

Models on Datascale

May 2024



Overview

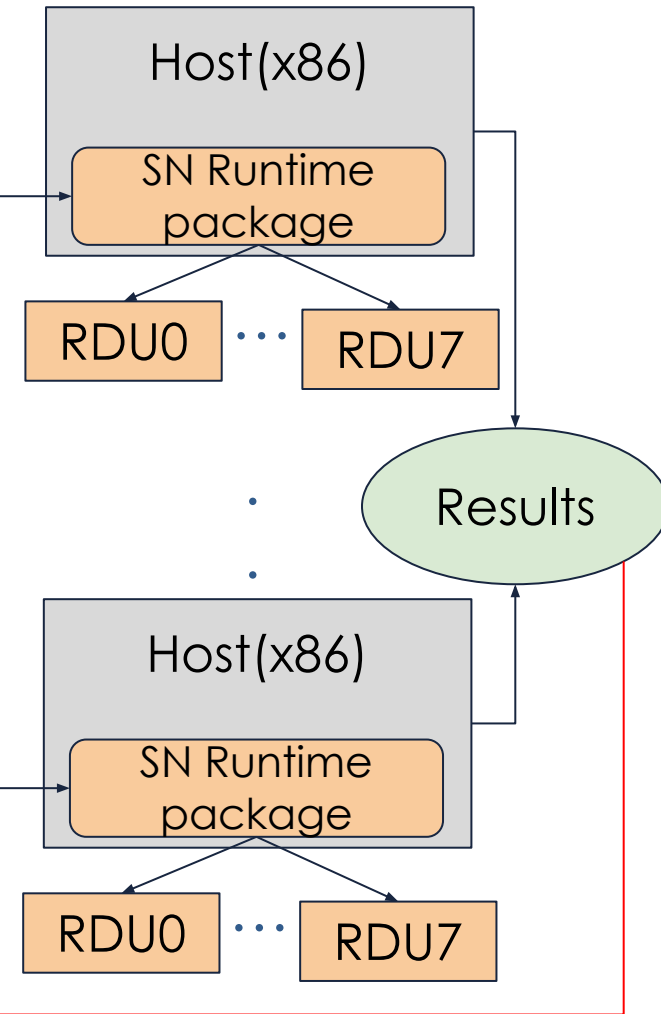
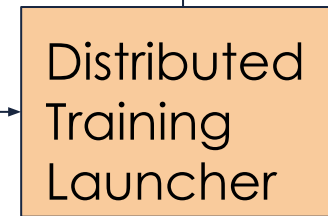
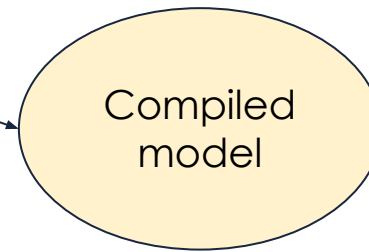
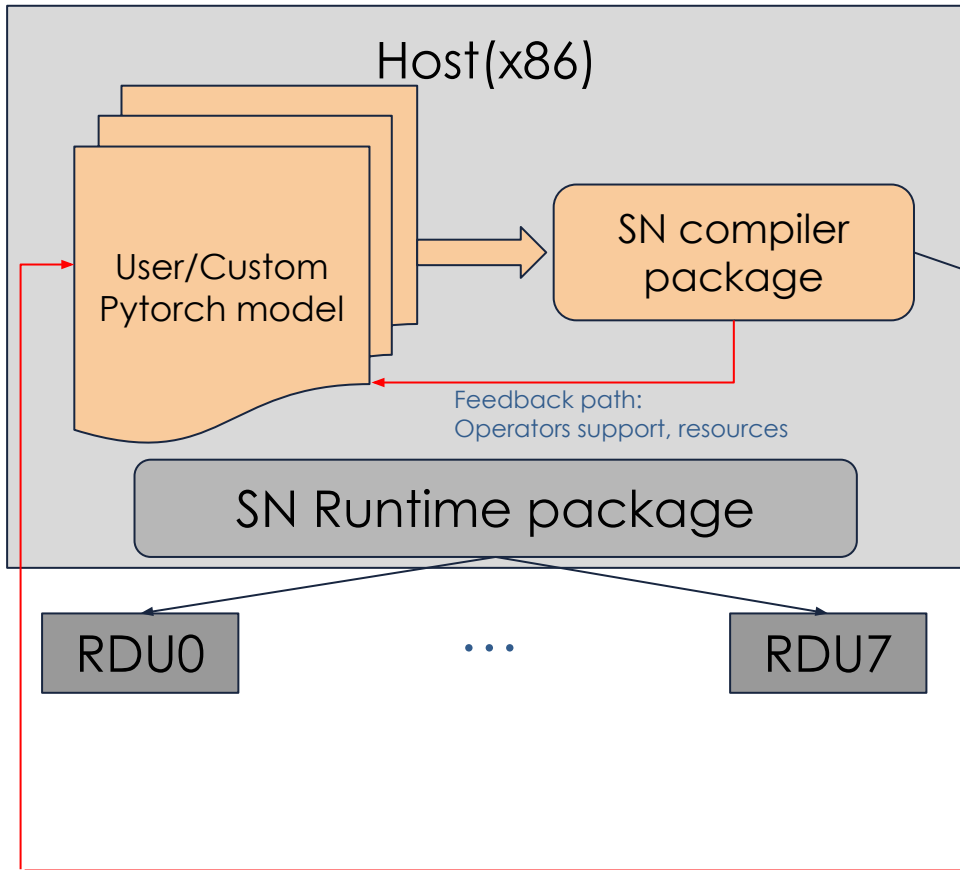
- SambaFlow Python SDK - user interface to compile and run their models on a Datascale System
- Train / Finetune the compiled model on the RDUs by passing in the PEF file and the training dataset
- Validate the model or start inference by passing the checkpoint and testing dataset
- 3 different options of distribution
 - + Bare metal - ideal for custom models
 - + Model Zoo - ideal for adapting OSS models
 - + Model Box - ideal for execution



