Introduction to Tenstorrent

Dalar Vartanians
Principal Field Application Engineer, Machine Learning

Customer Success Team

May 27, 2026

tenstorrent

About Me

- MS, Electrical and Computer Engineering from UCSD
- Machine Learning
- Customer Engineering
- Tesla, Xpeng, **Tenstorrent** ☺

- Epic of the Kings Enthusiast
- Ideas Resonate with modern worlds of engineering!

“Engineering is essentially the act of turning chaos into structure!”

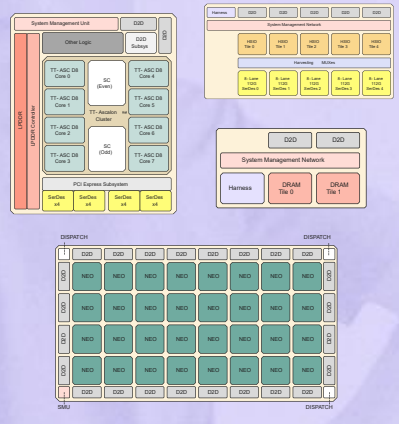
tenstorrent

Tenstorrent Product Summary

IP (TT-Ascalon™ / Tensix NEO)

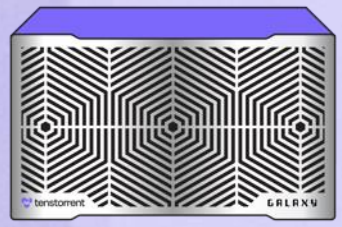
- Scales from mW to MW for efficiency and performance
- IP available for licensing
- Industry-leading performance
- Modular design available in varied configurations

Chips & Chiplets



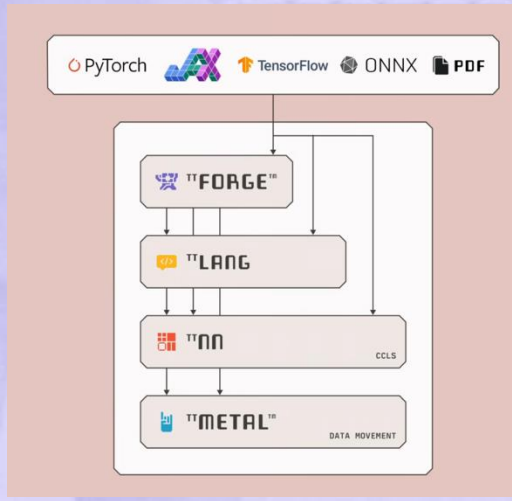
- Portfolio of products powered by scalable Tensix AI cores
- Inference and training, CNN and NLP, recommendation engines, all on the same silicon
- Hardware available for purchase as well as IP available for licensing
- Multi-component modular chiplets

Servers (Tenstorrent Galaxy™)



- Galaxy Server – 32 high performance ASICs in a custom chassis
- Easily combine servers into a Galaxy Rack with high bandwidth chip-to-chip connectivity

Software



- Fully open source software stack with multiple entry points
- TT-Forge™ - MLIR-based compiler (Q2 2025)
- TT-NN™ – Library of optimized operators
- TT-Metalium™ - Bare metal, low-level programming model

Evolution of Tenstorrent AI Chips

2023 (EOL) → **2024** → **Now**

Grayskull®

High Performance AI ASIC

e150

e75

- First generation Tensix Processor
- Up to 120 Tensix Cores
- PCIe Gen 4
- 8 GB 256-bit LPDDR4

Wormhole™

Scalability

n300
Dual Chip

n150
Single Chip

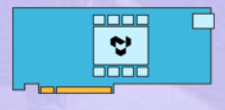
- Tensix Cores updated: 50% more SRAM per Tensix Core™, improved BLOCKFP8 performance, expanded precision format support
- Moved to 12 GB 192-bit GDDR6
- 100 Gbps connectivity
- Inter-card and inter-system expansion

Blackhole™

RISC-V & AI Generation

p300
Dual Chip

p150
Single Chip



p100
Single Chip
No Scale-Out

- Moved to 6nm manufacturing
- Updated NoC on Tensix Cores and added 16 x280 RISC-V cores
- Increased to 32 GB 256-bit GDDR6 at faster speed
- Moved to PCIe Gen 5.0
- Upgraded to 400 Gbps connectivity

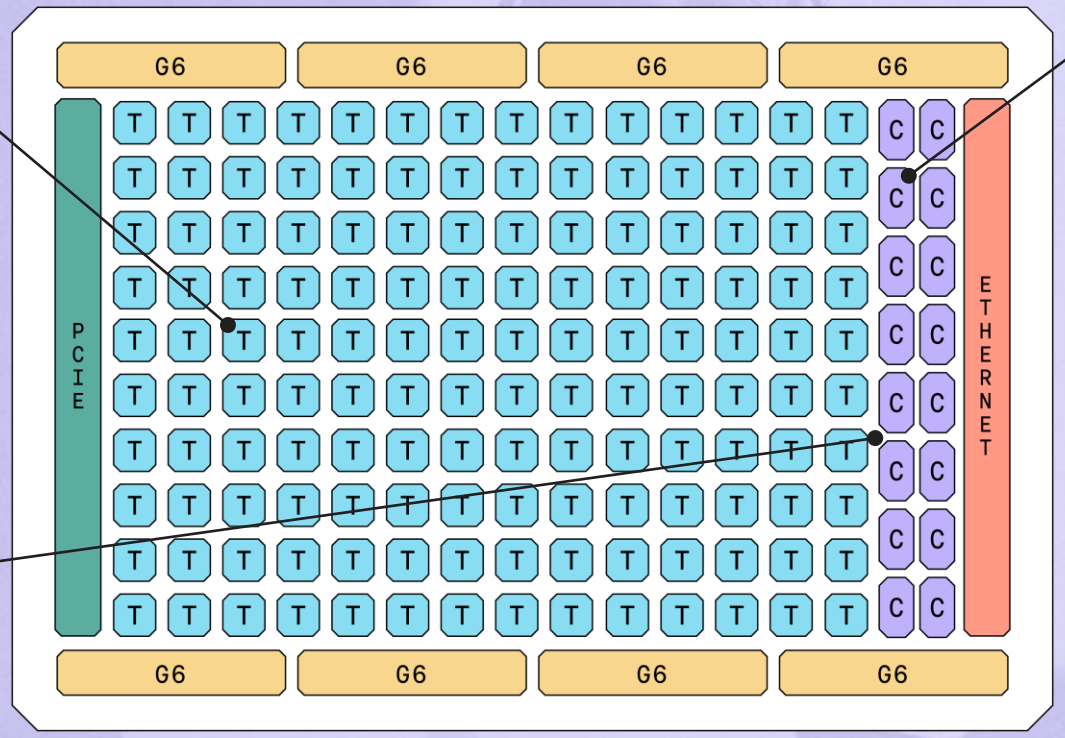
Why AI Needs Both RISC-V Cores and AI Accelerators

Tensix Cores are ideal for big math operations:

- Vector calculations
- Matrix arithmetic
- Large data sets

Merging Tensix Cores and CPU cores on the same die:

- Lowers latency
- Boosts utilization
- Increases ML performance



CPU cores are ideal for:

- Conditionality
- Traditional math
- High performance
- Robust programmability

ML Developers need both CPU and AI cores to build dynamic models of the future that are not possible today due to latency and utilization problems of using the host CPU.

Tenstorrent Galaxy Evolution

2023

Wormhole™

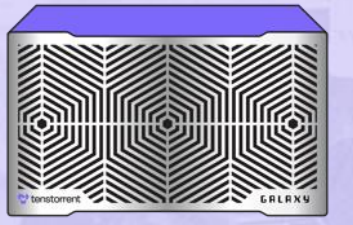
Prototype

- 32 Wormhole™ cards in a highly optimized, highly dense, custom 4U chassis
- 5.2 PetaFLOPS at BLOCKFP8
- 384 GB of globally accessible GDDR6 memory
- 3.8 GB SRAM
- Expansion beyond a single server accomplished via standard network cabling

2025

Wormhole™

ODM Redesign

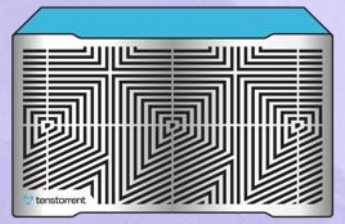


- 32 Wormhole™ cards in a highly optimized, highly dense, custom 6U chassis
- Internal head node for built-in host capability
- Increased PetaFLOPS and GDDR6 memory bandwidth
- UBB module for flexible customer infrastructure
- Expansion beyond a single server accomplished via standard network cabling

2025

Blackhole™

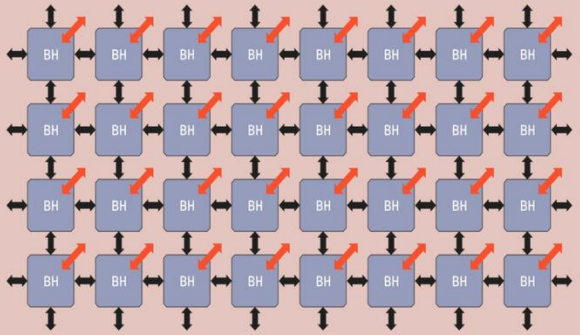
RISC-V & AI Generation



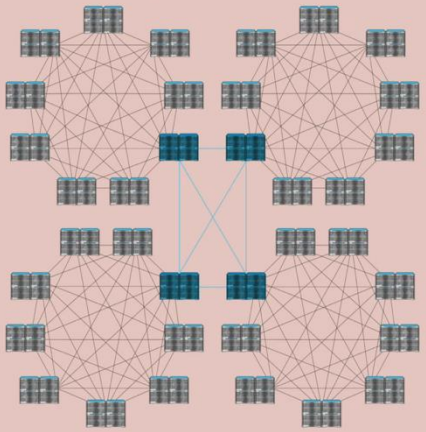
- Utilizes 6U Wormhole UBB design for easy customer transitions
- Replaces WH chips with BH chips

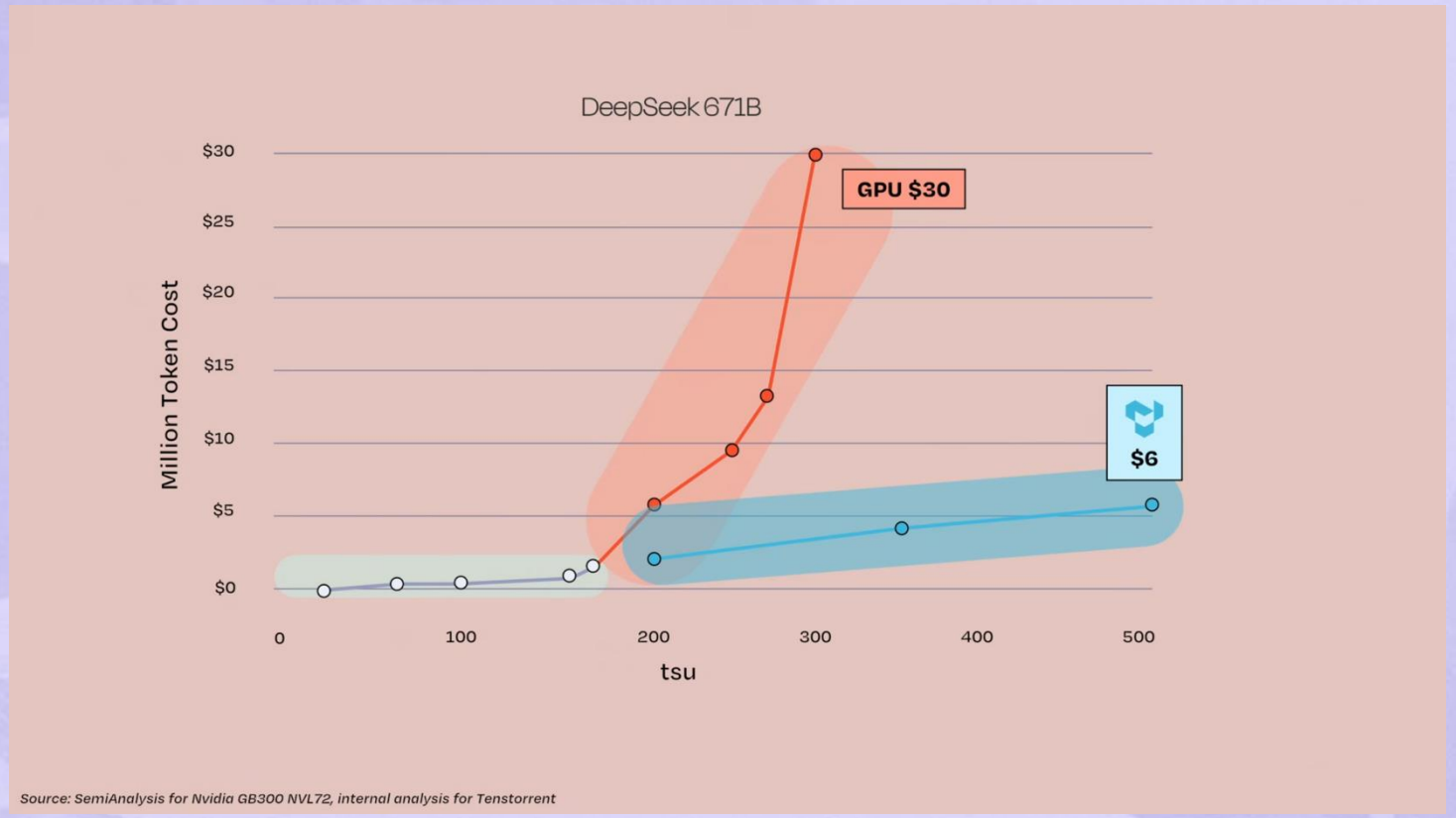
How We Scale with Galaxies

Tenstorrent Galaxy



Tenstorrent Galaxy supercluster 144





Source: SemiAnalysis for Nvidia GB300 NVL72, internal analysis for Tenstorrent

The only high-performance open-source AI Software

- **TT-Forge** – MLIR-based compiler integrated into various frameworks; AI/ML models from domain-specific compilers to custom kernel generation

