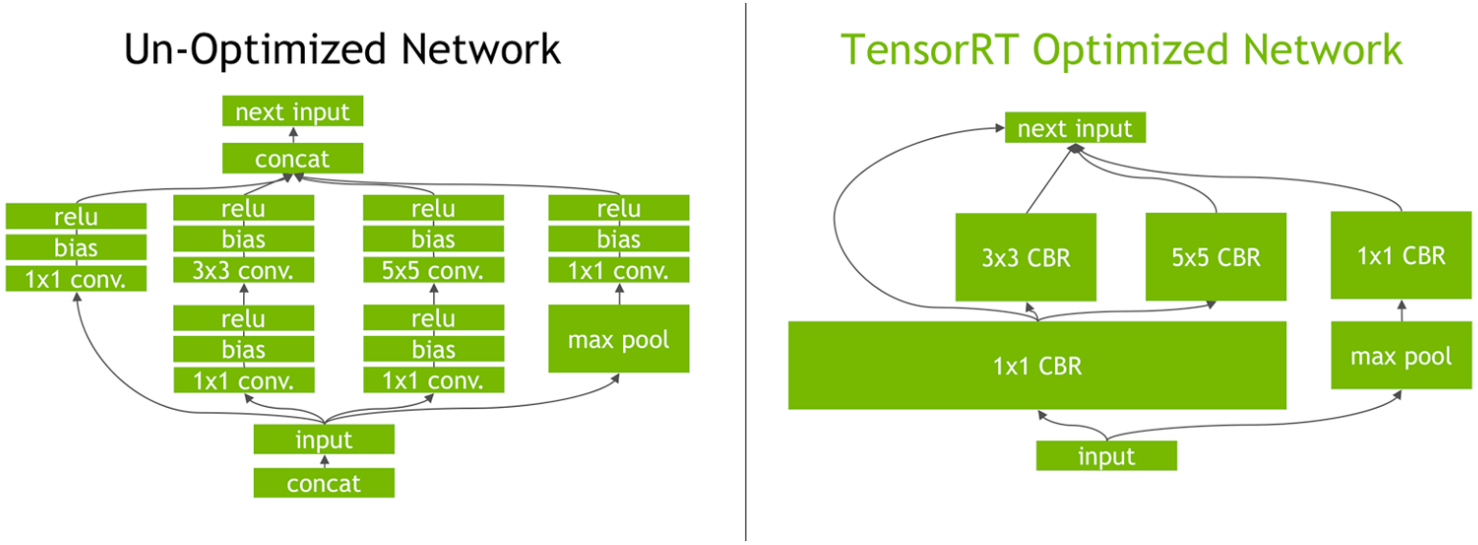
- During Build phase, TensorRT chooses from different of specialized kernels implementations for the same operation.
  - Many of them hand-tuned and optimized for a range of parameters and target platforms.
  - TensorRT will pick the implementation from a library of kernels that delivers the best performance for the target GPU, input data size, filter size, tensor layout, batch size and other parameters.

- This ensures that the deployed model is performance tuned for the specific deployment platform as well as for the specific neural network being deployed
  - This is part of the reason why engines are platform specific.

Copyright © 2025

Layer and tensor Fusion

- The fusion optimization step detects supported patterns of operations and fuses multiple layers into one
- **Example:**
  - Conv + ReLU: two separate kernel launches
  - Fused Implementation is automatically inserted by TensorRT



More: https://docs.nvidia.com/deeplearning/tensorrt/latest/performance/best-practices.html#types-of-fusions

Copyright © 2025

Reduced Precision Computation

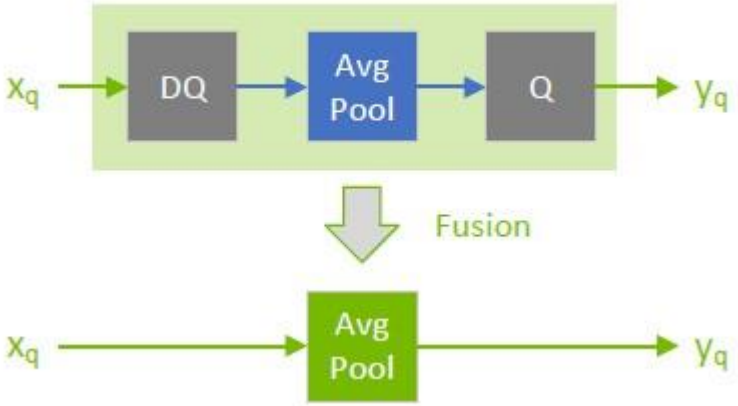
- TensorRT supports low-precision arithmetic to accelerate inference.
  - TensorRT quantizes activations and weights for INT8, FP8, and FP4 (Weight-only quantization is supported for INT4.)
- TensorRT supports two modalities
  - **Implicit Quantization:** (Deprecated in TensorRT > 10): during build phase, TensorRT use the (provided) scale to opportunistically to optimize layer execution (if the layer runs faster in INT8, then is quantized).
  - **Explicit Quantization:** quantization information encoded directly in the model. ONNX can encode this information as Q/DQ layers. The Build phase will stick to the model definition.

More: https://docs.nvidia.com/deeplearning/tensorrt/latest/inference-library/work-quantized-types.html

Copyright © 2025

Explicit Quantization

- When TensorRT detects the presence of Q/DQ layers in a network, it builds an engine using explicit-precision processing logic.



- Q/DQ Layers can be placed manually when exporting ONNX model.
- In Part 3 we will use Nvidia modelopt for Post-Traning Quantization

Copyright © 2025

Quantization Modes

- There are three supported quantization scale granularities:
  1. **Per-tensor quantization**: a single scale value (scalar) is used to scale the entire tensor.
  2. **Per-channel quantization:** a scale tensor is broadcast along the given axis - for convolutional neural networks, this is typically the channel axis.
  3. **Block quantization:** the tensor is divided into fixed-size 1-dimensional blocks along a single dimension. A scale factor is defined for each block.

Copyright © 2025

Quantization Schemes and Precision

Note: INT4 and FP4 quantized weights are stored by packing two elements per byte.

Quantization Schemes	INT8	FP8	MXFP8	INT4	NVFP4
Representation	8-bit signed 2's complement	S1E4M3 floating point	• S1E4M3 floating point	4-bit signed 2's complement	S1E2M1 floating point
Weight quantization	Per-tensor/per-axis	Per-tensor/per-axis	Per-block (block size = 32)	Per-block (block sizes = `{64, 128}`)	Per-block (block size = 16)
Activation quantization	Per-tensor	Per-tensor	Dynamic, per-block (block size = 32)	No	Dynamic, per-block (block size = 16)
Implicit quantization	Yes	No	No	No	No
Explicit quantization	Yes	Yes	Yes	Yes	Yes
Scale data type	FP32, FP16, BF16	FP32, FP16, BF16	E8MO	FP32, FP16, BF16	FP32, FP16, BF16

Copyright © 2025

ONNX

ONNX (Open Neural Network Exchange) is a platform agnostic representation for ML Models.

- An ONNX model contains:
  - Metadata: Model version, producer info, and optional custom properties.
  - Model Graph: Computational graph (layers, operations, data flow).
  - Weights & Parameters: Stored in a binary format inside the model file.
- Framework Agnostic: does not contain framework or platform specific informations.
- Operator Set (**Opset**): Defines supported operations (e.g., Conv, Relu, Softmax).

