Ahead-of-time (AOT) compilation of Tensorflow models

Introduction to Tensorflow (TF) with Accelerated Linear Algebra (XLA) and Ahead-of-time (AOT) compilation in CMSSW

Marcel Rieger, Peter Schleper, Bogdan Wiederspan

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

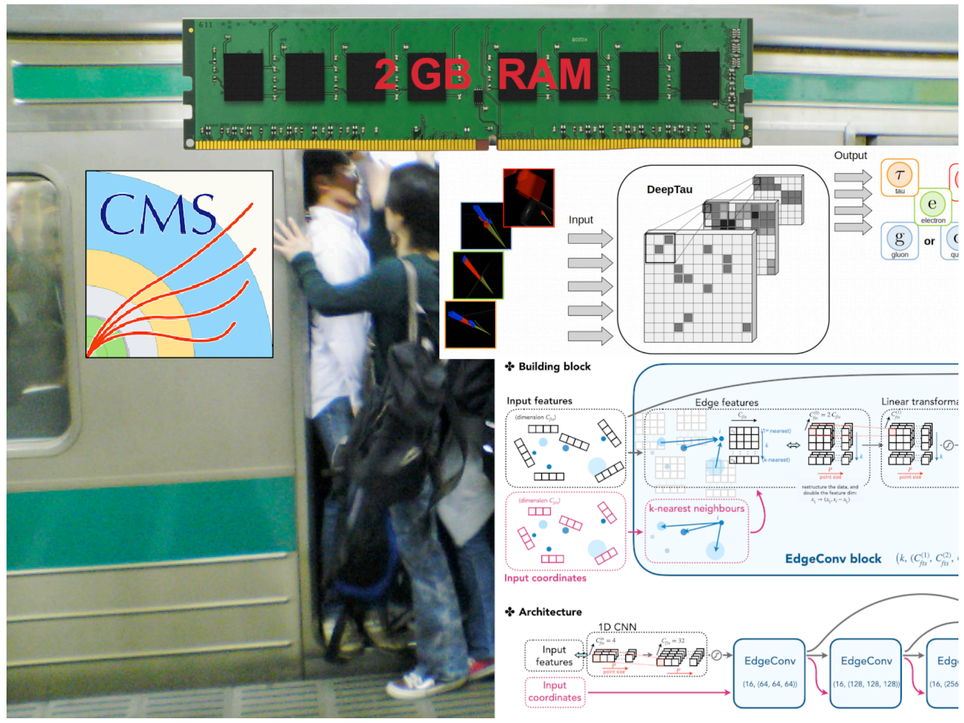
SPONSORED BY THE
Federal Ministry of Education and Research

FSP CMS
Erforschung von Universum und Materie

Motivation: Why CMS care about efficiency

- **CMS workflow**
  - strong memory constrained: **≈2GB per cpu core**
- **Many production models already implemented**
  - e.g. DeepJet, DeepMET, DeepTau (50-100 MB each) ⟶ and even more in the pipeline!

In Summary: its getting crowded

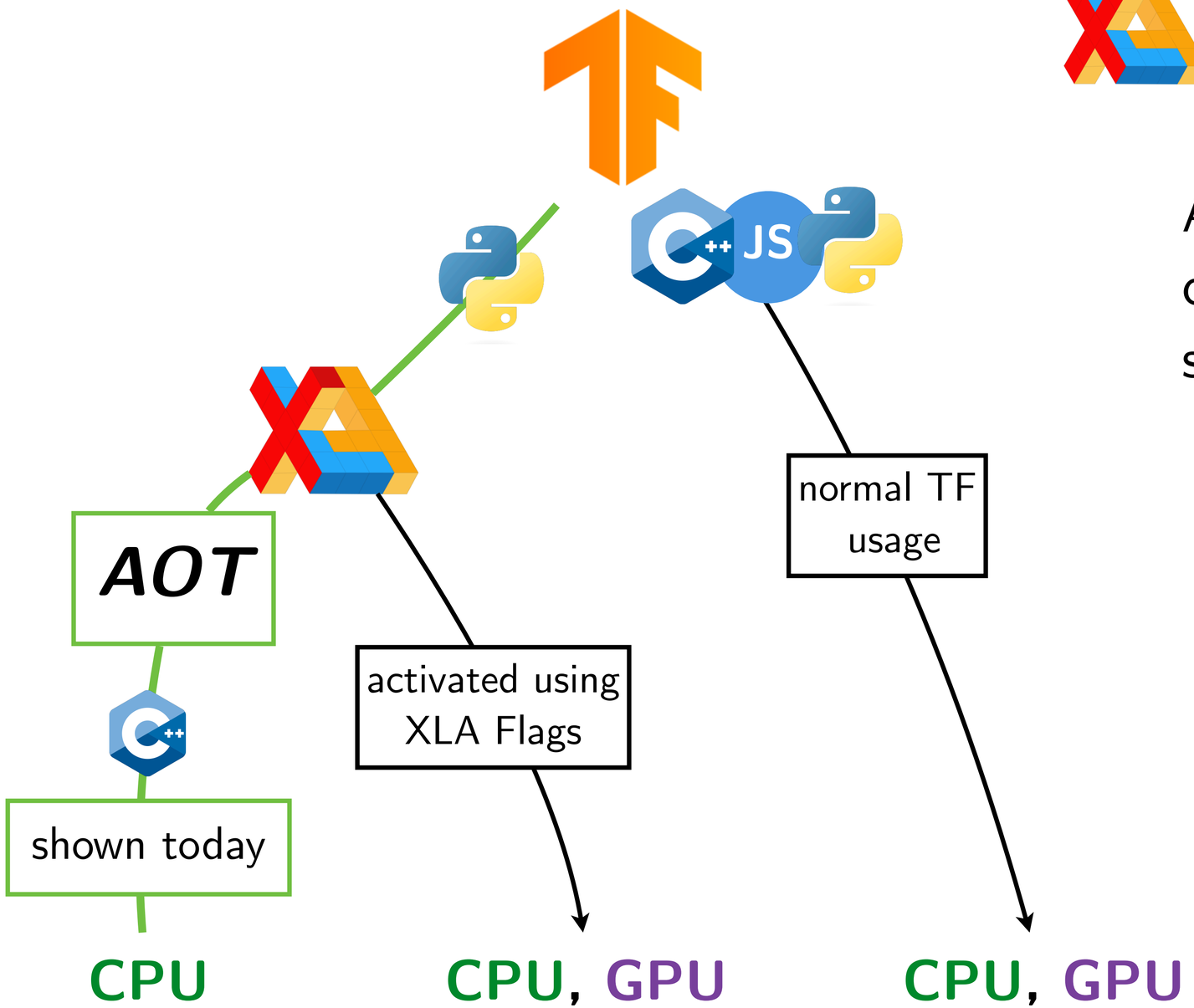


- **Naive solutions**
  - buy more hardware
  - **deploy only "important" models**
  - **more efficient deployment of models (Focus of this talk)**

Overview: Different ways to run Tensorflow (TF) models

Focus of this talk: (left/green path)

deploy **TF models** using **XLA and AOT on CPU**

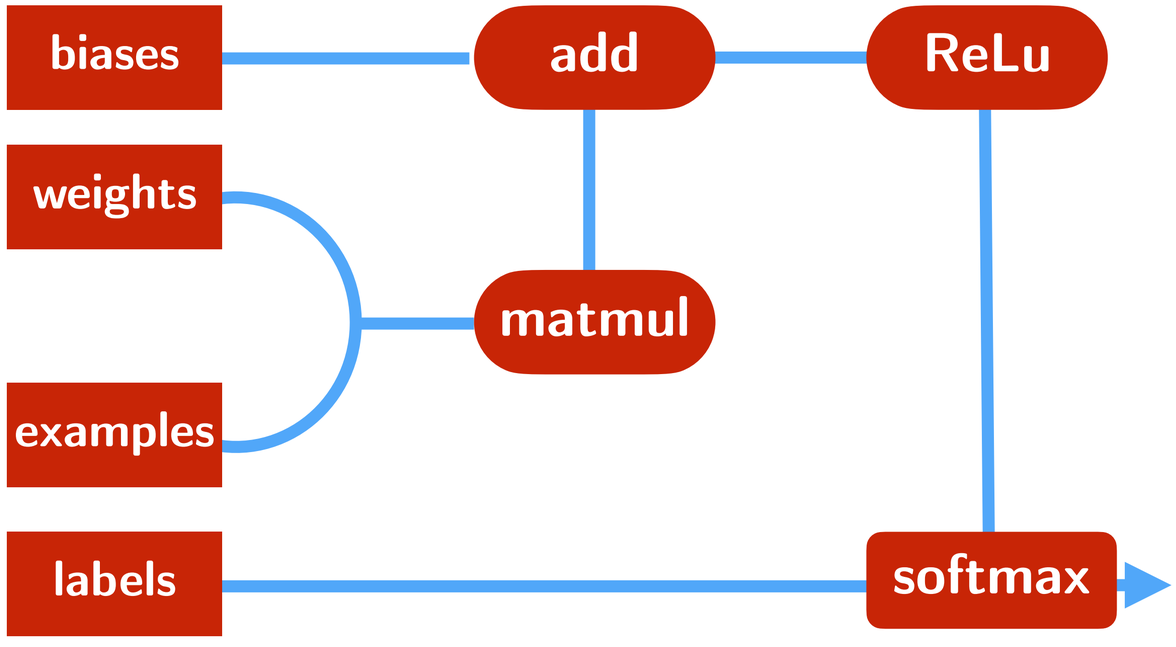


XLA ="Accelerated Linear Algebra"

AOT = "Ahead of time",
compile code at build time into
system dependent binary

How TensorFlow operates in graph mode

- TF generates a data flow graph representing the ML model
- Graphs consist out of kernel and edges
  - **Kernel** represent operations (Add, MatMul, Conv2D, ...)
    - TF runtime, called session, executes graph kernel
    - Operation kernels are written in C++ for CPU or CUDA for GPU
  - **Edges** represent data flowing (Tensors, control dependencies, resource handles, ...)



Example of a 1-layer network

- Beyond modus operandi of TensorFlow: **optimizations (XLA)** and **independence of Tensorflow (AOT)**

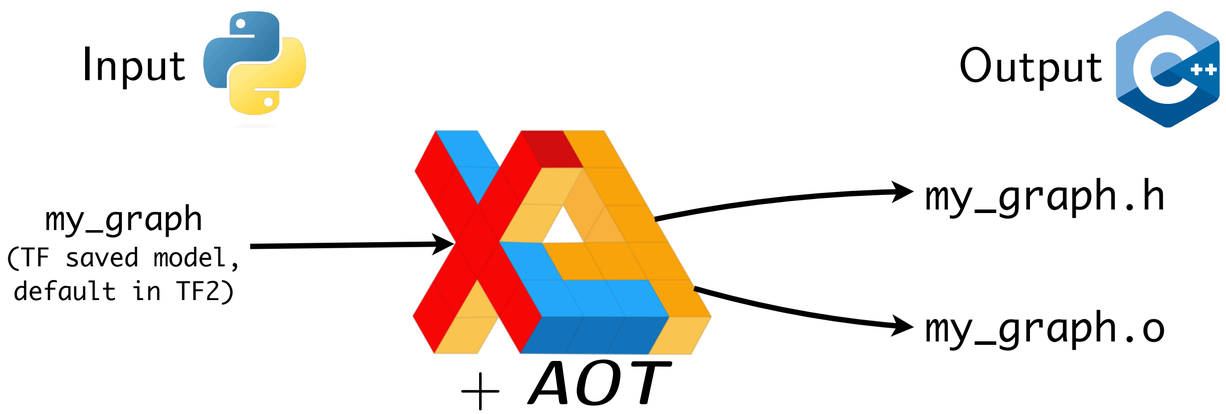
What does XLA and AOT do?

Enables several types of graph optimizations

- On graph level:
  - kernel fusion (main speed benefit)
  - Buffer analysis for allocating runtime memory (eliminates intermediate caches)
  - Common subexpression elimination
  - Pruning of unused kernel
- On hardware level:
  - TPU, GPU or CPU (different backends)
- Universal: JAX, PyTorch and ONNX use XLA

AOT

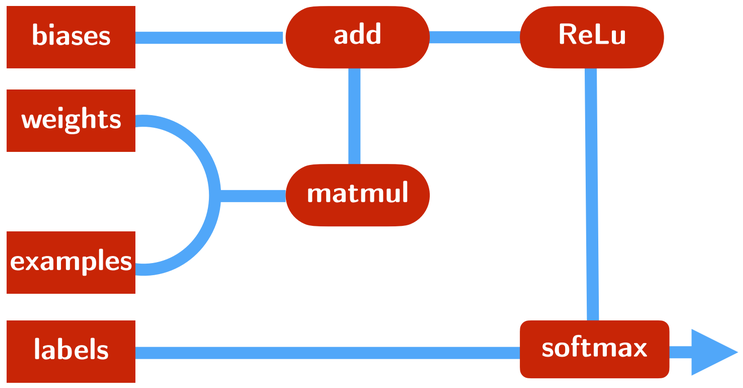
- Converts graphs into **self-contained library (header-object-pair)**
  - Graph becomes series of compute kernels in C++
  - **No need to load TensorFlow**



AOT Compilation workflow summary

Step 1: Prepare graph

save model with fixed batch size
(static memory layout)



Step 2: Tensorflow Optimization

optimization level
configurable

MetaOptimizer

Step 3: XLA Optimization

optimization level
configurable

Step 4: AOT compiled using TF "saved_model_cli" tool

Input

```
XLA_optizmized
static my_graph
```

AOT

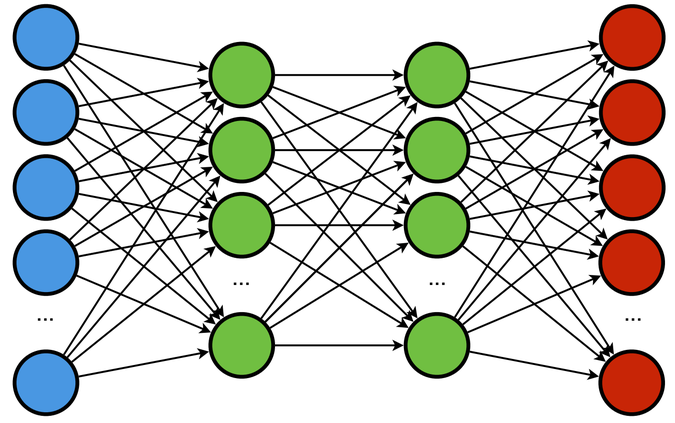
Final Output

my_graph.h

my_graph.o

CPU runtime and Memory Study

Performance study outline



Network and payload

- Created several feed-forward toy models
  - 12/25 layers, 128/256 units, batch-norm and SELU activation

1. CPU runtime tests performed on login-node with CMSSW:

- Compared **forward pass runtime** of **TF C++ vs. AOT in 1 SingleThread scenario**
  - inputs of the network are random values
  - Averaged runtimes over 500 calls, after 100 "warm up" calls (to eliminate caching effects)
  - Tested **event forward time** for different batch sizes

2. Memory tests using MemoryProfiler (IgProf)

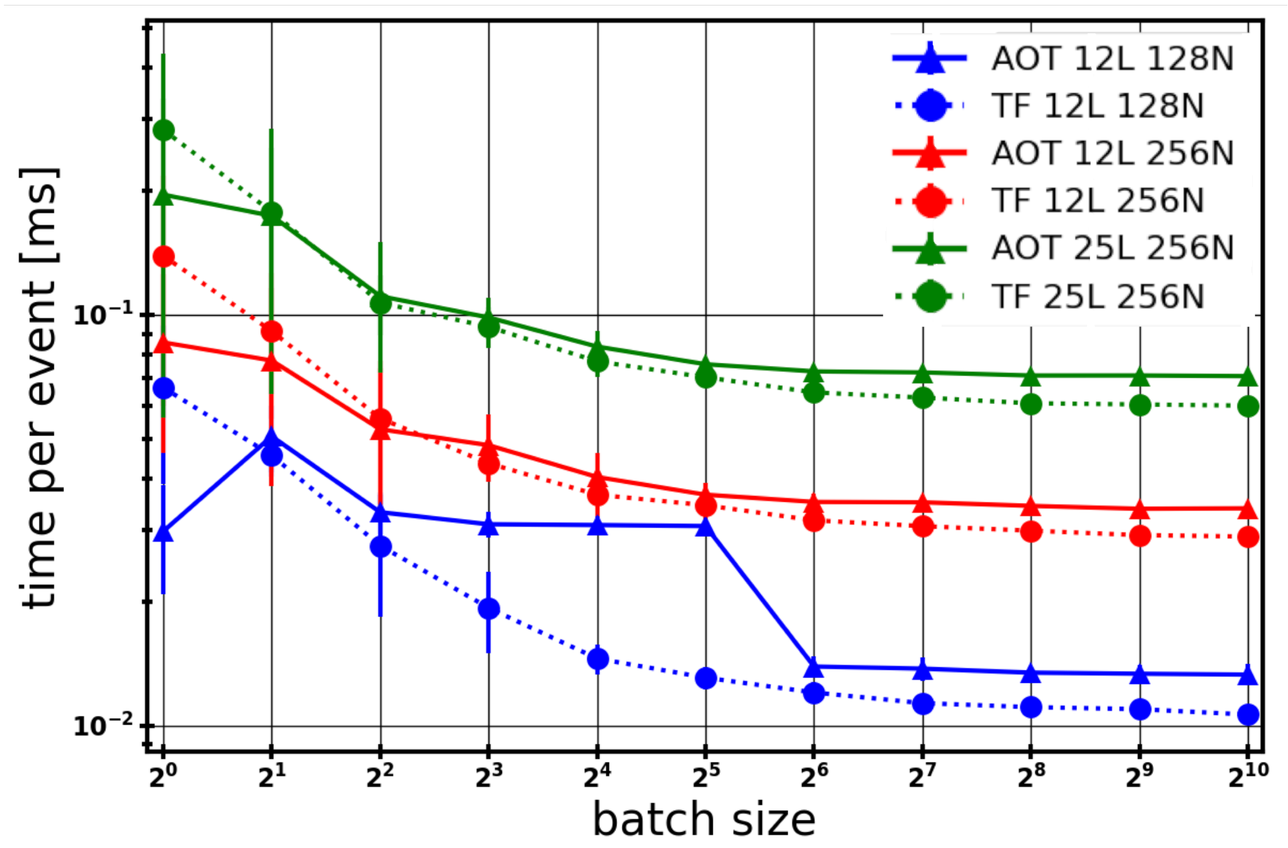
3. Development of batching strategies for different scenarios and compare CPU runtime

(not shown (due ⏱): study can be found here)

CPU performance

- big batch sizes can be vectorised (decrease of time per event, till reaching saturation)
- **In vectorised regime:** TF slightly better than AOT
- **Single batch: AOT is always better than TF**

- **overall comparable performance between C++ TF and AOT**
  - **default XLA** optimizations applied ➡ bare minimum



L = # Layer
N = # Nodes

dotted = TF
solid = AOT

(Link: cpu performance study)

Memory reservation comparison

AOT reserve ...

- **only once** its models weights (*.o file)
- **every time** buffer for **input**, **output** and **intermediate layer** (scales with batch size)

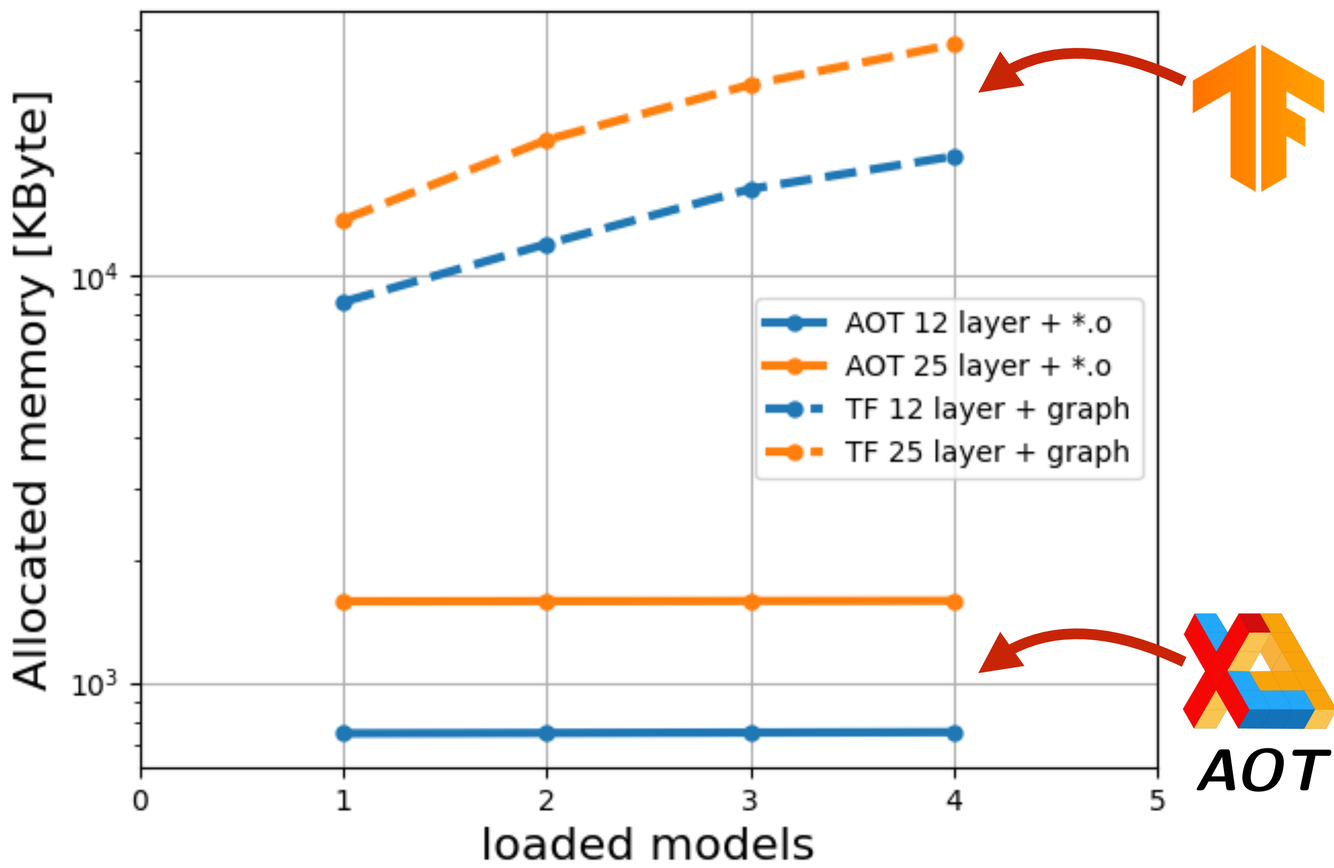
TF reserve ...

- **only once** its graph=(models weights + meta data)
- **every time** its session=(runtime enviroment)
- **every time** buffer for input and output (not included here)

In comparions:

- **AOT size by factor 10 smaller than TF**
  - e.g. DeepTau would be 10 MB instead of 100MB
- AOT scales great with number of loaded models
  - barely seeable slope
- AOT fully independent of TensorFlow at runtime, saves ~300 MBs on top
  - saves also 1 min of loading TF

Memory usage comparison
between 12 and 25 layers networks



(Link: memory study)

Limits of AOT compilation

When does AOT compilation fail?