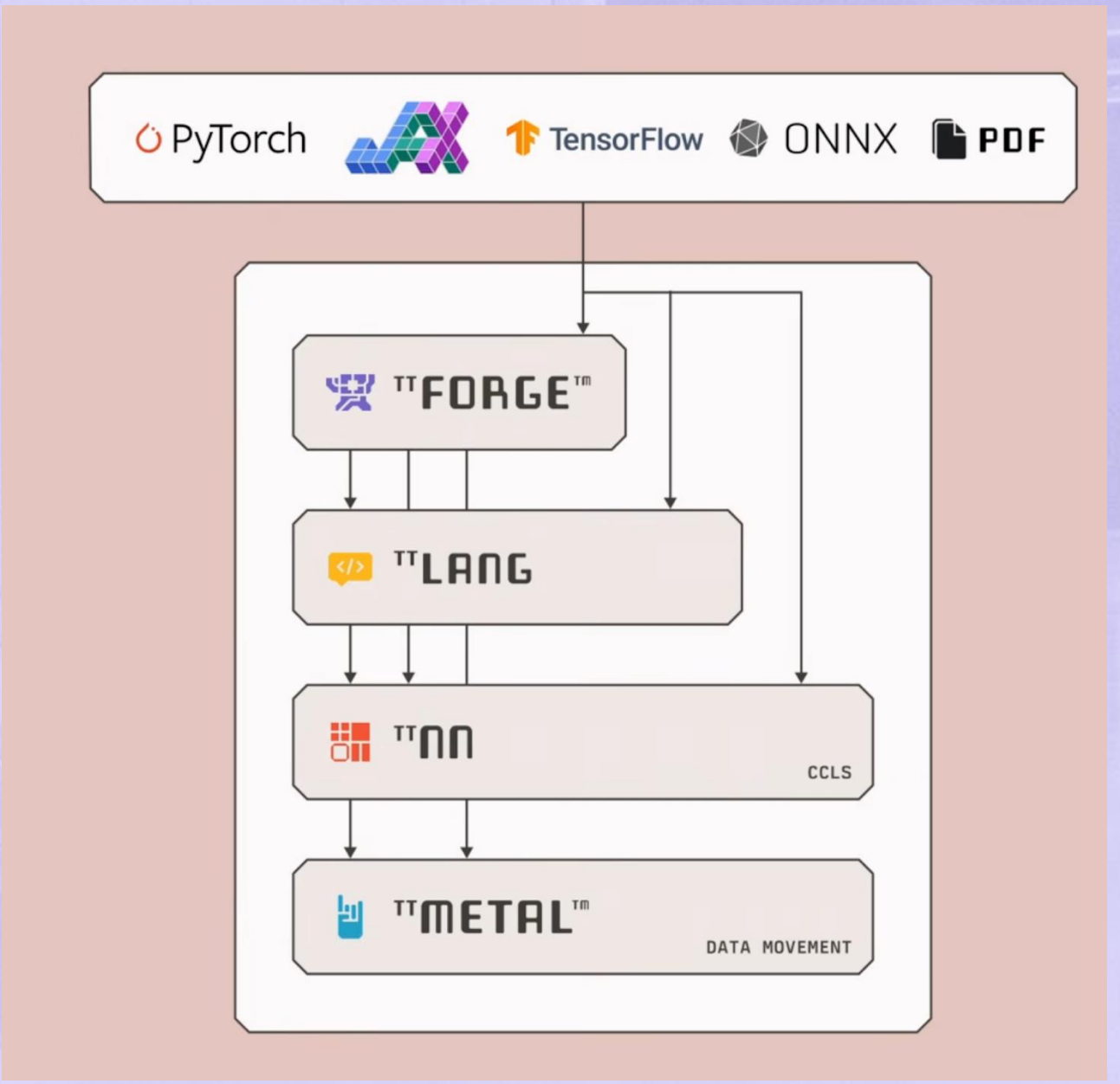
- **TT-Lang** – Python-based DSL

- **TT-NN™** – Library of optimized operators
  - ATen coverage
  - PyTorch-like API

- **TT-Metalium™** – Low-level programming model and entry point
  - Build your own kernels
  - User-facing host API

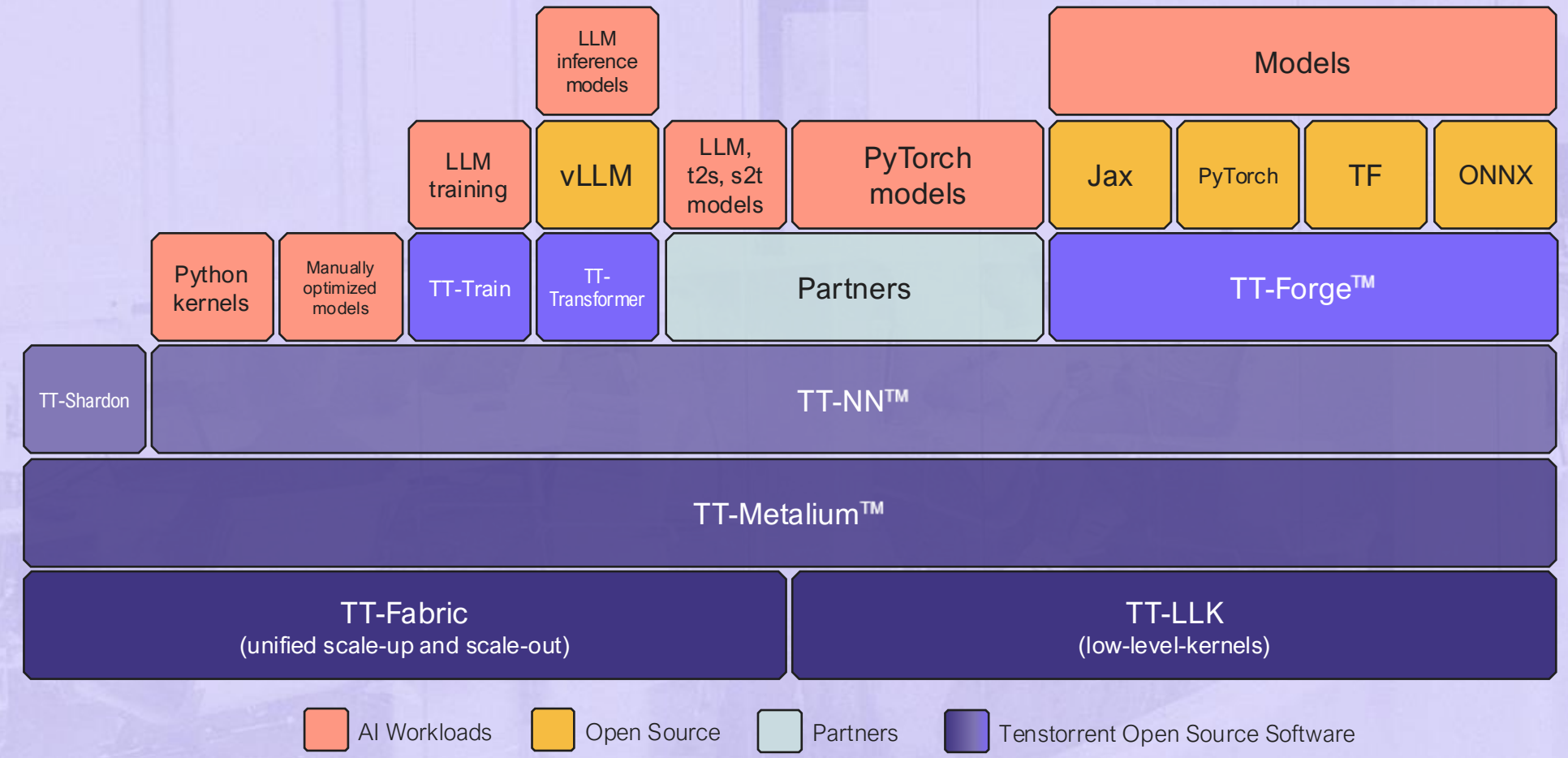
Try it today.

Tenstorrent GitHub



Software Ecosystem and Integrations

General: https://github.com/tenstorrent
TT-Metalium™: https://github.com/tenstorrent/tt-metal
TT-Forge™: https://github.com/tenstorrent/tt-forge

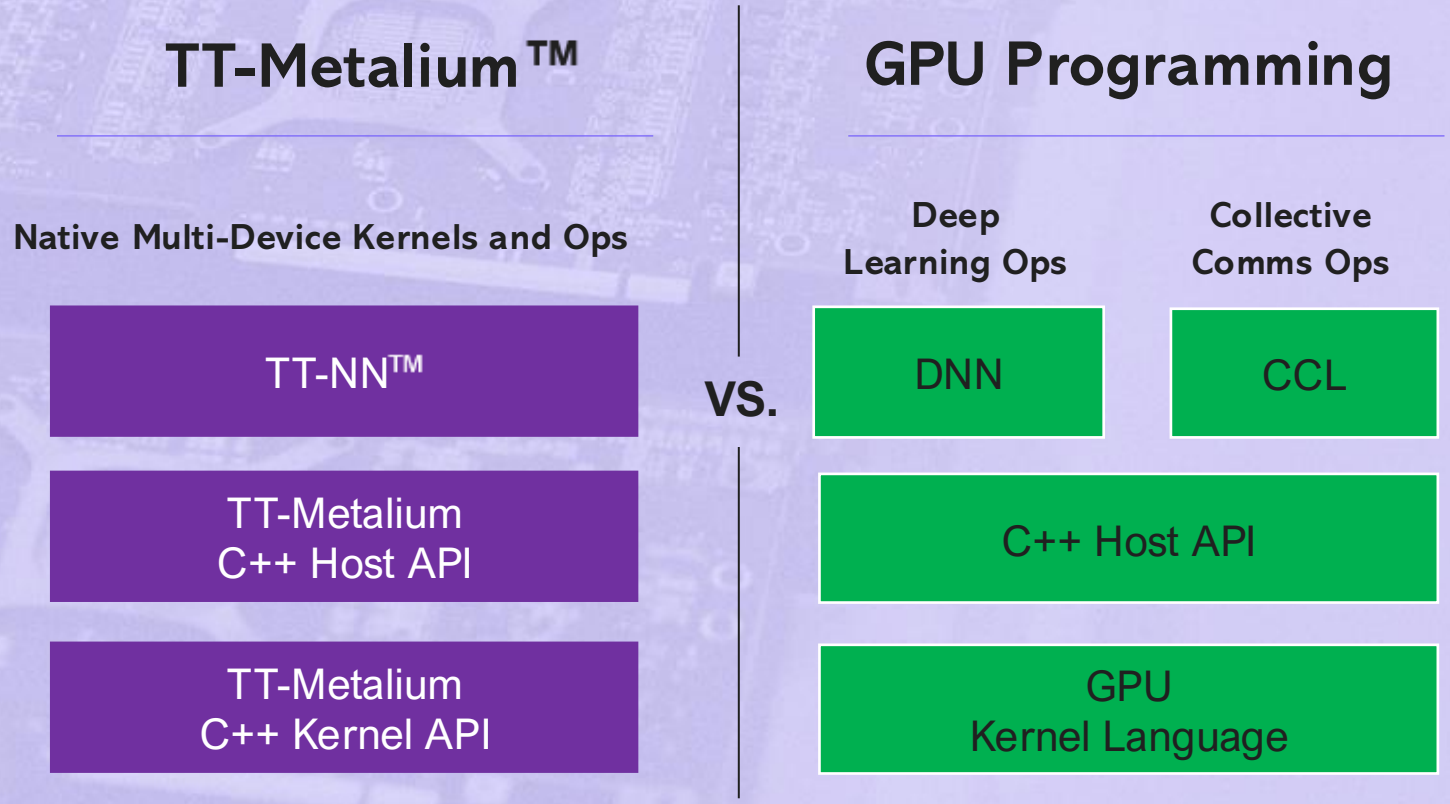


Third-Party Developers

- In-Game AI
- Datacenter AI Visualization
- AI Training
- ML Compiler
- ML Framework

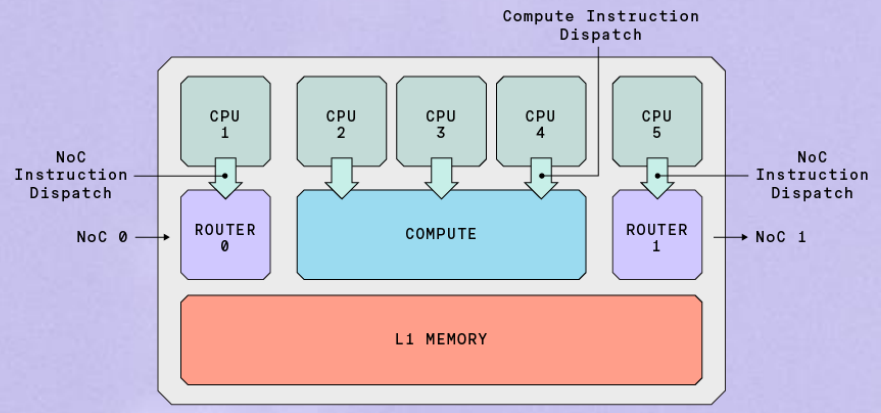
TT-Metalium™: Built for AI and Scale-Out

- **Kernels are plain C++ with APIs**
- **Dedicated data movement and compute kernels**
  - Optimize data movement and compute overlap directly
- **Any core can read/write/sync to any core or chip directly**
- **Full control of data layout and persistency in SRAM and DRAM**
- **Different cores can run different kernels and flow data directly between them**
- **Native multi-device kernels**
  - Fused and overlapped compute and inter-chip communication within the kernels

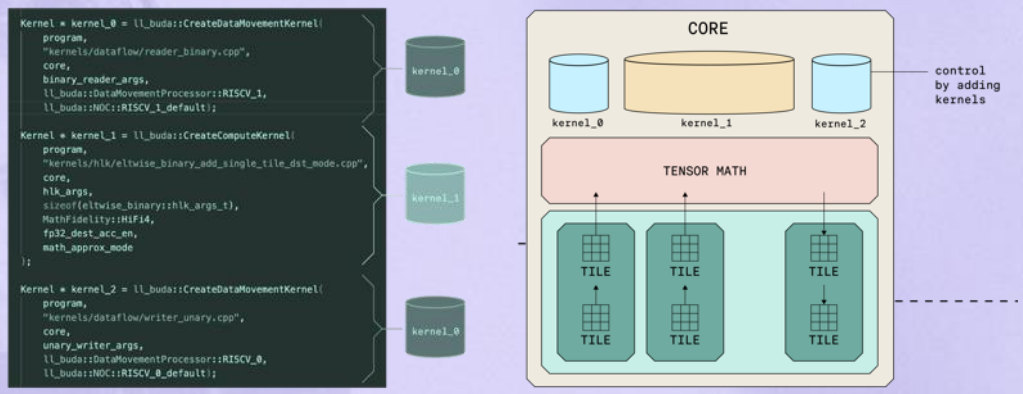


TT-Metalium™: Tensix Core to Multi-Chip Scale-Out

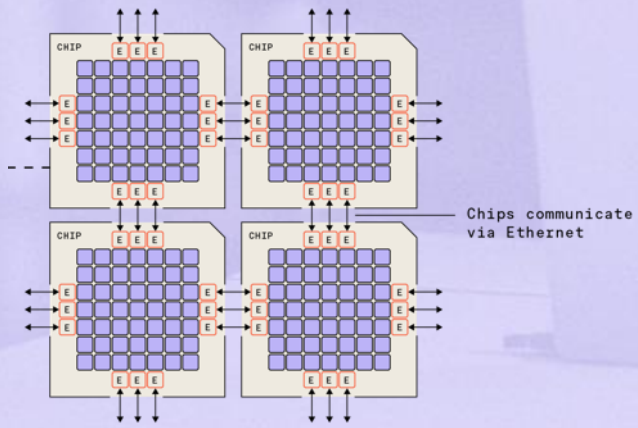
Tensix Core (Direct Access)