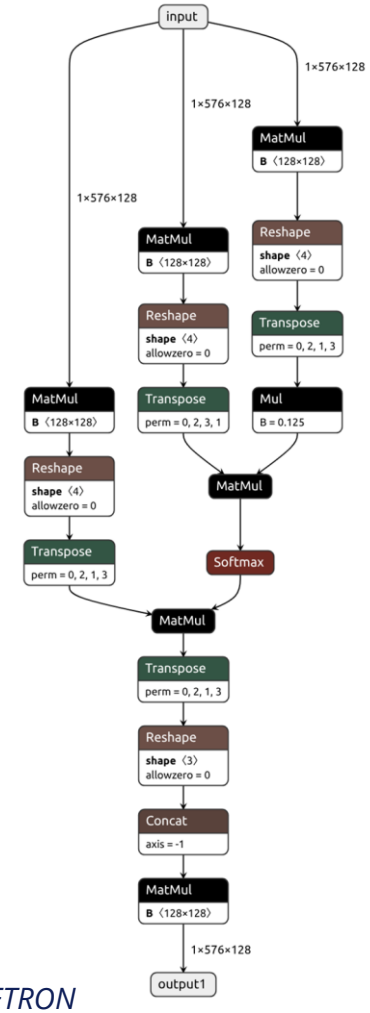
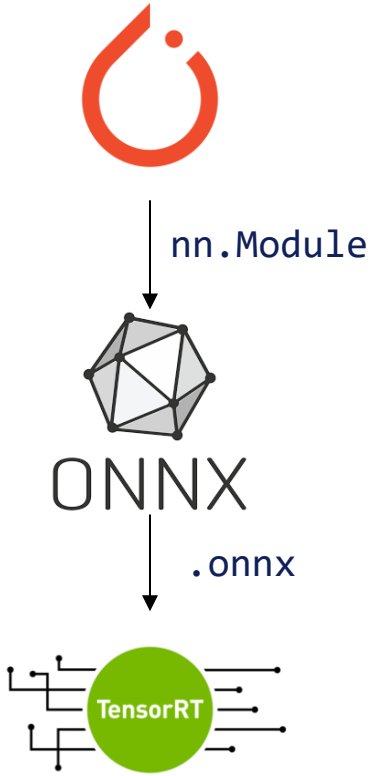


Image: MultiHead Self Attention Layer ONNX graph, image obtained using NETRON

Copyright © 2025

JIT Tracing PyTorch Models

- Tracing allow to build a static representation from a Just In Time (JIT) compiled pytorch model.
  - torch.onnx.export() is the tdefault Pytorch eporter API.
  - Alternatively, TorchScript-based ONNX exporter is available since 1.2.0



Limitations:

- Do not record input-based control flow
- Python-based dynamic behaviour is lost
  - Only tensor operations are recorded
  - Incomplete or invalid ONNX Graphs can be produced

More: https://docs.pytorch.org/docs/stable/onnx.html

Copyright © 2025

Preliminary Steps

- Cleanup the forward() path of any tracking-breaking logic
  - Therefore some commonly used constructs incompatible with export.

```
def forward(self, x):
    result = x
    for _ in range(int(x.sum().item())):
         result = result * 0.9
    return result
```

- Data dependant python loop

```
def forward(self, x):
    n = int(x[0].item())
    y = x.view(n, -1)
    return y
```

- Data dependant reshaping

```
def forward(self, x):
    ops = []
    for _ in range(3):
        ops.append(x * 0.5)
    return sum(ops)
```

- Operation o Python data structures

Copyright © 2025

Preliminary Steps

- Monkey-Patching is a clever technique to simplify the forward path at export time, without compromising the original code.

```
def forward(self, x, y):
    x = self.layers(x)
    loss = self.loss(x,y)
    return x, loss
```

- Forward function compute and returns loss for training purposes.

```
model = MyModel()

def clean_forward(self, x):
    x = self.layers(x)

# Patch the method
model.forward = clean_forward.__get__(model, MyModel)

# Now export to ONNX without print
torch.onnx.export(model, torch.randn(1, 3), "model.onnx")
```

- Patch model on-the-fly with clean forward function, without breaking the original implementation.

Copyright © 2025

Building Engine from ONNX

- Quick way trtexec:
  - trtexec is a command-line tool that integrate advanced benchmarking capabilities and can also be uset to serialize trt engines
  - **Example:** `trtexec --onnx=model.onnx --saveEngine=model.trt --skipInference`
- Python API: programmatic export offers finer control over the building and serialization phases.

```
# Create builder, network and parser
builder = trt.Builder(TRT_LOGGER)
network_flags = 1 <<
int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
network = builder.create_network(network_flags)
parser = trt.OnnxParser(network, TRT_LOGGER)

# Parse ONNX model
with open("model.onnx", "rb") as f:
    parser.parse(model_file.read())
```

```
# Create builder config
config = builder.create_builder_config()
config.max_workspace_size = 1 << 30  # 1 GB

# Create optimization profile (for dynamic shapes)
profile = builder.create_optimization_profile()
config.add_optimization_profile(profile)

# Build engine
engine = builder.build_engine(network, config)

# Serialize and save engine
with open(engine_path, "wb") as f:
    f.write(engine.serialize())
```

Copyright © 2025

Handling Dynamic Shapes

- TensorRT supports also Dynamic **Input** shapes, defined at runtime.
  - At build time, the *optimization profile* must specify the permitted range of dimensions.
  - Example:

```
# Assume input name and shape from ONNX
input_name = network.get_input(0).name
profile.set_shape(input_name,
                  min=(1, 3, 224, 224),
                  opt=(8, 3, 512, 512),
                  max=(16, 3, 1024, 1024))
config.add_optimization_profile(profile)
```

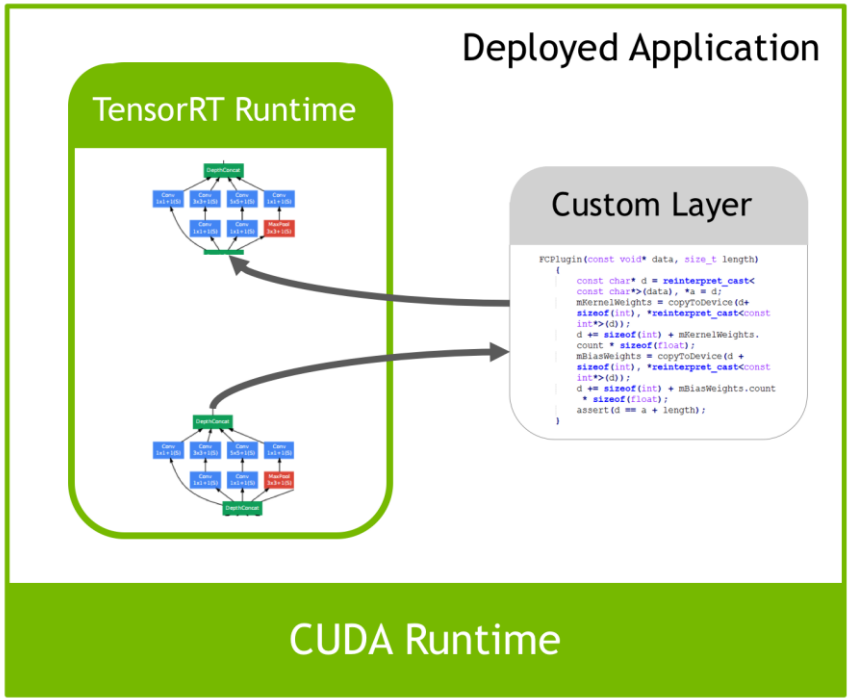
- Handling dynamically shaped **output** requires explicit handling of output memory allocation (more on this later!)

Copyright © 2025

Unsupported Operators

- Some specialized operators might not be supported:
  - Custom layers, intricate post-processing…
- TensorRT can be extended with custom C++/Cuda plugins
  - Since TensorRT 10, Python is also supported for plugins development

- C++/CUDA plugins are compiled as static object (.so) and loaded during both Building and Execution phases.
- TensorRT Plugin interface has gone through several changes, since TensorRT10, **`IPluginV3`** is the default plugin interface.



Copyright © 2025

Use Case

- As an example, consider the model developed by **INSAIT** as part of the dAIEdge project (not public yet, under review).
  - Based on Prisadnikov et al, "*A Simple and Generalist Approach for Panoptic Segmentation*" (https://arxiv.org/pdf/2408.16504).

- Architecture:
  - DINO-V2 as shared feature extractor.
  - Multi-task approach on 4 vision tasks (object detection, semantic and instance segmentation, depth estimation).
    - Yolo-Style object detection head.
    - Similar segmentation setup to Prisadnikov's paper.

Copyright © 2025

Implementing a Plugin

- Developing a CUDA plugin is a challenging process, therefore are some tips and tricks.

1. Start by developing the required CUDA kernels as standalone C++/Cuda Code.
2. Thoroughly check the results against the reference Pytorch Implementation*
3. (Optional) Wrap the CUDA code to write a Pytorch extension for further testing (https://docs.pytorch.org/tutorials/advanced/cpp_extension.html).
4. TensorRT plugins Involve a lot of boilerplate code, an useful tool (https://github.com/NVIDIA-AI-IOT/tensorrt_plugin_generator) can help you generating Plugin skeleton (at the moment of writing IPLuginV3 is not supported).

* You can use **cupy** to run CUDA code in python, with the convenience of easier memory management and numpy input/outputs checking.