Bare metal workflow

ALCF: Execution phase

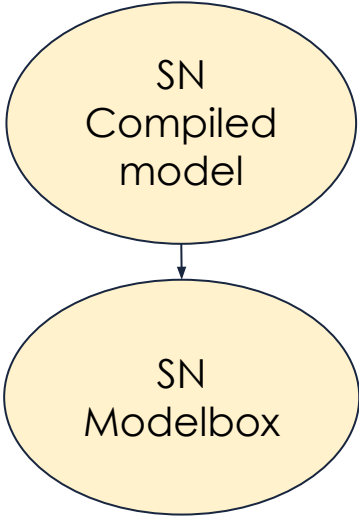
ALCF: Compilation phase



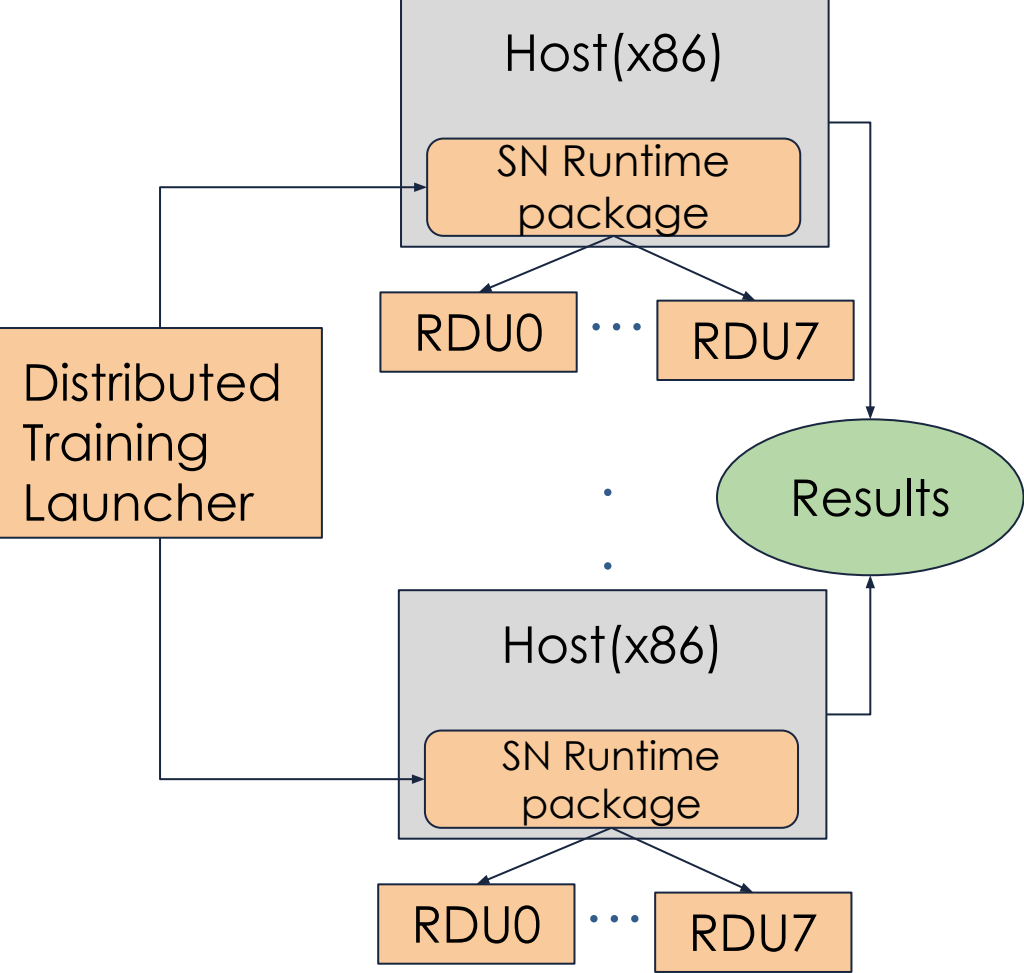
Feedback path : precision choice, performance

ModelBox workflow

SN: Compilation phase



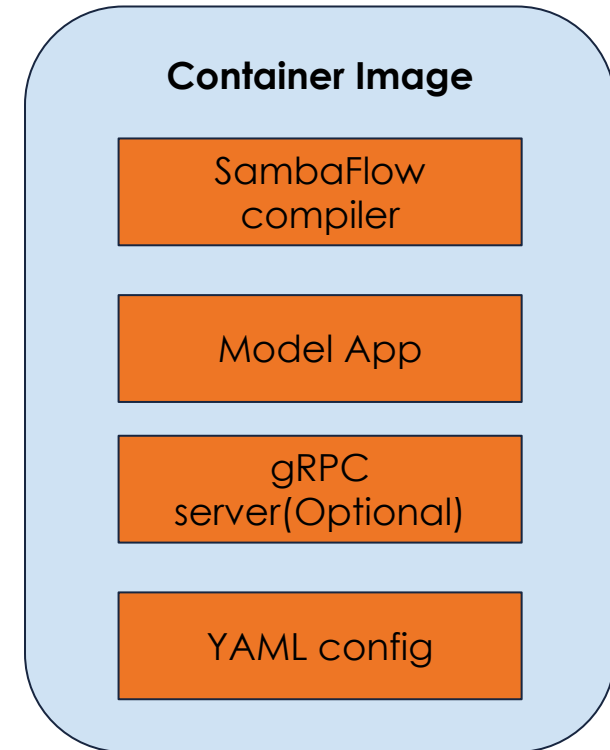
ALCF: Execution phase



Modelbox: virtualized container with model driver code and compatibility for execution phase

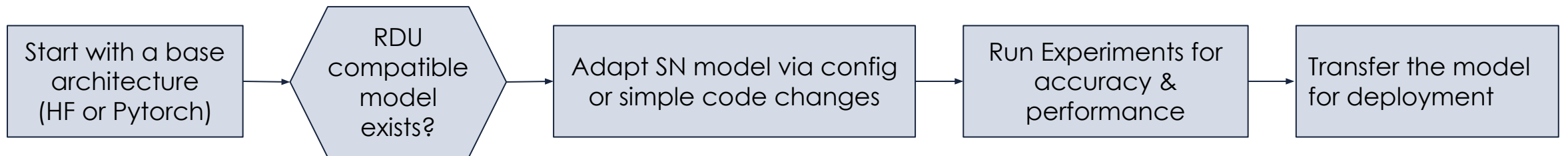
What is a ModelBox

- It is a container image for encapsulating a common API, model and a version of the compiler
- It is essentially a microservice to compile & run a single model on RDU
- Provides a standard set of interfaces to allow consumption of data and outputs
- The bare metal version of the Runtime is used to execute the container
- Container images are docker images that are OCI compliant