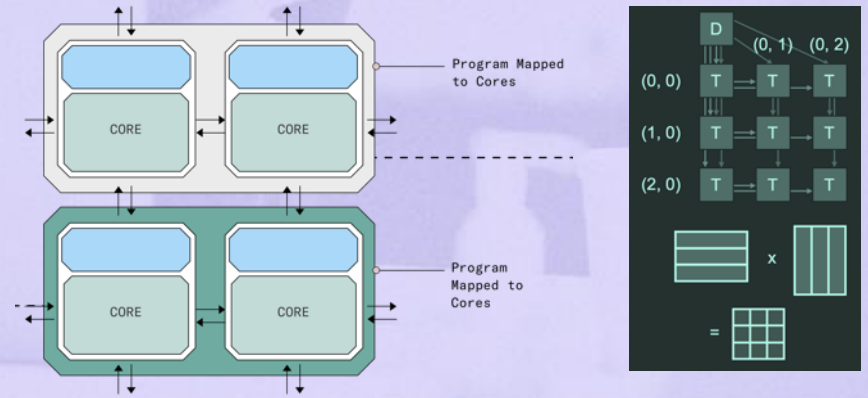
Compute & Data Movement Kernels (Decoupled)



Multi-Chip Scale-Out (Sea of Cores)



SIMD/MIMD & Multi-Cast (Across Chip)



AI model Bring-up

Model Bring-up Entry Points

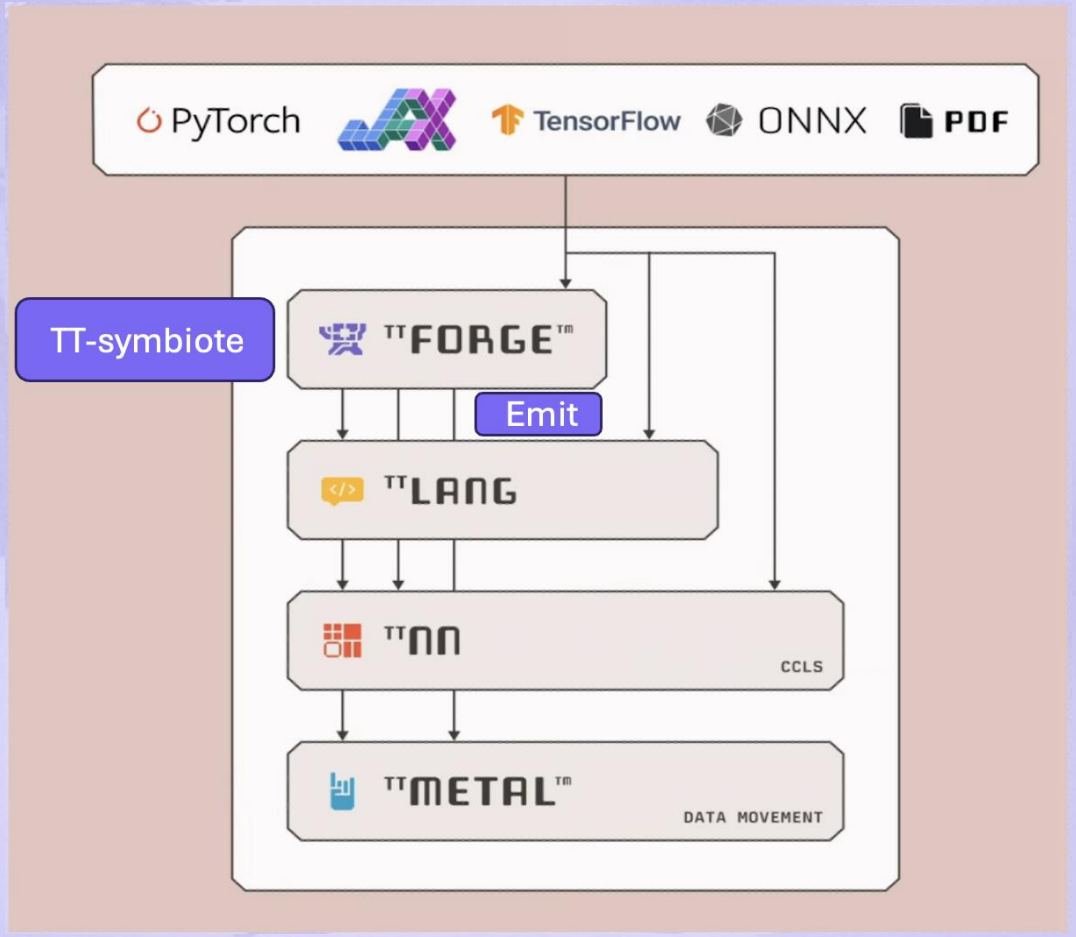
- **TT-Forge**: Quickest path to run new models on TT HW.
  - **Emit**: Internal compiler pipeline translating model graphs into hardware-optimized C++ code.

- **TT-Symbiote**: Automation added to modular bring-up
  - No TTNN code generated

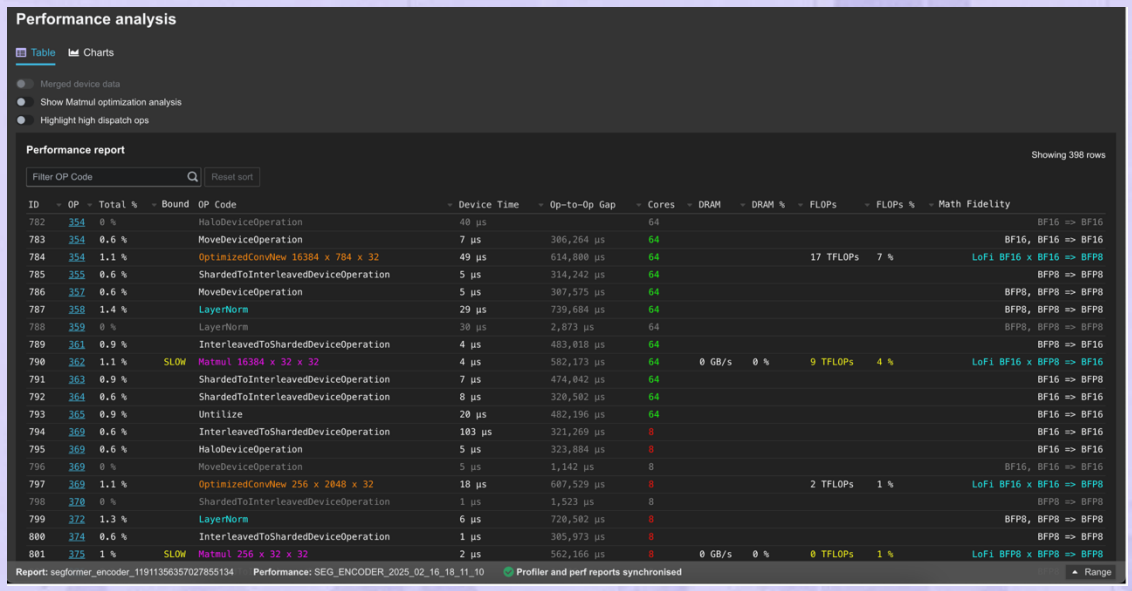
- **TTNN**: Direct implementation and optimizations via python APIs similar to Pytorch.
  - Manual
  - Agentic

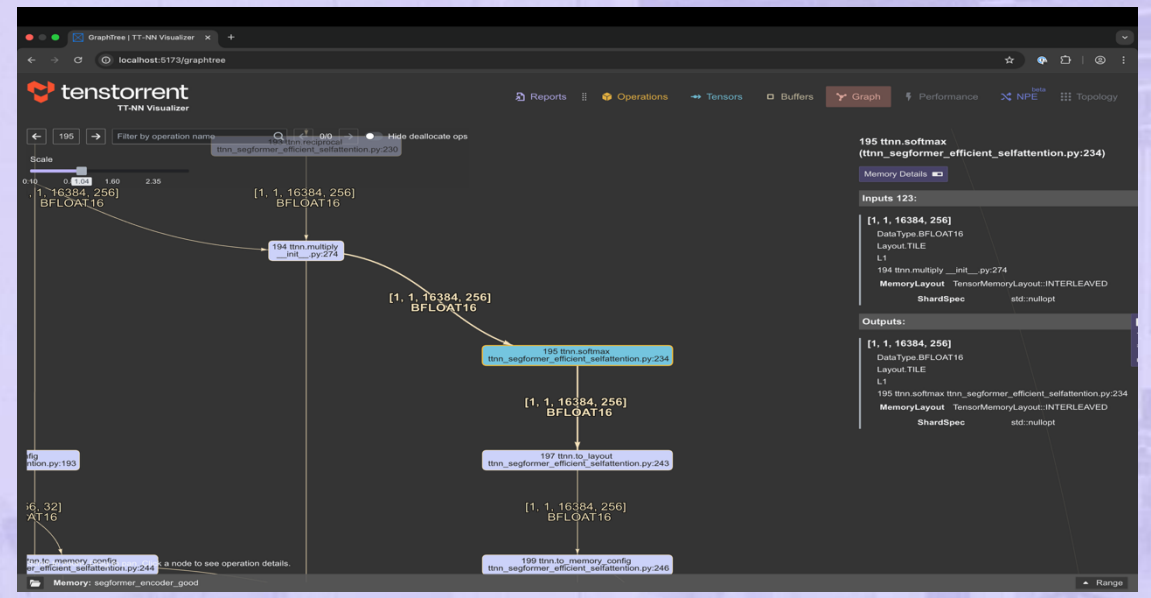
Writing/Optimizing Kernels

- **TT-Lang**: Implement potentially missing kernels using python DSL
- **TT-Metal**: C++ kernels



Performance Profiling Tools





Tracy Profiler

- **Custom Tracy Profiler Fork** – Adapted specifically for Tenstorrent hardware architecture.
- **Full-Stack Execution Tracing** – Visualizes execution flows across host and device simultaneously.
- **Host System Telemetry** – Captures performance data from x86 CPUs and Python code.
- **Tensix Processor Tracking** – Monitors on-device compute cores and data movement.

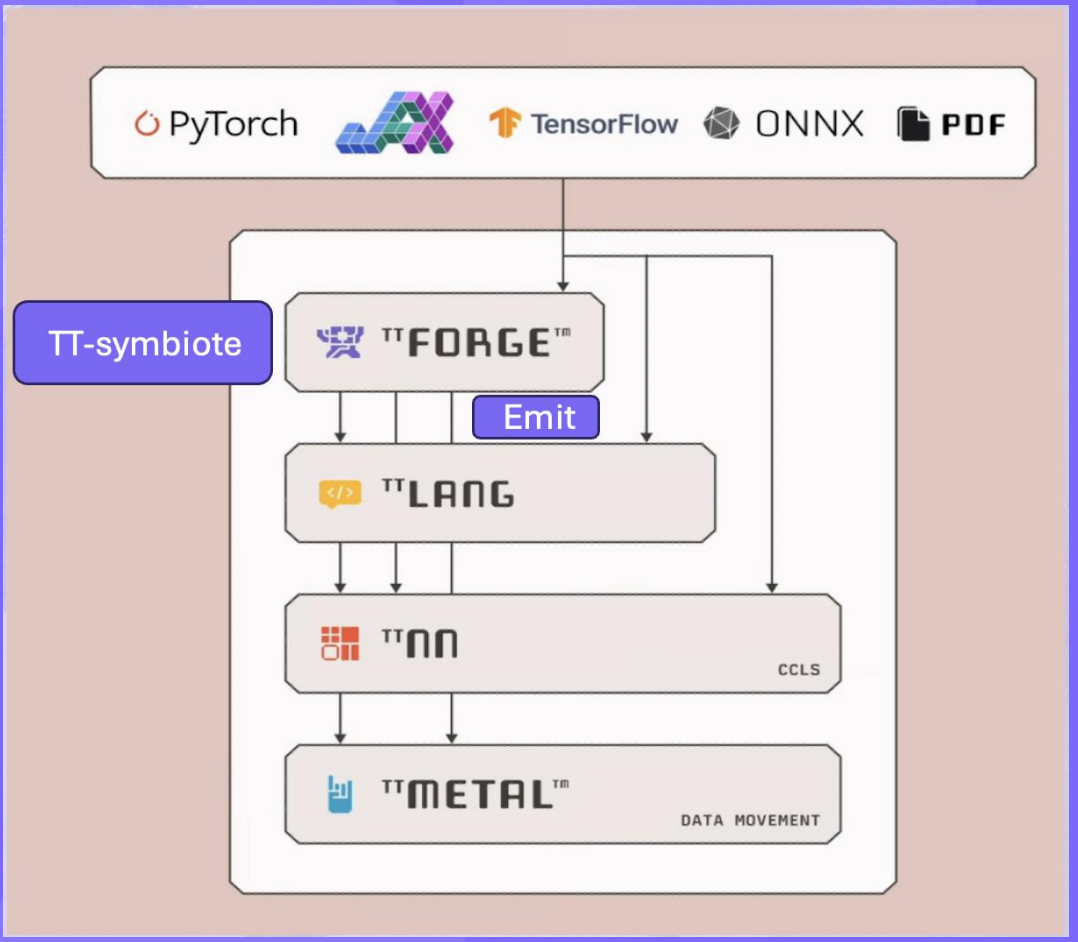
TTNN-Visualizer

- **Graph & Flow Diagrams** – Maps the full operation graph to trace layer dependencies.
- **Memory & Buffer Plots** – Tracks L1/DRAM allocation, tensor layouts, and data movement.
- **Performance Analysis** – Reads profiling telemetry to catch hardware timing imbalances or stalls.
- **Multi-Instance Support** – Loads reports locally or syncs remotely via SSH.