Copyright © 2025

Implementing a Plugin

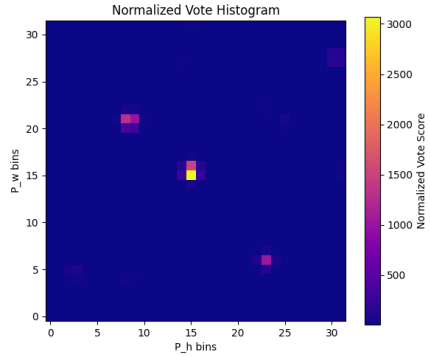
- **Example:** deploying instance segmentation layer.
  - Coordinate of instance centroids (x,y) are encoded as sin/cos positional encoding (PE).
  - Each output location predict PE coding of the closest centroid, or zero if no centroid is close enough.
  - Output locations cast ‘votes’ for the presence of a centroid at any possible location.
  - Candidate centroids are selected with local maxima search, candidate masks are defined with a distance threshold to the closest centroid.

Copyright © 2025

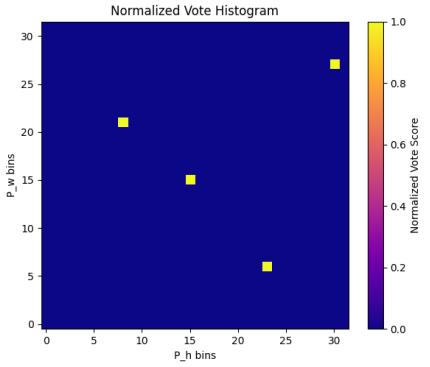
Implementing a Plugin

- The Instance segmentation plugin can be built with 3 kernels.

1. `fused_vote_kernel`: each thread accumulate votes for a specific candidate location. Intermediate results are stored in a buffer (H'xW').



1. `local_maxima_discrete_derivative`: implements discrete derivative over a 3x3 grid to suppres non local maxima values. The remaining non-zero values are candidate mask centroids.



1. `masks_aggregation`: To avoid dynamic output shapes, all H'xW' masks are computed (zeros for non-candidate locations).

Copyright © 2025

Implementing a Plugin

Several functions need to be implemented. Therefore are some notable ones.

- **IPluginV3OneRuntime::enqueue**: is the most important functions, where the kernels are launched.

```
int32_t enqueue(PluginTensorDesc const* inputDesc, PluginTensorDesc const* outputDesc, void const* const* inputs, void* const*
outputs, void* workspace,
cudaStream_t stream) noexcept override
{
    // populate outputs and return status code
}
```

- `IPluginV3OneBuild::getOutputDataTypes` and `IPluginV3OneBuild::getOutputShapes` are used at build time to know shapes and data types for proper allocation.
- `IPluginV3OneBuild::getWorkspaceSize` is used to specify extra space that needs to be allocated beside input/output buffers (e.g, for intermediate results).
- **Note:** Memory should never be explicitly allocated (cuda_malloc()) inside plugin.

Copyright © 2025

Exporting custom layers

- The Symbolic API can be used to export custom operators to ONNX

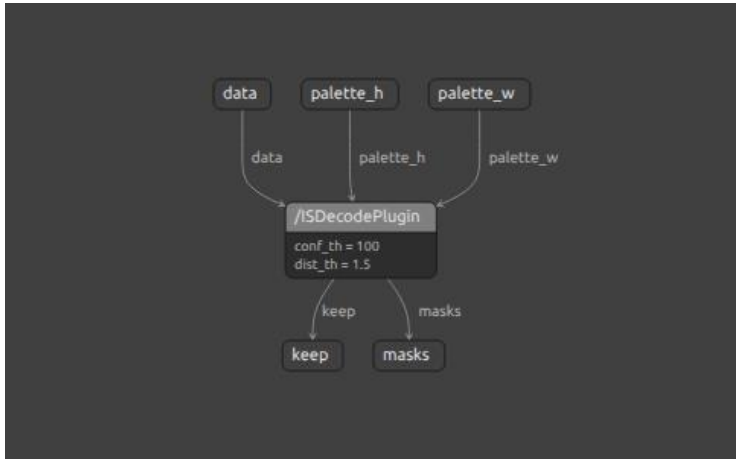
```
class InstSegDecode(Function):
    @staticmethod
    def symbolic(g, data, palette_h, palette_w, conf_th, dist_th):
        node = g.op("CustomTorchOps::ISDecodePlugin",
                    data, palette_h, palette_w,
                    conf_th_f=conf_th, dist_th_f=dist_th, outputs=2)

        num_masks = palette_h.type().sizes()[0] * palette_w.type().sizes()[0]
        H, W = data.type().sizes()[1:3]

        masks = node[0].setType(
            data.type().with_sizes([num_masks, H, W])
        )
        keep = node[1].setType(
            data.type().with_sizes([num_masks])
        )

        return masks, keep

    @staticmethod
    def forward(ctx, data, palette_h, palette_w, conf_th, dist_th):
        return instseg_decode.forward(data, palette_h, palette_w, conf_th, dist_th)
```



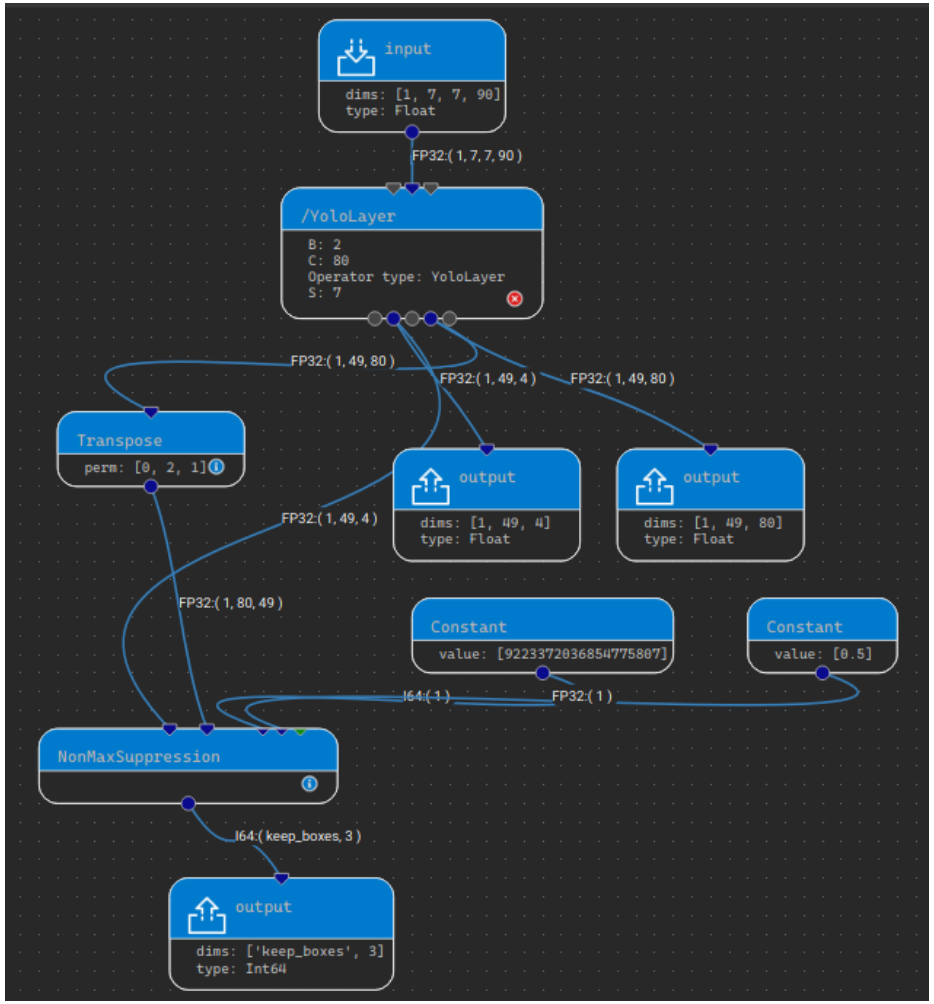
Instance Segmentation Decoding Plugin In ONNX Model graph.

Copyright © 2025

Graph Surgery

In some cases it can be useful to modify the ONNX graph before converting it.