Model Zoo Workflow

- The Model Zoo is a library of RDU compatible model source code along with necessary example runner scripts needed for compiling and running the model on RDUs.
- The SambaFlow compiler and other software dependencies made available through a container image
- The Model Zoo is compatible with open source checkpoints (eg: HuggingFace)
- Ideal for further adapting an OSS model that is already ported and released by Sambanova



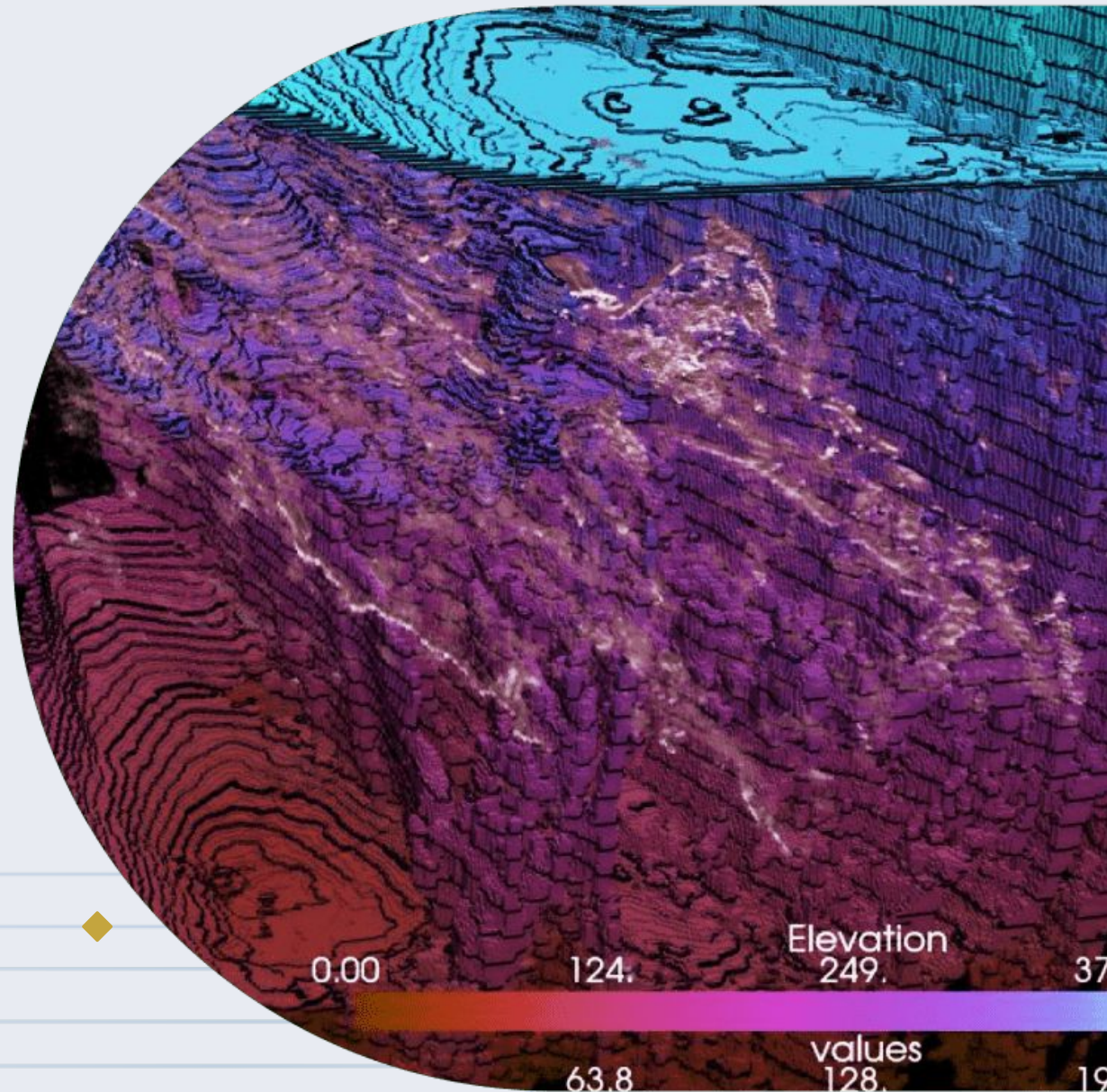
Bare metal vs Model Zoo vs Model Box



	Baremetal	Modelzoo	Modelbox
Mechanism			
Purpose	End2end development	Modify/Develop existing SN model implementations	Execute preverified SN model implementations
Execution			
Compiler			
Modifications			
Model definition(Pytorch)			
Example Use Case	Develop custom model from scratch, e.g. CosmicTagger	Alter/Enhance popular models, such as llama	Train default model definition, such as llama from Hugging Face

Porting Models

PyTorch to SambaFlow



How Does It Work?

- Import your model from PyTorch
- Run **samba.from_torch_model_...** to convert model parameters to SambaTensors
 - + Convert input Torch Tensors with **samba.from_torch_tensor(...)**