Thank You

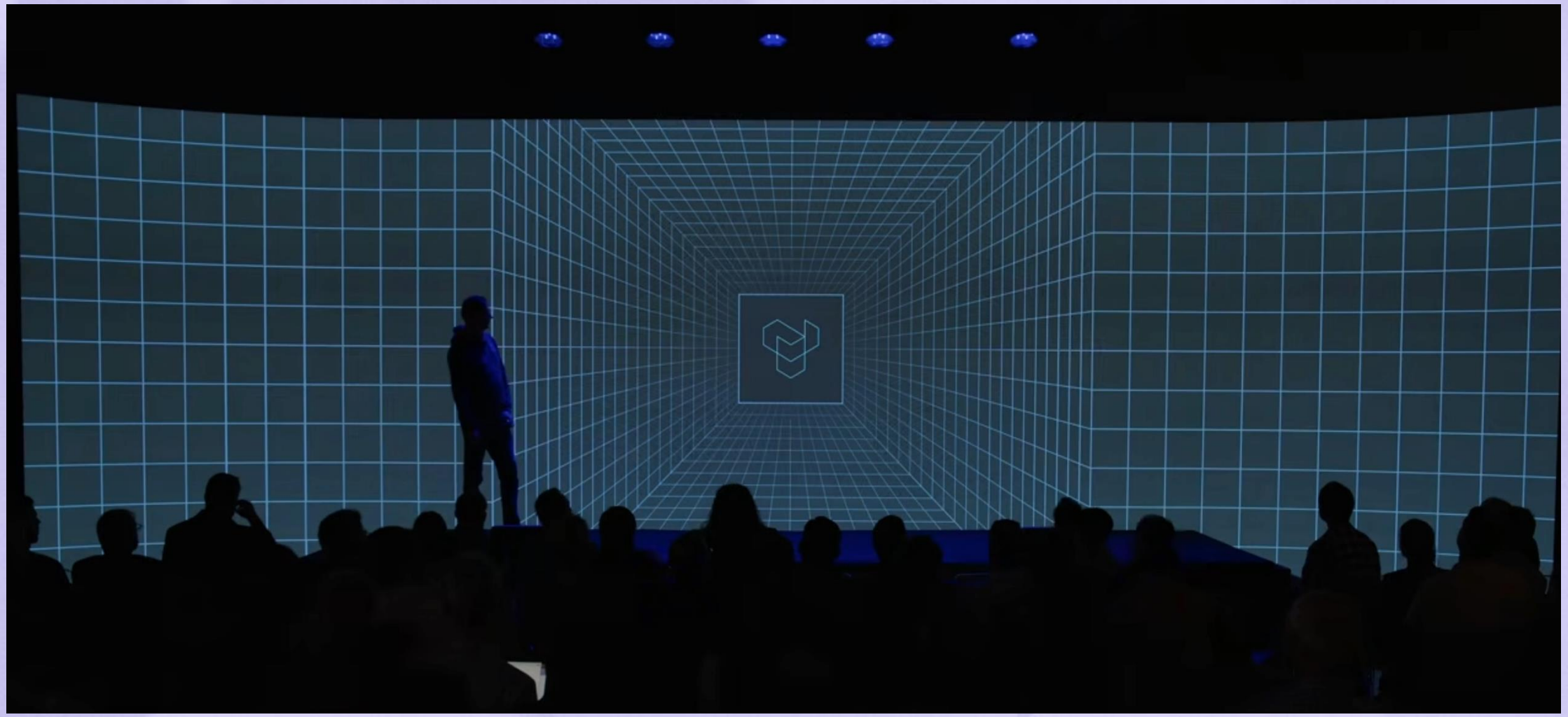
TT-Forge

Run Anything!



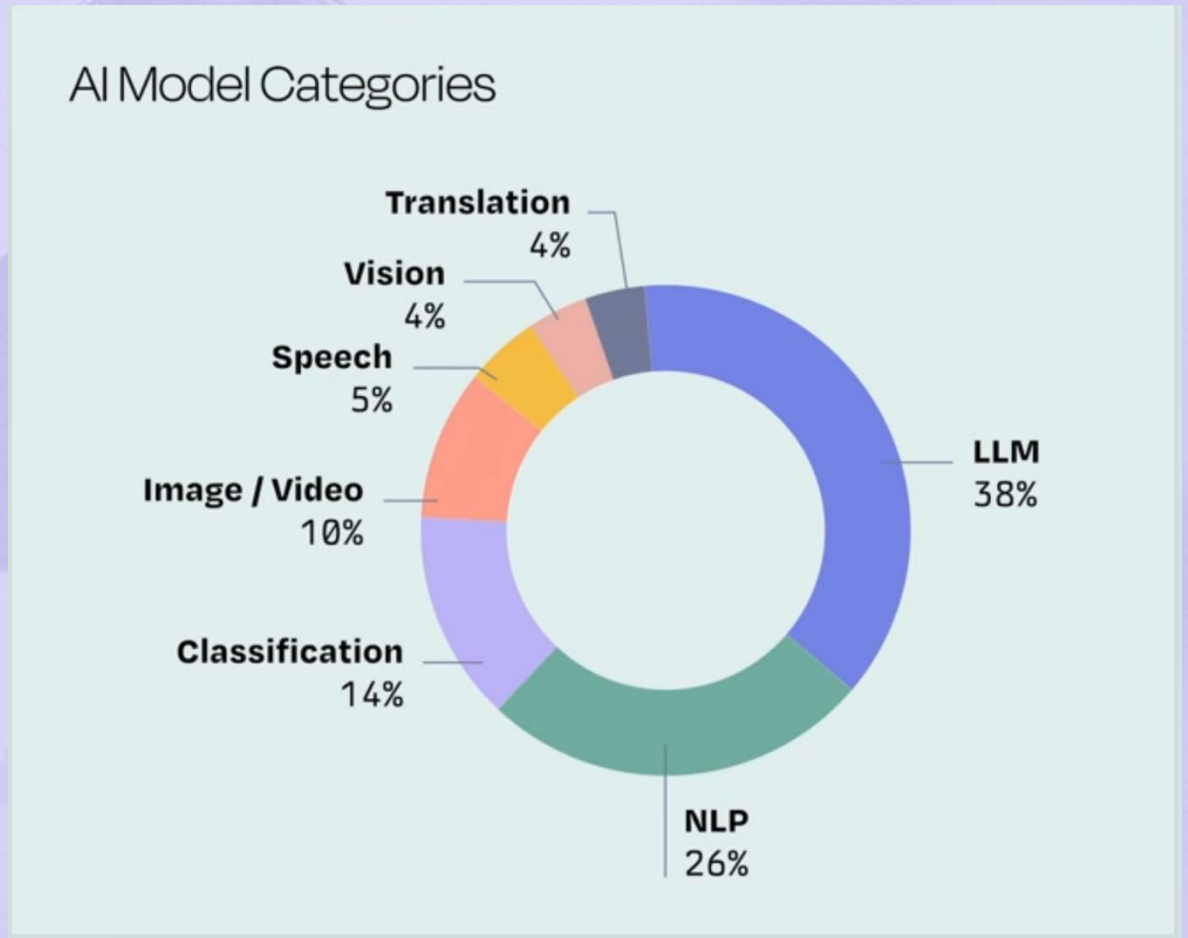
tenstorrent

TT-Deploy Day!



Generality Test

Thousands of Models from Hugging Face Running on TT-Quiet Box
90% Success on Random models



TT-Forge Sub-projects

Project	What It Does
TT-XLA	Primary frontend for **PyTorch** and **JAX** models. Uses the PJRT interface to compile models into StableHLO graphs for TT-MLIR. Supports single and multi-chip.
TT-Forge-ONNX	TVM-based frontend for **ONNX**, **TensorFlow**, and **PaddlePaddle** models. Single-chip only.
TT-MLIR	Core MLIR-based compiler. Defines TTIR, TTNN, and TTKernel dialects, applies optimization passes (fusion, sharding, layout), and lowers to TT-Metalium.
TT-Lang	Python DSL for custom high-performance kernels. Write fused ops in Python with built-in simulation, profiling, and AI-assisted translation from Triton-class DSLs. *(Early preview)*
TT-Blacksmith	Optimized training recipes and experiments. 40+ examples spanning PyTorch, JAX, and Lightning across vision models, LLMs, and NLP.
TT-Forge-Models	800+ model variants continuously tested in CI. Standardized loaders for LLMs, vision, NLP, multimodal, detection, segmentation, speech, and more.

800+ model variants

continuously tested in CI – thousands more ran internally

Category	Models
LLMs	Llama 3.1/3.2 (1B–70B), Qwen 2.5/3 (0.5B–32B), Falcon-3 (1B–10B), Phi-1/2/3/3.5, Gemma 1.1/2 (2B–7B), Mistral/Ministral (7B–24B), Mamba 2.8B
Vision	ResNet-50, ViT, EfficientNet, MobileNetV1/V2, Swin, VoVNet, SegFormer, U-Net, UFLD/UFLDv2, MNIST
NLP / Encoders	BERT, ALBERT, BGE-M3, Qwen3-Embedding, RoBERTa, SqueezeBERT
Multimodal	BLIP (vision-language), Stable Diffusion XL (UNet)
Multi-chip (N300+)	Llama 3.1 8B/70B, Falcon-3 7B/10B, Mistral 7B/Nemo/Small 24B, Qwen 2.5/3 up to 32B

Ease of Use Example: Loading Torch Vision Models

Get ResNet-50 model Running on TT HW in minutes

```
# Requires Ubuntu 24.04 and Python 3.12
python3.12 -m venv tt-forge-env
source tt-forge-env/bin/activate
pip install tt-forge --extra-index-url https://pypi.eng.aws.tenstorrent.com/
tt-forge-install
pip install torchvision
```

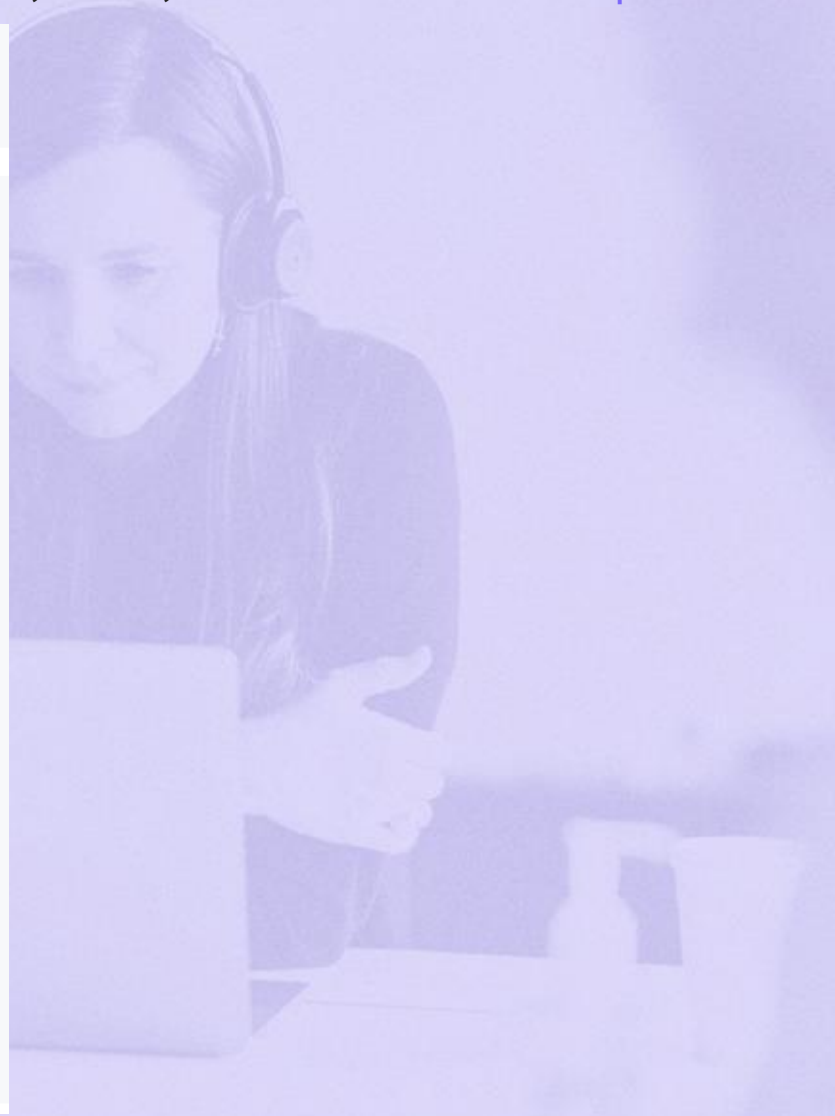
```
import torch
import torch_xla.core.xla_model as xm
import torch_xla.runtime as xr
import tt_torch
from torchvision.models import resnet50, ResNet50_Weights

# Set device to Tenstorrent
xr.set_device_type("TT")
device = xm.xla_device()

# Load ResNet-50
model = resnet50(weights=ResNet50_Weights.DEFAULT).to(torch.bfloat16).eval()
compiled_model = torch.compile(model, backend="tt")
compiled_model = compiled_model.to(device)

# Run inference on Tenstorrent
input_tensor = torch.randn(1, 3, 224, 224, dtype=torch.bfloat16).to(device)
with torch.no_grad():
    output = compiled_model(input_tensor)

predicted_class = output.cpu().argmax(dim=-1).item()
print(f"Predicted ImageNet class: {predicted_class}")
```

Train a Model

Standard PyTorch training example runs on Tenstorrent hardware via TT-Blacksmith
40+ ready-to-run training recipes - Llama, Gemma, Qwen, ViT, NeRF, and more. See the experiments table.

```
git clone https://github.com/tenstorrent/tt-blacksmith.git && cd tt-blacksmith
source env/activate --xla
```

```
import os
import torch
import torch_xla
import torch_xla.runtime as xr

os.environ["PJRT_DEVICE"] = "TT"
os.environ["XLA_STABLEHLO_COMPILE"] = "1"

# Standard PyTorch — the only difference is the device
device = torch_xla.xla_device()

model = MyModel().to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
loss_fn = torch.nn.CrossEntropyLoss()

for inputs, targets in train_loader:
    inputs, targets = inputs.to(device), targets.to(device)

    outputs = model(inputs)
    loss = loss_fn(outputs, targets)
    loss.backward()

    optimizer.step()
    torch_xla.sync(wait=True)
    optimizer.zero_grad()
```



Write a Custom Kernel

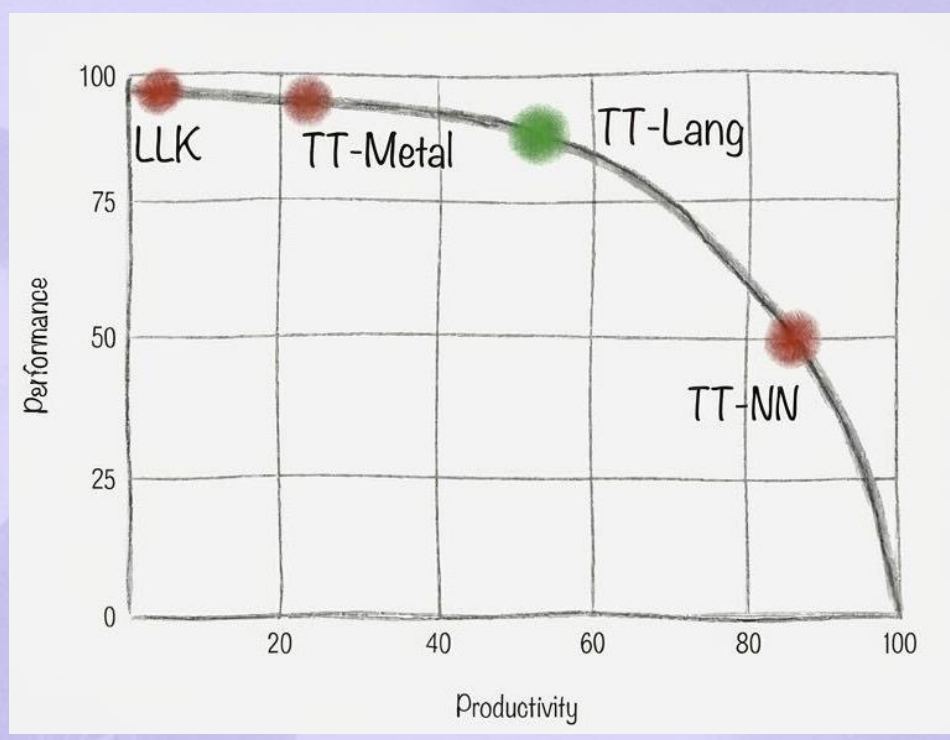
Python in, optimized hardware code out!