- **Example:** Non-Maxima Suppression after custom layer.
  - Pytorch NMS layer doesn't export cleanly to ONNX, producing redundant operations.
  - Hand-wiring NonMaxSuppression operator produce a cleaner graph.
- **Nvidia Nsight DL Designer** can e used to visually operate on the ONNX graph.
- The Graphsurgeon API allow programmatic operation (https://docs.nvidia.com/deeplearning/tensorrt/latest/_static/onnx-graphsurgeon/index.html)



Copyright © 2025

Inference

- Deserialize engine and create execution context.

```
import cupy as cp

# Build runtime
runtime = trt.Runtime(TRTInference.TRT_LOGGER)
with open(engine_path, 'rb') as engine_file:
    self.engine =
runtime.deserialize_cuda_engine(engine_file.read())

# Create engine execution context
self.context = self.engine.create_execution_context()
self.stream = cp.cuda.Stream()
```

- Allocate Input and output bindings:
  - TensorRT uses explicit memory management.
  - On Jetson, unified memory avoid repeated host-device copy overhead.
  - The engine contains only layer definitions and tactics, actual memory must be allocated before execution.

Copyright © 2025

Inference

- In Pythons, device memory can be managed in several ways:
  - Explicitly pyCUDA calls, preferably with pinned memory;

    ```
    h_input = cuda.pagelocked_empty(input_shape, dtype=np.float32)
    ```

    - Cupy numpy-like interface

      ```
      buffer = cp.empty(size, dtype=dtype)
      ```

- Copy host to device (populate input buffers with input data)
- Set allocated buffers:

```
for i in range(self.engine.num_io_tensors):
    self.context.set_tensor_address(self.engine.get_tensor_name(i), self.bindings[i])
```

- Enqueue:

```
self.context.execute_async_v3(stream_handle=self.stream.ptr)
```

- Copy device to host (retrieve outputs from output buffers)

Copyright © 2025

Dynamic Output Shapes

1. To handle data-dependant, dynamic output shapes, it is possible to associate a custom IOutputAllocator to handle memory allocation at runtime.
   a. Derive allocator class from `IOutputAllocator`.
   b. Override the `reallocateOutput` and `notifyShape` methods. TensorRT calls the first when it needs to allocate the output memory and the second when it knows the output dimensions.
2. Allocator is then associated with the appropriate output:

```
self.context.set_output_allocator(tensor_name, self.out_allocator)
```

Example: NonMaxSuppression as dynamic output shape depending on how many bounding boxes are kept in the final prediction.

More: https://docs.nvidia.com/deeplearning/tensorrt/latest/inference-library/work-dynamic-shapes.html#dynamically-shaped-output

Copyright © 2025

Pythonic Inference

- **Tripy** is a debuggable, Pythonic frontend for TensorRT, a deep learning inference compiler (still in beta!)
- https://nvidia.github.io/TensorRT-Incubator/index.html

Copyright © 2025

Part 2: Benchmarking and Profiling

- Baseline wall-clock timing in Pytorch and TensorRT
- Detailed Memory Analysis
- Advanced profiling strategies

Copyright © 2025

Common Pitfalls

- The wall clock time asses the full model end-to end latency.
- Common pitfalls include:

1. Relying solely on time.time()

```
# Improper benchmarking
start = time.time()
model(input)
elapsed = time.time() - start
```

- time.time() has low resolution on some systems
- Error: not accounting for asynchronous GPU operation

1. Not Synchronizing GPU Operations

```
torch.cuda.synchronize()
start = time.perf_counter()
model(input)
torch.cuda.synchronize()
elapsed = time.perf_counter() - start
```

- perf_counter() provide maximum resolution.
- torch.cuda.sinchronize() ensure GPU operations are complete.

Copyright © 2025

Common Pitfalls

3. Not accounting for variability

- Run several (5-10) warm-up iterations before measuring
- Average across (50-100) inference iterations
- (bonus) always remember to disable gradient computation!

```
# Warm-up iterations (to stabilize results)
for _ in range(warmup_iters):
    with torch.no_grad():
        _ = MM.model(input_tensor)
```

```
start_time = time.perf_counter()
with torch.no_grad():
for _ in range(test_iters):
    _ = MM.model(input_tensor)
elapsed_time = time.perf_counter() - start_time

avg_inference_time = round((elapsed_time / test_iters *
1000),3)
```

3. Unlocked GPU Frequencies

- NVIDIA GPUs use dynamic frequency scaling (GPU clocks are adjusted based on the load)
- Benchmark results fluctuate unpredictably!
- Always remember to (temporarily) LOCK frequency clocks before benchmarking

```
$ nvidia-smi -q -d SUPPORTED_CLOCK # List supported clocks
$ sudo nvidia-smi -ac <mem_clock>,<graphics_clock>  # Lock clocks
```

Copyright © 2025

TensorRT Benchmarking

The trtexec tool is the official NVIDIA command-line utility for:

- Running inference on TensorRT engine
- Measuring latency, throughput, and execution time
- Stress-testing and validating engine performance

Example:

```
$ trtexec --loadEngine=model.engine \
        --useCudaGraph \
        --noDataTransfers \
        --useSpinWait \
        --iterations=100 \
        --avgRuns=100 \
        --exportTimes=stats.json
```

- Can also specify precision, load plugins any more.
- Statistics are stored to json file for later analysis (e.g, with pandas).

Note: Run jetson_cloks before benchmarking on Nvidia Jetson:

```
$ sudo jetson_clocks
```

- Disables dynamic scaling and lock all clocks to max frequency.

Copyright © 2025

TensorRT Benchmarking

- CUDA Events are a powerful tool to profile execution when using programmatic APIs (Python/C++).

```
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);

cudaEventRecord(start, stream);
context->enqueueV3(stream);
cudaEventRecord(end, stream);

cudaEventSynchronize(end);
float totalTime;
cudaEventElapsedTime(&totalTime, start, end);
```

- With profiling we will get finer per-layer and even per-operator performance metrics.

Copyright © 2025

Tracking Memory: Pytorch

- Memory benchmarking in Pytorch:
  - Peak memory: highest in-use memory observed.
  - Allocated memory: current live tensor memory

```
torch.cuda.empty_cache()
torch.cuda.reset_peak_memory_stats(device)
torch.cuda.synchronize()

start_mem = torch.cuda.memory_allocated(device)

# Perform inference
with torch.no_grad():
    output = model(input)

end_mem = torch.cuda.memory_allocated(device)
peak_mem = torch.cuda.max_memory_allocated(device)

# results
memory_used = (end_mem - start_mem) / (1024 ** 2)
peak_memory = peak_mem / (1024 ** 2)
```

- Always clear stats and reset caches before starting.
- Peak memory captures the minimum memory requirement to run the model.
- `memory_reserved()` can be used to detect memory allocated by pytorch caching allorcator (`memory_reserved` >= `memory_allocated`)

Copyright © 2025

Tracking Memory: TensorRT

- TensorRT does not expose built-in runtime memory profiling.
- The recommended approach programmatically track TensorRT memory usage is to implement a custom memory allocator.
  - Override `trt.IGpuAllocator` class.
  - Implement allocation/deallocation functions, keeping track of the allocated memory.

```
class TrackingAllocator(trt.IGpuAllocator):
    def __init__(self):
        super().__init__()
        self.total_allocated = 0
        self.peak_allocated = 0

    def allocate(self, size, alignment, flags):
        ptr = cuda.mem_alloc(size)
        self.total_allocated += size
        self.peak_allocated = max(self.peak_allocated, self.total_allocated)
        return int(ptr)
```

More: https://docs.nvidia.com/deeplearning/tensorrt/latest/performance/best-practices.html#tracking-memory

Copyright © 2025

Profiling TensorRT

- Collecting profiling information is essential to understand performance results of the optimized model.
  - TensorRT has a lightweight builtin Profiler API (IProfiler)

```
class LayerProfiler(trt.IProfiler):
    def __init__(self):
        super().__init__()
        self.layer_times = {}

    def report_layer_time(self, layer_name, time_ms):
        if layer_name not in self.layer_times:
            self.layer_times[layer_name] = []
        self.layer_times[layer_name].append(time_ms)

# Attach to execution context
context.profiler = LayerProfiler()
```

Copyright © 2025

Profiling TensorRT

- trtexec can also extract per-layer profiler information by specify `--profilingVerbosity=detailed` and `--dumpLayerInfo`

Example:

```
$ trtexec --onnx=resnet50-v1-12-quantized.onnx --shapes=data:4x3x224x224 --stronglyTyped --noDataTransfers --
useCudaGraph --useSpinWait --profilingVerbosity=detailed --dumpLayerInfo --dumpProfile --separateProfileRun
```