- Run **samba.session.compile(...)** to compile the model
 - + Sometimes requires adaptations for compatibility (more details coming up)
- Start running via **samba.session.run(...)** and **samba.utils.trace_graph(...)**

Best Practices, dos and don'ts

- Inputs should be well-defined including internal tensors
 - + Need to allow the compiler to see every symbol
- Try to contain operations to 1 continuous graph
- Avoid control flow within model
- Avoid synchronizing between CPU/RDU too often
- Stick with PyTorch, converting models and tensors as needed

What are SambaTensors?

- Wrappers around Torch Tensors, with special SN capabilities
- Each one gets a unique **name** for interfacing with the chip
- Specify **batch_dim** for optimization
- Methods to transfer to/from RDU
- Can be treated like normal Torch Tensor
 - + e.g., `tensor3 = tensor1 + tensor2`
or `tensor2 = tensor1.reshape(-1, 5)`

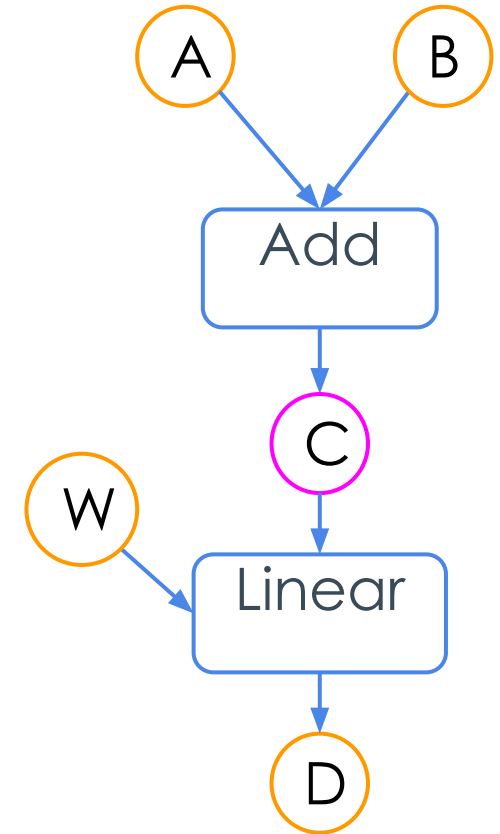
SambaTensor

.data (torch tensor)
.batch_dim (int)
.dtype (torch dtype)
.sn_name (string)
.sn_grad (gradient on chip)

.cpu() (transfer to CPU)
.rdu() (transfer to RDU)

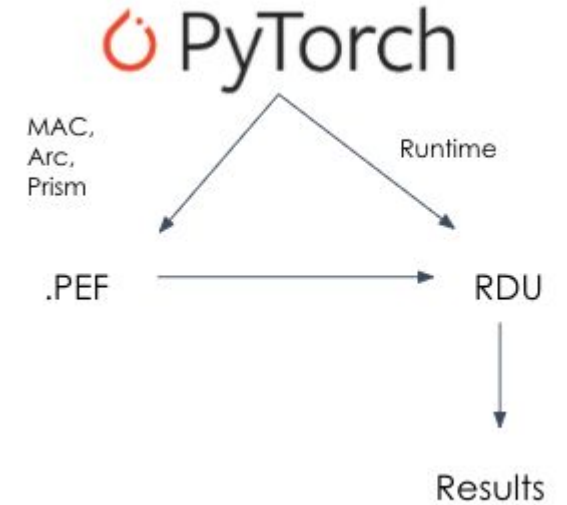
What is tracing?