TT-Lang: write high-performance kernels in Python instead of low-level C++. Here's a matmul with bias (Y = A @ B + C):

```
# Matmul with bias: Y = A @ B + C
a_dfb = ttl.make_dataflow_buffer_like(A, shape=(1, 1, 1))
b_dfb = ttl.make_dataflow_buffer_like(B, shape=(1, 1))
c_dfb = ttl.make_dataflow_buffer_like(C, shape=(1, 1))
y_dfb = ttl.make_dataflow_buffer_like(Y, shape=(1, 1, 1))

@ttl.compute()
def matmul_compute():
    for _ in range(IT):
        for _ in range(MT):
            for _ in range(NT):
                with y_dfb.reserve() as y_blk:
                    with c_dfb.wait() as c_blk:
                        y_blk.store(c_blk, acc=True)       # y = c
                    for _ in range(KT):
                        with (
                            a_dfb.wait() as a_blk,
                            b_dfb.wait() as b_blk,
                        ):
                            y_blk.store(a_blk @ b_blk, acc=True)  # y += a @ b
```

TT-Lang

- **Python-based Domain-Specific Language (DSL)**
  for writing high-performance custom kernels on TT hardware
- **Python enables AI tools to translate GPU DSL kernels**
  (Triton, CUDA, cuTile, TileLang) to TT hardware more reliably than direct TT-Metal translation
- **Functional simulation via AI agents**:
  propose kernels, validate correctness, and iterate autonomously
- **Catch errors and performance issues in-IDE**
  via functional simulator — no hardware required
- **Line-by-line performance metrics and data flow graphs**
  help identify bottlenecks and optimization opportunities



Compiler handles:
- NOC addressing
- Register allocation
- Memory management

TT-Lang repo includes:
- Tutorials
- Simulation Tools
- Profiling Tools

TT-Lang Example

Goal: Fuse a * b + c into one kernel

```
y = (a * b + c) * d    on 8192x8192 bf16 tensors
```

Fuse the inner a*b+c — keep intermediates in L1

TT-NN baseline: 3 separate op launches

```
y = ttnn.multiply(
        ttnn.add(
            ttnn.multiply(a, b),    # op 1 -> DRAM
            c),                     # op 2 -> DRAM
        d)                          # op 3 -> DRAM

Three launches. Two intermediate DRAM round-trips.

With TT-Lang, a*b+c becomes ONE fused kernel:
  - a*b stays in L1 (no DRAM write)
  - +c happens immediately (no DRAM read)
  - One launch instead of two
```

WHY FUSE?

- Eliminate DRAM round-trips
- Reduce op launch overhead
- Keep intermediates in L1
- One kernel, three threads

TT-Lang Example Cont'd

TT-NN baseline — no custom kernel

Three separate TT-NN ops, each dispatched independently.

step_0_ttnn_base.py

```
import ttnn, torch

a = from_torch(torch.rand((8192, 8192), dtype=torch.bfloat16))
b = from_torch(torch.rand((8192, 8192), dtype=torch.bfloat16))
c = from_torch(torch.rand((8192, 8192), dtype=torch.bfloat16))
d = from_torch(torch.rand((8192, 8192), dtype=torch.bfloat16))

# 3 ops, 2 DRAM round-trips for intermediates
y = ttnn.multiply(ttnn.add(ttnn.multiply(a, b), c), d)

# a*b+c alone: 3.995 ms on 64 cores
```

WHAT HAPPENS

- multiply(a,b): compute + write DRAM
- add(..., c): read DRAM + compute + write
- multiply(..., d): read DRAM + compute
- Each op uses all 64 cores

TT-Lang Example Cont'd

Fused kernel — single node, one tile at a time

One core, one 32x32 tile per iteration. Fused a*b+c in a single kernel.

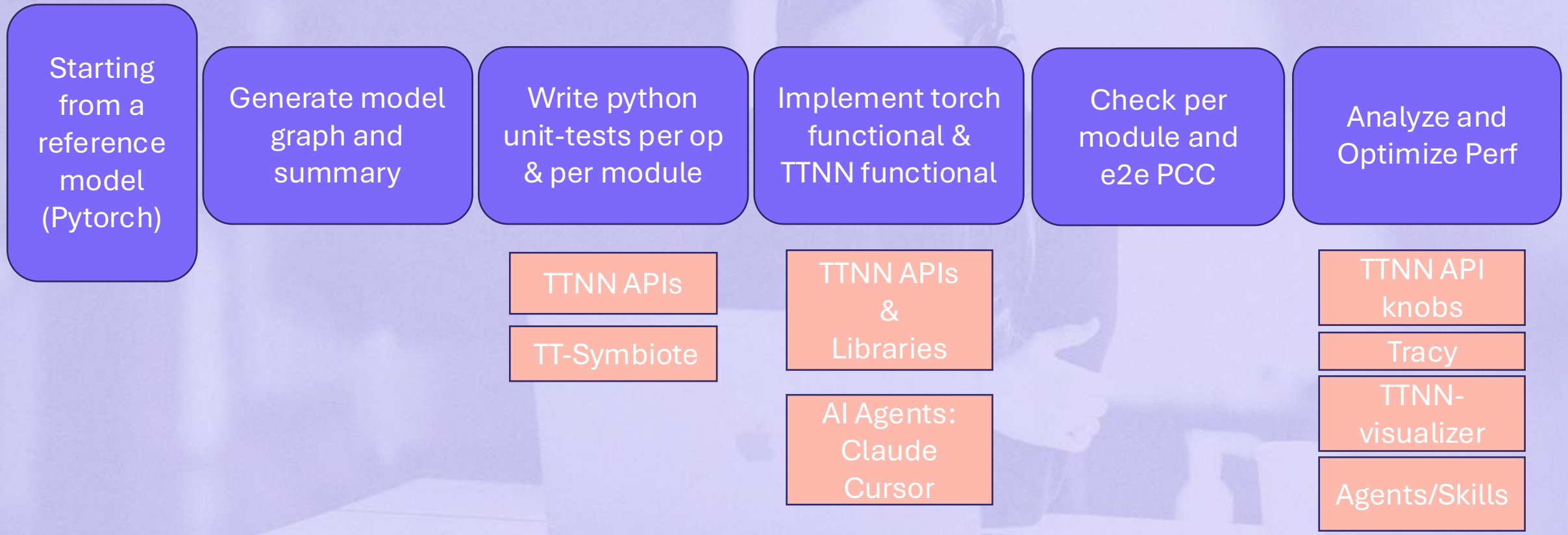
```
@ttl.operation(grid=(1, 1))              # 1 core
def __tutorial_operation(a, b, c, y):
    a_dfb = ttl.make_dataflow_buffer_like(a, shape=(1,1), block_count=2)
    ...

    @ttl.compute()
    def compute():
        for _ in range(rows):
            for _ in range(cols):
                with a_dfb.wait() as a_blk, b_dfb.wait() as b_blk,\
                     c_dfb.wait() as c_blk, y_dfb.reserve() as y_blk:
                    y_blk.store(a_blk * b_blk + c_blk)  # fused!

    @ttl.datamovement()     # reader: DRAM -> L1
    @ttl.datamovement()     # writer: L1 -> DRAM
```

CONCEPTS INTRODUCED

- @ttl.operation: declare grid
- make_dataflow_buffer: L1 DFB
- @ttl.compute: fused math
- @ttl.datamovement: NOC I/O

TT-Lang Demo – hands on session

Systematic Model Bring-Up and Optimization Flow



Model Summary & Unit Test

Layer (type)	Input Shape	Output Shape	Param #	Kernel Size	Stride	Padding	Dilation	Groups	
Conv2d-1	[-1, 3, 480, 640]	[-1, 32, 480, 640]	864	[3, 3]	[1, 1]	[1, 1]	[1, 1]	1	(1, 32, 3, 480, 640, 3, 3, 1, 1, 1, 1, True, None, False)
Conv2d-4	[-1, 32, 480, 640]	[-1, 64, 480, 640]	18,432	[3, 3]	[1, 1]	[1, 1]	[1, 1]	1	(1, 64, 32, 480, 640, 3, 3, 1, 1, 1, 1, True, None, False)
Conv2d-7	[-1, 64, 480, 640]	[-1, 32, 480, 640]	2,048	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 32, 64, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-10	[-1, 32, 480, 640]	[-1, 32, 480, 640]	288	[3, 3]	[1, 1]	[1, 1]	[1, 1]	32	(1, 32, 32, 480, 640, 3, 3, 1, 1, 1, 1, True, None, False)
Conv2d-13	[-1, 32, 480, 640]	[-1, 32, 480, 640]	1,024	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 32, 32, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-16	[-1, 32, 480, 640]	[-1, 32, 480, 640]	1,024	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 32, 32, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-19	[-1, 32, 480, 640]	[-1, 32, 480, 640]	288	[3, 3]	[1, 1]	[1, 1]	[1, 1]	32	(1, 32, 32, 480, 640, 3, 3, 1, 1, 1, 1, True, None, False)
Conv2d-22	[-1, 32, 480, 640]	[-1, 32, 480, 640]	1,024	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 32, 32, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-25	[-1, 64, 480, 640]	[-1, 32, 480, 640]	2,048	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 32, 64, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-28	[-1, 32, 480, 640]	[-1, 64, 480, 640]	2,048	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 32, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-32	[-1, 64, 480, 640]	[-1, 128, 480, 640]	73,728	[3, 3]	[1, 1]	[1, 1]	[1, 1]	1	(1, 128, 64, 480, 640, 3, 3, 1, 1, 1, 1, True, None, False)
Conv2d-35	[-1, 128, 480, 640]	[-1, 64, 480, 640]	8,192	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 128, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-38	[-1, 64, 480, 640]	[-1, 64, 480, 640]	576	[3, 3]	[1, 1]	[1, 1]	[1, 1]	64	(1, 64, 64, 480, 640, 3, 3, 1, 1, 1, 1, True, None, False)
Conv2d-41	[-1, 64, 480, 640]	[-1, 64, 480, 640]	4,096	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 64, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-44	[-1, 64, 480, 640]	[-1, 64, 480, 640]	4,096	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 64, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-47	[-1, 64, 480, 640]	[-1, 64, 480, 640]	576	[3, 3]	[1, 1]	[1, 1]	[1, 1]	64	(1, 64, 64, 480, 640, 3, 3, 1, 1, 1, 1, True, None, False)
Conv2d-50	[-1, 64, 480, 640]	[-1, 64, 480, 640]	4,096	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 64, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-53	[-1, 128, 480, 640]	[-1, 64, 480, 640]	8,192	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 128, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-56	[-1, 64, 480, 640]	[-1, 128, 480, 640]	8,192	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 128, 64, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-60	[-1, 128, 480, 640]	[-1, 64, 480, 640]	8,192	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 128, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-63	[-1, 64, 480, 640]	[-1, 64, 480, 640]	576	[3, 3]	[1, 1]	[1, 1]	[1, 1]	64	(1, 64, 64, 480, 640, 3, 3, 1, 1, 1, 1, True, None, False)
Conv2d-66	[-1, 64, 480, 640]	[-1, 64, 480, 640]	4,096	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 64, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-69	[-1, 64, 480, 640]	[-1, 64, 480, 640]	4,096	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 64, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-72	[-1, 64, 480, 640]	[-1, 64, 480, 640]	576	[3, 3]	[1, 1]	[1, 1]	[1, 1]	64	(1, 64, 64, 480, 640, 3, 3, 1, 1, 1, 1, True, None, False)
Conv2d-75	[-1, 64, 480, 640]	[-1, 64, 480, 640]	4,096	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 64, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)
Conv2d-78	[-1, 128, 480, 640]	[-1, 64, 480, 640]	8,192	[1, 1]	[1, 1]	[0, 0]	[1, 1]	1	(1, 64, 128, 480, 640, 1, 1, 1, 1, 0, 0, True, None, False)

```
@pytest.mark.parametrize(
    "batch_size, output_channels, input_channels, input_height, input_width, filter_height, filter_width, stride_h, stride_w, pad_h, pad_w, use_1d_systolic_array, config_override",
    (
        # unique convs in rn50 (complete list)
        # first conv post folding and input_channels padding to tile width
        (8, 64, 16, 115, 115, 4, 4, 1, 1, 0, 0, True, None),
        (16, 64, 16, 115, 115, 4, 4, 1, 1, 0, 0, True, {"act_block_h": 32}),
        (20, 64, 16, 115, 115, 4, 4, 1, 1, 0, 0, True, {"act_block_h": 32}),
        # rn50 layer1
        (8, 64, 64, 56, 56, 3, 3, 1, 1, 1, 1, True, None),
        (16, 64, 64, 56, 56, 3, 3, 1, 1, 1, 1, True, None),
        (20, 64, 64, 56, 56, 3, 3, 1, 1, 1, 1, True, None),
        # rn50 layer2
        (8, 128, 128, 56, 56, 3, 3, 2, 2, 1, 1, True, None),
        (16, 128, 128, 56, 56, 3, 3, 2, 2, 1, 1, True, None),
        (20, 128, 128, 56, 56, 3, 3, 2, 2, 1, 1, True, {"act_block_h": 32}),
        (8, 128, 128, 28, 28, 3, 3, 1, 1, 1, 1, True, None),
        (16, 128, 128, 28, 28, 3, 3, 1, 1, 1, 1, True, None),
        (20, 128, 128, 28, 28, 3, 3, 1, 1, 1, 1, True, None),
        # rn50 layer3
        (8, 256, 256, 28, 28, 3, 3, 2, 2, 1, 1, False, None),
```

Unit test:

- **Check PCC**
- **Parameterize op arguments & optimization knobs** – heigh-shard, width-shard, etc
- **Automate via agents**
- **Modular baseline via TT-Symbiote**:
  - Leverage existing modules in TT-Symbiote
  - Register any missing ones
  - Quick modular experimentation