Result:

```
[04/26/2024-00:42:55] [I]    Time(ms)     Avg.(ms)   Median(ms)   Time(%)   Layer
[04/26/2024-00:42:55] [I]       56.57       0.0255       0.0256       1.8   resnetv17_stage4_conv7_weight +
...
[04/26/2024-00:42:55] [I]      103.86       0.0468       0.0471       3.3   resnetv17_stage4_conv8_weight +
...
[04/26/2024-00:42:55] [I]       46.93       0.0211       0.0215       1.5   resnetv17_stage4_conv9_weight +
...
[04/26/2024-00:42:55] [I]       34.64       0.0156       0.0154       1.1   resnetv17_pool1_fwd
[04/26/2024-00:42:55] [I]       63.21       0.0285       0.0287       2.0   resnetv17_dense0_weight + ...
[04/26/2024-00:42:55] [I]     3142.40       1.4149       1.4162     100.0   Total
```

Copyright © 2025

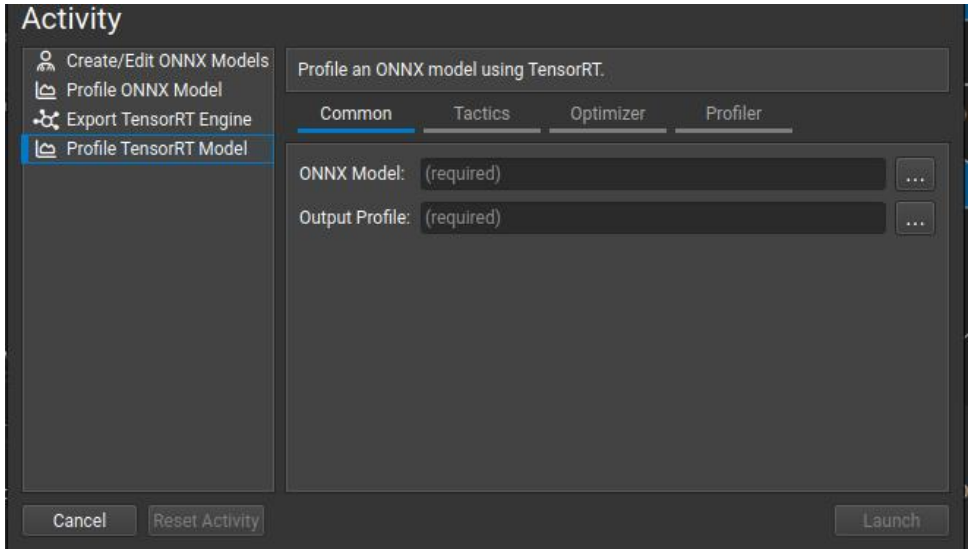
Profiling TensorRT

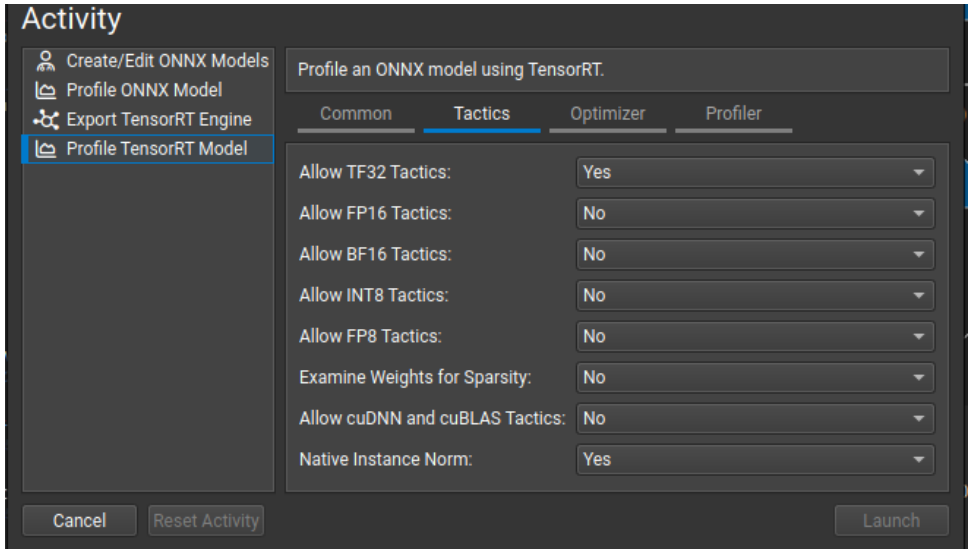
- Specialized tools allow to analyze full GPU execution timeline, far beyond the simple layer timing stats obtainable with profiling.
  - This is essential to debug and understand the computing behaviour of the optimized model.
  - Understanding memory bottlenecks, SM/Tensor Core utilization, Streams usage and more.

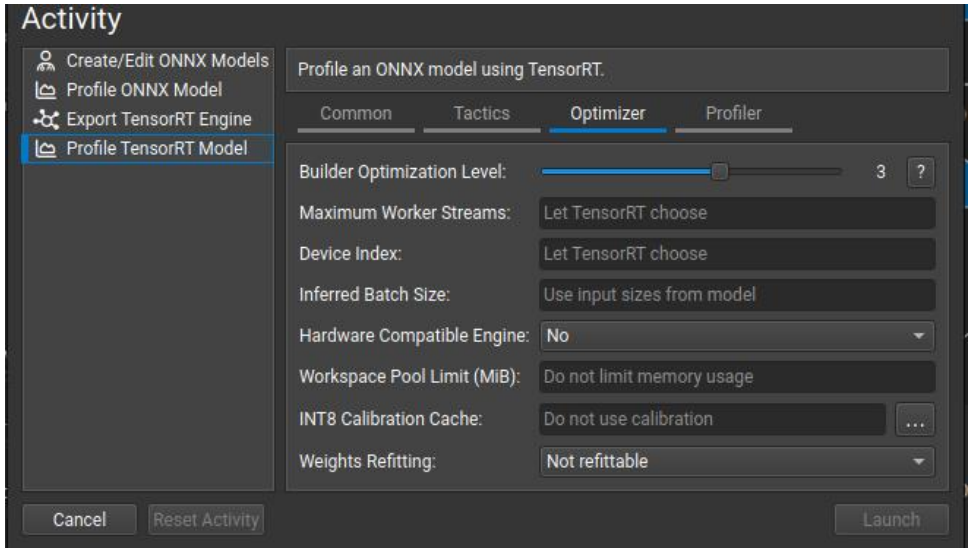
- **NVIDIA Nsight Deep Learning Designer** include ..
  - Automatically export onnx model to tensorrt (no need for pre built engine).
  - Benchmarking of local or ssh (e.g, Jetson) target devices.
  - Fine-grained control over the desired optimization parameters for engine building.