Possible reasons to fail AOT compilation:

1. Kernel produce **no predictable shapes** (=no fix memory layout) at compile time (e.g tf.where)

   ➡ known model affected by this: LSTMs without padding

2. No existing TF XLA implementation of the kernel

   - frequent checking: more XLA kernels added with each TF update

Created tool to check
if models op nodes
has XLA implementation

example:
AOT compatibility of
feed-forward network for old TF v. 1.4

Operation	has XLA
MatMul	True
Identity	True
Sub	True
BiasAdd	True
Mul	True
Softmax	True
NoOp	True
Rsqrt	True
Selu	True
AddV2	False
ReadVariableOp	True
Const	True

network is not
AOT compatible

(Link: compatiblity study)

What was done:

- shown new compilation and deployment method for Tensorflow graphs
- showed that AOT is on average comparable fast as TF C++ models
  - AOT open gates to more optimization with different XLA level
- AOT memory footprint of models is about factor 10 smaller than TF

Your turn with AOT:

- Dev tool to check model compatibility (here)
- Dev tools for preparation and compilation (here)
- Example for compilation without helper tools and basic use (here)
- Recommended usage: Use an wrapper made by us (here)
  - wrapper takes care of:
    - dynamic types
    - emulation of batch sizes
    - pointer handling

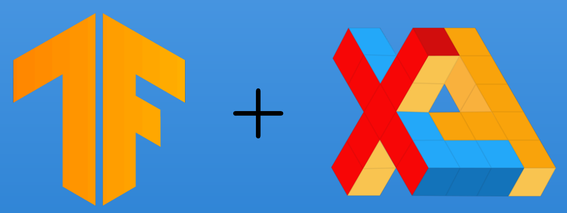
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

CLUSTER OF EXCELLENCE
QUANTUM UNIVERSE

SPONSORED BY THE
Federal Ministry
of Education
and Research

"Ahead-of-time" compilation of Tensorflow models

Marcel Rieger, Peter Schleper, Bogdan Wiederspan
Email: bogdan.wiederspan@uni-hamburg.de



Thank you for your attention!

FAQ: in descending frequency

- **Can you use this on PyTorch models?**
  - No. You need to rebuild your model in TensorFlow
  - Well actually, there is an experimental PyTorch AOT (here)
- **You said "no predictable shapes" are a problem, can you convert LSTM, RNNs etc.?**
  - No, but padded time-series-networks (fixed shape) should work in theory (we didn't tested this yet)
- **You only showed that this works for FeedForward Model what about other model architectures?**
  - Testing all models is impossible, currently FeedForward and DeepTau V1 were converted successfully
  - Harsh truth: Welcome on Board Beta-Tester!
- **What about support for different CPU Architectures?**
  - can be extended if LLVM supported is there, (see here), x64, ARM is tested by google
    - ▻ given by target-triple string: "ARCHITECTURE-VENDOR-OPERATING_SYSTEM"
    - ▻ e.g. (x86_64_pc_linux)

BackUp FPS 2022

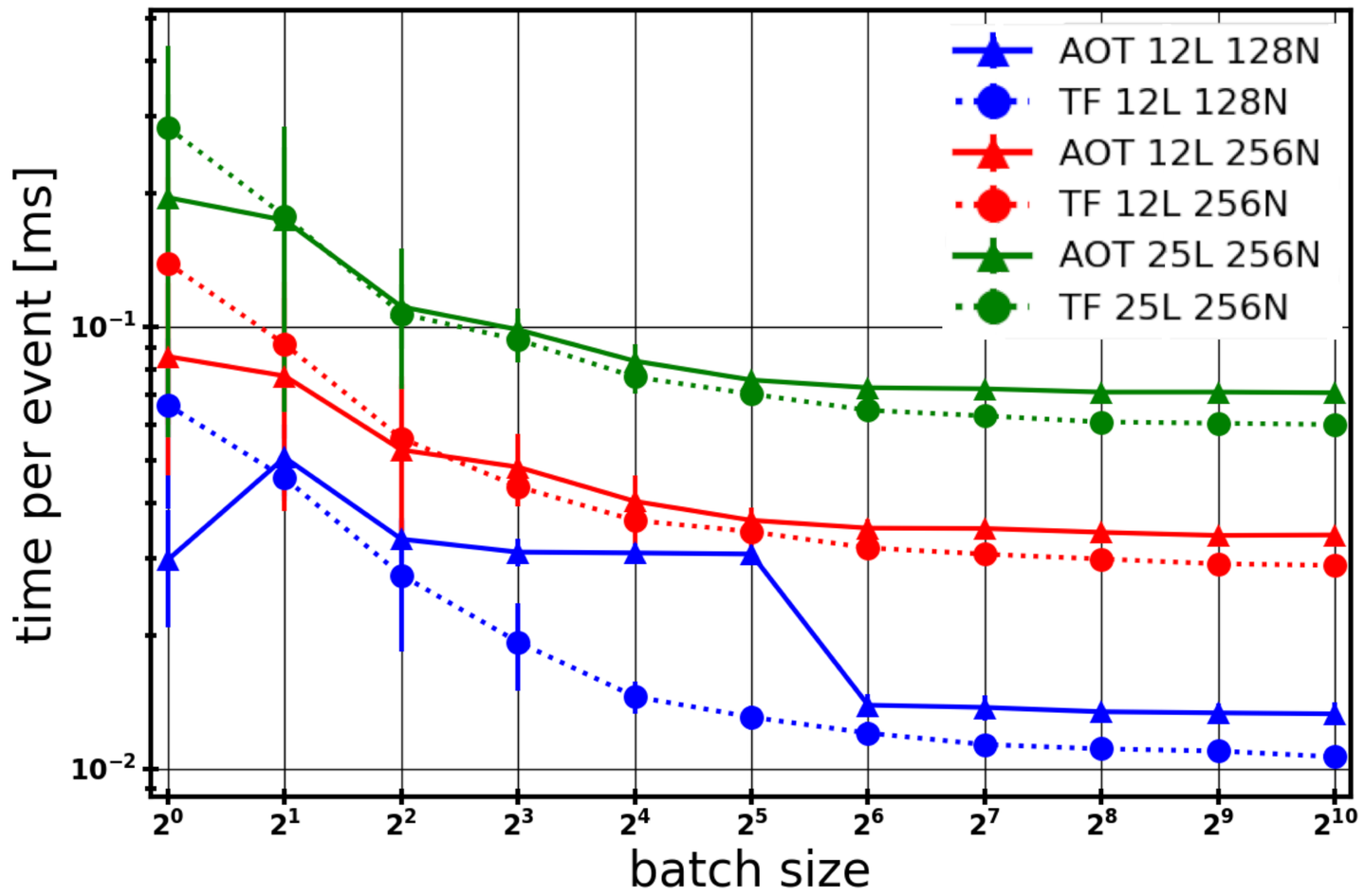
CPU performance

1. Runtimes

- bigger batch sizes can be vectorised (decrease of time per event, till reaching saturation)
- **Equal performance between TF and AOT**
  - compare same color lines
    dotted (TF) vs solid (AOT)
- **batch size 1 AOT is always better than TF**

2. Multi-threading

- Performance test restricted to 1 thread
- CMS production models run in single thread



Memory comparison AOT vs TF

memory comparison
TF graph vs AOT model

N layers	TF graph [kB]	AOT model *.o [kB]	trainable weights [kB]
12	1997	749	768
25	3539	1581	1619

- TF graph more than 2 times bigger than AOT
  ⟶ TF graphs contain many meta information
- AOT model consist of (pruned) trainable weights

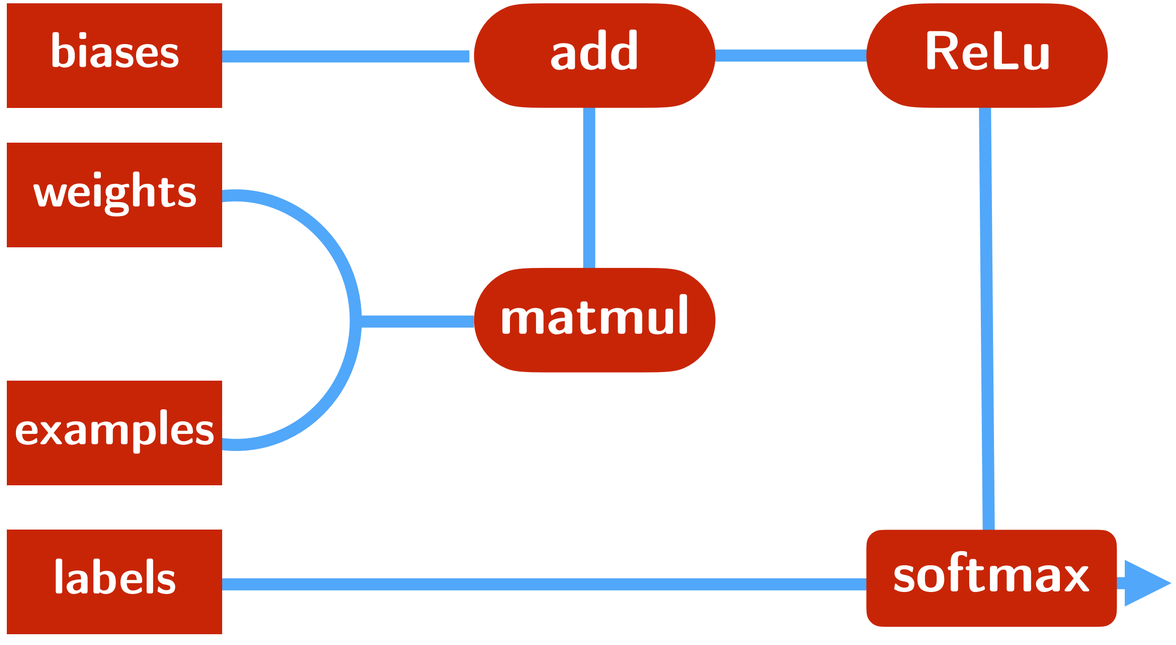
total memory consumption
AOT (batch size 1) vs. TF (loading 1 model)

N layers	AOT model *.o + Wrapper [kB]	TF Session + Graph [kB]
12	751	8599
25	1583	13688

- **AOT by factor 10 smaller than TF**
- AOT wrapper = buffer for **input**, **output** and **intermediate layer** (scales with batch size)
- wrapper size = total - model size ⟶ small (2kB)
- TF I/O tensors are not included in the measurement
- AOT fully independent of `libTensorflow.so` at runtime, saves ~300 MBs on top

How TensorFlow in graph mode operates

- TF generates a data flow graph representing the ML model
- Graphs consist out of kernel and edges
  - **Kernel** represent operations (Add, MatMul, Conv2D, ...)
    - ▻ TF runtime, called session, executes graph kernel
    - ▻ Operation kernels are written in C++ for CPU or CUDA for GPU
  - **Edges** represent data flowing (Tensors, control dependencies, resource handles, ...)

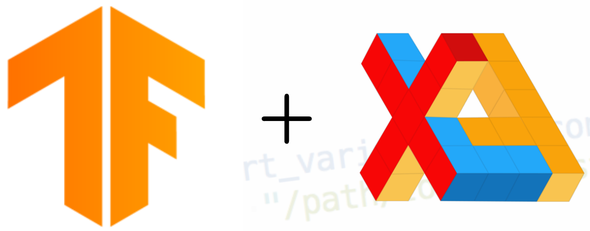


Example of a 1-layer network

- We are going: beyond modus operandi of TensorFlow: **optimizations (XLA)** and **independence of Tensorflow (AOT**

Old Presentations

FSP 2022



in CMSSW

Ahead-of-time (AOT) compilation of Tensorflow models

First steps with Tensorflow (TF) with
Accelerated Linear Algebra (XLA) and
Ahead-of-time (AOT) compilation in CMSSW

Marcel Rieger, Peter Schleper, Bogdan Wiederspan

SPONSORED BY THE
Federal Ministry of Education and Research

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bigger picture: The "usual" workflow

- Workflow:
  - Train your model in python ⟶ environment of **your choice**
  - Run the trained model (e.g. for your analysis) in **the same environment**

Training engine
(Python in *your* env.)

TensorFlow+Keras

PyTorch

scikit-learn

XGBoost

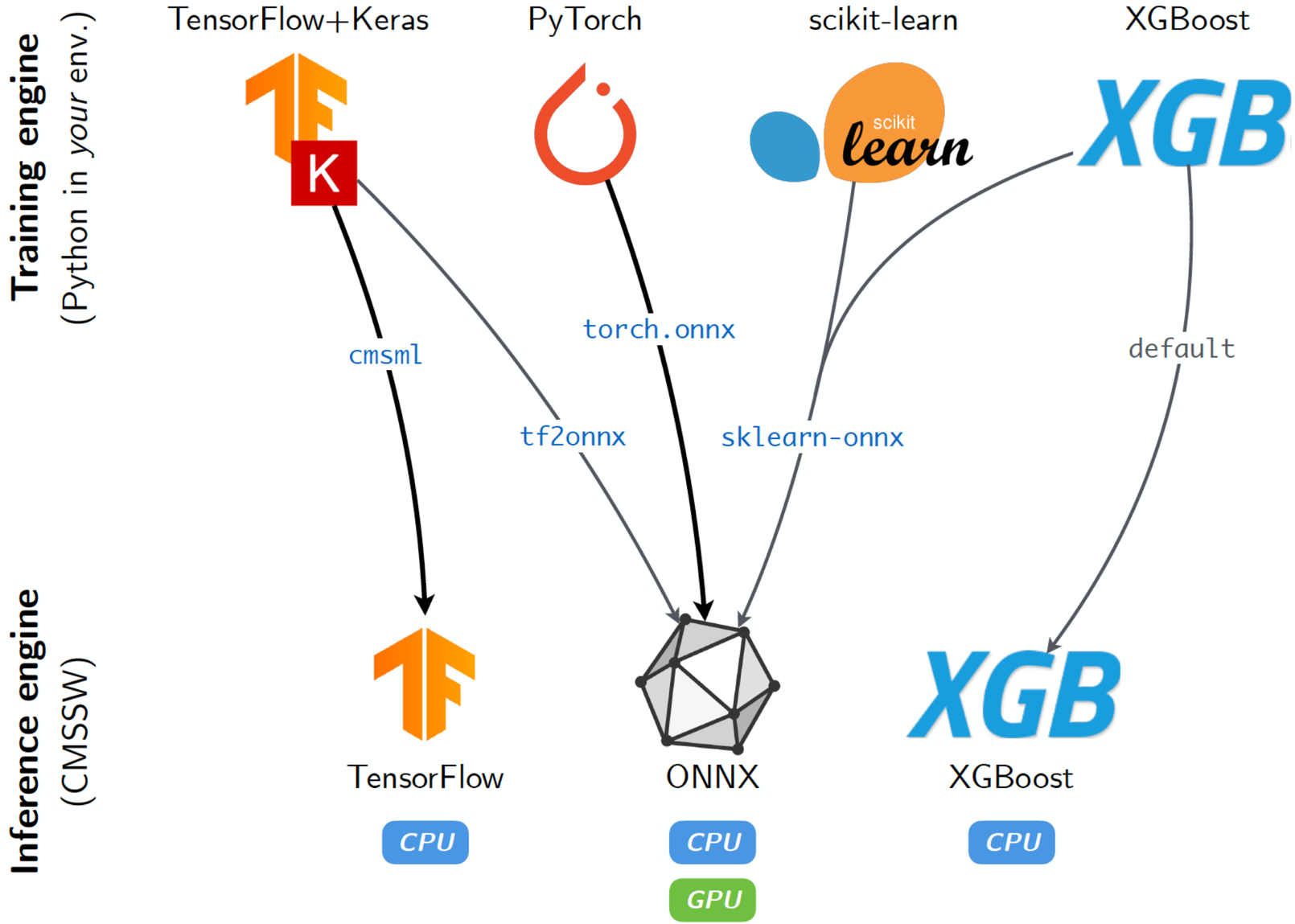
This talk: Inference in CMSSW

- CMS runs your model in CMSSW (through an *inference engine)*
- Why we should care: **limited memory resources per core (≈2GB)**

- **Many models already implemented**
  - DeepJet tagging
  - DeepFlavor
  - DeepMET
  - ....

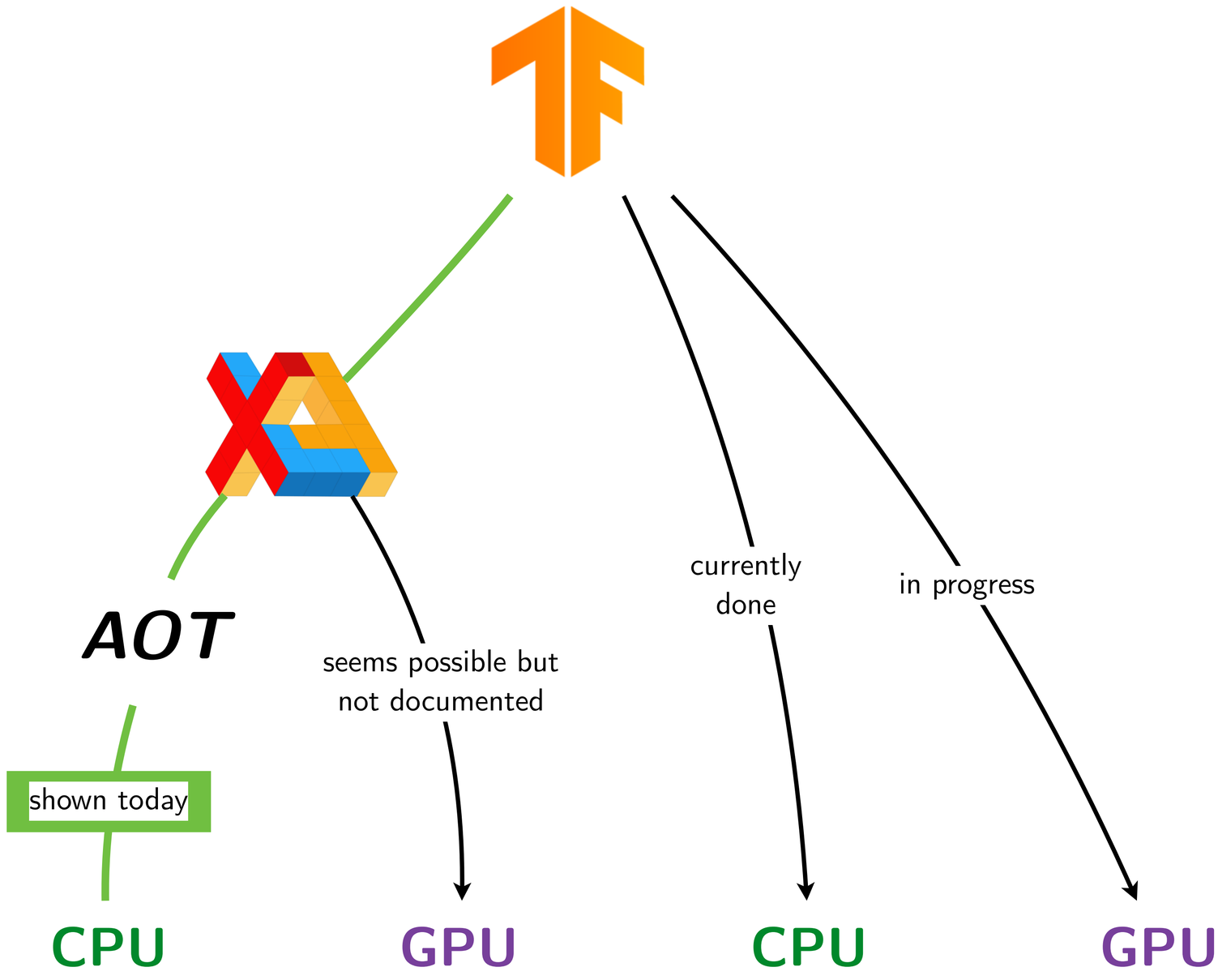
  → and even more in the pipeline!

- **Naive solutions**
  - buy more hardware
  - **deploy only "important" models**
  - be more efficient



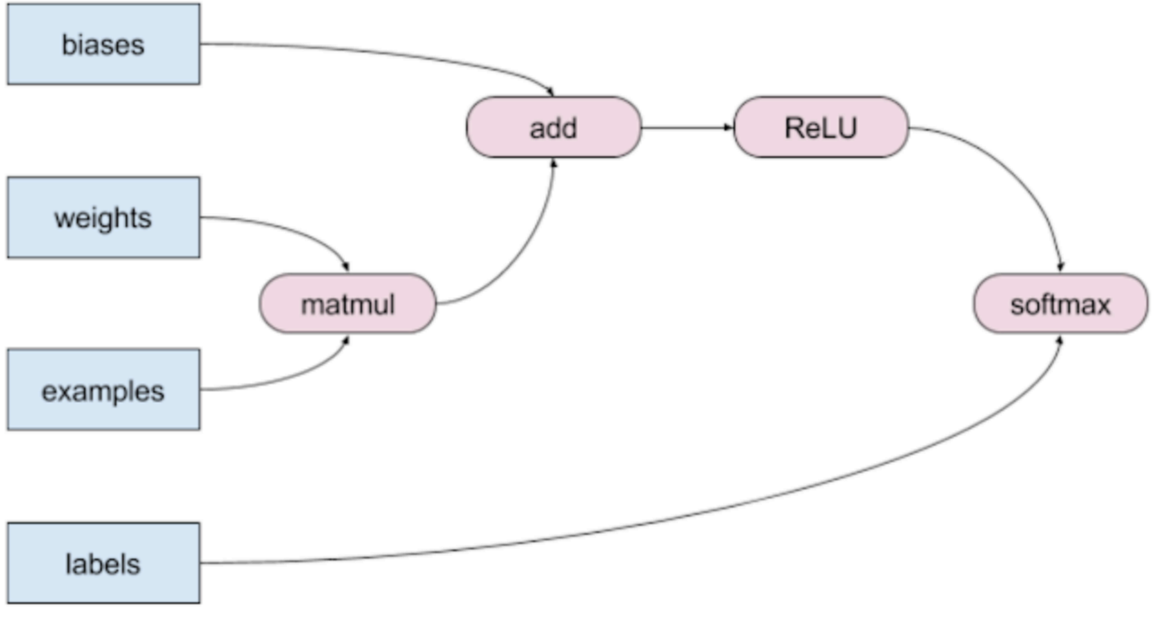
Bigger picture: Tensorflow in CMSSW

This talk focuses on: being more efficient with **Tensorflow models** using XLA and AOT on CPU (left/green path)



How TensorFlow operates

- TF generates a data flow graph representing the ML algorithm (model)
- Graphs consist out of nodes/kernel and edges
  - Nodes represent operations (Add, MatMul, Conv2D, ...)
    - ▻ TF runtime execute graph nodes
    - ▻ Operation kernels are written in C++ for CPU or GPU (e.g. with Cuda)
    - ▻ Execution runtime depends on the number of calls and complexity of the kernel
  - Edges represent data flowing (Tensors, control dependencies, resource handles, ...)