- Tracing walks through the model with “dummy” tensors to form the graph
- This is carried out automatically during compilation or it can be done manually by using **trace_graph()** before running on RDU
- Beneficial because the RDU+compiler have knowledge of the **full graph** to optimize, not just individual components



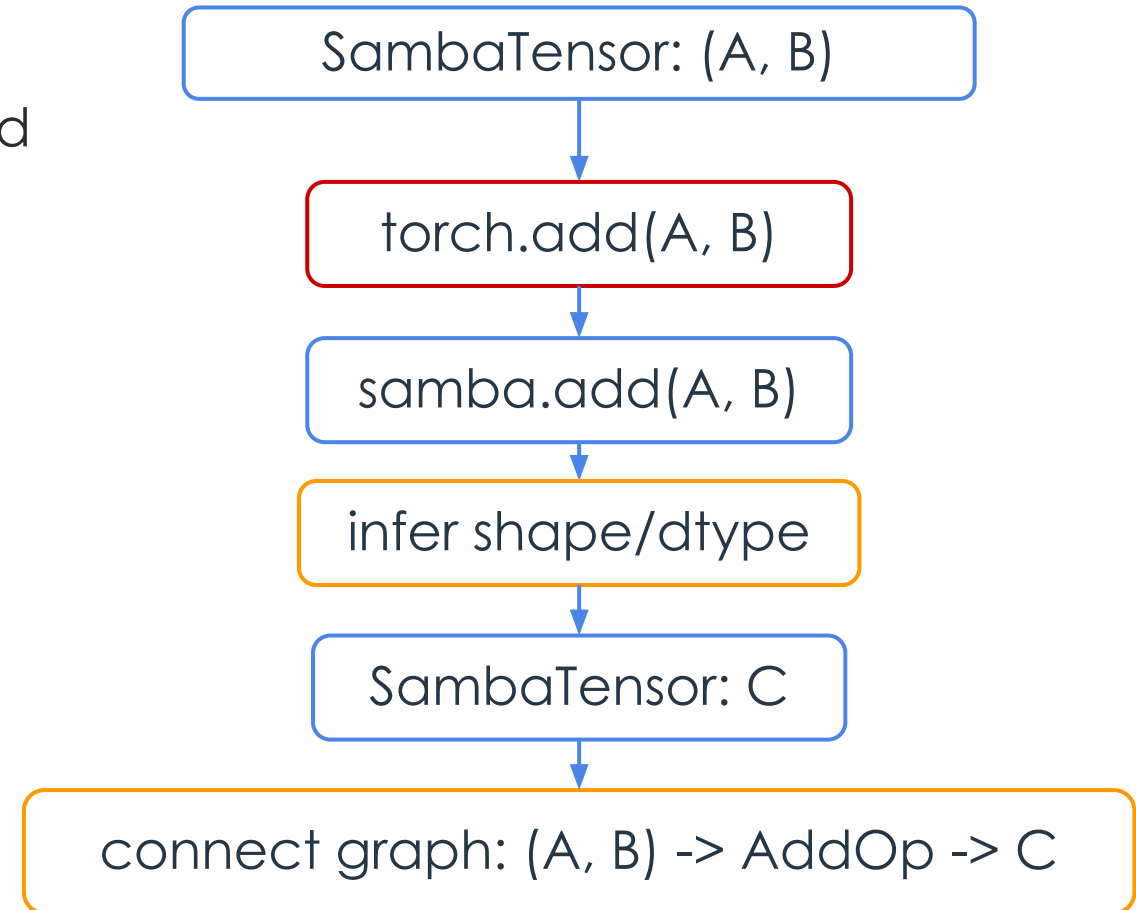
How do we trace your graph?