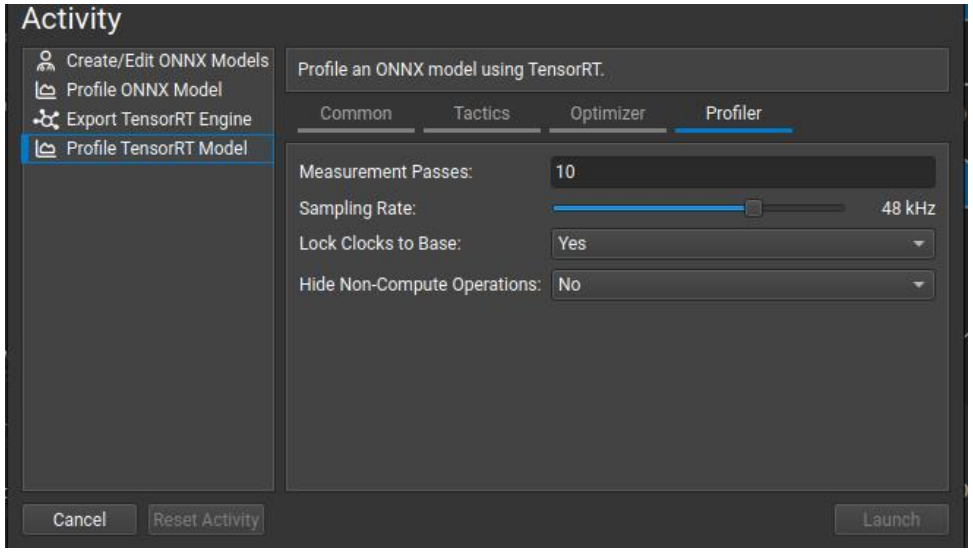
Copyright © 2025

Profiling TensorRT









Copyright © 2025

Profiling TensorRT

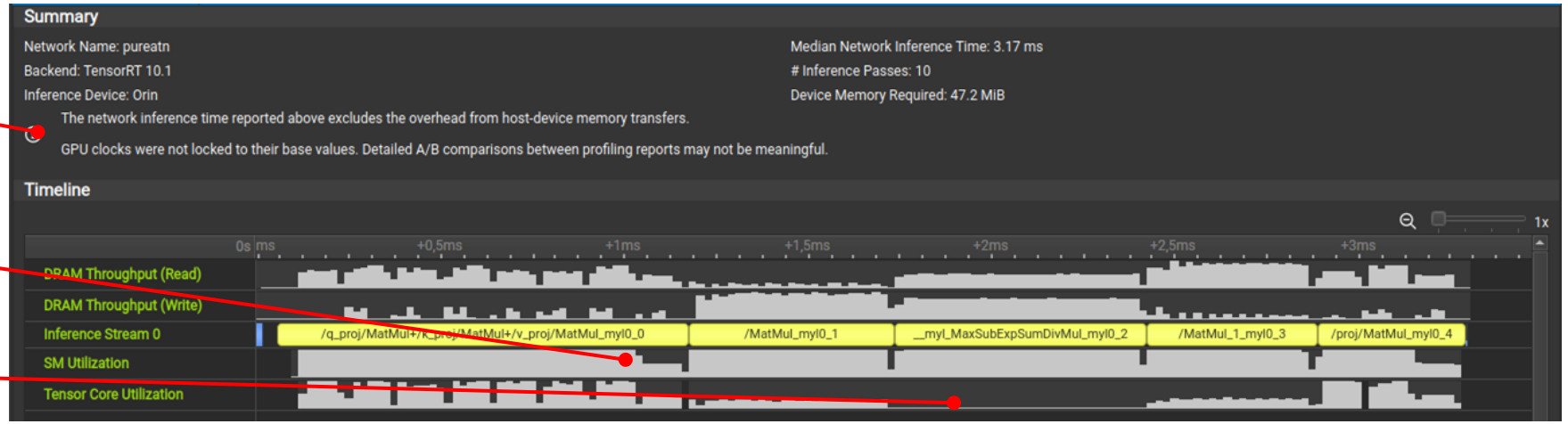
- **Example:** FP32 vs FP16 Multi-Head Self Attention

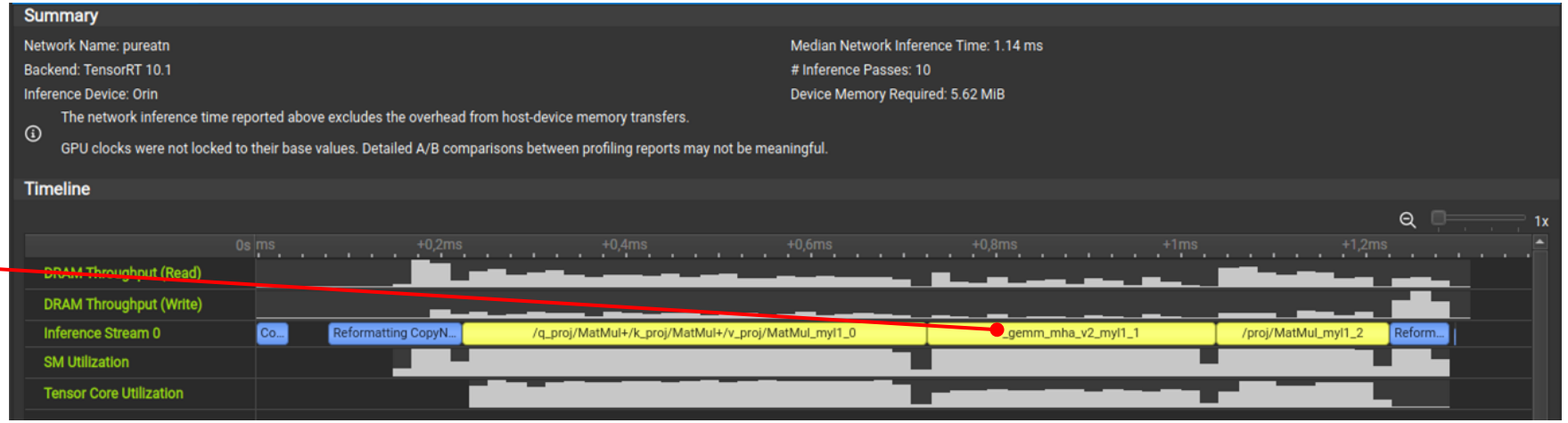
Possibly bug, GPU clocks were locked!

Fused Q,K,V Projections

Sofmax doesnt' benefit of Tensor Cores

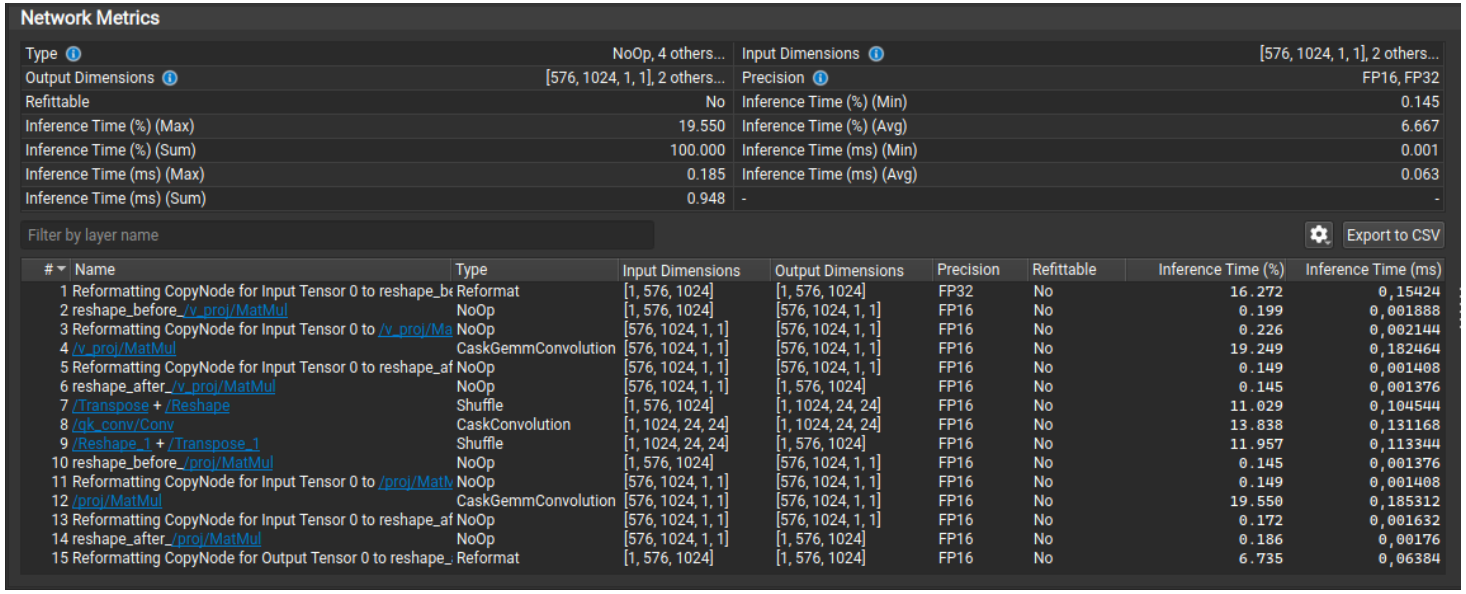
Silently swapped in Flash-Attention V2!