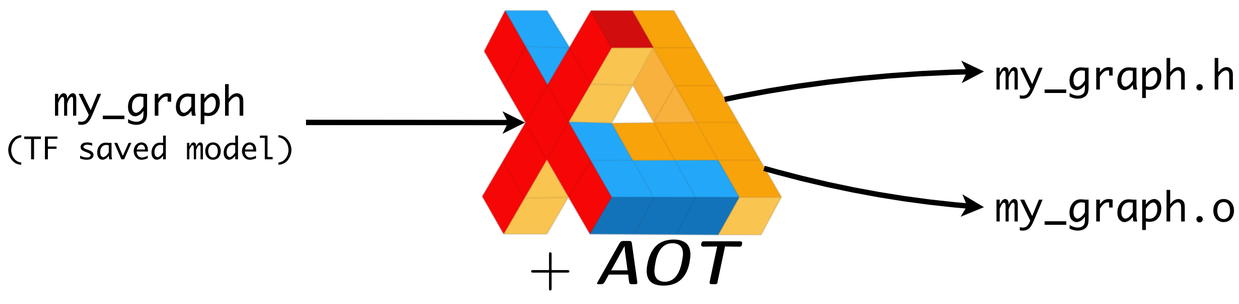
Example of a 1-layer network
Nodes are math operations and placeholder variables, connecting lines are edges

- This how tensorflow currently operates! The new part is: **optimizations (XLA)** and **independence (AOT)**

XLA and AOT

XLA

- Compiler framework for linear algebra
- Enables several types of **graph optimizations**:
  - Hardware-dependent:
    - ▻ TPU, GPU or CPU
  - Hardware-independent:
    - ▻ Operation/kernel fusion
    - ▻ Common subexpression elimination
    - ▻ Buffer analysis for allocating runtime memory
    - ▻ Pruning of unused nodes

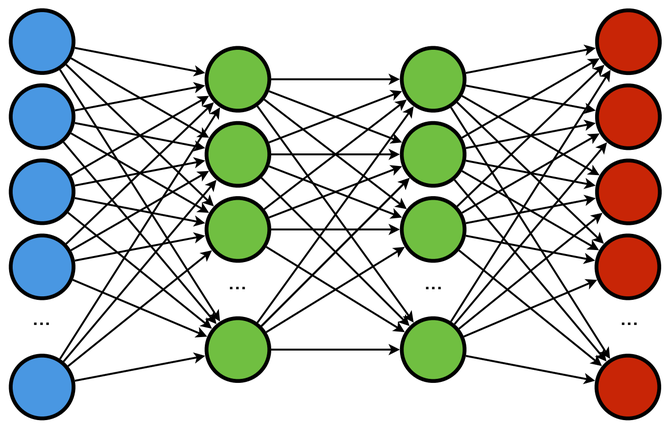


AOT

- Converts graph into **self-contained library**
  - Graph becomes a series of standalone compute kernels
  - No dependence on main `libtensorflow.so`
- Pros (more on next slides):
  1. Reduced memory footprint
  2. Trivial multi-threading behavior
  3. Runtime potentially faster ⟶ depends on degree of enabled XLA optimizations and model
- Cons:
  - No dynamic batching, but can be conveniently emulated (stitching)
  - models with unpredictable shape at compile time are not AOT compatible
    - **– DeepTau (combined) already AOT compiled!**

Study outline

- **Software stack**
  - `slc7_amd64_gcc10` with `CMSSW_12_4_0`
  - Using custom CMSDIST stack with TF XLA enabled and patched "eigen" library
- **Network and payload**
  - Created several feed-forward toy models
    - ▻ Up to 25 layers with 256 units, SELU activation
  - **CPU runtime tests performed on login-node:**
    - ▻ **Compared forward pass runtime of TF and AOT in CMSSW**
    - ▻ Averaged runtimes over 500 calls, after ~100 "warm up" calls (measured with plain `std::chrono`)
    - ▻ Tested **event forward time** for different batch sizes from 2^0 to 2^{10} (saturation)
    - ▻ **No XLA** optimizations applied so far
  - **Memory tests using IgProf:**
    - ▻ **Measure memory consumption for setup phase in multithreading scenario**
    - ▻ **Compare TF multiple sessions vs. multiple AOT models**



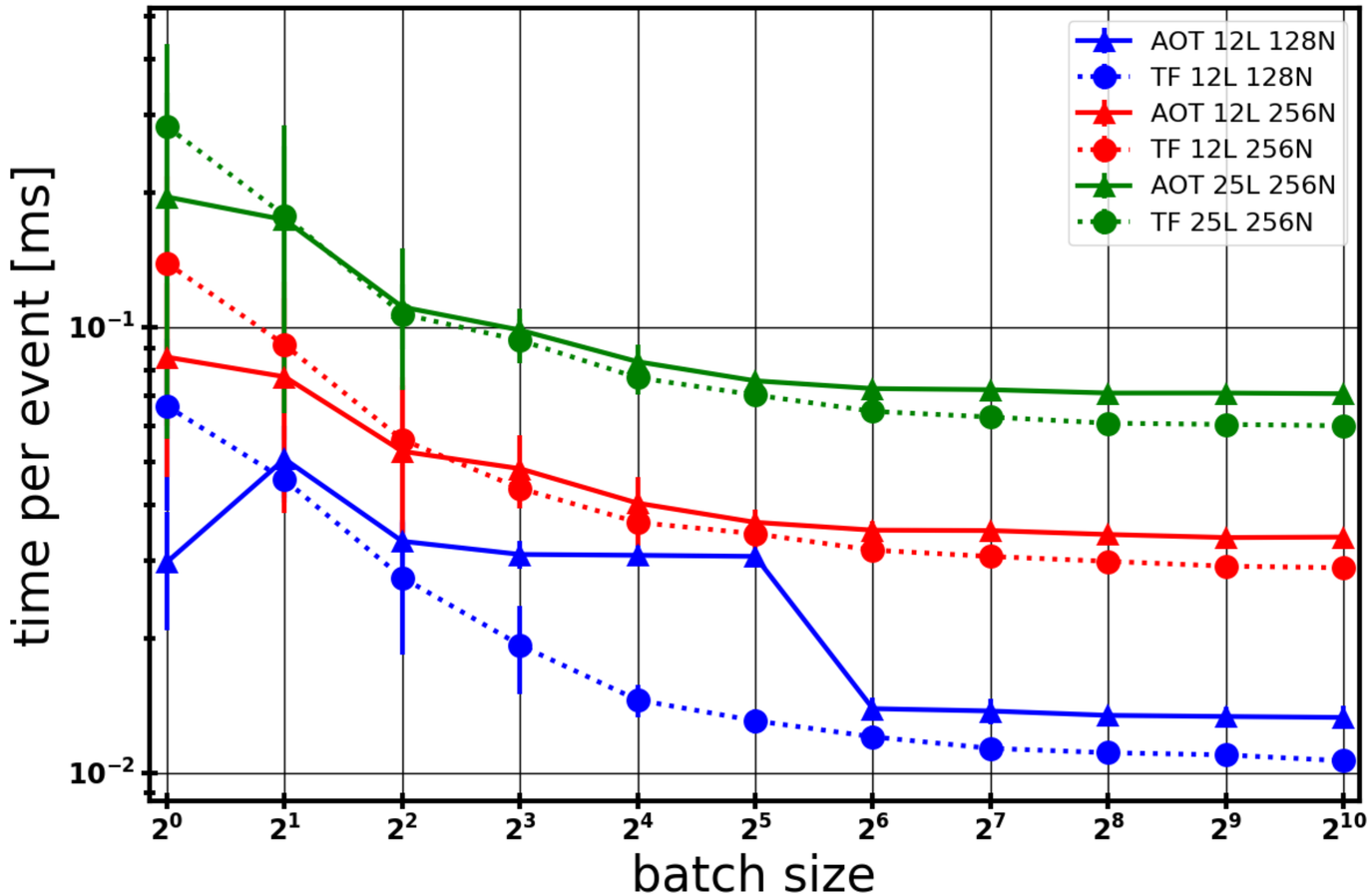
CPU performance

1. **Runtimes**
   - **Equal performance!**

     (comparison same color plots)
   - **AOT without XLA optimizations**
     - ▻ **see bare minimum**

2. **Multi-threading**
   - CMSSW restrict program to 1 thread
   - Weights and series of compute kernels exist **once** in memory (`extern "C"`)
   - accessible by TF through lightweight wrapper
     - ▻ can be loaded into multiple threads
     - ▻ **negligible** multi-threading overhead

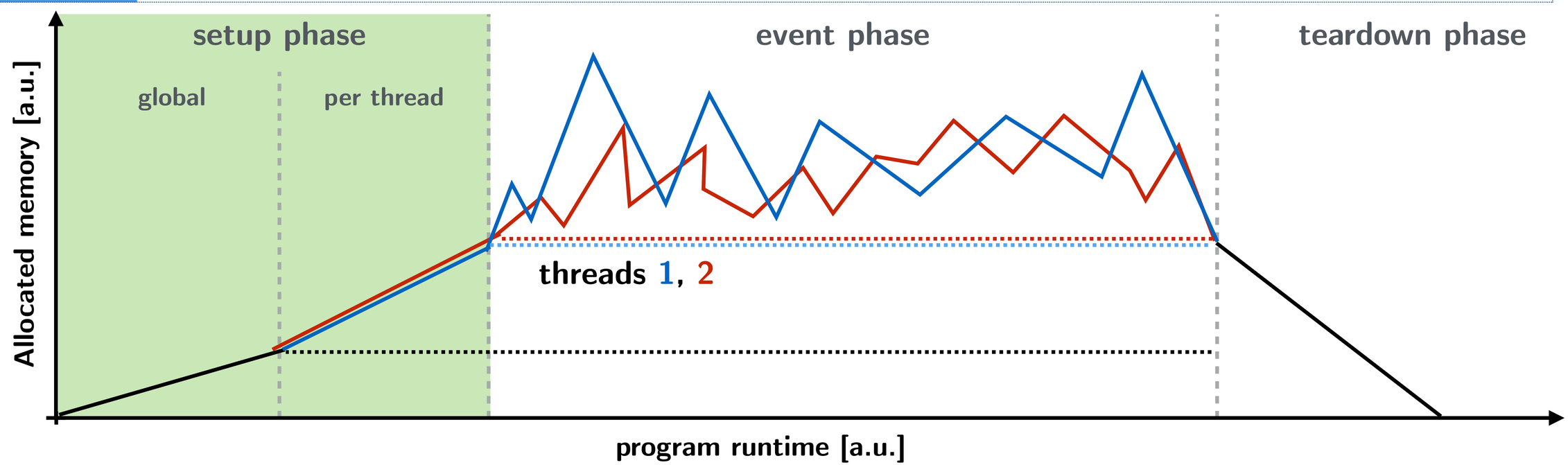
       (see next slides)



Understanding of memory for different phases

- Talk: **Focus on setup phase**, since we can not measure event phase yet.
- global allocate memory once, per thread for each thread
- Comparison: **TF multiple sessions vs. loading multiple AOT models**

Phase	Description	Action in TF	Action in AOT
global setup	Before threads (stream modules) are launched	`tf::loadGraph();` (load model and weights into memory)	loading compiled model (external c-function in *.o file)
setup per thread	Footprint per thread, but before events are analyzed	`tf::loadSession();` (device placement & caching per thread)	`CppWrapper w;` (access to c-function & reserve buffers for input, output)
event phase	Resource consumption during event processing	`tf::run(session, ...);` (book inputs/outputs & evaluate model)	`w.run();` (evaluate model)



TensorFlow setup phase

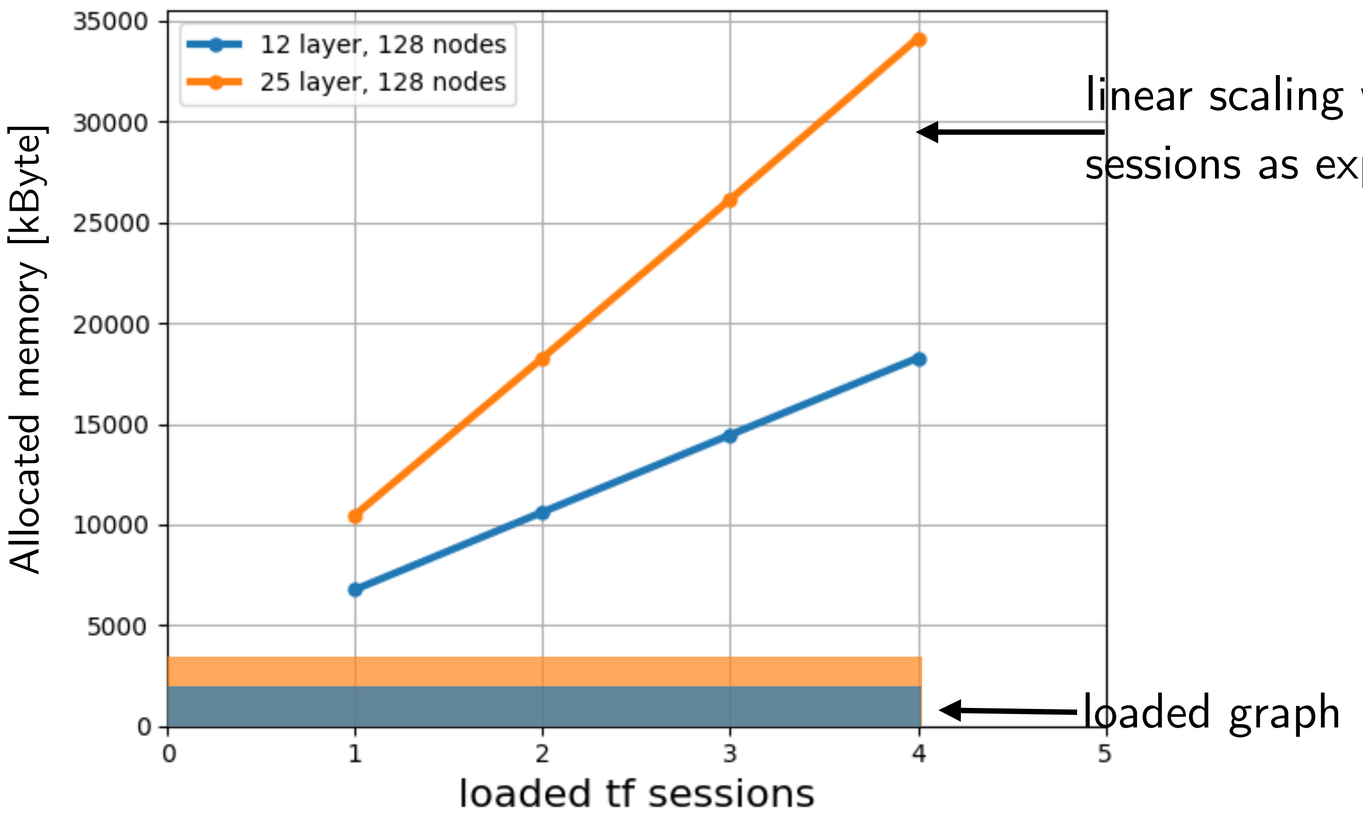
- Launch multiple sessions with same frozen graph
  - Load graph once (global)
  - Copy graph into session (per-thread)
- Does not include creation of tensors yet!

```cpp
void PerfTesterTF::loadGraph() {
  graph_def = tensorflow::loadGraphDef(graph_path);
}
```

```cpp
void PerfTesterTF::loadSessions() {
  std::vector< tensorflow::Session* > loaded_sessions;
  loaded_sessions.reserve(number_of_loaded_sessions);
  for (int i = 0; i < number_of_loaded_sessions; i++) {
    session = tensorflow::createSession(graph_def);
    loaded_sessions.push_back(session);
    }
}
```

code to load graph and sessions

Memory allocation of `loadSessions()`



Graph contains plenty of meta data

N layers	`loadGraph()` [kB]	trainable weights [kB]
12	1997	768
25	3539	1619

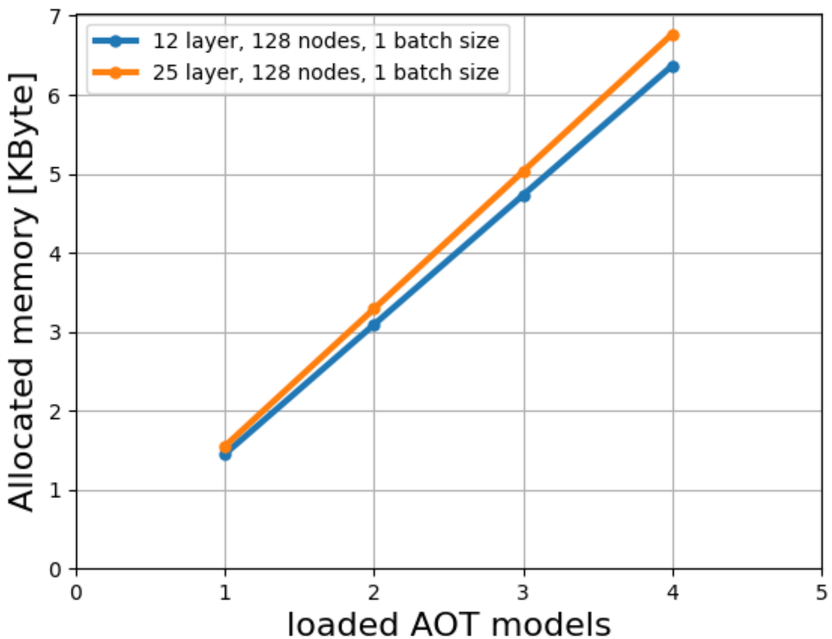
AOT setup phase

- Load multiple AOT models of same batch size
  - Cpp wrapper created to call c-function (per thread)
    - ▻ Reserves **buffers** for **inputs**, **outputs per model**
    - ▻ off-set = model depending **(slope)**
  - Reserves buffer for **intermediate layers once (*.o)**
- Important: each Cpp wrapper handles only one batch size
  - c-function with model weights might be shared between different batch sizes (check ongoing, important for stitching)
  - cpp wrapper is neglectable small compared to layers

c-function consists mainly of model weights
difference is result of pruning (bias and batch norm)

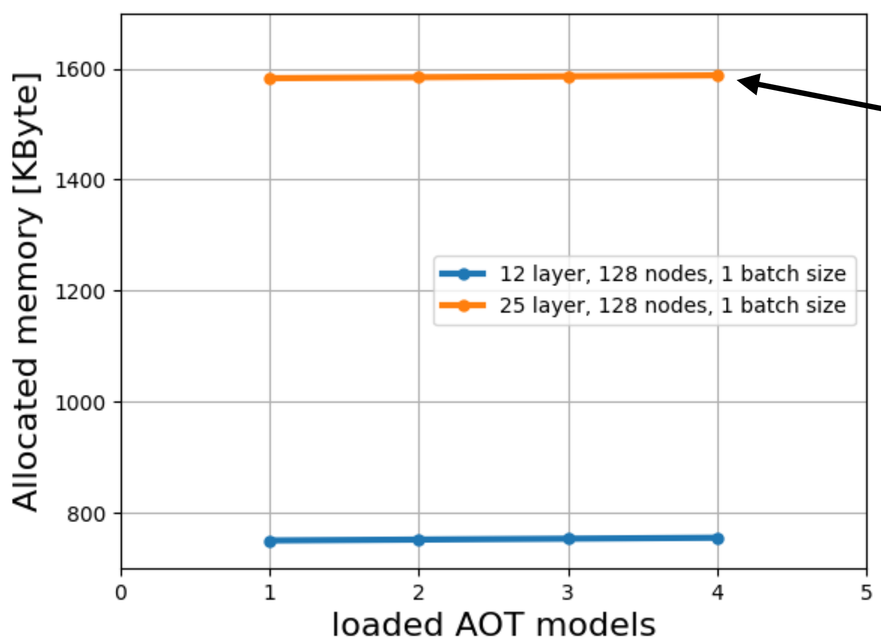
N layers	AOT model *.o [kB]	trainable weights [kB]
12	749	768
25	1581	1619

Memory allocation of Cpp wrapper



add model size
as offset

total Memory allocation

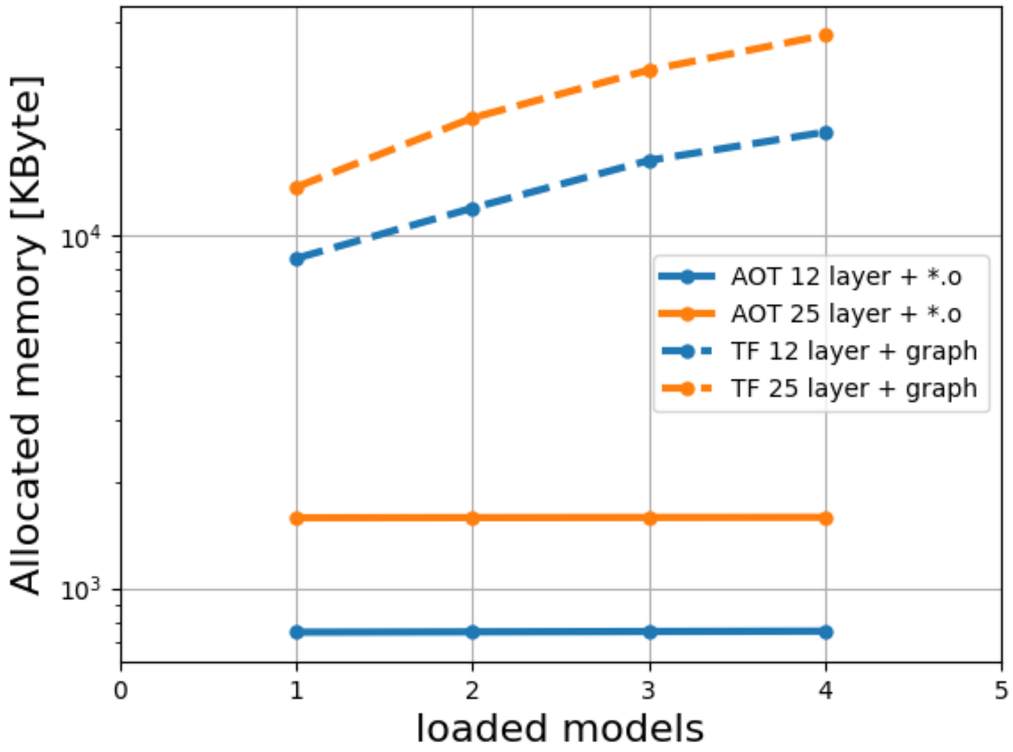


slope
barely seeable

TF

- `tf::Graph` larger than model weights
  - About twice as large in shown tests
- `tf::Session` always larger than `tf::Graph`
  - Model weights copied into each session
- Absolute size per session quite large

(Session + Graph) vs. (Buffer + *.o)



AOT

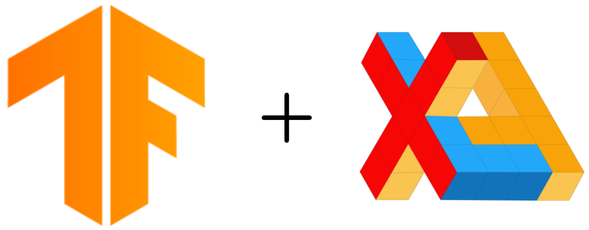
- Footprint of compiled c-function solely driven by size of model weights
- Cpp wrappers reserve buffers for input, output **and all** intermediate layers, overhead negligible
  - One wrapper needed per expected batch size
  - c-function (weights) might be sharable
- Absolute sizes small compared to TF objects
- independent of TF (save ~ several hundred MB RAM)

What was done:

- First steps with a new compiler method for Tensorflow graphs were made
- CPU runtime and RAM usage of conventional method was compared
  - ▻ CPU runtime is equally good
  - ▻ RAM usage is promising

The future of AOT in CMSSW: Roadmap

- ***Short term***
  - Understand memory measurements in event processing phase
  - Repeat CPU, RAM measurements with production model
    - ▻ DeepTau (combined) already AOT compiled
  - Enable AOT in central CMSDIST stack (bring patch live)
- ***Medium Term***
  - Automate model compilation within scram
  - Provide tools for working with compiled model
    (input / output access + dynamic batching)



in CMSSW

Ahead-of-time (AOT) compilation of Tensorflow models

Thank you for your attention!