PyTorch vs TTNN

PyTorch	TT-NN™
• Doesn't handle block-floats, tiles, layouts, or sharding	• **Native layouts and sharding**
• Doesn't natively handle multi-device • Custom libraries currently developed for multi-device	• **Native Distributed Shared Memory (SRAM and DRAM)**
• Doesn't support performance OP configurations • Model developer can't configure OPs for performance	• **Developer can configure OPs for performance** • **OPs are designed to be a great target for compilers / MLIR**
• Not truly native multi-device • Additional layers on top	• **Native multi-device / multi-host**

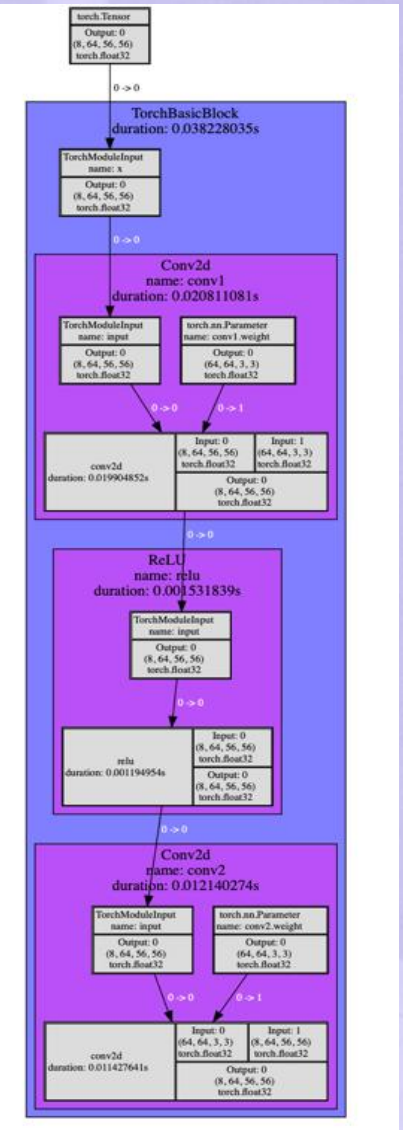
PyTorch vs TTNN Code

```python
# SPDX-FileCopyrightText: © 2023 Tenstorrent Inc.

# SPDX-License-Identifier: Apache-2.0


import torch
import torch.nn as nn


class DownSample1(nn.Module):
    def __init__(self):
        super().__init__()
        self.c1 = nn.Conv2d(3, 32, 3, 1, 1, bias=False)
        self.b1 = nn.BatchNorm2d(32)
        self.relu = nn.ReLU(inplace=True)

        self.c2 = nn.Conv2d(32, 64, 3, 2, 1, bias=False)
        self.b2 = nn.BatchNorm2d(64)

        self.c3 = nn.Conv2d(64, 64, 1, 1, 0, bias=False)
        self.b3 = nn.BatchNorm2d(64)

        self.c4 = nn.Conv2d(64, 64, 1, 1, 0, bias=False)
        self.b4 = nn.BatchNorm2d(64)

        self.c5 = nn.Conv2d(64, 32, 1, 1, 0, bias=False)
        self.b5 = nn.BatchNorm2d(32)

        self.c6 = nn.Conv2d(32, 64, 3, 1, 1, bias=False)
        self.b6 = nn.BatchNorm2d(64)

        self.c7 = nn.Conv2d(64, 64, 1, 1, 0, bias=False)
        self.b7 = nn.BatchNorm2d(64)

        self.c8 = nn.Conv2d(128, 64, 1, 1, 0, bias=False)
        self.b8 = nn.BatchNorm2d(64)

    def forward(self, input: torch.Tensor):
        x1 = self.c1(input)
        x1_b = self.b1(x1)
        x1_m = self.relu(x1_b)

        x2 = self.c2(x1_m)
        x2_b = self.b2(x2)
        x2_m = self.relu(x2_b)

        x3 = self.c3(x2_m)
        x3_b = self.b3(x3)
        x3_m = self.relu(x3_b)

        x4 = self.c4(x2_m)
        x4_b = self.b4(x4)
        x4_m = self.relu(x4_b)

        x5 = self.c5(x4_m)
        x5_b = self.b5(x5)
        x5_m = self.relu(x5_b)

        x6 = self.c6(x5_m)
        x6_b = self.b6(x6)
        x6_m = self.relu(x6_b)
        x6_m = x6_m + x4_m

        x7 = self.c7(x6_m)
        x7_b = self.b7(x7)
        x7_m = self.relu(x7_b)
        x7_m = torch.cat([x7_m, x3_m], dim=1)

        x8 = self.c8(x7_m)
        x8_b = self.b8(x8)
        x8_m = self.relu(x8_b)

```

```python
class Down1:
    def __init__(self, model) -> None:
        if type(model) is str:
            torch_model = torch.load(model)
        else:
            torch_model = model.torch_model
        self.torch_model = torch_model
        self.conv1 = Conv(torch_model, "down1.conv1", [1, 320, 320, 3], (1, 1, 1, 1), act_block_h=128)
        self.conv2 = Conv(torch_model, "down1.conv2", [1, 320, 320, 32], (2, 2, 1, 1), reshard=True)
        self.conv3 = Conv(torch_model, "down1.conv3", [1, 160, 160, 64], (1, 1, 0, 0), deallocate=False)
        self.conv4 = Conv(torch_model, "down1.conv4", [1, 160, 160, 64], (1, 1, 0, 0), reshard=True, deallocate=False)
        self.conv5 = Conv(torch_model, "down1.conv5", [1, 160, 160, 64], (1, 1, 0, 0), deallocate=False)
        self.conv6 = Conv(torch_model, "down1.conv6", [1, 160, 160, 32], (1, 1, 1, 1))
        self.conv7 = Conv(torch_model, "down1.conv7", [1, 160, 160, 64], (1, 1, 0, 0))
        self.conv8 = Conv(torch_model, "down1.conv8", [1, 160, 160, 128], (1, 1, 0, 0))
        self.convs = [self.conv1, self.conv2, self.conv3, self.conv4, self.conv5, self.conv6, self.conv7, self.conv8]

    def __call__(self, device, input_tensor):
        output_tensor = self.conv1(device, input_tensor)
        output_tensor_split = self.conv2(device, output_tensor)

        output_tensor_left = self.conv3(device, output_tensor_split)

        res_block_split = self.conv4(device, output_tensor_split)
        output_tensor = self.conv5(device, res_block_split)
        output_tensor = self.conv6(device, output_tensor)
        output_tensor = res_block_split + output_tensor

        ttnn.deallocate(res_block_split)
        output_tensor = self.conv7(device, output_tensor)

        output_tensor = ttnn.experimental.tensor.sharded_to_interleaved(output_tensor, ttnn.L1_MEMORY_CONFIG)
        output_tensor_left = ttnn.experimental.tensor.sharded_to_interleaved(output_tensor_left, ttnn.L1_MEMORY_CONFIG)
        output_tensor = ttnn.concat([output_tensor, output_tensor_left], dim=3, memory_config=ttnn.L1_MEMORY_CONFIG)

        output_tensor = self.conv8(device, output_tensor)
        return output_tensor
```

```python
class Conv:
    def __init__(
        self, model, path, input_params, conv_params, *, act_block_h=None, reshard=False, deallocate=True
    ) -> None:
        self.weights, self.bias = fold_bn_to_conv_weights_bias(model, path)
        self.input_params = input_params
        self.kernel_size = (self.weights.shape[2], self.weights.shape[3])
        self.conv_params = conv_params
        self.out_channels = self.weights.shape[0]
        self.act_block_h = act_block_h
        self.reshard = reshard
        self.deallocate = deallocate

    def __str__(self) -> str:
        return f"Conv: {self.weights.shape} {self.bias.shape} {self.kernel_size}"

    def __call__(self, device, input_tensor):
        conv_config = ttnn.Conv2dConfig(
            dtype=ttnn.bfloat8_b,
            weights_dtype=ttnn.bfloat8_b,
            math_fidelity=ttnn.MathFidelity.LoFi,
            activation="relu",
            height_sharding=True,
            math_approx_mode_enabled=True,
            fp32_dest_acc_enabled=False,
            packer_l1_accum_enabled=False,
            input_channels_alignment=16 if self.input_params[3] < 16 else 32,
            transpose_shards=True,
            reshard_if_not_optimal=self.reshard,
            deallocate_activation=self.deallocate,
            reallocate_halo_output=False,
        )
        if self.act_block_h is not None:
            conv_config.act_block_h_override = self.act_block_h

        [output_tensor, _out_height, _out_width, self.weights, self.bias] = ttnn.conv2d(
            input_tensor=input_tensor,
            weight_tensor=self.weights,
            bias_tensor=self.bias,
            in_channels=self.input_params[3],
            out_channels=self.out_channels,
            device=device,
            kernel_size=self.kernel_size,
            stride=(self.conv_params[0], self.conv_params[1]),
            padding=(self.conv_params[2], self.conv_params[3]),
            batch_size=self.input_params[0],
            input_height=self.input_params[1],
            input_width=self.input_params[2],
            conv_config=conv_config,
        )
        return output_tensor
```

Tracy for Performance Analysis



OP CODE	DEVICE KERNEL DURATION [ns]
UntilizeWithHaloV2	5685
OptimizedConv	124923
Move	5394
UntilizeWithHaloV2	26953
MaxPool	121956
Tilize	3572
Move	1584
tt::operations::primary::Matmul	8480
UntilizeWithHaloV2	11512
OptimizedConv	56452
tt::operations::primary::Matmul	22560
tt::operations::primary::Matmul	22607
EltwiseBinary	8953
Move	5528
tt::operations::primary::Matmul	16542
UntilizeWithHaloV2	11418
OptimizedConv	56567
tt::operations::primary::Matmul	22512
EltwiseBinary	8944
tt::operations::primary::Matmul	16617

TTNN Visualizer

- Comprehensive list of all operations in the model or in a module
- Interactive graph visualization of operations
- Interactive and detailed L1, DRAM and Circular Buffer memory plots
- Overview of all buffers for the entire model run
- Filterable list of tensor details
- Visualize tensor allocations per core

Export variables to set for generating visualizer reports:

```
export TT_METAL_HOME=$(pwd) # Adjust if your root path differs
export PYTHONPATH=$TT_METAL_HOME

export TTNN_CONFIG_OVERRIDES='{
  "enable_logging": true,
  "enable_fast_runtime_mode": false,
  “report_name”: “ResNet50”,
  "enable_comparison_mode": true,
  "enable_detailed_buffer_report": true,
  "enable_detailed_tensor_report": true,
  "enable_graph_report": true
}'
```

Variables	Value	Description
↳ *"enable_logging"*	`true` / `false`	Captures operational logs required for timeline tracking.
↳ *"enable_fast_runtime_mode"*	`true` / `false`	Controls dispatch tracing depth; set `false` for sub-graph debugging and comparison modes.
"report_name"	`“ResNet50”`	Give a unique name for your report
↳ *"enable_comparison_mode"*	`true` / `false`	Activates TT-NN Comparison Mode to automatically validate operation outputs against a known reference (requires `enable_fast_runtime_mode: false`).
↳ *"enable_detailed_buffer_report"*	`true` / `false`	Generates per-page buffer structures for the Buffers tab.
enable_detailed_tensor_report"	`true` / `false`	Enables visualization of the actual input and output tensor values for every operation.
enable_graph_report”	`true` / `false`	Generate graph report

- **Run:**

```
pytest models/demos/vision/classification/resnet50/blackhole/demo/demo.py::test_demo_sample”
```

- **Reports generated under a path like:**

```
generated/ttnn/reports/11300304066889687934
```

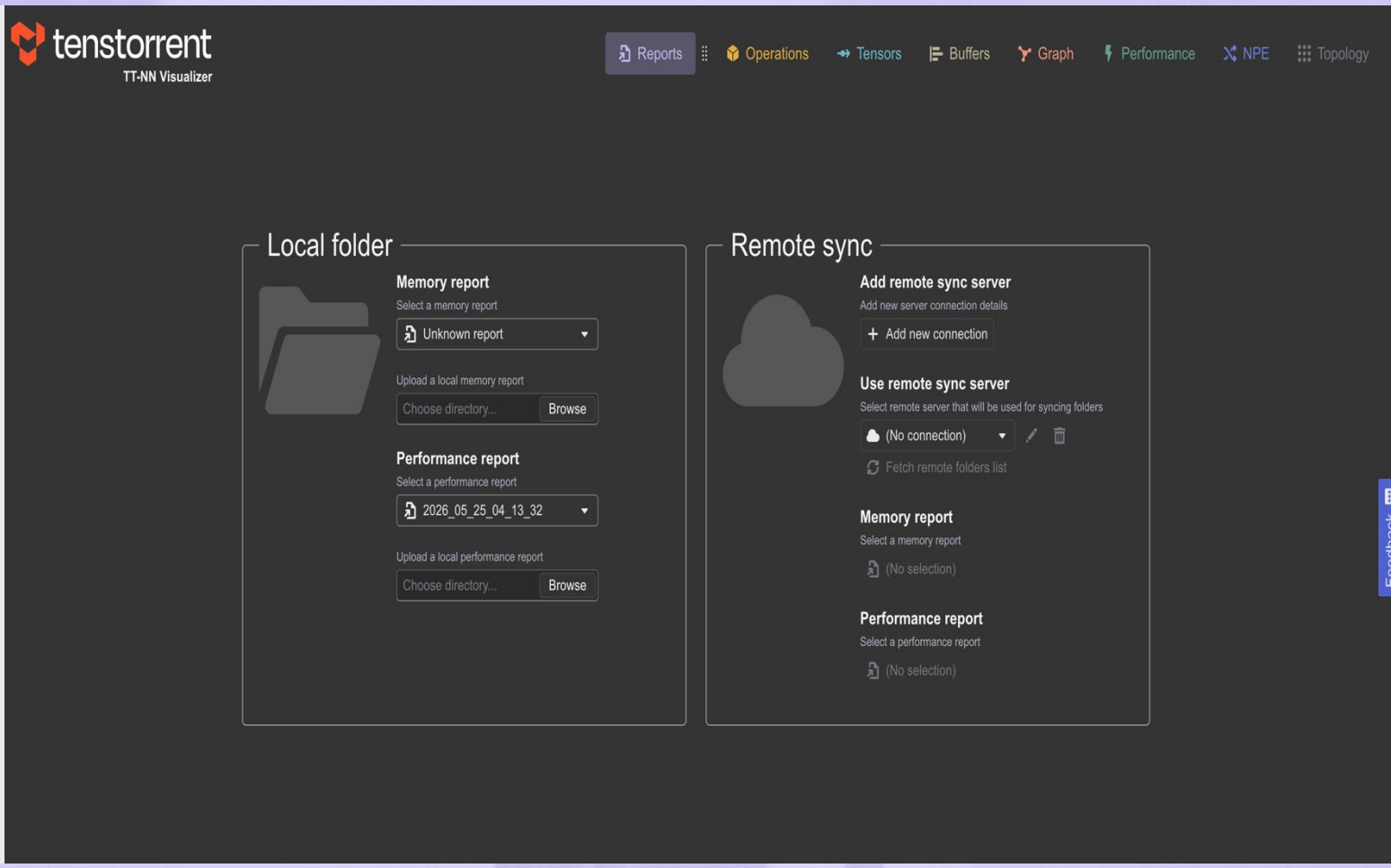
- **To generate the tracy perf sheet along with the visualizer reports:**

```
unset TTNN_Visualizer_CONFIGs
Run: tools/tracy/profile_this.py –m “pytest --disable-warnings
models/demos/vision/classification/resnet50/blackhole/demo/demo.py::test_demo_sample”
```

- **You will see:**

```
OPs csv generated at: /home/ubuntu/tt-
metal/generated/profiler/reports/2025_03_21_19_40_57/ops_perf_results_2025__03_21_19_40_57.csv
```

TTNN Visualizer – Live demo – ResNet50 – QB2 (BH)



TT-Symbiote

Linear & Activations:

- nn.Linear → TTNNLinear
- nn.GELU → TTNNGelu
- nn.SiLU → TTNNSilu
- nn.ReLU → TTNNReLU
- …

Normalization:

- nn.LayerNorm → TTNNLayerNorm
- …

Convolutions:

- nn.Conv2d → TTNNConv2dNHWC
- nn.MaxPool2d → TTNNMaxPool2dNHWC
- …

Attention:

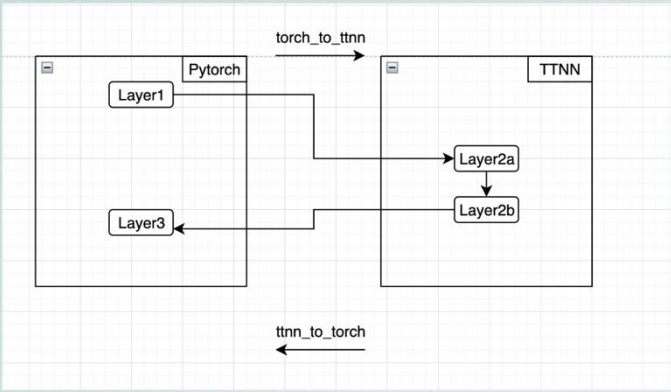
- Self-attention → TTNNSelfAttention
- Whisper attention → TTNNWhisperAttention
- …

What is TT-Symbiote?