Copyright © 2025

Profiling TensorRT



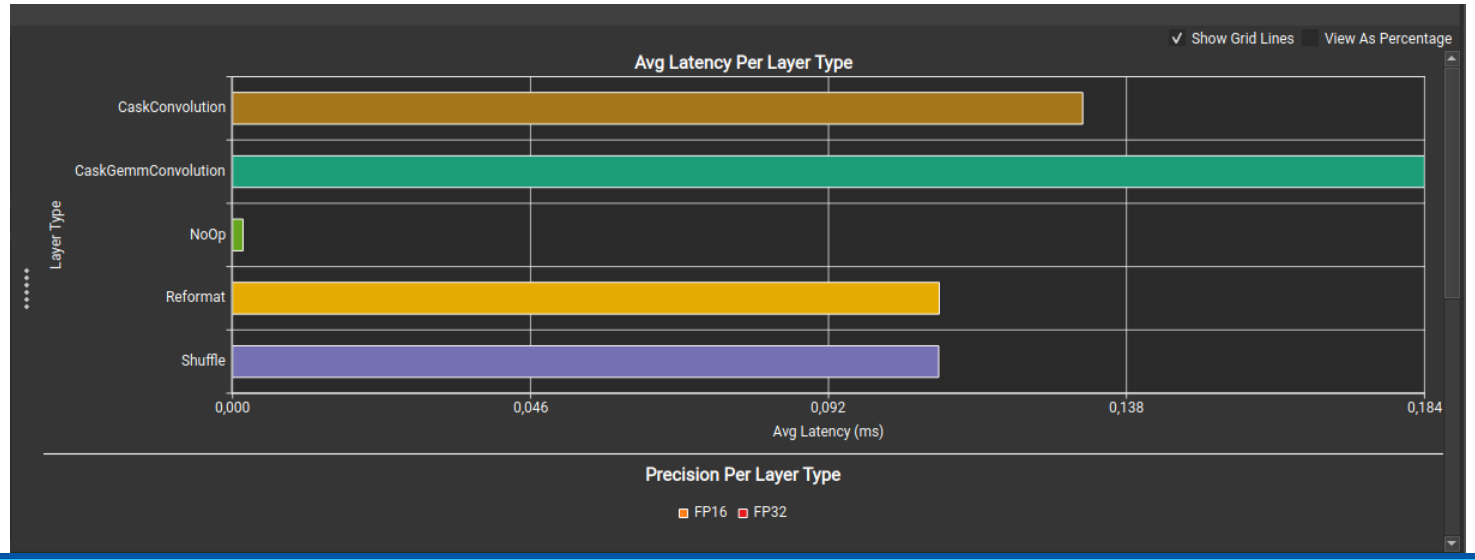
Network Metrics

Type	NoOp, 4 others...	Input Dimensions	[576, 1024, 1, 1], 2 others...
Output Dimensions	[576, 1024, 1, 1], 2 others...	Precision	FP16, FP32
Refittable	No	Inference Time (%) (Min)	0.145
Inference Time (%) (Max)	19.550	Inference Time (%) (Avg)	6.667
Inference Time (%) (Sum)	100.000	Inference Time (ms) (Min)	0.001
Inference Time (ms) (Max)	0.185	Inference Time (ms) (Avg)	0.063
Inference Time (ms) (Sum)	0.948	-	-

Filter by layer name

Export to CSV

#	Name	Type	Input Dimensions	Output Dimensions	Precision	Refittable	Inference Time (%)	Inference Time (ms)
1	Reformatting CopyNode for Input Tensor 0 to reshape_be	Reformat	[1, 576, 1024]	[1, 576, 1024]	FP32	No	16.272	0,15424
2	reshape_before_/v_proj/MatMul	NoOp	[1, 576, 1024]	[576, 1024, 1, 1]	FP16	No	0.199	0,001888
3	Reformatting CopyNode for Input Tensor 0 to /v_proj/Ma	NoOp	[576, 1024, 1, 1]	[576, 1024, 1, 1]	FP16	No	0.226	0,002144
4	/v_proj/MatMul	CaskGemmConvolution	[576, 1024, 1, 1]	[576, 1024, 1, 1]	FP16	No	19.249	0,182464
5	Reformatting CopyNode for Input Tensor 0 to reshape_af	NoOp	[576, 1024, 1, 1]	[576, 1024, 1, 1]	FP16	No	0.149	0,001408
6	reshape_after_/v_proj/MatMul	NoOp	[576, 1024, 1, 1]	[1, 576, 1024]	FP16	No	0.145	0,001376
7	/Transpose + /Reshape	Shuffle	[1, 576, 1024]	[1, 1024, 24, 24]	FP16	No	11.029	0,104544
8	/qk_conv/Conv	CaskConvolution	[1, 1024, 24, 24]	[1, 1024, 24, 24]	FP16	No	13.838	0,131168
9	/Reshape_1 + /Transpose_1	Shuffle	[1, 1024, 24, 24]	[1, 576, 1024]	FP16	No	11.957	0,113344
10	reshape_before_/proj/MatMul	NoOp	[1, 576, 1024]	[576, 1024, 1, 1]	FP16	No	0.145	0,001376
11	Reformatting CopyNode for Input Tensor 0 to /proj/Matl	NoOp	[576, 1024, 1, 1]	[576, 1024, 1, 1]	FP16	No	0.149	0,001408
12	/proj/MatMul	CaskGemmConvolution	[576, 1024, 1, 1]	[576, 1024, 1, 1]	FP16	No	19.550	0,185312
13	Reformatting CopyNode for Input Tensor 0 to reshape_af	NoOp	[576, 1024, 1, 1]	[576, 1024, 1, 1]	FP16	No	0.172	0,001632
14	reshape_after_/proj/MatMul	NoOp	[576, 1024, 1, 1]	[1, 576, 1024]	FP16	No	0.186	0,00176
15	Reformatting CopyNode for Output Tensor 0 to reshape_	Reformat	[1, 576, 1024]	[1, 576, 1024]	FP16	No	6.735	0,06384



- **Layer Table**
  - TensorRT layers executed by the network, type, dimensions, precision, and inference time.
  - raw time measurements and percentages of the inference pass
- **Network Graphs**
  - Display average inference latency for each type of layer in the TensorRT engine.

Copyright © 2025

Profiling TensorRT

- In some cases Nsight System can be preferred
  - DL Designer does not currently support loading custom plugins
  - Nsight Systems provides GPU/system-wide profiler (Kernel launched, memory transfer, stream usage, synchronization events…).
- Nsight Systems can capture profile information when launching inference with trtexec

```
$ trtexec --onnx=foo.onnx --profilingVerbosity=detailed --saveEngine=foo.plan
nsys profile -o foo_profile --capture-range cudaProfilerApi trtexec --profilingVerbosity=detailed --
loadEngine=foo.plan --warmUp=0 --duration=0 --iterations=50
```

Copyright © 2025

Part 3: Performance optimization

- **Introduction to performance optimization: modelopt toolkit.**
- **Basic techniques and common pitfalls.**

Note: we won’t cover specialized optimization techniques in details. This section is meant to understand how basic techniques apply in TensorRT

Copyright © 2025

Modelopt

NVIDIA TensorRT Model Optimizer (ModelOpt) is a collection of state-of-the-art model optimization techniques (quantization, sparsification, pruning, distillation).

- Modelopt manly work in Pytorch and ONNX, before TensorRT build phase.
- TensorRT is therefore agnostic to the usage of modelopt, any other toolkit can be used.
- Several Techniques are meant for optimizing LLMs.

Copyright © 2025

Modelopt: Optimization

Quantization

- ModelOpt implement several algorithms for Post-training quantization (PTQ) and Quantization-aware training (QAT).

Sparsification

- Structured N:M weights sparsification can reduce memory footprint and accelerate inference on recent architectures (>= Ampere) to skip zero-multiplication with structured sparsity.

Pruning

- ModelOpt provides three main pruning methods, that aims to find a subnet that satisfy specific performance constraints. Mostly aimed at LLMs, we won’t cover pruning in this lectures.

Copyright © 2025

Post-Training Quantization Using Calibration

PTQ is the easiest way to reduce computation by 2x-4x on TensorRT.

- **`mtq.quantize`** API requires the model, the appropriate quantization configuration, and a forward loop as inputs.

```
import modelopt.torch.quantization as mtq

# Load data
_, val_data = setup_datasets()

# Create quantization dataloader
indices = torch.randperm(len(val_data))[:512 - 1].tolist()
val_set = Subset(val_data, indices)
val_loader = DataLoader(val_set, batch_size=1,
collate_fn=custom_collate_fn)

# Select quantization config
quant_config = copy.deepcopy(mtq.INT8_SMOOTHQUANT_CFG)

# Define forward_loop.
def forward_loop(model):
for batch in tqdm(coco_quant_loader):
    # copy to device
    batch = batch.to('cuda')
    model(batch.image)

# Quantize the model and perform calibration (PTQ)
model = mtq.quantize(model, quant_config, forward_loop)
```

- mtq.quantize() inserts TensorQuantizer (quantizer modules) into the model layers and patches their forward method to perform quantization.
- The quantization algorithm is used to calibrate weights and activation scales.
- TensorQuantizer will automatically export as Q/DQ ONNX layers.

Copyright © 2025

Post-Training Quantization Using Calibration

A quantization method comprises three primary components: (1) Weight precision format, (2) Activation precision format, (3) Calibration algorithms.

- For vision models, the following options are recommended.