Buffer

Allocated Buffers

- First 2 bits are used to define "kind", else is value
- kind = 0 ⟶ layers, 1 ⟶ temp (output), 2 ⟶ entrypoint (input), 3 ⟶ stackbuffer (intern)
- sum of 0 = *.o file (loaded once)

```
// Number of buffers for the compiled computation.
static constexpr size_t kNumBuffers = 25;

static const ::xla::cpu_function_runtime::BufferInfo* BufferInfos() {
  static const ::xla::cpu_function_runtime::BufferInfo
    kBufferInfos[kNumBuffers] = {
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({131072ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({32768ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({20480ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({514ULL, 0ULL}),
::xla::cpu_function_runtime::BufferInfo({161ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({33ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({16ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({4097ULL, ~0ULL})
      };
  return kBufferInfos;
}
```

Buffer value	Buffer size (B)	Buffer size (kB)	kind
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
131072	32768	32.0	0
32768	8192	8.0	0
20480	5120	5.0	0
514	128	0.125	2
161	40	0.0390625	1
33	8	0.0078125	1
16	4	0.00390625	0
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
4097	1024	1.0	1

Layer weights

- Formula to calculate number of parameters:
- 25 Hidden Layer network:
- $\underbrace{32 * 64 + 3 * 64}_{\text{Input Layer}} + \underbrace{64 * 128 + 128 * 3 + 24*(128^2 + 128 * 3)}_{\text{Hidden Layer}} + \underbrace{128 * 10 + 10}_{\text{Output Layer}} = 414538$ --> 1658152 Bytes

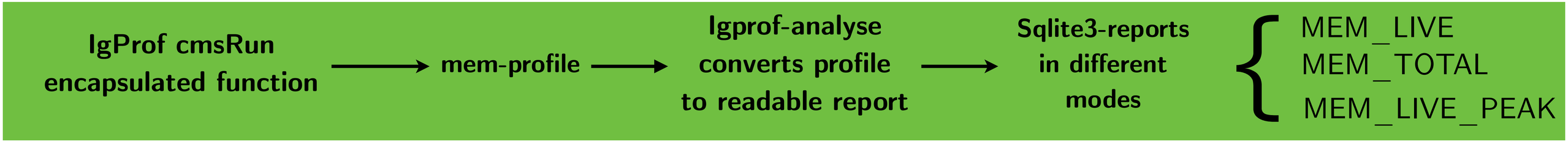
12 Hidden Layer network:

- $\underbrace{32 * 64 + 3 * 64}_{\text{Input Layer}} + \underbrace{64 * 128 + 128 * 3 + 11*(128^2 + 128 * 3)}_{\text{Hidden Layer}} + \underbrace{128 * 10 + 10}_{\text{Output Layer}} = 196554$ --> 786216 Bytes

IgProf

Overview: Memory measurement with IgProf & Test Setup

- Use **“Ignominious profiler”** (IgProf, talk) in memory-profile mode
- Encapsulate the code to be measured in a function
  - Trackable by name
  - Easy readout with with SQLite and Python (sqlite3)
  - Separation of functions into (e.g.) graph loading, session loading, inference calls
- **Workflow:**



- IgProf measures `malloc()`, `free()` of heap memory
  - Different memory metrics:
    - ▻ **MEM_TOTAL**: accumulated `malloc()`
    - ▻ **MEM_LIVE**: difference `malloc()` and `free()` for given interval
    - ▻ **MEM_LIVE_PEAK**: biggest single `malloc()`

Available metrics

- IgProf measures `malloc()`, `free()` of heap memory
  - Different memory metrics:
    - ▷ **MEM_TOTAL**: accumulated `malloc()`
    - ▷ **MEM_LIVE**: difference `malloc()` `free()` for given interval
    - ▷ **MEM_LIVE_PEAK**: biggest single `malloc()`
- Typical IgProf Report:

Rank	% total	Counts: to / from this	Counts: Total	Calls: to / from this	Calls: Total	Paths: Including child / parent	Paths: Total	Symbol name
	0.12	4,329,002	4,329,002	56,530	56,530	1	1	virtual thunk to PerfTesterTF::analyze(edm::Event const&, edm::EventSetup const&)
[1697]	**0.12**	**0**	**4,329,002**	**0**	**56,530**	**1**	**1**	**PerfTesterTF::loadGraph()**
	0.12	4,329,002	4,329,002	56,530	56,530	1	1	tensorflow::loadGraphDef(std::__cxx11::basic_string<char, std::char_traits<char>,

Encapsulated function to measure

Allocated memory in bytes

Number of allocations

Function call history

How does profiling (**IgProf**) works?

- Profiling divided into 3 steps:
- 1: Link Profiler with your program
  - **IgProf** set hooks before every function call
    - ▻ No change to application/build process necessary
- 2: Produce profile reports by execute the program
  - **IgProf** can run in 3 modes:
    - ▻ CPU-runtime (-pp) statistical sampling based performance profiles
    - ▻ memory (-mp) total dynamic memory allocations
    - ▻ Instrumentation () time spent in given function (±ns)
  - Store information of the parent and child stack frame
- 3: Analyse your profile
  - **IgProf-analyse** creates human-readable profile reports

Linking	Mode	Destination of Profile	Program to profile	Arguments of the program	Redirect stdout/-err to logfile

```
igprof -mp -o "profile_dir.mp" cmsRun ./cmssw_cfg.py &> "$log_dir.mp.log"
```

General information about “profiler”

- Typical program problems: too slow, consumes too much memory, its doing both!
- Computer programs can be large and complex with multiple subprocess being invoked
  - Making it hard to identify inefficient parts of the program or bugs
- Profiling gives quantified answer about:
  “How much does each function in this program consume resource X?”
- Profiler collecting data mid execution of the program
  - If certain feature is not used, it will not show up in the profile
- CMS own profiler **“Ignominious profiler”** (IgProf) the (https://igprof.org/index.html)

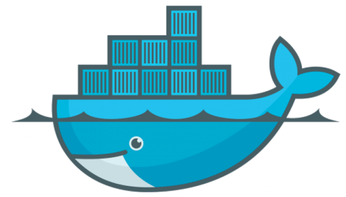
Analyse the profile

- Process profile statistics using the **“IgProf-Analyser”** tool
  - Possible report output: ASCII-text and sqlite3-database
  - Preferred way: sqlite3 (need command line sqlite3 app)

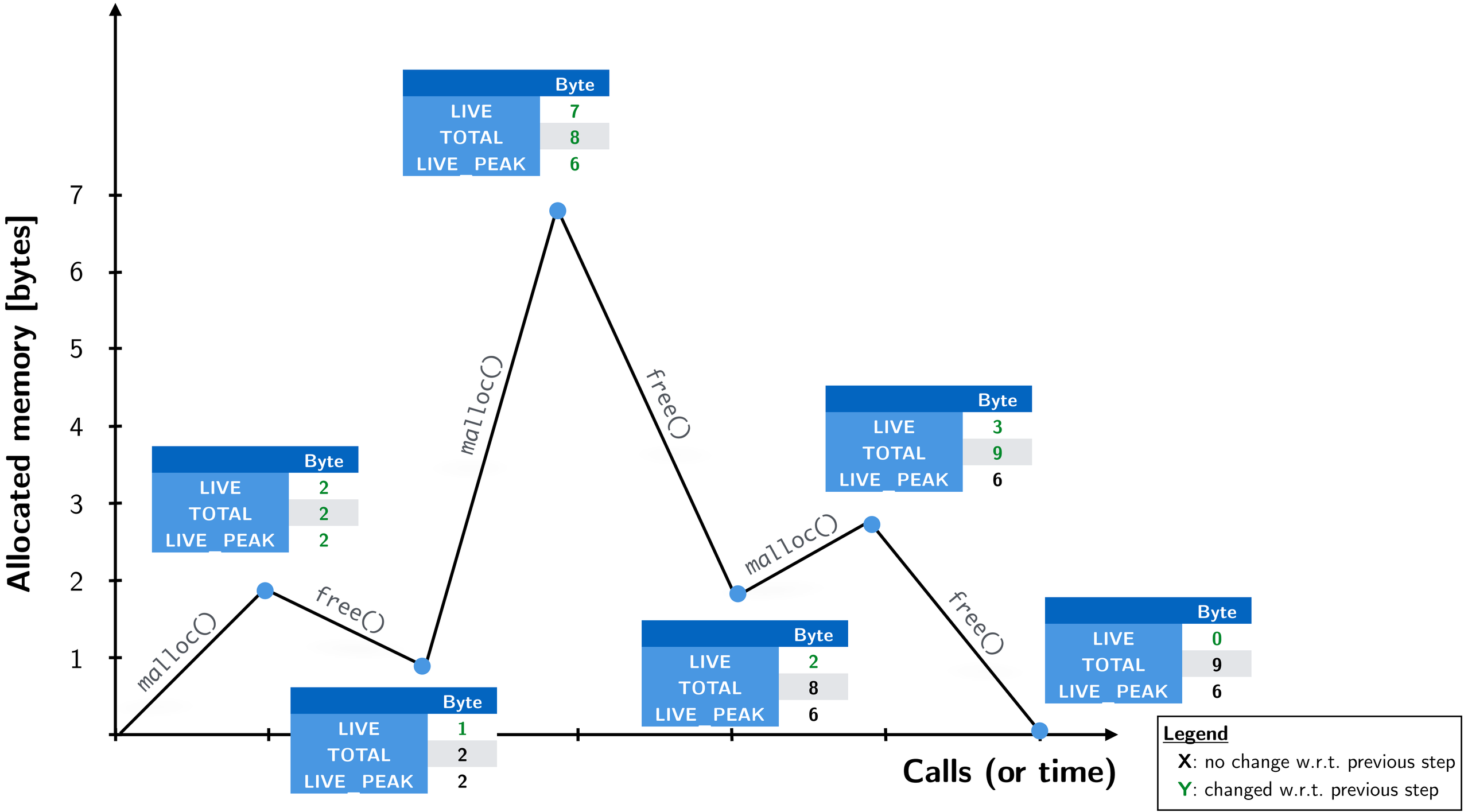
```
igprof-analyse --sqlite -d -v -g -r "MEM_TOTAL" "profile_dir.mp" | sqlite3 "dst_report.sql3"
```

  - ▻ Enables post-processing and plotting (pythons sqlite3)
  - ▻ Enables web-navigation using **“IgProf-Navigator”**, need CGI-open area

- Easiest way to navigate is Docker image (docker pull igprof/igprof) with port forwarding



Visualization of metrics



Inspect logs with the browser: 127.0.0.1:PORT

Sorted by cumulative cost

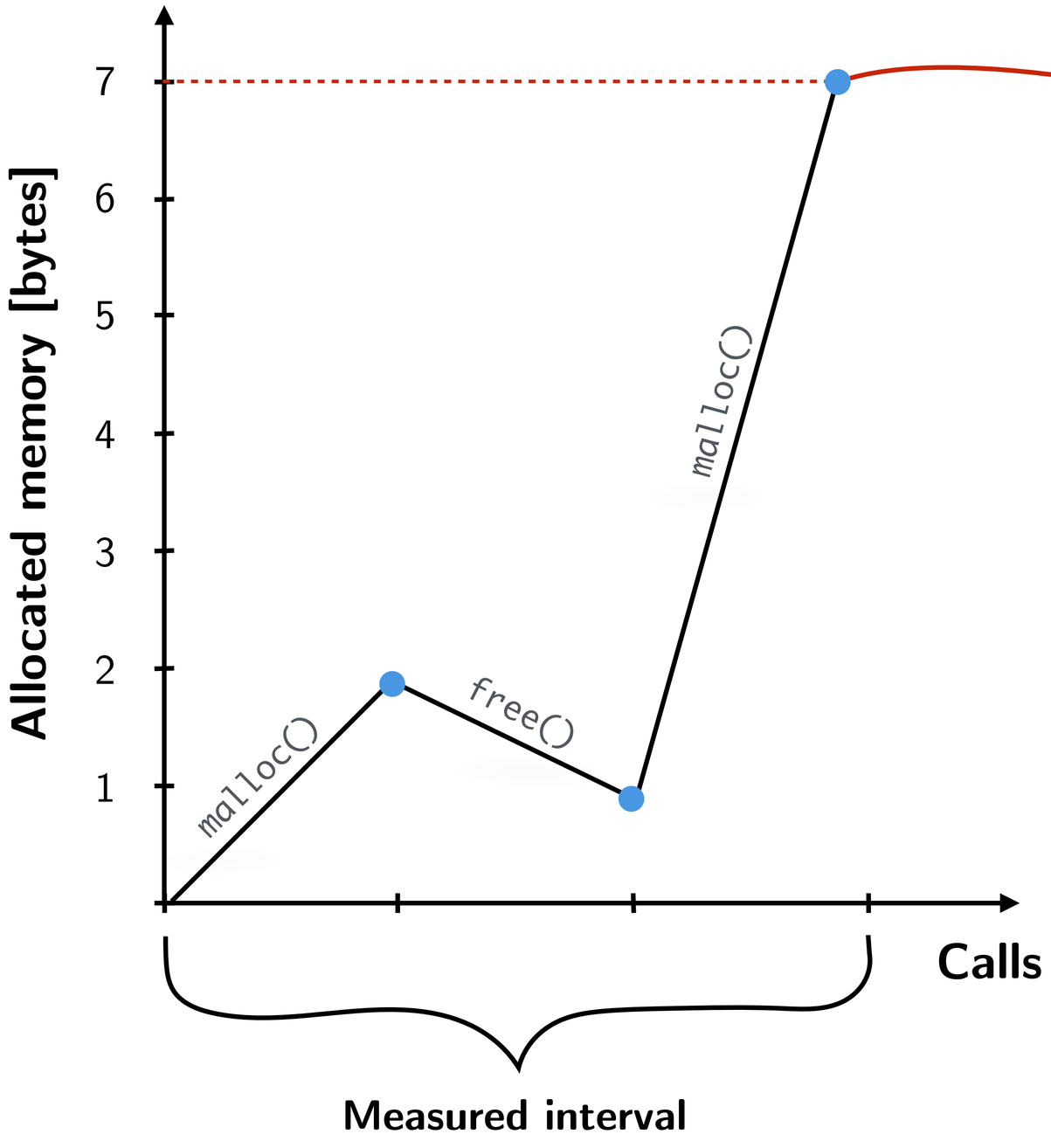
(Sort by self cost)

Rank	Total %	Cumulative	Calls	Symbol name
795	0.79	28,413,804	8,042	_PyObject_MakeTpCall
796	0.78	28,397,192	222,976	std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, s
797	0.78	28,354,666	23,778	ClingMemberIterInternal::DCIter::DCIter(clang::DeclContext*, cling::Interpreter*)
798	0.78	28,052,796	302,319	PerfTesterTF::loadSessions()
800	0.78	28,052,788	302,318	tensorflow::createSession(tensorflow::GraphDef const*, tensorflow::SessionOptions&)
799	0.78	28,052,788	302,318	tensorflow::createSession(tensorflow::GraphDef const*, int)
801	0.77	28,046,280	26,956	TClass::CallShowMembers(void const*, TMemberInspector&, bool) const'2
802	0.77	28,007,926	302,218	tensorflow::DirectSession::Create(tensorflow::GraphDef const&)
1698	0.12	4,329,002	56,530	tensorflow::loadGraphDef(std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
1697	0.12	4,329,002	56,530	PerfTesterTF::loadGraph()

Create 1 or more Session

Load 1 or more Graphs

Measurement of the setup phase



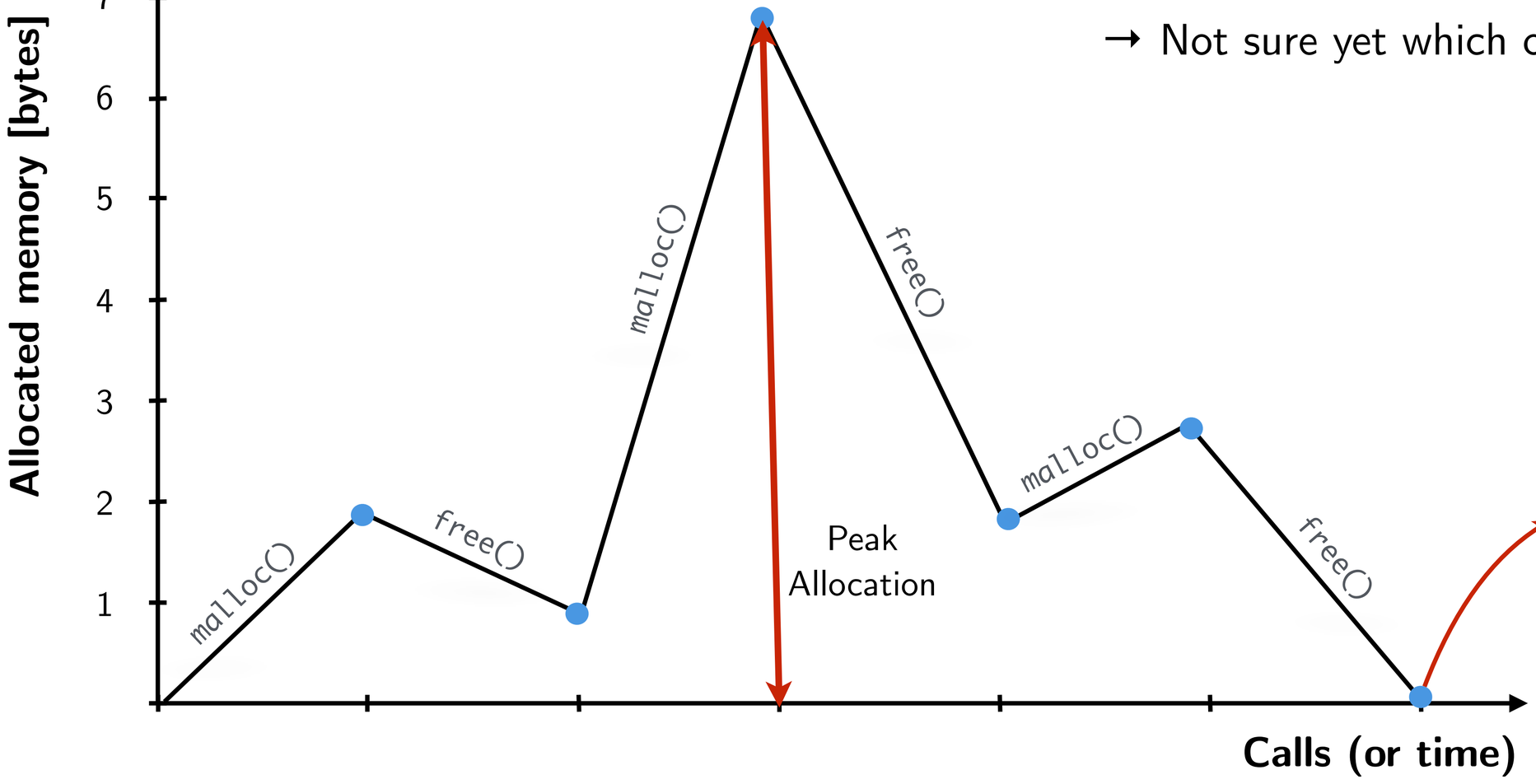
- For the **setup phase**, we are only interested in MEM_LIVE, i.e., amount of physical memory blocked by setup functions
- We know exactly which function / interval to measure (e.g. `tf::loadGraph()`)
- Applies to global & per-thread measurement

	Byte	
LIVE	7	→ `malloc()` - `free()`
TOTAL	8	→ Σ `malloc()`
LIVE_PEAK	6	→ max single `malloc()`

Measurement of the event phase

- For inference calls in event loop (`analyze()`), need to know maximum memory allocation ↕ within the evaluation call (e.g. `tf::run(session, ...)`)
- Open question: which metric to use?
  - **LIVE**: difference between "after" - "before", should be 0 in absence of leaks
  - **LIVE_PEAK**: only measures largest, single allocation
  - **MAX**: "records the largest single allocation by any function" (link)

→ Not sure yet which one to use



	Byte
LIVE	0
TOTAL	9
LIVE_PEAK	6

*.o file offset

- Pruning removed nodes that does not change the values:
  - ▻ Bias tensors removed (default value = 0)
  - ▻ Batch norm Gamma and Beta removed (default = 1, 0)
- (3 * 64 + 3 * 12 * 128 + 10)*4/1024 = 18.7890625
  - 64, 128 , 10 are shape of network
  - bias, gamma and beta have same shape (thus * 3)
  - 4 since 32 Bit-float weights

c-function consists mainly of model weights
difference is result of pruning (bias and batchnorm)

N layers	AOT model *.o [kB]	trainable weights [kB]
12	749	768
25	1581	1619

Potential advantages in CMSSW

- **Current TF workflow**
  - Training in python, then export to frozen graph
  - Load graph in C++ (once per process)
  - Mount graph onto session (once per stream module instance)
  - Use TF C++ API to feed, run and read
    - ▻ No simple handles for optimization

- **Potential AOT benefits**
  - Easy to use graph and compiler optimizations
  - Potentially faster (?), might depend on model
  - Reduced memory footprint
    - ▻ No need to load the full `libtensorflow.so`
    - ▻ Model weights only once in memory
    - ▻ Virtually no memory overhead over multiple threads

Requirements / implications

- **For TF installation in CMSDIST**
  - Needs to be installed with XLA support, not done yet but mostly (but described e.g. here)
    - ▻ One incompatibility "eigen" in CMSDIST, but resolvable

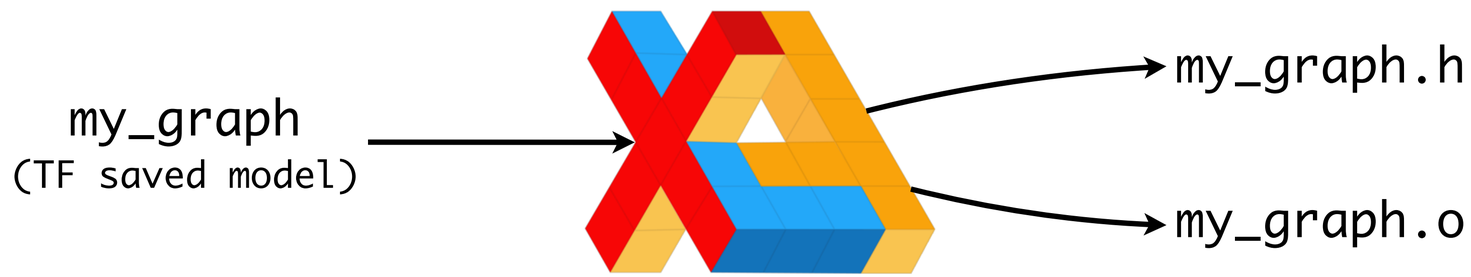
- **For models**
  - Graph to be in "saved model" format (standard), exported with target signature(**s**), i.e., input shapes
  - Custom operations need to be compositions of TF ops; currently not supported:
    - ▻ Some ops of standalone keras (use of `tf.keras` is anyway encouraged)
    - ▻ Some TF ops with unpredictable shape (e.g. `tf.where`)

- **For inference code in CMSSW**
  - For now: created *.o files must be linked manually in BuildFiles
    - ▻ Vision: add hooks to BuildFiles and implement them to work with scram, e.g.

```
<aot_compile model="/data/my_model" class="MyModel" batch_sizes="1 2 4
8"/>
```

  - No more access to `tensorflow::Tensor`, input / output manipulation done on bare pointers
    - ▻ Simple set of tools could provide convenience

- Graphs can be pre-compiled with the `saved_model_cli` tool

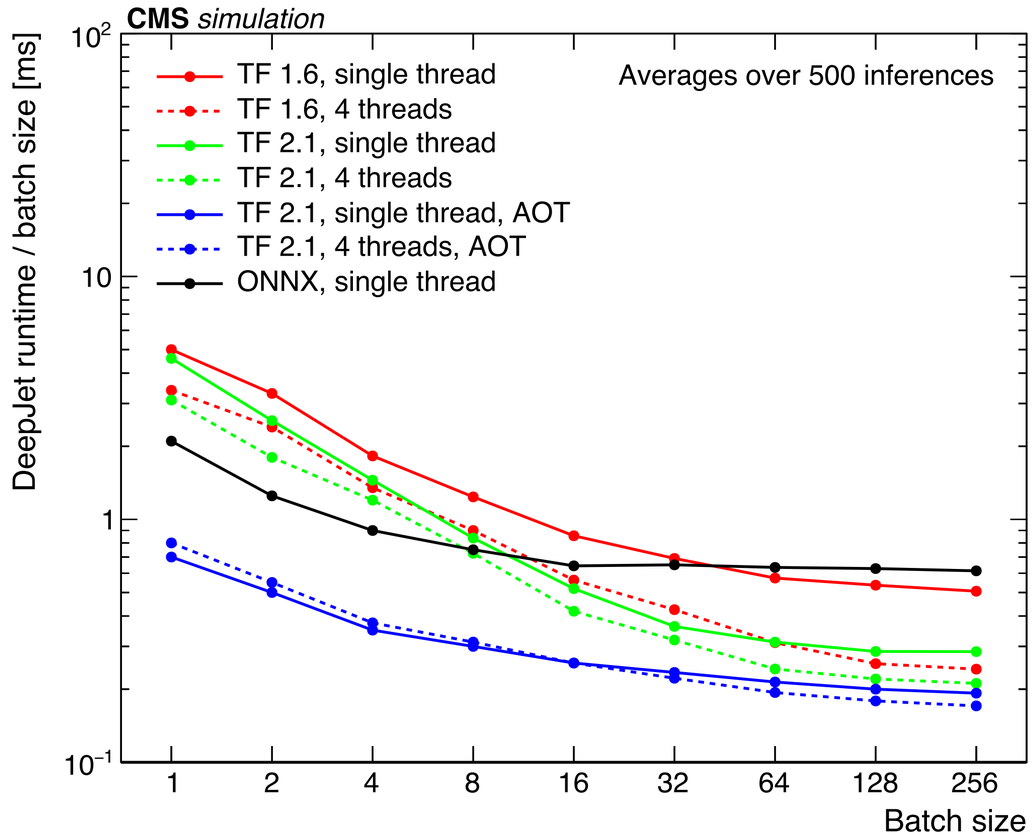


- Based on domain-specific XLA (Accelerated Linear Algebra) compiler
- Pros:
  - Generated files self-contained and, in particular, do not require the TensorFlow runtime
  - Provide a simple function that can be simply called with network inputs
  - Can greatly improve performance
- Cons:
  - Compilation process challenging to automate in production
  - Batch sizes are fixed and need to be known a priori, or either padded or stitched