Think of it as a translator:

- Model lives in PyTorch
- Tenstorrent hardware requires TTNN
- TT-Symbiote seamlessly integrates TTNN execution with Pytorch execution

No free lunch. Bouncing between them introduces time penalty



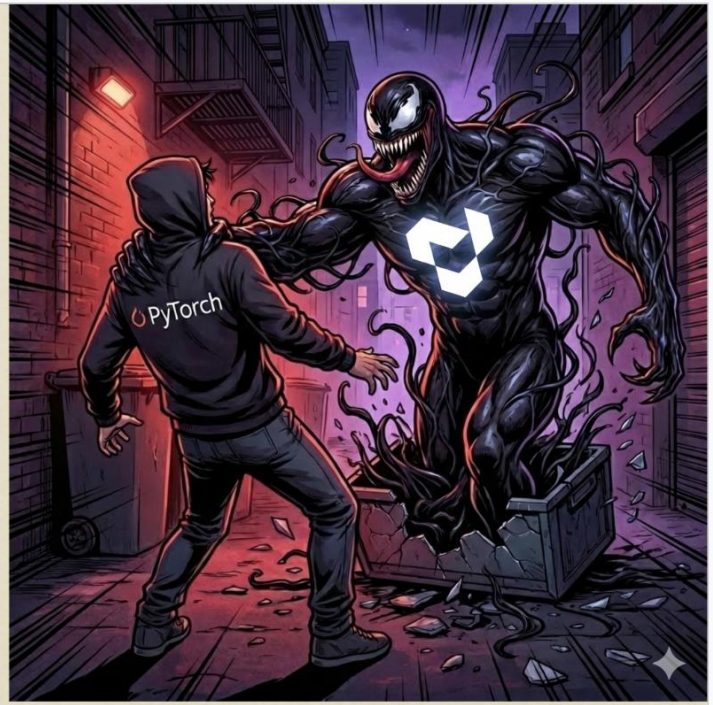
How it looks

Before

```
model = AutoModelForImageClassification.from_pretrained("google/vit-base-patch16-224")
result = model(torch.randn(1, 3, 224, 224))
```

After

```
model = AutoModelForImageClassification.from_pretrained("google/vit-base-patch16-224")

# ✨ Add 3 lines ✨
nn_to_ttnn = {nn.Linear: TTNNLinear, nn.LayerNorm: TTNNLayerNorm}
register_module_replacement_dict(model, nn_to_ttnn)
set_device(model, ttnn_device)

result = model(torch.randn(1, 3, 224, 224))
```



That's it!
All Linear + LayerNorm layers automatically replaced.

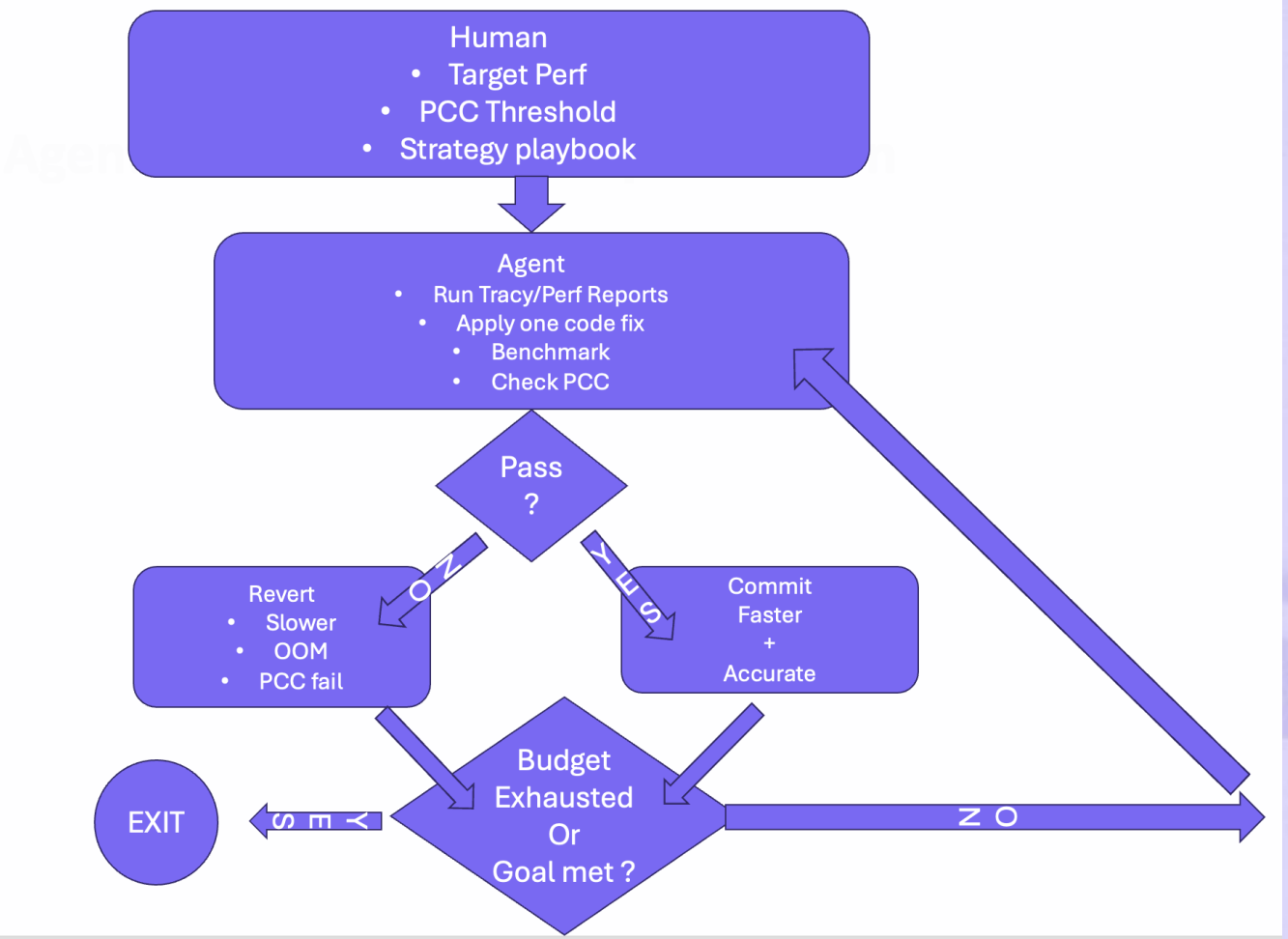
TT-symbiote unique position

1. Model runs on CPU = can run on TT
2. Framework built to optimize manual model bring up process using the following:
   a. Automatic trace capture/replay
   b. Automatic signpost insertion for tracy
   c. Automatic module dispatch time tracking
   d. Reusable modules
3. TTNN top perf = Symbiote top perf
4. Framework invisible once all pytorch modules translated to TTNN

MOE models:
- Sparse MMs
- CCLs

Agents for TTNN Model Optimization

automate measure → patch → verify



Examples – Swin_L Model Optimization Reference

- Target Perf: 10 FPS
- PCC > 0.99
- Strategy Playbook
- Budget Definition

"Budget exhausted" = experiment budget, not memory

Kind of budget	What it means (examples)
Attempt count	e.g. Run at most 60 optimization experiments (Swin agent run)
Time	e.g. Stop after 4 hours or before a CI window ends
Cost	Token/API spend, GPU hours, or review time cap
No progress	Stop if the last N tries show no measurable gain

Not device memory — OOM / L1 limits are handled on the Revert path in the agent loop.

Loop exits when: goal met OR experiment budget exhausted.

Example: Strategy Playbook (for coding agents)

1. Goal & guardrails	
Target	e.g. "Optimize Swin-L until 10 FPS" or "BGE-M3 Batch-32 under 65 ms"
PCC threshold	Must pass model accuracy check before keep
Experiment budget	Max N attempts / time cap (NOT device memory)
Iteration rule	One change per iteration; revert on slower / OOM / PCC fail

2. Allowed levers (pick one per loop)	
Data movement	ROW_MAJOR, L1 handoffs, replace concat chains (Swin: grid_sample vs 753 concats)
Fusion	Fuse GroupNorm 9→2 kernels; reduce op count (BGE: 368→318 ops)
Matmul sweeps	in0_block_w, subblock, core grid — respect shared L1 budget
Precision / fidelity	bf8, LoFi, fp32_dest_acc_en — re-check PCC after every change

3. Workflow the agent follows	
Loop	Tracy profile → tt-perf-report → hypothesize ONE fix → benchmark → PCC → commit or revert → repeat

4. Example priorities (BGE-M3 / Swin-L)	
BGE-M3	Batch 1: 11→4.3 ms; Batch 32: 200→61 ms; matmul + precision tuning
Swin-L	2.27→5.66 FPS; data movement biggest win; GroupNorm fusion

5. Tools	
Profile	python -m tracy -p -r -v -m pytest perf.py
Report	tt-perf-report .../ops_perf_results_*.csv --start-signpost start --end-signpost stop

Internal engineering playbook excerpt — one lever per loop; experiment budget ≠ device memory

Swin Example

Metric	Value
Instruction	Optimize to 10 FPS
Playbook	Prior BGE-M3 strategies + 5-step workflow
Budget	60 coding attempts
Kept	19 commits
Reverted	Failures: Slower, OOM, PCC degradation, Crash

Optimization Levers

Lever	Agent typically tries
Data movement	ROW_MAJOR, L1 handoffs, replace concat chains
Fusion	Custom ttnn fused ops; reduce per-layer op count
Matmul	Sweep subblock / in0_block_w under shared L1 budget
Precision	bf8, LoFi, fp32_dest_acc_en — always re-run PCC

Results at a Glance

	BGE-M3	ATSS Swin-L DyHead
Throughput	B1: 11→4.3 ms · B32: 200→61 ms	2.27 → 5.66 FPS
Data movement	L1 handoff; skip DRAM trips	ROW_MAJOR + grid_sample; drop 753 concats
Fusion	−26 ops (~130 µs)	9 GroupNorm kernels → 2
Precision	Lower fidelity + acc settings → larger subblocks	bf8 + LoFi (~25% jump)
Matmul	Attention + MLP chunk tuning	MLP-focused sweep under shared L1

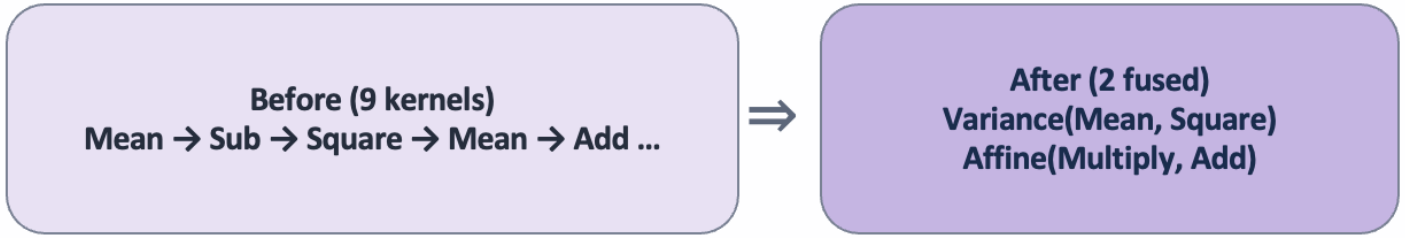
Data Movement and Layout

	Problem	Fix	Impact
Swin-L	753× ttnn.concat mask stitching	ttnn.grid_sample + ROW_MAJOR	Biggest win (~+1.8 FPS)
BGE-M3	DRAM round-trips between steps	Keep tensors in L1 between ops	24 fewer DRAM trips / layer path

Principle: change how you view tensors before launching reshape kernels.

Fusion And Op-Count Reduction

GroupNorm on Swin-L: fewer kernel launches → less dispatch overhead.