Quantization Methods	Performance small-batch	Performance large-batch	Accuracy degradation	Details
FP8	Medium	Medium	Very Low	• FP8 per-tensor weight & activation quantization with min-max calibration. • Compresses FP16/BF16 model to 50% of original size. • Calibration time: minutes**. • Deploy via TensorRT, TensorRT-LLM. Supported GPUs: Ada, Hopper and later.
INT8 SmoothQuant	Medium	Medium	Medium	• 8-bit integer quantization with a variant of SmoothQuant calibration. • Per-channel weight quantization, per-tensor activation quantization. • Compresses FP16/BF16 model to 50% of original size • Calibration time: minutes**. • Deploy using TensorRT, TensorRT-LLM. Supported on most GPUs.

More: https://nvidia.github.io/TensorRT-Model-Optimizer/guides/_pytorch_quantization.html

Copyright © 2025

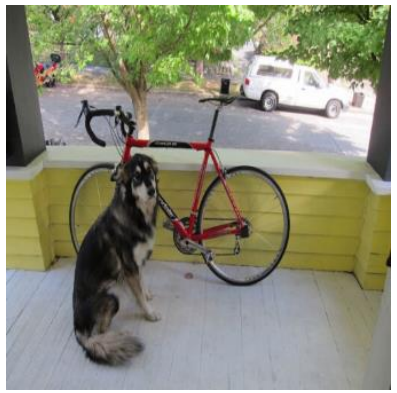
Post-Training Quantization Using Calibration

PTQ is not a magic wand, common pitfalls:

- Quantizing specific subgraphs.
- Incorrect or insufficient calibration dataset.
- Ignoring dynamic input shapes.

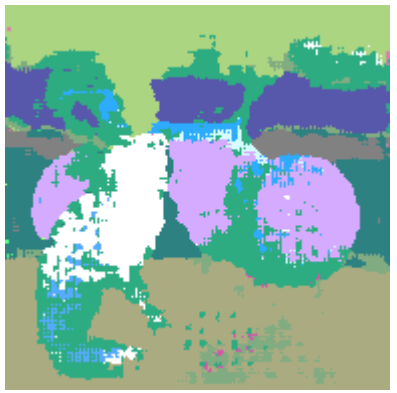
Example: segmentation decoder blocks are sensitive to quantization

```
# Disable quantization of decoder to preserve accuracy
quant_config = copy.deepcopy(mtq.INT8_SMOOTHQUANT_CFG)
quant_config["quant_cfg"]["*decoder*"] = {"enable": False}
quant_config["quant_cfg"]["*coco_sem_seg*"] = {"enable": False}
```



Source

Result: FP32



Result: INT8 - ALL

Result: INT8 - no decoder

Copyright © 2025

Quantization Aware Finetuning (PTQ)

QAT is the technique of fine-tuning a quantized model to recover model quality degradation due to quantization.

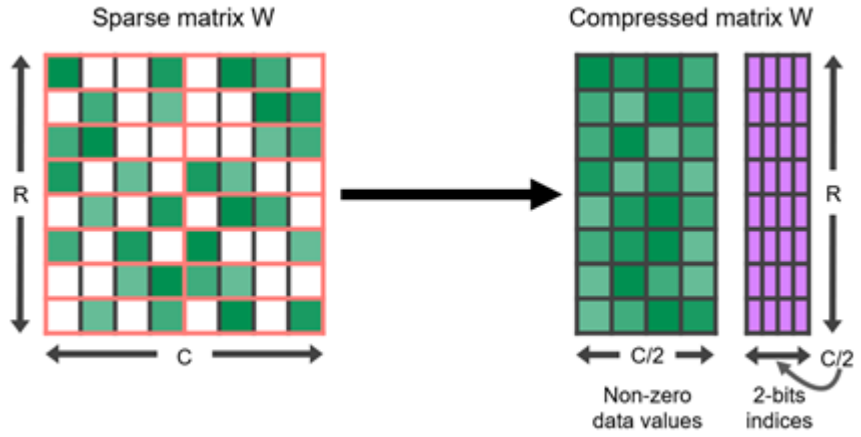
- After PTQ, TensorQuantizer weights are frozen and model weights are finetuned with the appropriate training loop.
- 10% of the original training epochs should be enough for vision models.
- **Note:** In pytorch INT8 precision is simulated, no speedup should be visible in training.

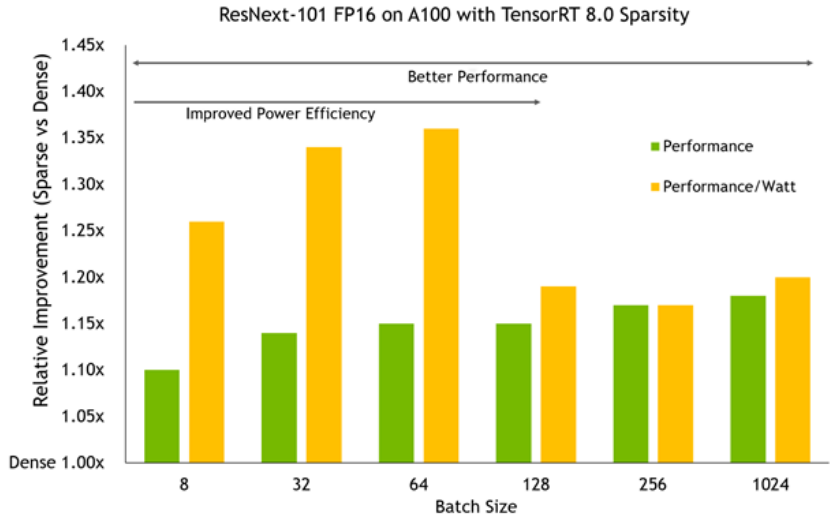
Copyright © 2025

Sparsification

Weight sparsity is a broad concept in model optimization.

- Unstructured sparsty: zero weights are randomly distributed.
- **N:M** sparsity: structured sparsity pattern where in each block of M contiguous elements, at most N are nonzeros. Ampere and later architecture support **2:4** sparsity for matmul operations. Sparse weights save memory and computation.
- Sparsification Algorithm: Magnitude-based sparsity simply zero the smallest 2 weights in a block of 4 contiguous weights. Surprisingly, this work.





Copyright © 2025

Sparsification

- Modelopt sparsification is based on Nvidia Automatic Sparsity (**ASP**) (https://github.com/NVIDIA/apex/tree/master/apex/contrib/sparsity) .
  - ASP does not require calibration data.

Example:

```
import modelopt.torch.sparsity as mts

# Configure and convert for sparsity
sparsity_config = {
        "collect_func": lambda x: x,
}

sparse_model = mts.sparsify(
        model,
        "sparse_magnitude",
        config=sparsity_config,
)
```

- Post-Training Sparsification can lead to severe degradation.
- Fine-tuning is likely required, sparsity mask fixed during finetuing stage.

Copyright © 2025

Conclusion

- The NVIDIA Jetson and TensorRT ecosystem has matured into one of the most capable, performance-oriented toolchains for edge AI deployment, balancing raw performance, deployment flexibility, and profiling depth.

- While the platform remains largely closed-source and proprietary in key areas, it provides transparency, tooling, and low-level access (e.g., TensorRT tactics, Nsight profiling, custom plugin integration) that are typically hidden or inaccessible in other edge AI ecosystems.

- This combination of high-performance inference, insightful profiling, and customization capability makes NVIDIA's stack a solid choice for advanced edge AI workloads, especially for computer vision models.

Copyright © 2025

Thank You

carmelo.scribano@unimore.it

EAI

https://edge-ai-tech.eu/

The two-day innovative educational initiative is designed for students, young researchers, and professionals eager to expand their understanding and skills in these cutting-edge technologies.

Participants will engage in interactive discussions, hands-on sessions, and lectures covering foundational concepts and innovative applications of edge AI across various domains with leading experts, gain insights into real-world applications, and work on team-based projects to foster collaboration and innovation.

Copyright © 2025

EDGE AI Academy 2025

Summer School Objectives

- The EDGE AI Academy Summer School aims to empower participants with advanced knowledge, hands-on experience and an in-depth understanding of edge AI, which combines Artificial Intelligence (AI) with edge computing and the Internet of Things (IoT) to process data closer to its source.
- Edge AI technology is gaining prominence due to its ability to enhance real-time decision-making, reduce latency, improve data privacy, and diminish reliance on continuous internet connectivity.
- The topics will extend to RISC-V architectures as ultra-low-power high-performance RISC-V-based processors and ultra-low-power vector-based accelerators are advancing smart IoT and edge AI computing capabilities.

Copyright © 2025