Potential advantages in CMSSW

- **Current TF workflow**
  - Training in python, then export to frozen graph
  - Load graph in C++ (once per process)
  - Mount graph onto session (once per stream module instance)
  - Use TF C++ API to feed, run and read
    - ▻ No simple handles for optimization

- **Potential AOT benefits**
  - Easy to use graph and compiler optimizations
  - Potentially faster (?), might depend on model
  - Reduced memory footprint
    - ▻ No need to load the full `libtensorflow.so`
    - ▻ Model weights only once in memory
    - ▻ Virtually no memory overhead over multiple threads



DPG 2023

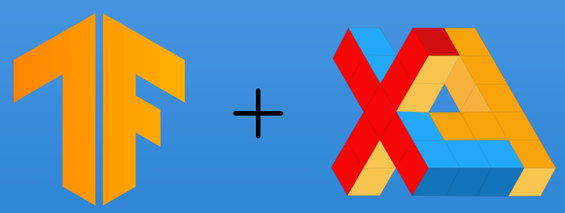
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

CLUSTER OF EXCELLENCE
QUANTUM UNIVERSE

SPONSORED BY THE
Federal Ministry of Education and Research

"Ahead-of-time" compilation of Tensorflow models

Marcel Rieger, Peter Schleper, Bogdan Wiederspan



DPG

Inference of neural network models in CMS

Context of this talk:

- Service work for "Machine Learning Production Group" for the CMS Experiment
- Talk focuses on improvement of **computing performance** of neural networks

---

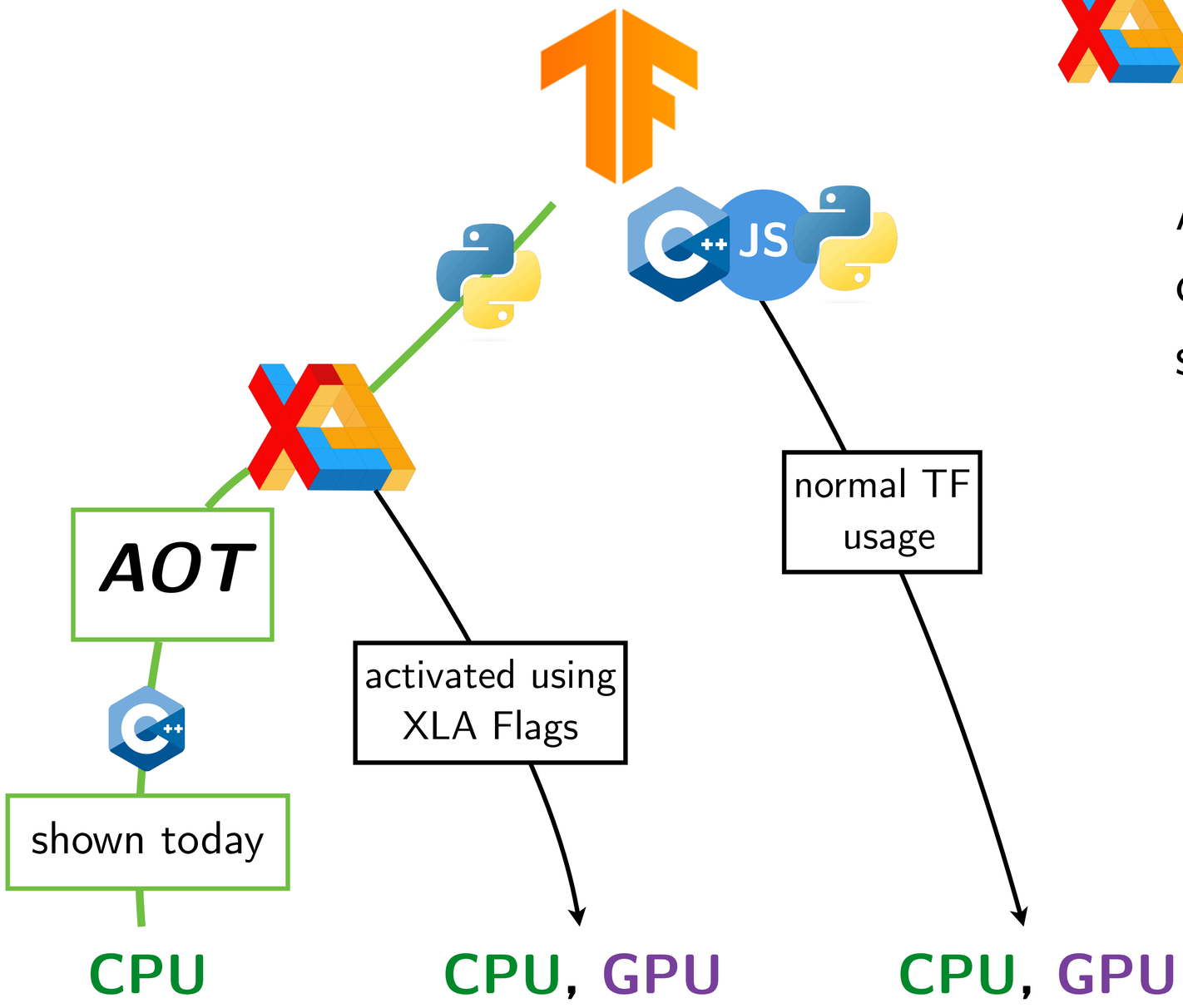
Inference of neural network models in CMS

- Inference engine of CMS called CMSSW
- models run single threaded in CMSSW
- Why we in CMS care: **limited memory resources per core (≈2GB)**

- **Many production models already implemented**
  - DeepJet tagging, DeepFlavor, DeepMET (50-100 MB each) ⟶ and even more in the pipeline!
- **most models run in CPU only mode**

- **Naive solutions**
  - buy more hardware
  - **deploy only "important" models**
  - **be more efficient**

Overview: Different ways to run Tensorflow (TF) models



XLA ="Accelerated Linear Algebra"

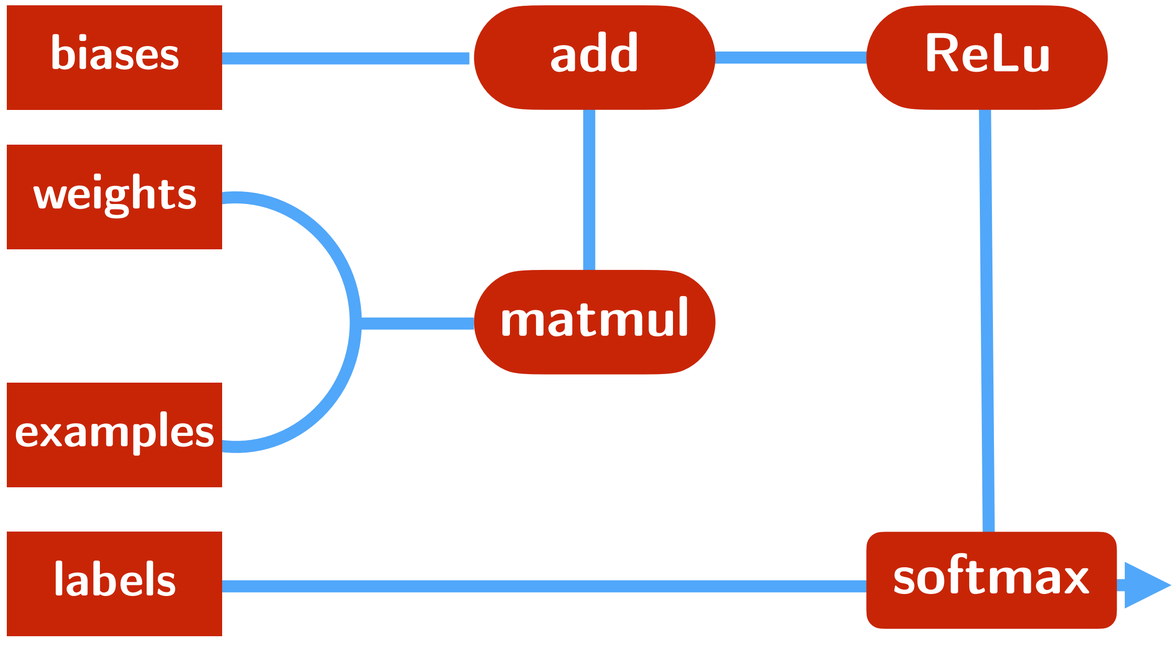
AOT = "Ahead of time",
compile code at build time into
system dependent binary

Focus of this talk:

being more efficient with **TF models** using **XLA and AOT on CPU (left/green path)**

How TensorFlow in graph mode operates

- TF generates a data flow graph representing the ML model
- Graphs consist out of kernel and edges
  - **Kernel** represent operations (Add, MatMul, Conv2D, ...)
    - TF runtime, called session, executes graph kernel
    - Operation kernels are written in C++ for CPU or CUDA for GPU
  - **Edges** represent data flowing (Tensors, control dependencies, resource handles, ...)



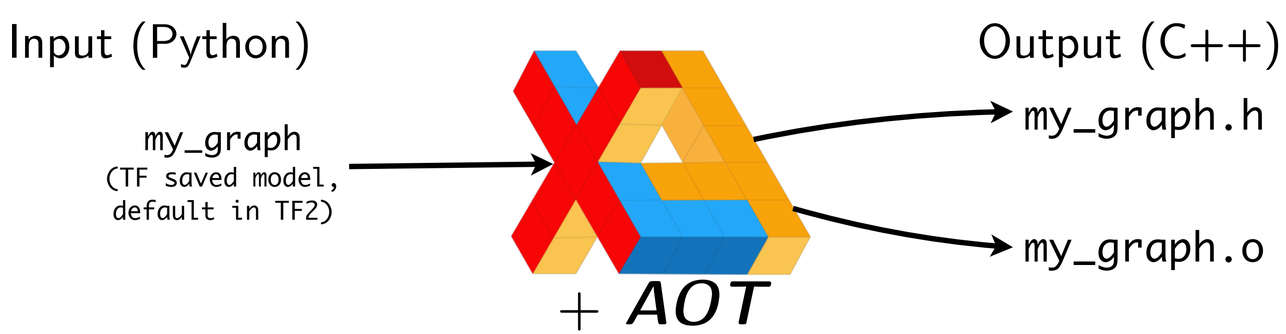
Example of a 1-layer network

- Beyond modus operandi of TensorFlow: **optimizations (XLA)** and **independence of Tensorflow (AOT)**

What does XLA and AOT do?

Enables several types of graph optimizations

- On graph level:
  - kernel fusion
  - Buffer analysis for allocating runtime memory (eliminates intermediate caches)
  - Common subexpression elimination
  - Pruning of unused kernel
- On hardware level:
  - TPU, GPU or CPU (different backends)



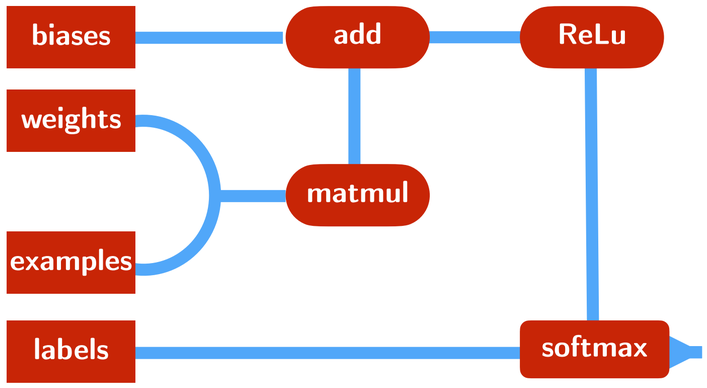
AOT

- Converts graphs into self-contained library
  - Graph becomes a series of compute kernels in C++
  - **No dependence** on main `libtensorflow.so`
- Pros:
  1. Reduced memory footprint (shown later)
  2. Trivial multi-threading behavior
  3. Runtime potentially faster (shown later)
- Cons:
  1. No dynamic batching (fixed memory layout)
  2. Graph needs to be XLA compatible (shown later)

AOT Compilation workflow summary

Step 1: Prepare graph

save model with fixed batch size
(static memory layout)



Step 2: Tensorflow Optimization

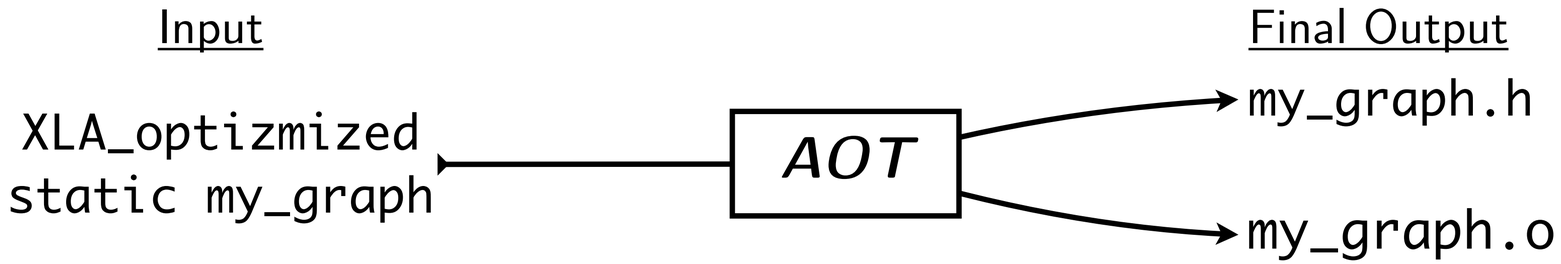
optimization level
configurable

MetaOptimizer

Step 3: XLA Optimization

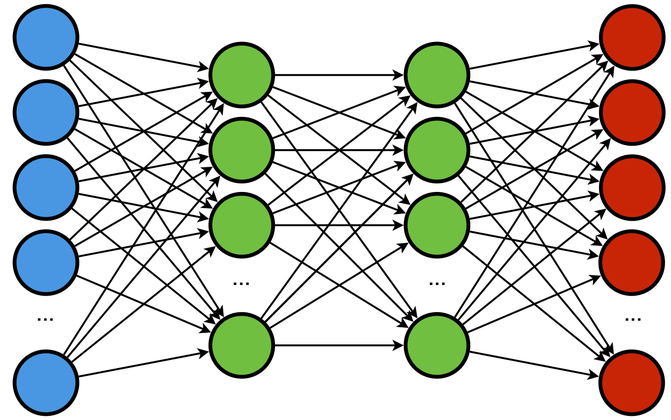
optimization level
configurable

Step 4: AOT compiled using TF "saved_model_cli" tool



Performance Study

Performance study outline



Network and payload

- Created several feed-forward toy model
  - 12/25 layers, 128/256 units, batch-norm and SELU activation
- **CPU runtime tests performed on login-node with CMSSW:**
  - Compared **forward pass runtime** of **TF C++ vs. AOT**
  - inputs of the network are random values
  - Averaged runtimes over 500 calls, after 100 "warm up" calls
  - Tested **event forward time** for different batch sizes from 2^0 to 2^{10}
  - **No XLA** optimizations applied
- **Memory tests using MemoryProfiler:**
  - **Measure memory consumption for setup phase**

CPU runtime Study

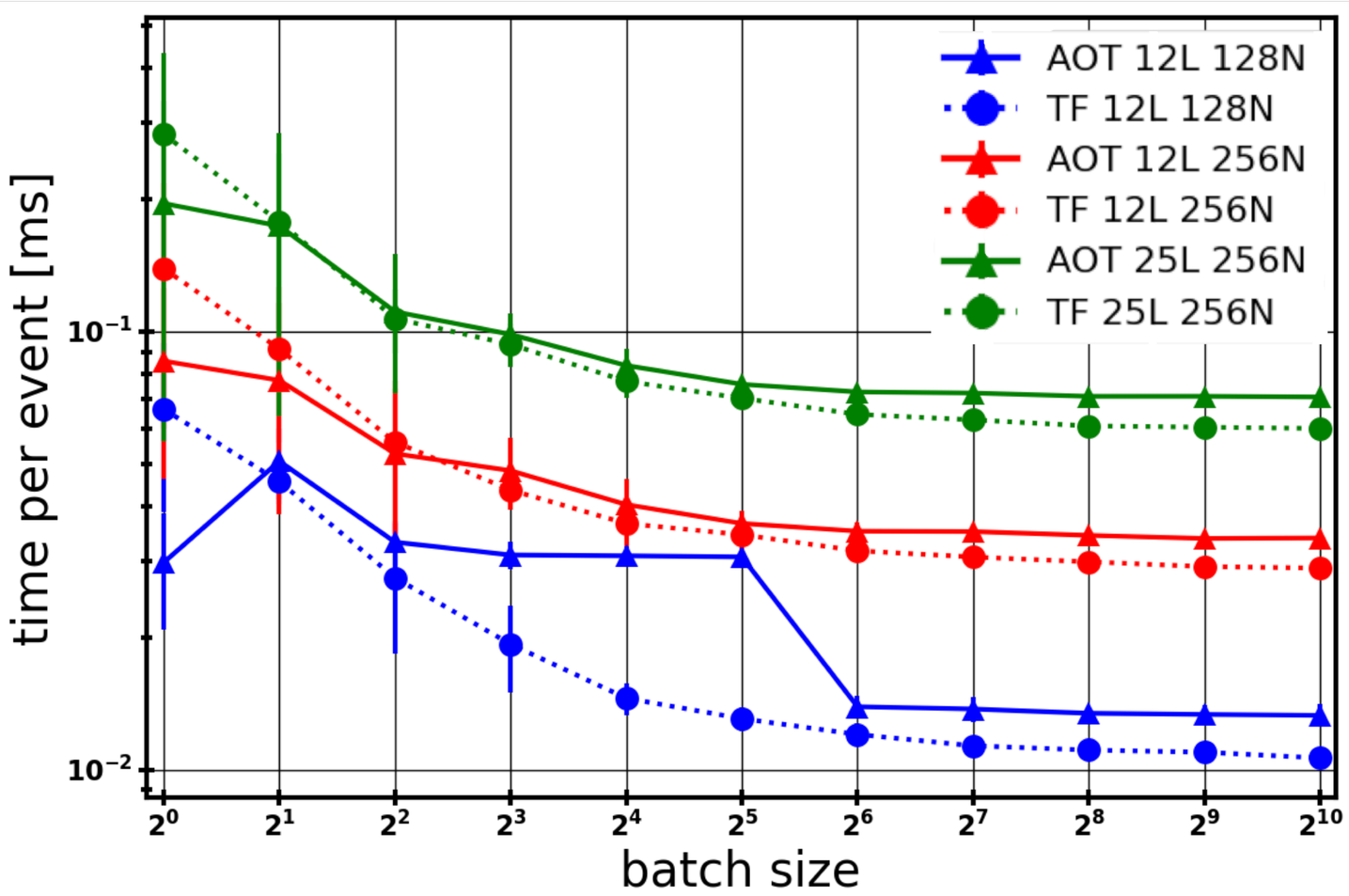
CPU performance

1. Runtimes

- bigger batch sizes can be vectorised (decrease of time per event, till reaching saturation)
- **Equal performance between TF and AOT**
  - compare same color lines
    dotted (TF) vs solid (AOT)
- **batch size 1 AOT is always better than TF**

2. Multi-threading

- Performance test restricted to 1 thread
- CMS production models run in single thread



Memory Study

Memory comparison AOT vs TF

memory comparison TF graph vs AOT model			
N layers	TF graph [kB]	AOT model *.o [kB]	trainable weights [kB]
12	1997	749	768
25	3539	1581	1619

- TF graph more than 2x bigger than AOT
  ⟶ TF graphs contain many meta information
- AOT model consist of (pruned) trainable weights

total memory consumption AOT (batch size 1) vs. TF (loading 1 model)		
N layers	AOT model *.o + Wrapper [kB]	TF Session + Graph [kB]
12	751	8599
25	1583	13688

- AOT by factor 10 smaller than TF
- AOT wrapper = buffer for **input**, **output** and **intermediate layer** (scales with batch size)
- wrapper size = total - model size ⟶ small (2kB)
- TF I/O tensors are not included in the measurement
- AOT fully independent of `libTensorflow.so` at runtime, saves several hundred MBs on top

Limits of AOT compilation

When does AOT compilation fail?

Possible reasons to fail AOT compilation:

1. Kernel produce no predictable shapes (=no fix memory layout) at compile time (e.g tf.where)
2. No existing TF XLA implementation of the kernel
   - more XLA kernels added with each TF update

How to check if an XLA implementation exist for a given model:

1. Create XLA operation table during TF compilation
2. Get all used nodes within graph
3. Find match between table and graph

example: check AOT compatibility of feed-forward network with batch-norm for TF v. 1.4

Operation	has XLA
MatMul	True
Identity	True
Sub	True
BiasAdd	True
Mul	True
Softmax	True
NoOp	True
Rsqrt	True
Selu	True
AddV2	False
ReadVariableOp	True
Const	True

→ network is not AOT compatible

Summary and Outlook:

- Summary:
  - a new method to run TF Graphs on CPU, called AOT, was presented
  - first runtimes showed that AOT is on average comparable fast as TF C++ models
    - batch size 1 in AOT is always faster than TF C++
    - AOT open gates to more optimization with different XLA level
  - AOT memory footprint is about factor 10 smaller than TF
  - presented limits of AOT compilation in the context of model building and constraints on ops

---

- CMS Outlook:
  - switch to AOT models would give room for more models in current hardware stack (save money)
  - slight increase in performance expected, since production models run on single batch mode
- General Outlook:
  - documentation of compiling workflow will be written
  - tools for model preparation are in development

you don't start from 0

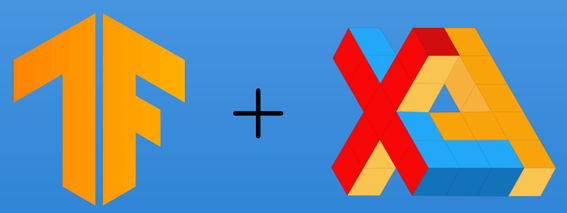
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

CLUSTER OF EXCELLENCE
QUANTUM UNIVERSE

SPONSORED BY THE
Federal Ministry
of Education
and Research

"Ahead-of-time" compilation of Tensorflow models

Marcel Rieger, Peter Schleper, Bogdan Wiederspan
Email: bogdan.wiederspan@uni-hamburg.de



Thank you for your attention!