Model	Effect
Swin-L	+~1.8 FPS; fused DCN offset/scale cut DRAM round-trips
BGE-M3 B1	368 → 318 ops; ~130 µs saved (host-bound)

Matmul tuning (sweeps)

Topic	Detail
What we sweep	Core grid, in0_block_w, output subblock — dozens of configs
BGE-M3 B32 QKV	1×3 subblock, in0_block_w=8 — max before L1 overflow
Reality check	Standalone sweeps show 30–50% gains; in-model gains shrink (shared L1)
Goal	Fast enough for the model, small enough to coexist with neighbors

Precision & fidelity

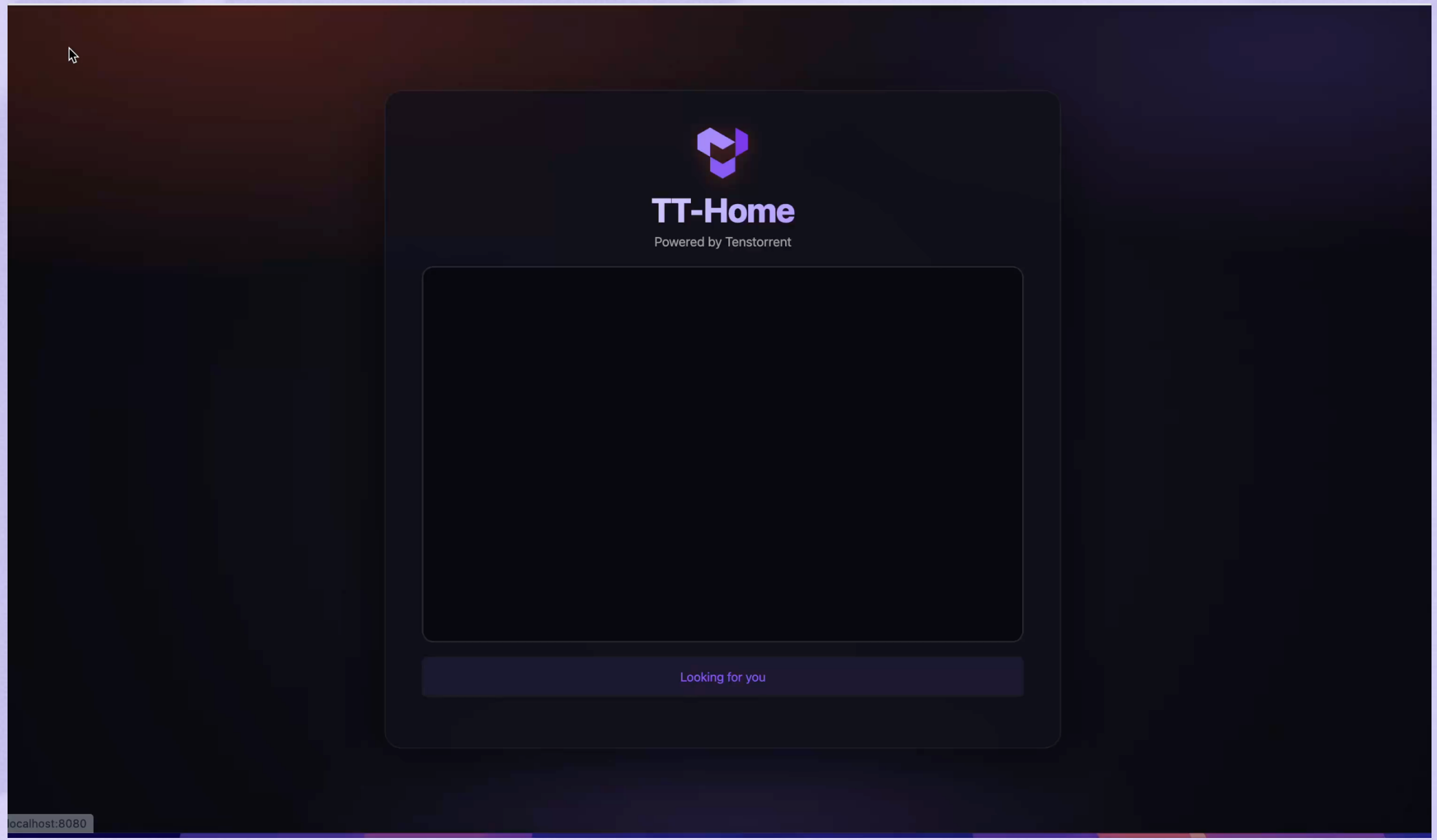
Knob	Trade-off	Model payoff
bf8 vs bf16	Half DRAM bandwidth	Swin-L ~25% perf jump
LoFi vs HiFi4	Faster math, less precision	Use when PCC still passes
fp32_dest_acc_en=False	Subblock limit 4→8	BGE B32: ~3.7 ms saved

TT-Inference-Server

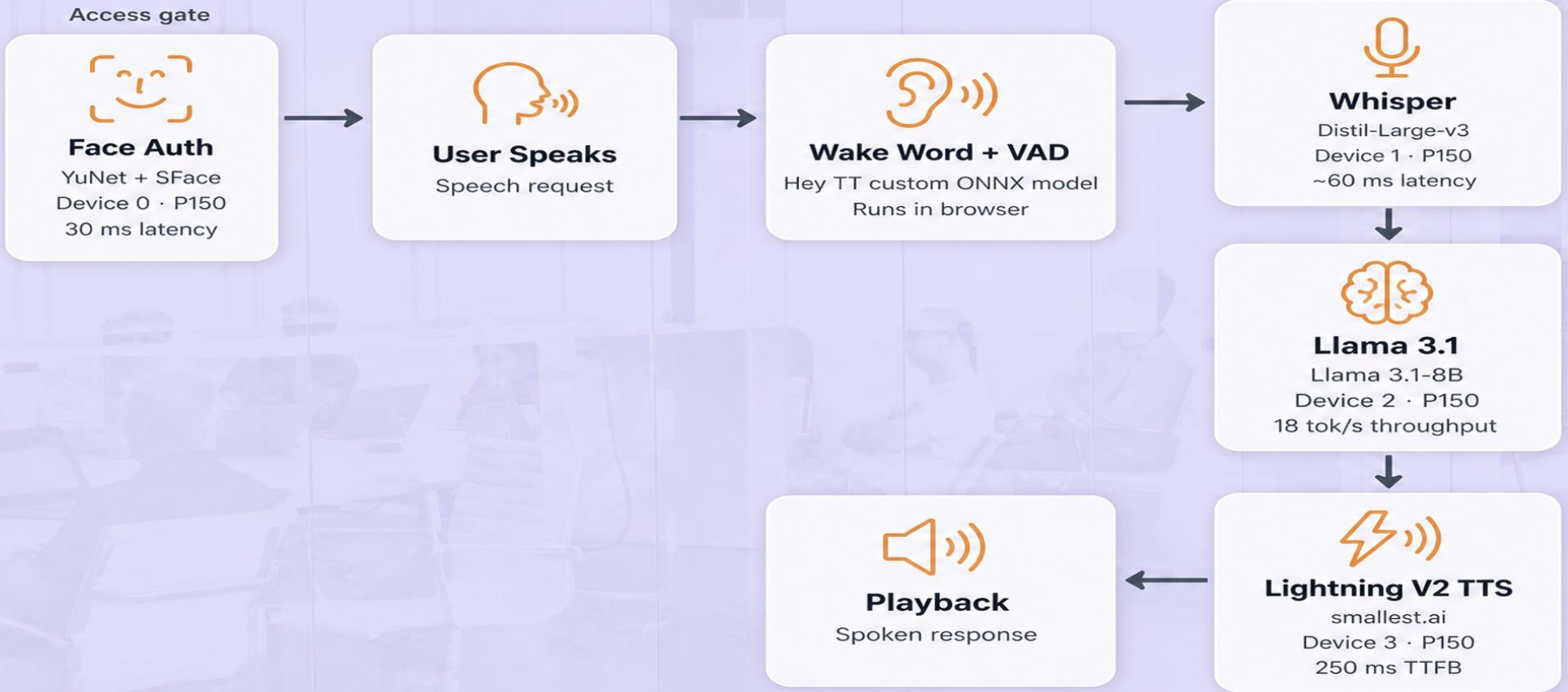
TT-Inference-Server: wrapper around VLLM

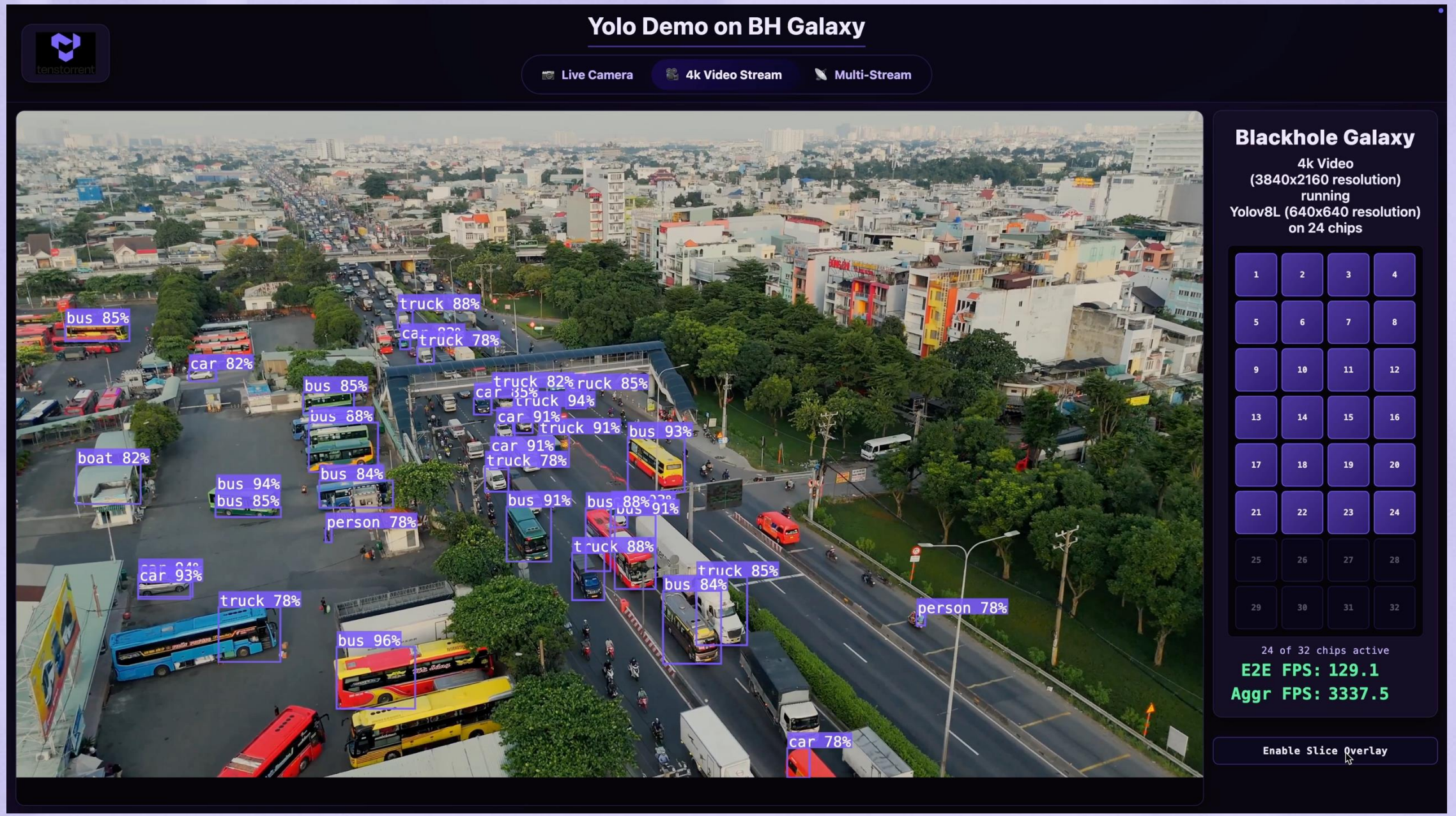
TT-Console: TT-Inference-Server under the hood

TT-Home Application



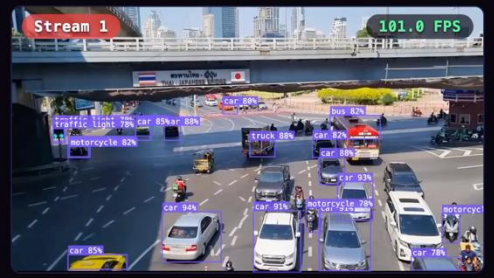
TT-Home Pipeline - QuietBox 2

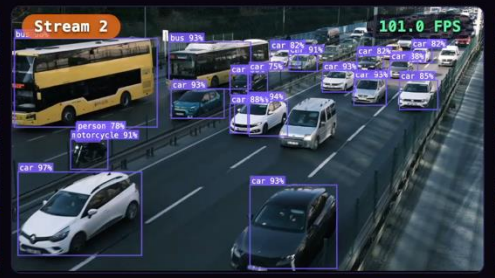


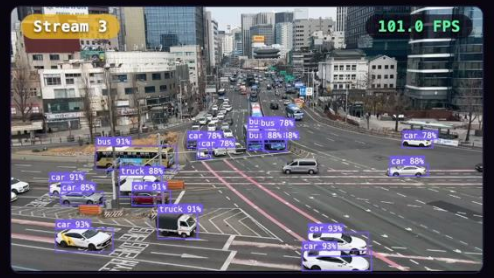

Yolo Demo on BH Galaxy
tenstorrent
Live Camera
4k Video Stream
Multi-Stream
bus 85%
truck 88%
truck 78%
car 82%
bus 85%
truck 82%
truck 85%
truck 94%
car 91%
truck 91%
bus 93%
car 91%
truck 78%
boat 82%
bus 84%
bus 94%
bus 85%
bus 91%
person 78%
truck 88%
car 93%
truck 85%
bus 84%
truck 78%
bus 96%
person 78%
car 78%
Blackhole Galaxy
4k Video
(3840x2160 resolution)
running
Yolov8L (640x640 resolution)
on 24 chips
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
25 26 27 28
29 30 31 32
24 of 32 chips active
E2E FPS: 129.1
Aggr FPS: 3337.5
Enable Slice Overlay

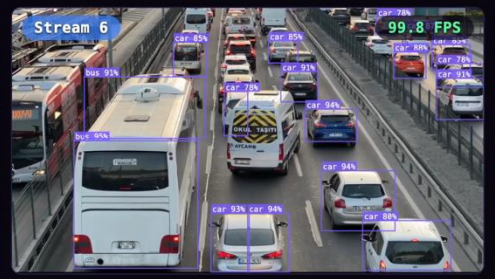
Yolo Demo on BH Galaxy

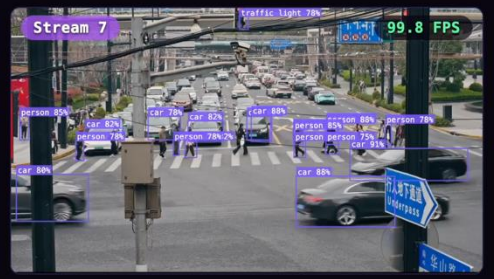
Live Camera | 4k Video Stream | Multi-Stream











Blackhole Galaxy

8 Parallel Stream
(1280x1280 resolution)
running
Yolov8L (640x640 resolution)
on 32 chips

1	2	3	4
	Stream 1		
5	6	7	8
	Stream 2		
9	10	11	12
	Stream 3		
13	14	15	16
	Stream 4		
17	18	19	20
	Stream 5		
21	22	23	24
	Stream 6		
25	26	27	28
	Stream 7		
29	30	31	32
	Stream 8		

32 of 32 chips active

E2E FPS: 111.2

Aggr FPS: 4425.4

Enable Slice Overlay