- Model definition is done in PyTorch
- The model's **forward pass** will determine the graph that is generated
 - + Note that there is no current support for control flow within model
- For compiling: **samba.session.compile**
- For training: **samba.utils.trace_graph**



Functional Overrides with SambaTensor

- SambaFlow understands and supports similar functional overrides as NumPy and PyTorch
- If you pass SambaTensors to a Torch function, SambaFlow will override and call the equivalent SambaFlow method
 - + Check the [SambaFlow API docs](#) for a listing of methods



Sample Code (Model)

- Model code all in PyTorch
- A few restrictions
 - + Modules with parameters are defined before forward
 - + The return type of forward should be simple (tensor, tuple/list of tensors)

```
class ResFFNLogReg(nn.Module):
    """Feed Forward Network with two different activation functions and a residual connection"""
    def __init__(self, num_features: int, hidden_size: int, num_classes: int) -> None:
        super().__init__()
        self.gemm1 = nn.Linear(num_features, hidden_size, bias=True)
        self.gemm2 = nn.Linear(hidden_size, hidden_size, bias=True)
        self.gemm3 = nn.Linear(hidden_size, num_classes, bias=True)

        self.norm1 = nn.LayerNorm(hidden_size)
        self.norm2 = nn.LayerNorm(hidden_size)

        self.tanh1 = nn.Tanh()
        self.sigmoid1 = nn.Sigmoid()

        self.criterion = nn.CrossEntropyLoss()

        self.apply(basic_weight_init)

    def forward(self, inputs: torch.Tensor, targets: torch.Tensor) -> Tuple[torch.Tensor]:
        out = self.gemm1(inputs)
        out = self.norm1(out)
        out = self.tanh1(out)
        residual = out
        out = self.gemm2(out)
        out = self.norm2(out)
        out = out + residual
        out = self.sigmoid1(out)
        out = self.gemm3(out)
        loss = self.criterion(out, targets)
        return loss, out
```

Sample Code (App Code)

- Running is 2 Steps, **compile** then **run**
- Need to pass **model**, **inputs**, and **optimizer** to compile/trace
- Write your own training loop!

```
#Input definition: "Dummy" SambaTensors
image = samba.randn(args.batch_size, args.num_features, name='image', batch_dim=0)
label = samba.randint(args.num_classes, (args.batch_size, ), name='label', batch_dim=0)
inputs = (image, label)

# Model definition
model = ResFFNLogReg(args.num_features, args.hidden_size, args.num_classes)
samba.from_torch_model_(model)

# Optimizer definition
optim = sambafLOW.samba.optim.SGD(model.parameters(),
                                   lr=args.lr,
                                   momentum=args.momentum,
                                   weight_decay=args.weight_decay)

# Compilation & Running, or training, a model must be explicitly carried out
if args.command == "run":
    # Trace the graph
    utils.trace_graph(model, inputs, optim, pef=args.pef, mapping=args.mapping)
    # Within the user defined train function, call: samba.session.run
    train(args, model)
else:
    samba.session.compile(
        model=model,
        inputs=inputs,
        optim=optim,
        name=model.__class__.__name__,
        init_output_grads=not args.inference,
    )
```


Sample Code

- Run the model with **samba.session.run**
 - + Provide all **input_tensors**
 - + Provide traced **output_tensors**
- Running specific sections
 - + **section_ids**
 - + **section_types**
- Sync parameter values between Host/RDU

```
outputs = samba.session.run(input_tensors=inputs,
                             output_tensors=traced_outputs,
                             section_types= ['fwd', 'bckwd', 'opt'])

# Sync the parameter values on rdu back to host
samba.session.to_cpu(model)

# Retrieve individual gradients
linear_grad = model.linear1.weight.sn_grad
```

Running all sections

```
for inputs in dataloader:
    # Running only fwd here for inference
    loss, out = samba.session.run(input_tensors = inputs,
                                  output_tensors = traced_outputs,
                                  section_types = ['fwd'])

    # Can do anything you like with the outputs
    process_inference_results(loss, out)
```