BackUp

Technical Details

Requirements / implications

- **For TF installation in CMSDIST**
  - Needs to be installed with XLA support, not done yet but mostly (but described e.g. here)
    - ▻ One incompatibility "eigen" in CMSDIST, but resolvable

- **For models**
  - Graph to be in "saved model" format (standard), exported with target signature(**s**), i.e., input shapes
  - Custom operations need to be compositions of TF ops; currently not supported:
    - ▻ Some ops of standalone keras (use of `tf.keras` is anyway encouraged)
    - ▻ Some TF ops with unpredictable shape (e.g. `tf.where`)

- **For inference code in CMSSW**
  - For now: created *.o files must be linked manually in BuildFiles
    - ▻ Vision: add hooks to BuildFiles and implement them to work with scram, e.g.

```
<aot_compile model="/data/my_model" class="MyModel" batch_sizes="1 2 4
8"/>
```

  - No more access to `tensorflow::Tensor`, input / output manipulation done on bare pointers
    - ▻ Simple set of tools could provide convenience

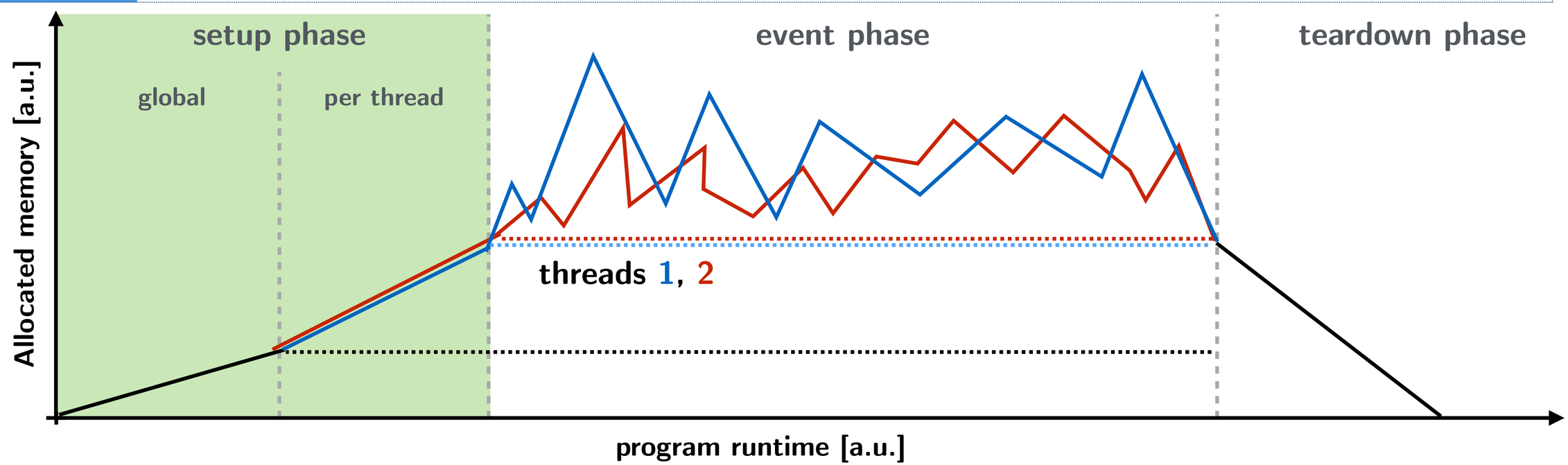
Advantages of using Graphs

- **Parallelism**:
  - identify operations that can be executed in parallel
- **Computation optimization**:
  - graphs are well-known data structure
  - optimisation possible.
    - e.g: prune unused nodes (size optimisation)
    - detect redundant operations or sub-optimal graphs and replace them with the best alternatives (speed optimisation)
- **Portability**:
  - graphs are language- and platform-neutral
- **Distributed execution**:
  - Every graph's node can be placed on an independent device and on a different machine.

Understanding of memory for different phases

- Talk: **Focus on setup phase**, since we can not measure event phase yet.
- global allocate memory once, per thread for each thread
- Comparison: **TF multiple sessions vs. loading multiple AOT models**

Phase	Description	Action in TF	Action in AOT
global setup	Before threads (stream modules) are launched	`tf::loadGraph();` (load model and weights into memory)	loading compiled model (external c-function in *.o file)
setup per thread	Footprint per thread, but before events are analyzed	`tf::loadSession();` (device placement & caching per thread)	`CppWrapper w;` (access to c-function & reserve buffers for input, output)
event phase	Resource consumption during event processing	`tf::run(session, ...);` (book inputs/outputs & evaluate model)	`w.run();` (evaluate model)



TensorFlow setup phase

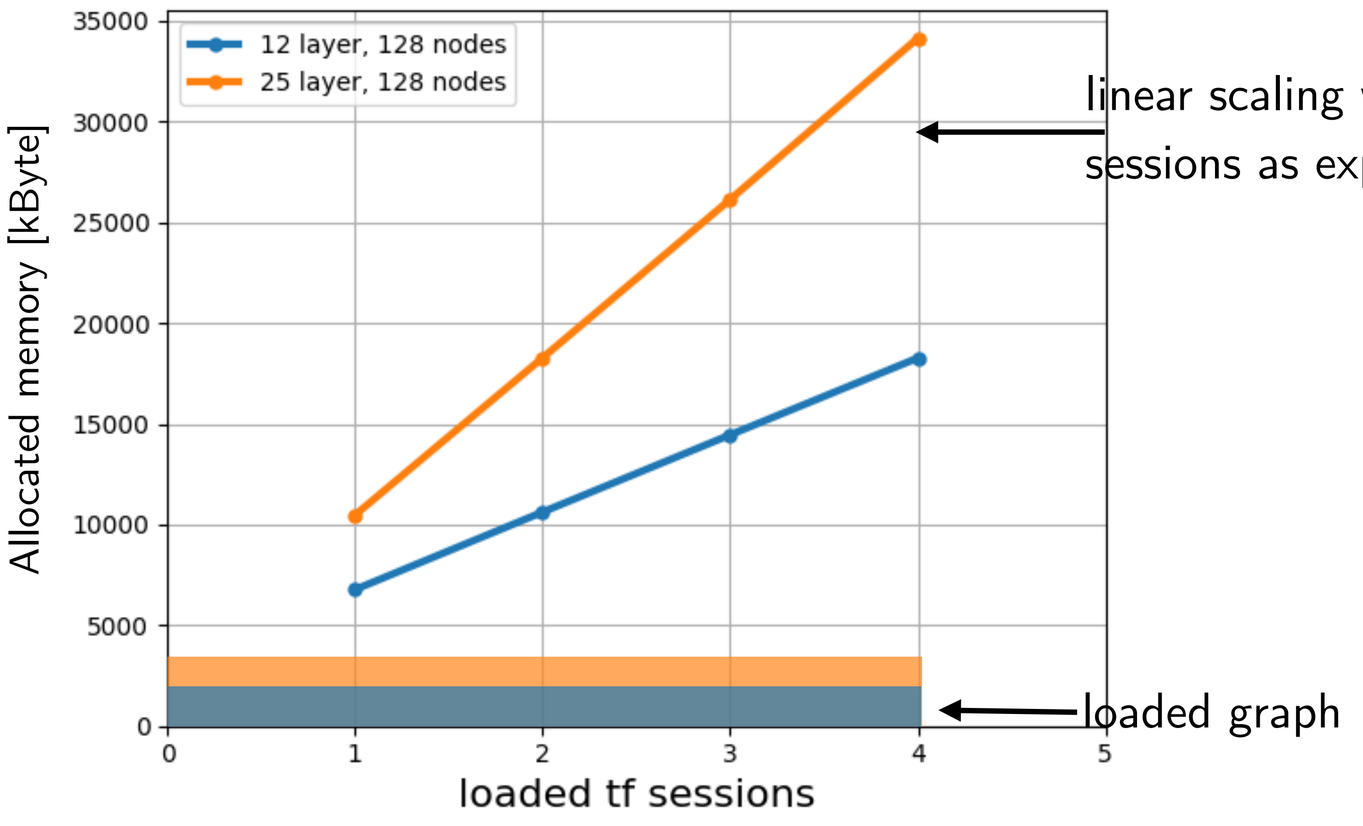
- Launch multiple sessions with same frozen graph
  - Load graph once (global)
  - Copy graph into session (per-thread)
- Does not include creation of tensors yet!

```cpp
void PerfTesterTF::loadGraph() {
  graph_def = tensorflow::loadGraphDef(graph_path);
}
```

```cpp
void PerfTesterTF::loadSessions() {
  std::vector< tensorflow::Session* > loaded_sessions;
  loaded_sessions.reserve(number_of_loaded_sessions);
  for (int i = 0; i < number_of_loaded_sessions; i++) {
    session = tensorflow::createSession(graph_def);
    loaded_sessions.push_back(session);
    }
}
```

code to load graph and sessions

Memory allocation of `loadSessions()`



Graph contains plenty of meta data

N layers	`loadGraph()` [kB]	trainable weights [kB]
12	1997	768
25	3539	1619

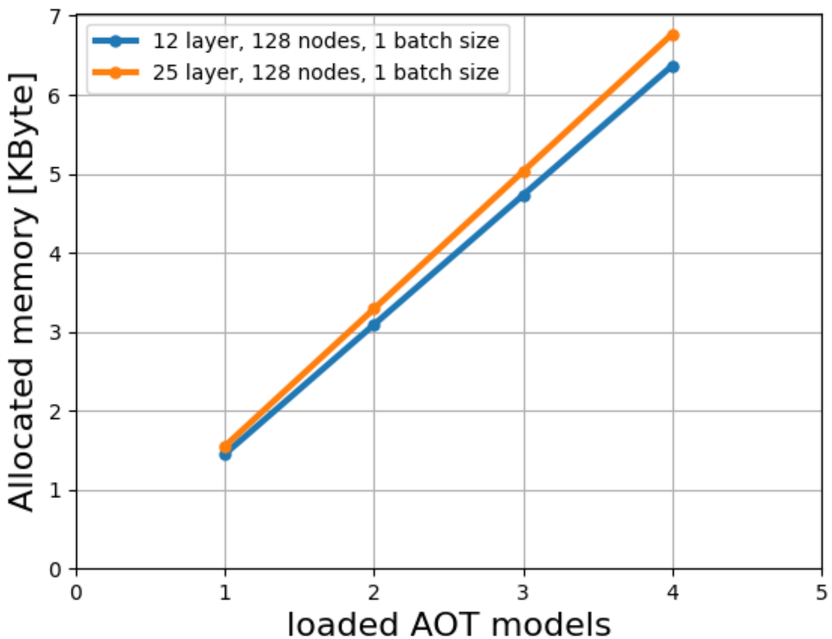
AOT setup phase

- Load multiple AOT models of same batch size
  - Cpp wrapper created to call c-function (per thread)
    - ▻ Reserves **buffers** for **inputs**, **outputs per model**
    - ▻ off-set = model depending **(slope)**
  - Reserves buffer for **intermediate layers once (*.o)**
- Important: each Cpp wrapper handles only one batch size
  - c-function with model weights might be shared between different batch sizes (check ongoing, important for stitching)
  - cpp wrapper is neglectable small compared to layers

c-function consists mainly of model weights
difference is result of pruning (bias and batch norm)

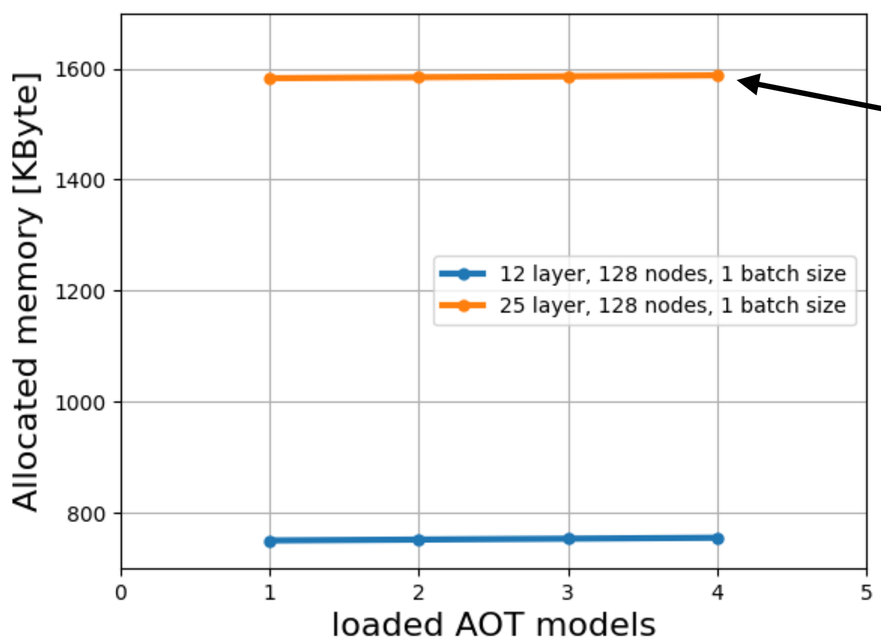
N layers	AOT model *.o [kB]	trainable weights [kB]
12	749	768
25	1581	1619

Memory allocation of Cpp wrapper



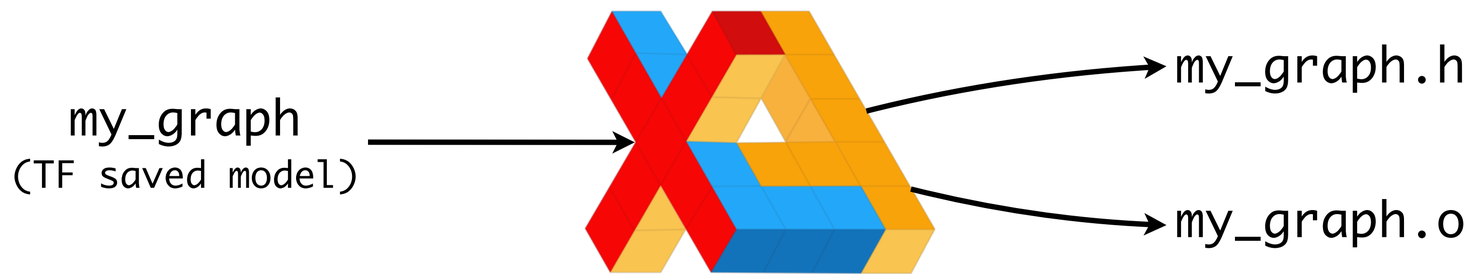
add model size
as offset

total Memory allocation



slope
barely seeable

- Graphs can be pre-compiled with the `saved_model_cli` tool

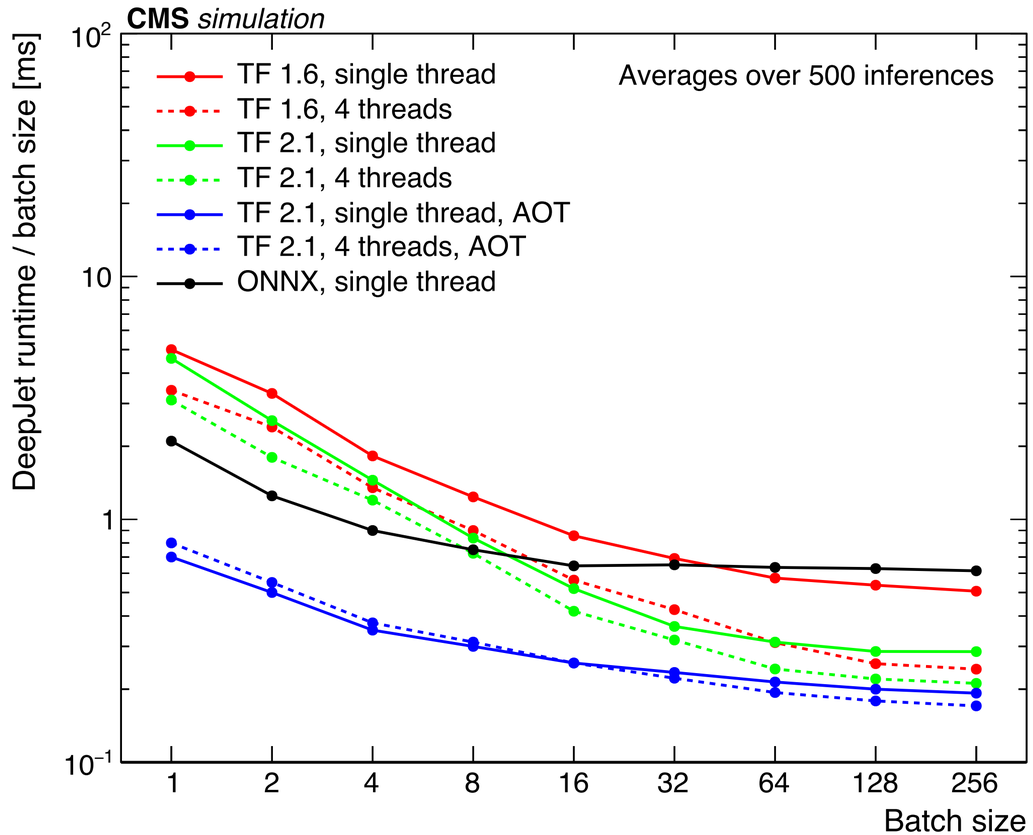


- Based on domain-specific XLA (Accelerated Linear Algebra) compiler
- Pros:
  - Generated files self-contained and, in particular, do not require the TensorFlow runtime
  - Provide a simple function that can be simply called with network inputs
  - Can greatly improve performance
- Cons:
  - Compilation process challenging to automate in production
  - Batch sizes are fixed and need to be known a priori, or either padded or stitched

Potential advantages in CMSSW

- **Current TF workflow**
  - Training in python, then export to frozen graph
  - Load graph in C++ (once per process)
  - Mount graph onto session (once per stream module instance)
  - Use TF C++ API to feed, run and read
    - ▻ No simple handles for optimization

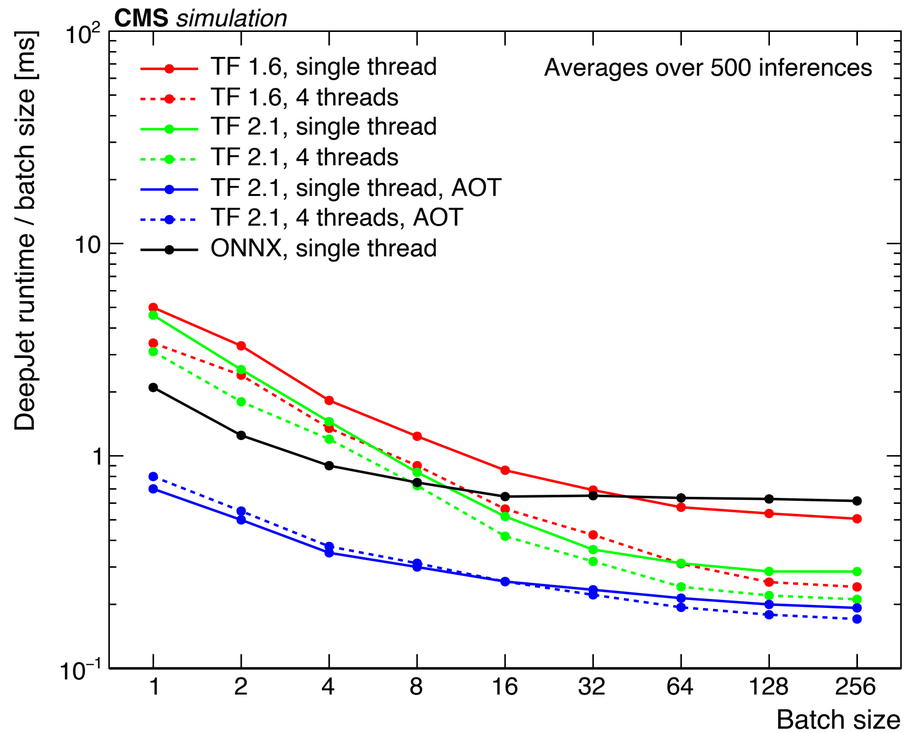
- **Potential AOT benefits**
  - Easy to use graph and compiler optimizations
  - Potentially faster (?), might depend on model
  - Reduced memory footprint
    - ▻ No need to load the full `libtensorflow.so`
    - ▻ Model weights only once in memory
    - ▻ Virtually no memory overhead over multiple threads



Potential advantages in CMSSW

- **Current TF workflow**
  - Training in python, then export to frozen graph
  - Load graph in C++ (once per process)
  - Mount graph onto session (once per stream module instance)
  - Use TF C++ API to feed, run and read
    - ▻ No simple handles for optimization

- **Potential AOT benefits**
  - Easy to use graph and compiler optimizations
  - Potentially faster (?), might depend on model
  - Reduced memory footprint
    - ▻ No need to load the full `libtensorflow.so`
    - ▻ Model weights only once in memory
    - ▻ Virtually no memory overhead over multiple threads

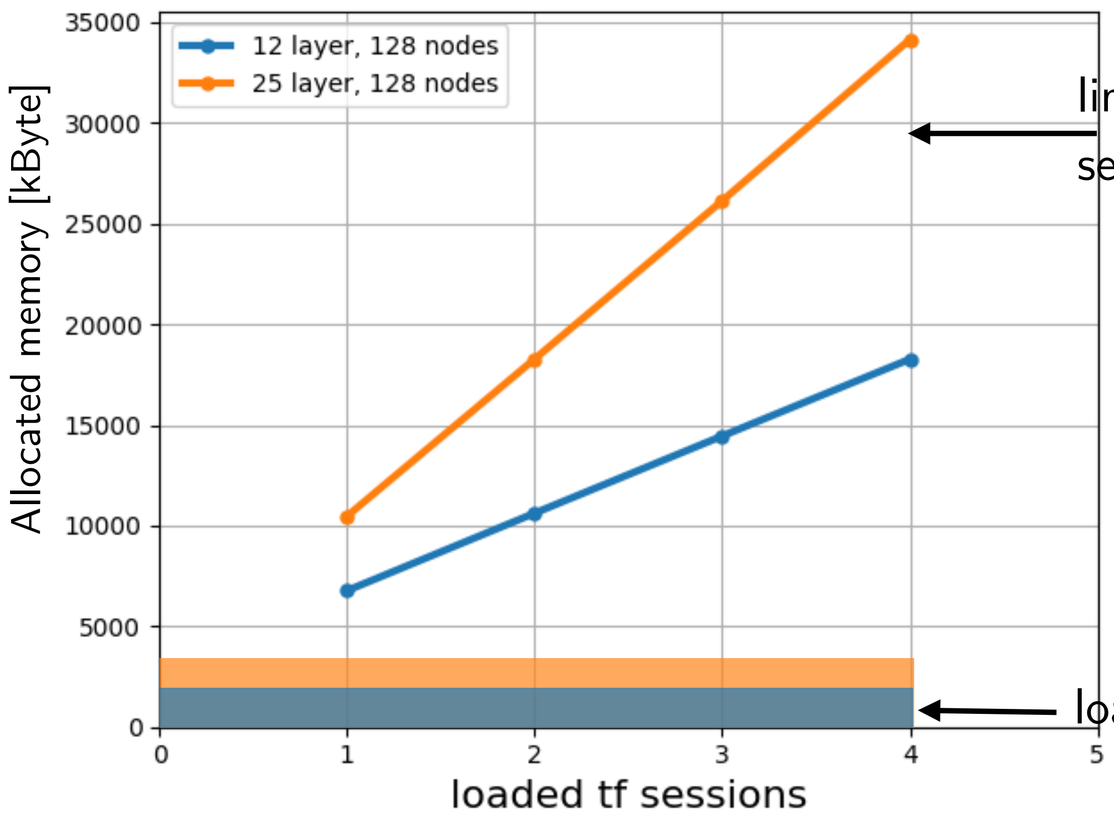


TensorFlow memory allocation in setup phase

- Session = environment that allocates resources and execute graph ⟶ session uses meta data saved in Graph
- Launch multiple sessions with same frozen graph
  - Load graph once (global)
  - Copy graph into session (per-thread)
- Does not include creation of tensors yet!

N layers	loaded Graph [kB]	trainable weights [kB]
12	1997	768
25	3539	1619

Memory allocation of loaded Sessions



linear scaling with sessions as expected

loaded graph

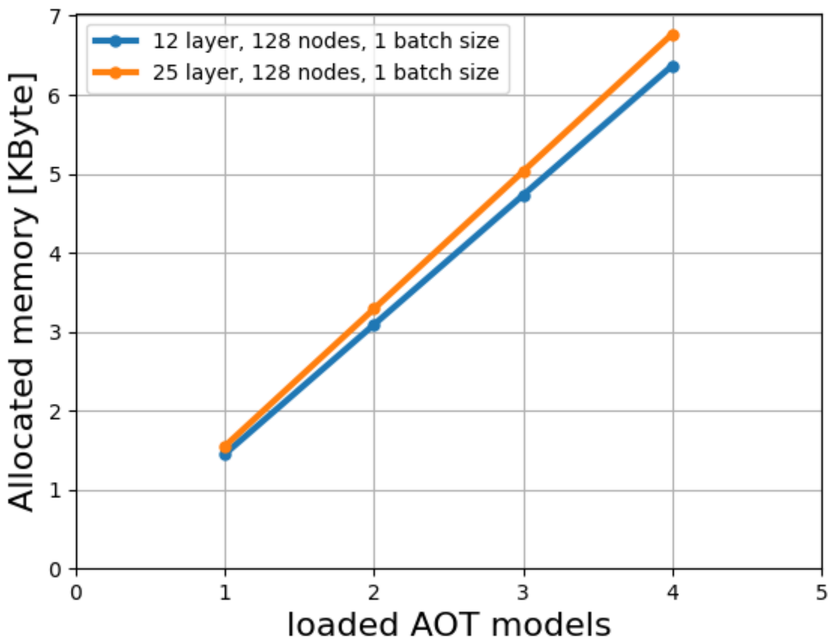
AOT memory allocation in the setup phase

- Load multiple AOT models of same batch size
  - Cpp wrapper created to call c-function (per thread)
    - ▻ Reserves **buffers** for **inputs**, **outputs per model**
  - Reserves buffer for **intermediate layers once (*.o)**
    - ▻ off-set = model depending **(slope)**
- Important: each Cpp wrapper handles only one batch size
  - c-function with model weights might be shared between different batch sizes (check ongoing)
  - cpp wrapper is neglectable small compared to layers

c-function consists mainly of model weights
difference is result of pruning (bias and batch norm)

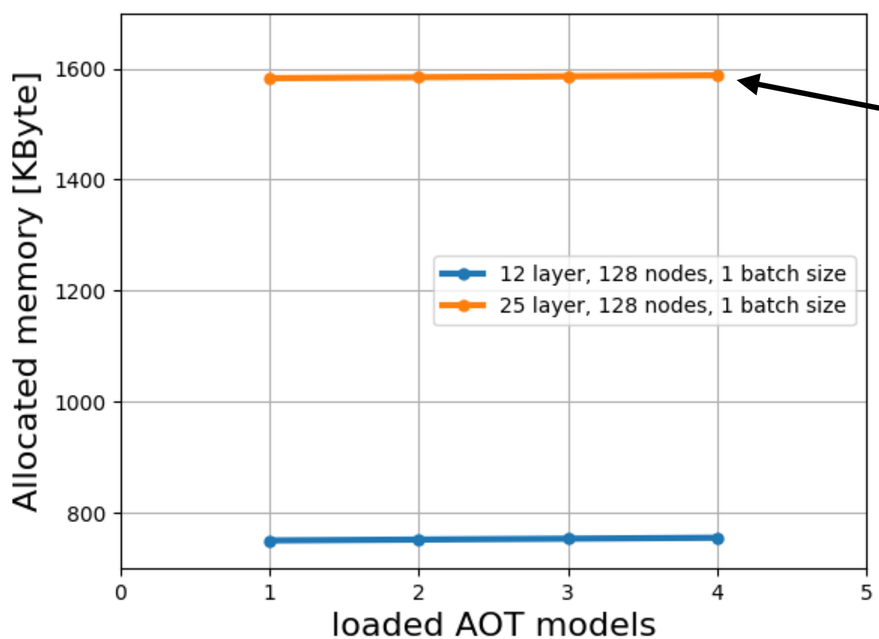
N layers	AOT model *.o [kB]	trainable weights [kB]
12	749	768
25	1581	1619

Memory allocation of Cpp wrapper



add model size
as offset

total Memory allocation (weights + wrapper)

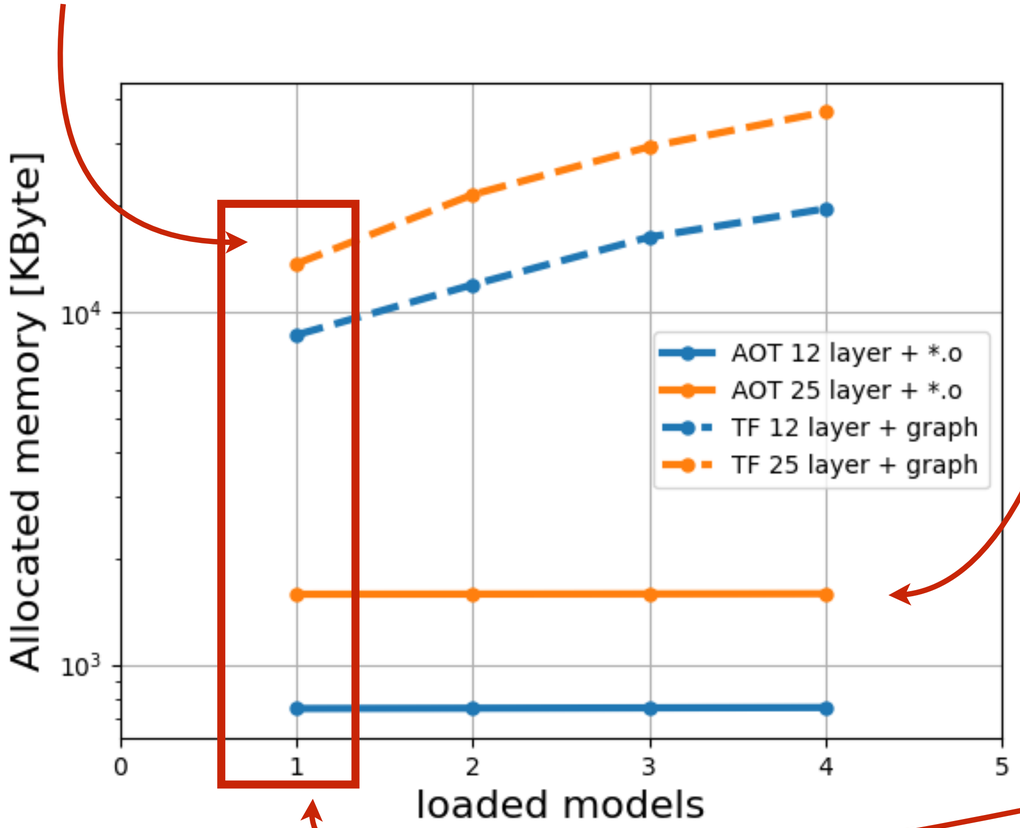


slope
barely seeable

Comparison of static memory allocation

Bare TF

1. `tf::Graph` **larger** than model weights (1/2 of graph are meta information)
2. `tf::Session` always larger than `tf::Graph`
   - Model weights are likely **copied** into each session
3. Absolute size per session quite large



AOT

1. Footprint of compiled c-function solely driven by **size of model weights (small slope)**
   - C++ wrappers reserve buffers for input, output **and all** intermediate layers, almost **no overhead**
2. One wrapper needed per expected batch size
   - Model weights in c-function might be **sharable**
3. Absolute sizes small compared to bare TF objects

- Also fully independent of `libTensorflow.so` at runtime, saves several hundred MBs on top

Note: only need to look at **1** loaded model once cmssw#40161 is merged

how buffer are allocated

Allocated Buffers

- First 2 bits are used to encode "kind", else is value
- kind = 0 ⟶ layers, 1 ⟶ temp (output), 2 ⟶ entrypoint (input), 3 ⟶ stackbuffer (intern)
- sum of 0 = *.o file (loaded once)

```
// Number of buffers for the compiled computation.
static constexpr size_t kNumBuffers = 25;

static const ::xla::cpu_function_runtime::BufferInfo* BufferInfos() {
  static const ::xla::cpu_function_runtime::BufferInfo
    kBufferInfos[kNumBuffers] = {
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({131072ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({32768ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({20480ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({514ULL, 0ULL}),
::xla::cpu_function_runtime::BufferInfo({161ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({33ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({16ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({4097ULL, ~0ULL})
    };
  return kBufferInfos;
}
```

Buffer value	Buffer size (B)	Buffer size (kB)	kind
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
131072	32768	32.0	0
32768	8192	8.0	0
20480	5120	5.0	0
514	128	0.125	2
161	40	0.0390625	1
33	8	0.0078125	1
16	4	0.00390625	0
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
4097	1024	1.0	1

SumBuffer: 750.2 kB,

ModelTypeBuffer Calculated: 767.8 kB

Layer weights

- Formula to calculate number of parameters:
- 25 Hidden Layer network:
- $\underbrace{32 * 64 + 3 * 64}_{\text{Input Layer}} + \underbrace{64 * 128 + 128 * 3 + 24*(128^2 + 128 * 3)}_{\text{Hidden Layer}} + \underbrace{128 * 10 + 10}_{\text{Output Layer}} = 414538$ --> 1658152 Bytes

12 Hidden Layer network:

- $\underbrace{32 * 64 + 3 * 64}_{\text{Input Layer}} + \underbrace{64 * 128 + 128 * 3 + 11*(128^2 + 128 * 3)}_{\text{Hidden Layer}} + \underbrace{128 * 10 + 10}_{\text{Output Layer}} = 196554$ --> 786216 Bytes

*.o file offset

- Pruning removed nodes that does not change the values:
  - ▻ Bias tensors removed (default value = 0)
  - ▻ Batch norm Gamma and Beta removed (default = 1, 0)
- (3 * 64 + 3 * 12 * 128 + 10)*4/1024 = 18.7890625
  - 64, 128 , 10 are shape of network
  - bias, gamma and beta have same shape (thus * 3)
  - 4 since 32 Bit-float weights

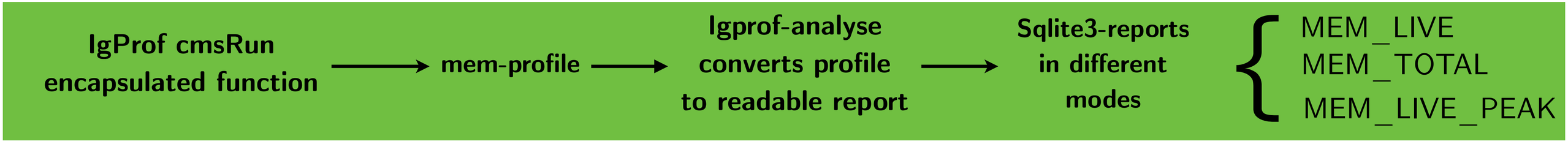
c-function consists mainly of model weights
difference is result of pruning (bias and batchnorm)

N layers	AOT model *.o [kB]	trainable weights [kB]
12	749	768
25	1581	1619

IgProf memory profiler

Overview: Memory measurement with IgProf & Test Setup

- Use **“Ignominious profiler”** (IgProf, talk) in memory-profile mode
- Encapsulate the code to be measured in a function
  - Trackable by name
  - Easy readout with with SQLite and Python (sqlite3)
  - Separation of functions into (e.g.) graph loading, session loading, inference calls
- **Workflow:**



- IgProf measures `malloc()`, `free()` of heap memory
  - Different memory metrics:
    - ▻ **MEM_TOTAL**: accumulated `malloc()`
    - ▻ **MEM_LIVE**: difference `malloc()` and `free()` for given interval
    - ▻ **MEM_LIVE_PEAK**: biggest single `malloc()`

Available metrics

- IgProf measures `malloc()`, `free()` of heap memory
  - Different memory metrics:
    - ▷ **MEM_TOTAL**: accumulated `malloc()`
    - ▷ **MEM_LIVE**: difference `malloc()` `free()` for given interval
    - ▷ **MEM_LIVE_PEAK**: biggest single `malloc()`
- Typical IgProf Report:

Rank	% total	Counts: to / from this	Counts: Total	Calls: to / from this	Calls: Total	Paths: Including child / parent	Paths: Total	Symbol name
	0.12	4,329,002	4,329,002	56,530	56,530	1	1	virtual thunk to PerfTesterTF::analyze(edm::Event const&, edm::EventSetup const&)
[1697]	**0.12**	**0**	**4,329,002**	**0**	**56,530**	**1**	**1**	**PerfTesterTF::loadGraph()**
	0.12	4,329,002	4,329,002	56,530	56,530	1	1	tensorflow::loadGraphDef(std::__cxx11::basic_string<char, std::char_traits<char>,

Encapsulated function to measure

Allocated memory in bytes

Number of allocations

Function call history

How does profiling (**IgProf**) works?

- Profiling divided into 3 steps:
- 1: Link Profiler with your program
  - **IgProf** set hooks before every function call
    - ▻ No change to application/build process necessary
- 2: Produce profile reports by execute the program
  - **IgProf** can run in 3 modes:
    - ▻ CPU-runtime (-pp) statistical sampling based performance profiles
    - ▻ memory (-mp) total dynamic memory allocations
    - ▻ Instrumentation () time spent in given function (±ns)
  - Store information of the parent and child stack frame
- 3: Analyse your profile
  - **IgProf-analyse** creates human-readable profile reports

Linking Mode Destination of Profile Program to profile Arguments of the program Redirect stdout/-err to logfile

```
igprof -mp -o "profile_dir.mp" cmsRun ./cmssw_cfg.py &> "$log_dir.mp.log"
```

General information about “profiler”

- Typical program problems: too slow, consumes too much memory, its doing both!
- Computer programs can be large and complex with multiple subprocess being invoked
  - Making it hard to identify inefficient parts of the program or bugs
- Profiling gives quantified answer about:
  “How much does each function in this program consume resource X?”
- Profiler collecting data mid execution of the program
  - If certain feature is not used, it will not show up in the profile
- CMS own profiler **“Ignominious profiler”** (IgProf) the (https://igprof.org/index.html)

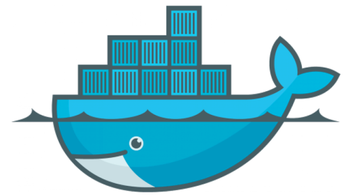
Analyse the profile

- Process profile statistics using the **“IgProf-Analyser”** tool
  - Possible report output: ASCII-text and sqlite3-database
  - Preferred way: sqlite3 (need command line sqlite3 app)

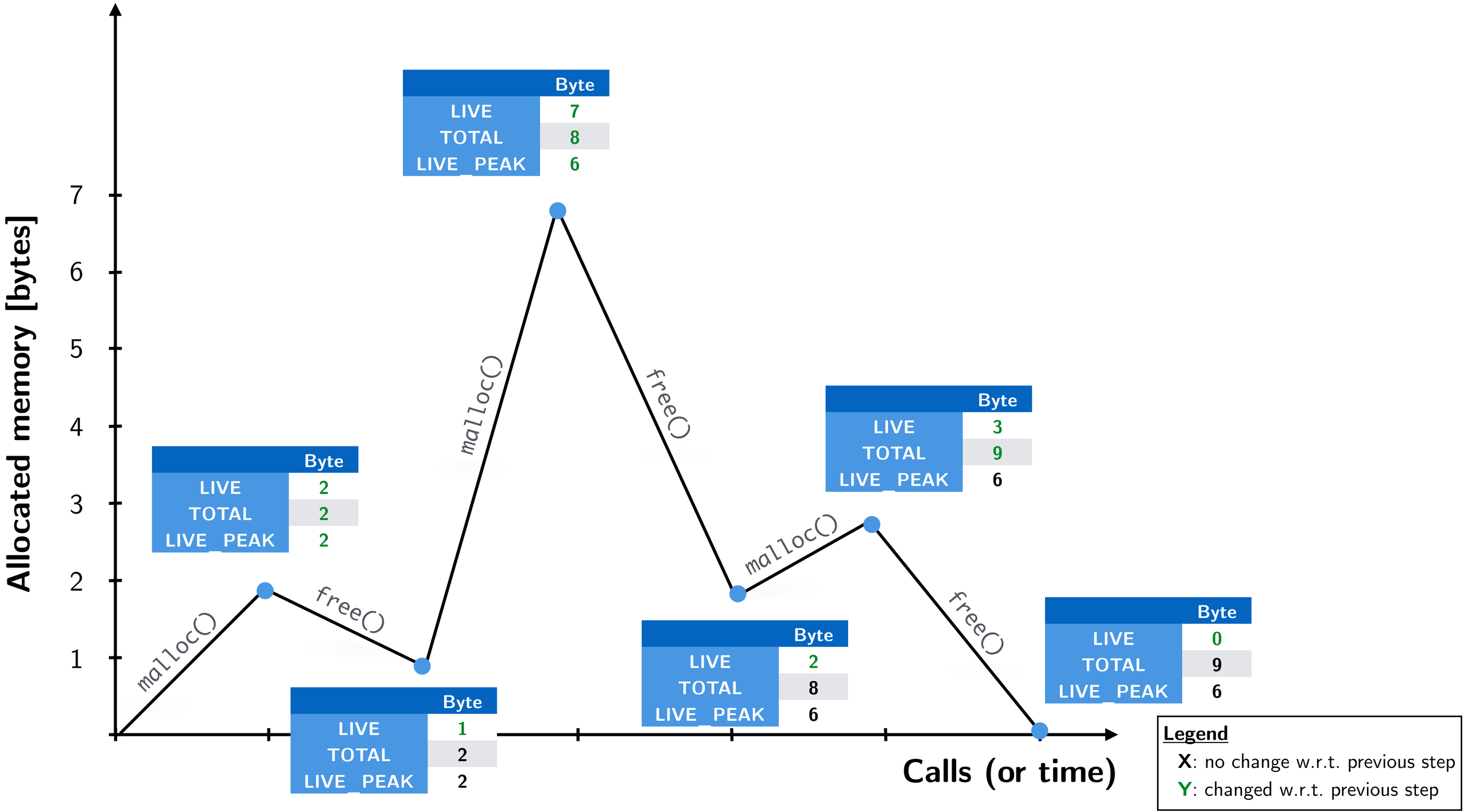
```
igprof-analyse --sqlite -d -v -g -r "MEM_TOTAL" "profile_dir.mp" | sqlite3 "dst_report.sql3"
```

  - ▻ Enables post-processing and plotting (pythons sqlite3)
  - ▻ Enables web-navigation using **“IgProf-Navigator”**, need CGI-open area

- Easiest way to navigate is Docker image (docker pull igprof/igprof) with port forwarding



Visualization of metrics



Inspect logs with the browser: 127.0.0.1:PORT

Sorted by cumulative cost

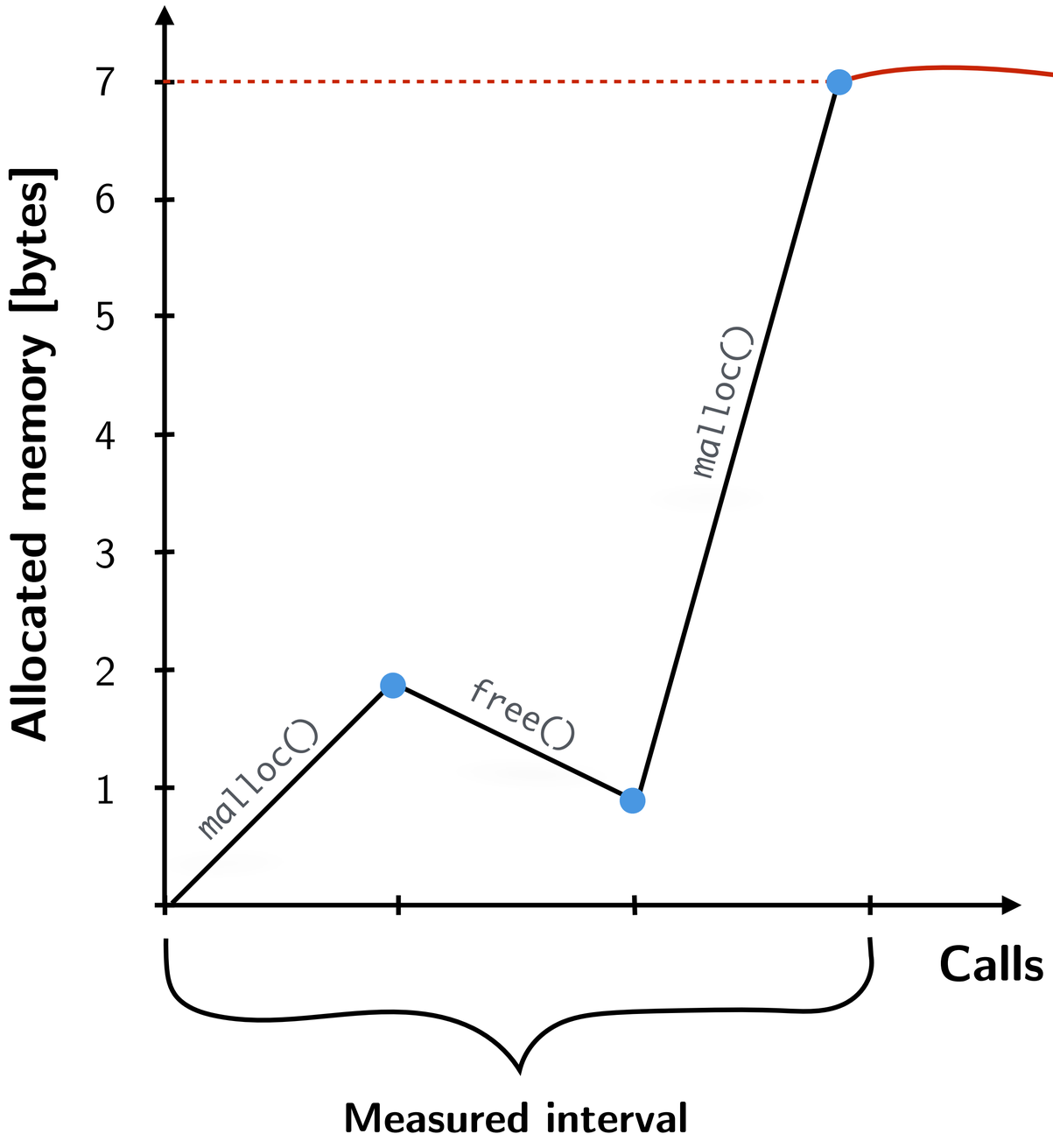
(Sort by self cost)

Rank	Total %	Cumulative	Calls	Symbol name
795	0.79	28,413,804	8,042	_PyObject_MakeTpCall
796	0.78	28,397,192	222,976	std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, s
797	0.78	28,354,666	23,778	ClingMemberIterInternal::DCIter::DCIter(clang::DeclContext*, cling::Interpreter*)
798	0.78	28,052,796	302,319	PerfTesterTF::loadSessions()
800	0.78	28,052,788	302,318	tensorflow::createSession(tensorflow::GraphDef const*, tensorflow::SessionOptions&)
799	0.78	28,052,788	302,318	tensorflow::createSession(tensorflow::GraphDef const*, int)
801	0.77	28,046,280	26,956	TClass::CallShowMembers(void const*, TMemberInspector&, bool) const'2
802	0.77	28,007,926	302,218	tensorflow::DirectSession::Create(tensorflow::GraphDef const&)
1698	0.12	4,329,002	56,530	tensorflow::loadGraphDef(std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
1697	0.12	4,329,002	56,530	PerfTesterTF::loadGraph()