Sample inference loop

Various args and run modes

- Args used internally expressed as command line args
- Some important args for compile/run:
 - + `command`
 - + `--inference`
 - + `--batch-size/-b, --microbatch-size/-mb`
 - + `--pef/-p`

An example compile command:

```
python <app>.py compile -b=64 -mb=4  
--inference -p <app.pef>
```



```
args = parse_app_args(argv)  
args.command == "compile"  
args.batch_size == 64  
args.microbatch_size == 4  
args.inference == True
```

An example run command:

```
python <app>.py run -b=64 -mb=4  
--inference -p <app.pef>
```



```
args = parse_app_args(argv)  
args.command == "run"  
args.batch_size == 64  
args.microbatch_size == 4  
args.inference == True
```

SambaLoader

- The SambaLoader is wrapper around the PyTorch DataLoader
- It helps to improve overall performance by better parallelizing load ops and graph ops
- As a bonus, it returns an iterator over SambaTensors so you don't need to explicitly do that conversion!

Basic Method Structure:

```
SambaLoader(torch_loader: Iterable[Iterable[torch.Tensor]], names: List[str])
```

Where `torch_loader` is an Iterable and `names` is a list of strings to be given to the input SambaTensors

Example:

```
from torch.utils.data import DataLoader
from sambaflow.samba.sambaloader import SambaLoader

data_loader = DataLoader(dataset, batch_size=args.bs,...)
samba_loader = SambaLoader(data_loader, ["sample", "label"])

for X_val, Y_val in samba_loader:
    samba.session.run(input_tensors=[X_val, Y_val],...)
```

More Details

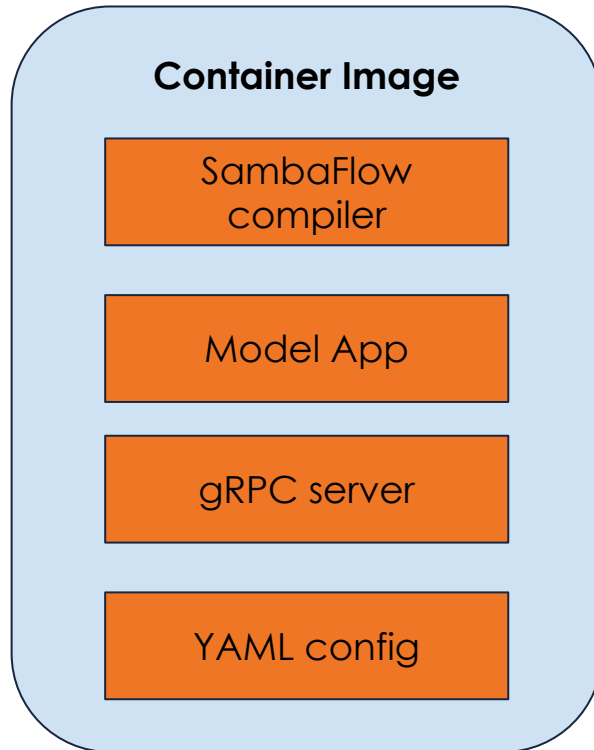
- Get more details on Sambanova Public Docs
 - + [SambaFlow developer documentation](#)
- Examples to try
 - + [SambaFlow Tutorials](#)
- Contact Sambanova Support team
 - + help@sambanova.ai
- Go to the Support Portal
 - + support.sambanova.ai



Appendix



What's inside a model box



- Binary, installed packages of SambaFlow compiler at a particular release version
- Samba application code for the model and required libraries or frameworks to compile & run (e.g. PyTorch)
- A set of gRPC APIs:
 - + Compile
 - + Train
 - + Infer
 - + preprocess
- A YAML configuration file that describes a) the model name, b) named parameters and legal values.

The artifacts like PEFs and Checkpoints are pulled from external and mounted on to the container.

Example Models to run on SN30 Datascale

	Bare Metal	Modelbox
NLP models	GPT1.5B, GPT13B, BERT	Llama V2 7B, 13B, 70B Llama V3 Genslm Mistral Deepseek Coder Falcon
Vision models	AutodiCNN CosmicTagger Unet2D, Unet 3D, DeepVIT, Rescalenet dcrnn Vit	AutophaseNN
Science models	Uno, BraggNN AI4Polymers	-