Create 1 or more Session

Load 1 or more Graphs

Measurement of the setup phase



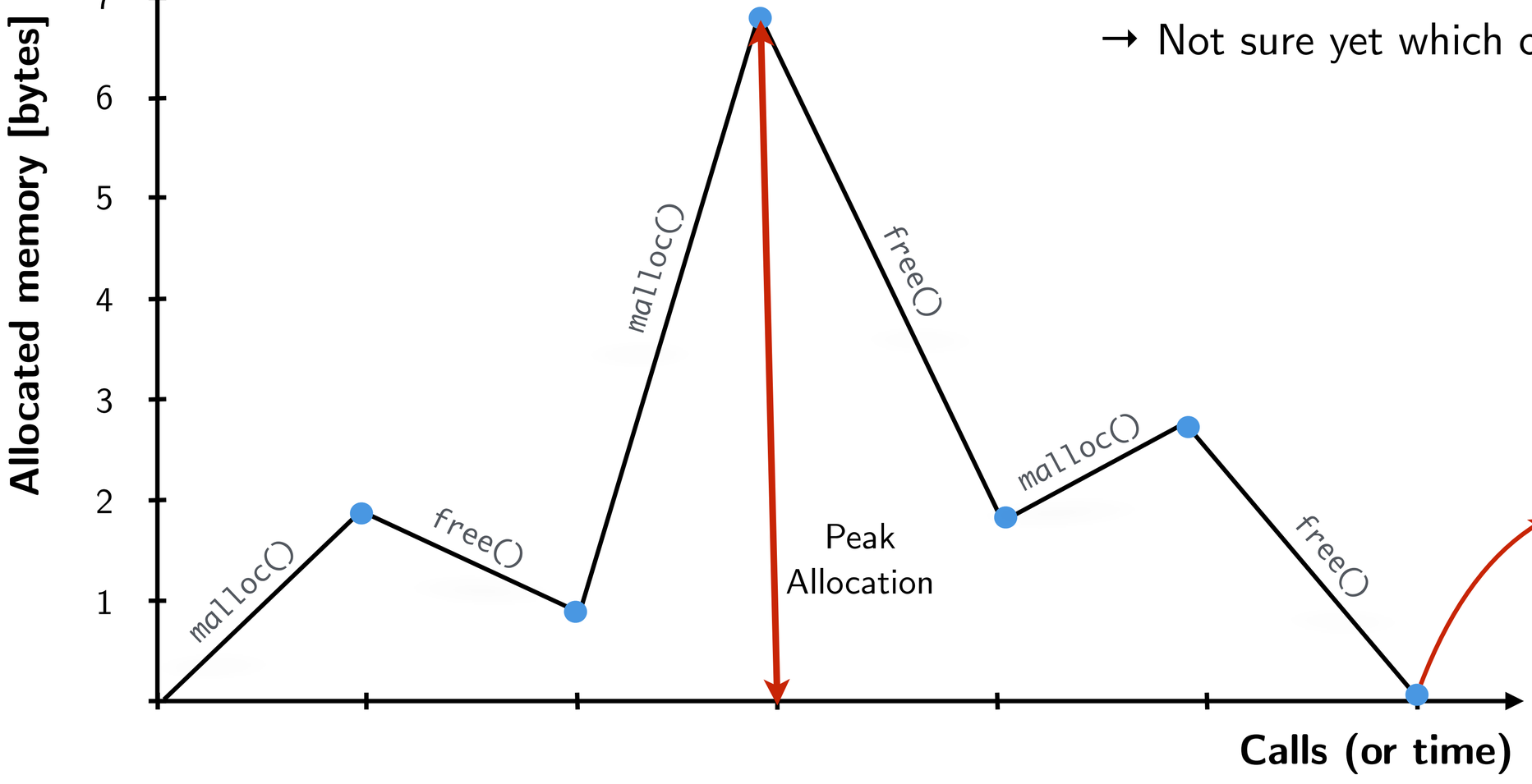
- For the **setup phase**, we are only interested in MEM_LIVE, i.e., amount of physical memory blocked by setup functions
- We know exactly which function / interval to measure (e.g. `tf::loadGraph()`)
- Applies to global & per-thread measurement

	Byte	
LIVE	7	→ `malloc()` - `free()`
TOTAL	8	→ Σ `malloc()`
LIVE_PEAK	6	→ max single `malloc()`

Measurement of the event phase

- For inference calls in event loop (`analyze()`), need to know maximum memory allocation ↕ within the evaluation call (e.g. `tf::run(session, ...)`)
- Open question: which metric to use?
  - **LIVE**: difference between "after" - "before", should be 0 in absence of leaks
  - **LIVE_PEAK**: only measures largest, single allocation
  - **MAX**: "records the largest single allocation by any function" (link)

→ Not sure yet which one to use



	Byte
LIVE	0
TOTAL	9
LIVE_PEAK	6

Potential advantages in CMSSW

- **Current TF workflow**
  - Training in python, then export to frozen graph
  - Load graph in C++ (once per process)
  - Mount graph onto session (once per stream module instance)
  - Use TF C++ API to feed, run and read
    - ▻ No simple handles for optimization

- **Potential AOT benefits**
  - Easy to use graph and compiler optimizations
  - Potentially faster (?), might depend on model
  - Reduced memory footprint
    - ▻ No need to load the full `libtensorflow.so`
    - ▻ Model weights only once in memory
    - ▻ Virtually no memory overhead over multiple threads

ML Production

Inference Strategies

in CMSSW

Updates on TensorFlow AOT and XLA in CMSSW

Inference and Batching Strategy

Marcel Rieger, Peter Schleper, Bogdan Wiederspan

Title on frontpage

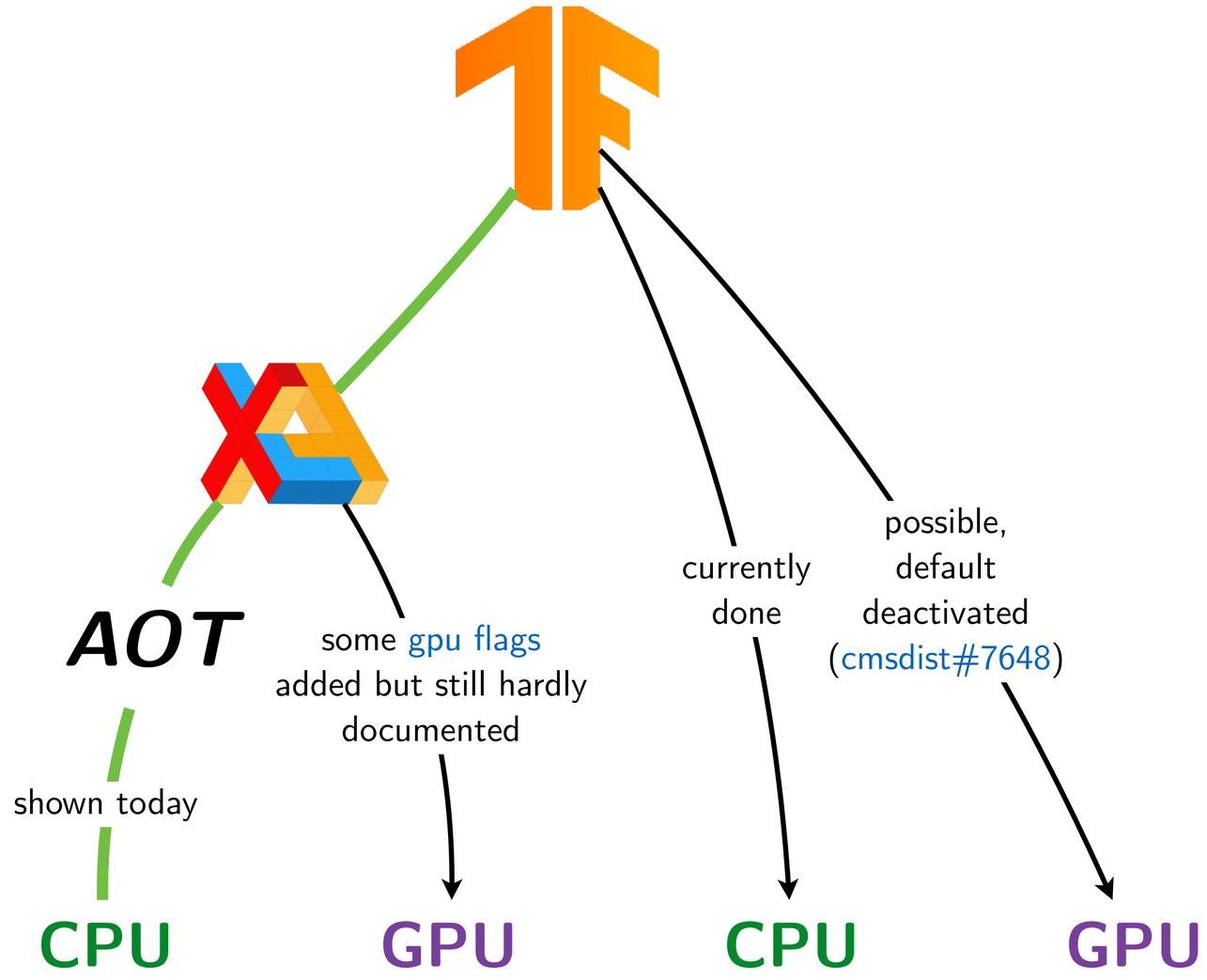
01.01.2022

SPONSORED BY THE
Federal Ministry of Education and Research

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Summary from previous presentations

Reminder: TensorFlow in CMSSW



Reminder: What does XLA and AOT do?

Enables several types of graph optimizations

- On graph level:
  - kernel fusion
  - Buffer analysis for allocating runtime memory (eliminates intermediate caches)
  - Common subexpression elimination
  - Pruning of unused kernel
- On hardware level:
  - TPU, GPU or CPU (different backends)

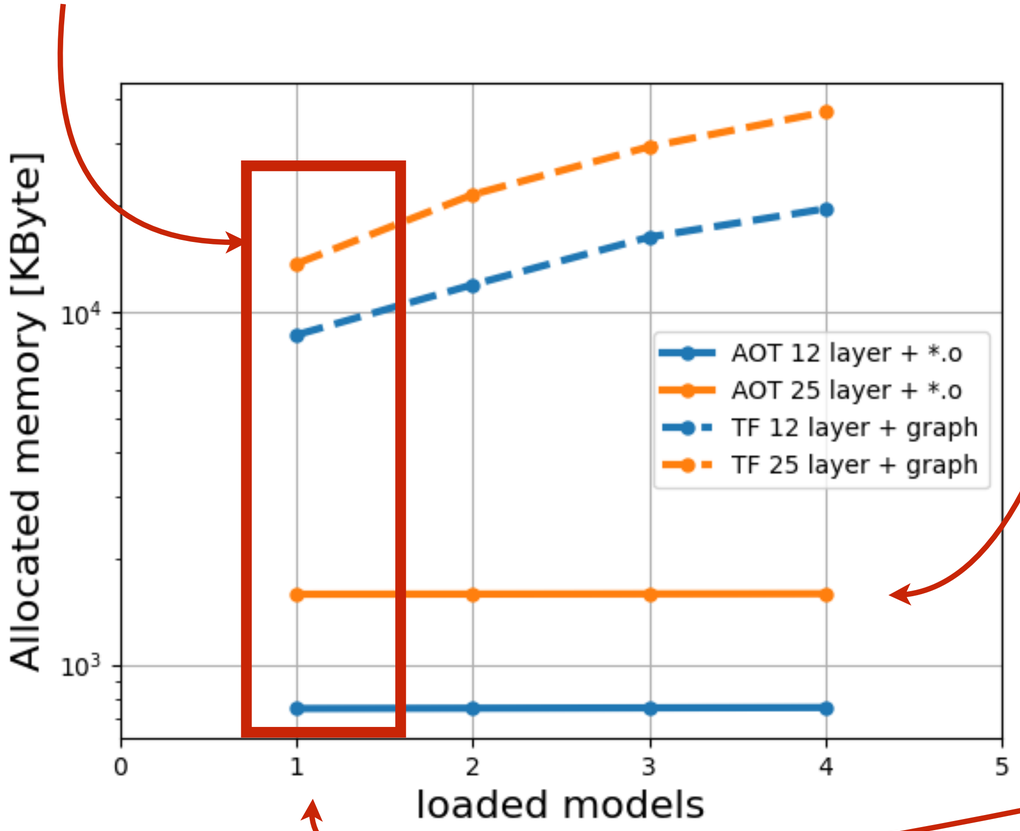
AOT

- Converts graphs into self-contained library
  - Graph becomes a series of C++ compute kernels
  - **No dependence** on main `libtensorflow.so`
- Pros:
  1. Reduced memory footprint
  2. Trivial multi-threading behavior
  3. Runtime potentially faster (depends on degree of optimization)
- Cons:
  1. No dynamic batching (fixed memory layout) but can be emulated (**padding/stitching**, **shown later**)
  2. Graph needs to be XLA compatible

Results of previous contribution (link ☞) on memory footprint

Bare TF

1. `tf::Graph` **larger** than model weights
   - About twice as large in shown tests
2. `tf::Session` always larger than `tf::Graph`
   - Model weights likely **copied** into each session
3. Absolute size per session quite large



AOT

1. Footprint of compiled c-function solely driven by **size of model weights**
   - C++ wrappers reserve buffers for input, output **and all** intermediate layers, almost **no overhead**
2. One wrapper needed per expected batch size
   - Model weights in c-function might be **sharable** (*)
3. Absolute sizes small compared to bare TF objects

- Also fully independent of `libTensorflow.so` at runtime, saves ~300 MBs on top

Note: only need to look at **1** loaded model once cmssw#40161 is merged

Results of previous contributions (link ☞) on model compatibility

Possible reasons to fail AOT compilation:

1. Kernel produce no predictable shapes (=no fix memory layout) at compile time (e.g tf.where)
2. No existing TF XLA implementation of the kernel
   - more XLA kernel implementation added with each TF update

XLA compatibility check for a given model:

1. requires TF-to-XLA conversion table, created by:

```
tf2xla_supported_ops --device="XLA_{CPU,GPU}_JIT"
```

2. Find match between table and nodes in graph, using cmsml tool:

example: check AOT compatibility of feed-forward network with batch-norm for TF v. 1.4

Operation	has XLA
MatMul	True
Identity	True
Sub	True
BiasAdd	True
Mul	True
Softmax	True
NoOp	True
Rsqrt	True
Selu	True
AddV2	False
ReadVariableOp	True
Const	True

network is not AOT compatible

Checklist for moving to production

Possible application workflow (for PAGs/POGs/DPGs)	**Action items (for us)**
1. Read documentation on how to integrate AOT models in CMSSW	→ Add section to cms-ml.github.io/documentation
2. Train network and store as SavedModel	→ Provide convenience function to `cmsml` (90% done)
▪ Either Keras model or `tf.function`	→ Provide AOT compatibility check to `cmsml` (90% done)
3. Integrate into CMSSW plugin	
▪ Add configuration to BuildFile.xml, e.g.: `<aot_compile model="/data/my_model" class="MyModel" batch_sizes="1 2 4 8"/>`	→ Automate compilation in scram tools
▪ Perform inference via `tfaot::Inference` helper class	→ Create class wrapping bare objects created by AOT, providing easier API, dynamic batching, ...
▪ Compile	→ Integrate AOT compilation tools into CMSDIST (85% done, need to integrate with general TF update)
4. Measure runtimes and memory metrics	→ Integrate with performance measurement tools (project by Nathan Prouvost, status shown soon)

107 More granular roadmap

- Setup
  - ☑ Update performance measurement techniques (→ created semi-automated tool based on IgProf)
  - ☑ Move to more stable test machine (→ moved setup to cmsdev test machine, less volatile)
- Updates to TF installation (**high priority**)
  - ☑ Enable AOT in central CMSDIST stack (→ done by Marcel, see update here)
  - ☑ Find method to check XLA/AOT compatibility (presented last talk, see here)
  - ☐ Provide tools for working with compiled model (tensor access + dynamic batching if needed) (→ ongoing ⏳)
  - ☐ Automate model compilation within scram
- Measurements
  - ☑ look into possible batch strategies (→ **shown today**)
  - ⏳ Use DeepTau as testing model
    - ☑ AOT compile model
    - ⏳ Adjust inference workflow in CMSSW to use AOT (→ ongoing ⏳)
    - ☐ Repeat performance measurements
  - ☐ Measure memory consumption in events processing phase

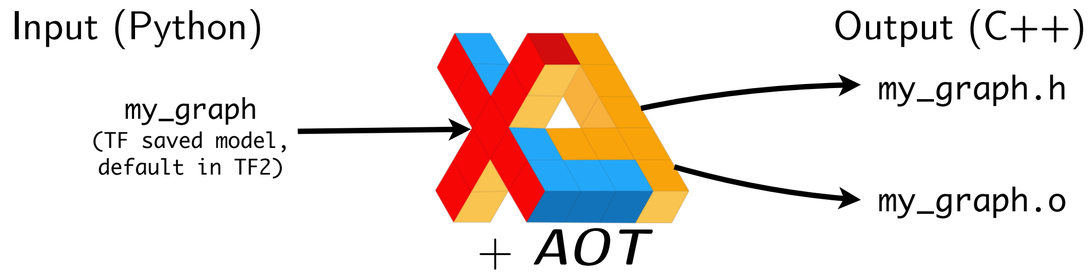
Inference & batching strategies

Static batching and consequences for AOT

- **AOT has no support for dynamic batching**
  - No interest expressed by Google to add support in the near future

- **Resulting challenges for AOT**
  - *Note:* most applications in CMSSW are single-batch anyway
  - Model inference that uses batching (e.g. per jet / tau / ...) would require **multiple compiled models** with different **static** batch sizes
    - Each compiled AOT model consist of one header and one object file (with weights)
    - Having multiple models requires loading weights (*.o file) **multiple times**

    → **Not feasible** to compile every possible batch size



- **Possible solution**

  ! Emulate dynamic batching with different strategies

Possible AOT inference strategies

- Batching strategies to emulate dynamic batching: **normal**, **stitching, padding**
- **Example**: batch size 3

Normal
(exact batch size)

normal 3
model: aot_batch_3.h
input: [e1, e2, e3]
output: [f(e1), f(e2), f(e3)]

Stitching
(pure additive combination)

stitching 2+1
model: aot_batch_{1,2}.h
input: [e1],[e2, e3]
output: concat([f(e1)], [f(e2), f(e3)])

stitching 1+1+1
model: aot_batch_1.h
input: [e1], [e2], [e3]
output: concat([f(e1)], [f(e2)], [f(e3)])

Padding
(allow marginal zero padding)

padding (4 - 1)
model: aot_batch_4.h
input: [e1, e2, e3, 0]
output: slice[f(e1), f(e2), f(e3), f(0)]

NB: more events require more sophisticated combinations **!**

AOT inference strategies: padding and stitching

- **batch strategy: padding**
  - runtime / batch always **equals** the **unpadded model** eq: (8 - {1,2,3}) ≈ 8

  → 0's are still fully evaluated

- **batch strategy: stitching**
  - runtime / batch **always slower** than **normal**
  - ***exception***: single batching → {16,32} * 1

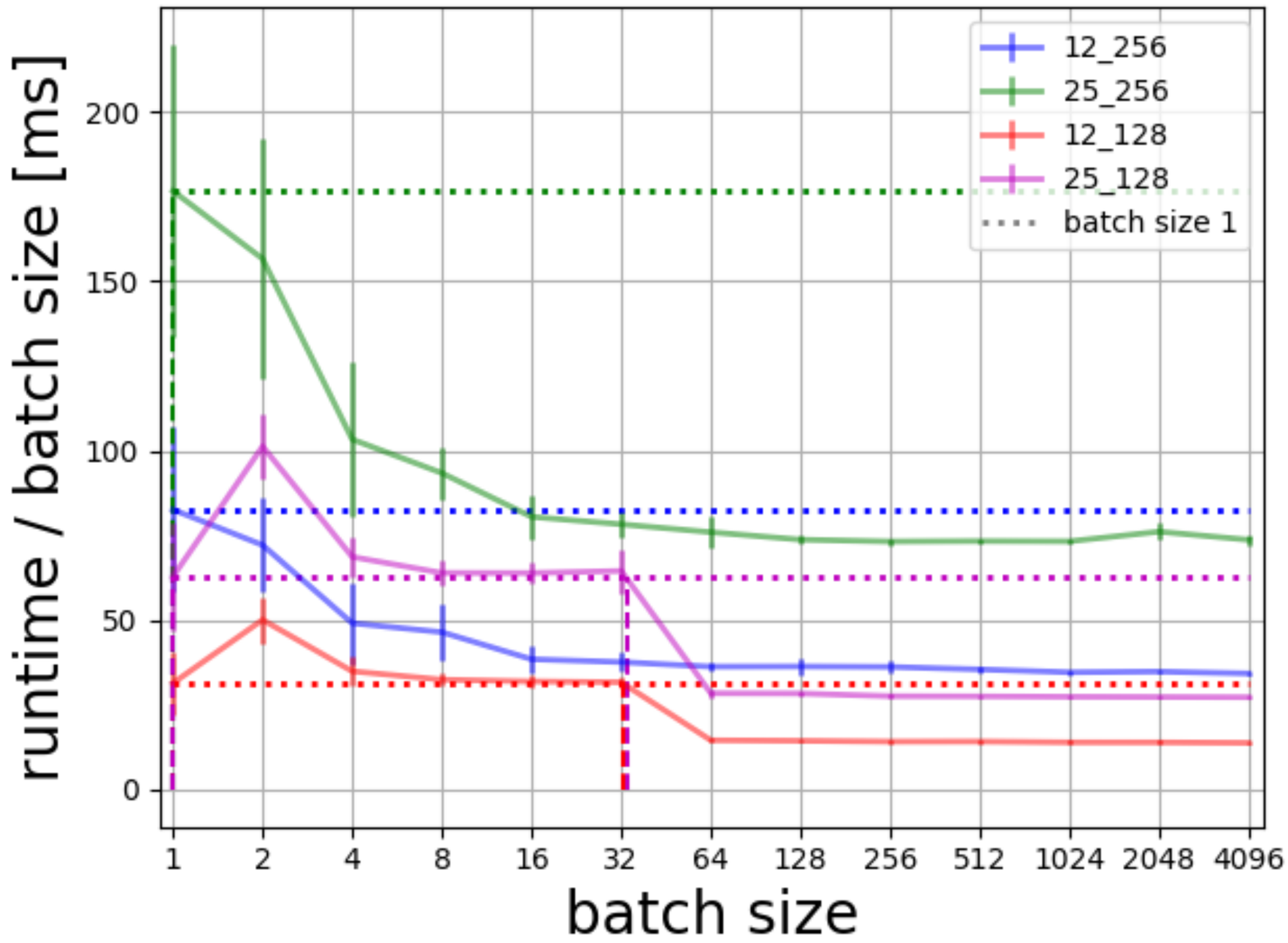
  → stitching batch size 1 is preferred strategy

runtime for network with 25 layers, 128 nodes				
Events	Batch strategy	Runtime [ms] / batch	Mean runtime [ms]	Std runtime [ms]
1	1	62.84	62.8	15.8
2	2	101.28	202.6	18.6
4	4	68.67	274.7	22.4
8	8	63.93	511.5	28.2
16	16	63.91	1022.5	50.3
32	32	64.53	2064.9	206.3
64	64	28.4	1817.8	79.0
16	16 * 1	62.84	1005.4	63.2
32	32 * 1	62.84	2010.9	89.4
5	(8 - 3)	64.48	515.9	38.4
6	(8 - 2)	64.46	515.6	38.5
7	(8 - 1)	64.17	513.4	33.9
5	4 + 1	131.51	337.52	27.4
6	4 + 2	169.95	477.2	29.1
7	4 + 2 + 1	232.79	540.08	33.1

AOT inference strategies: stitching batch size 1 models

- **preferred batch strategy is a hyper-parameter depending on models complexity**
  - **simple models (128 nodes)** runtime is dominated by **overhead**
  - **complex models (256 nodes)** or **large batch sizes** benefit from **vectorisation**

- **batch size 1 fast** for **simple models**
- threshold of strategy transition depends on the complexity of the model
  - vertical line indicates threshold
  - moves towards smaller batch size with increasing model complexity
  - independent of numbers of layers (no vectorisation benefit)
- preferred batch strategy:
  - "stitching" left to the threshold
  - "normal" right to the threshold



! *Note:* most applications in CMSSW are single-batch

Summary

- Gained insights into possible inference & batching strategies for AOT models to emulate dynamic batching
  - runtime most performant for smallest necessary batch size → not feasible to compile every batch size
- presented first measurements of padding and stitching strategy
  - "padding" shows no runtime benefit compared to "normal"
  - "stitching" multiple models of batch size 1 together is most performant for less complex models
  - "normal" batching strategy preferred for complex models
- General: choice of batching strategy depends on models complexity
  - threshold of strategy transition is a hyperparameter → needs to be measured for each model (we discuss about using Nathans automatised "ML Prof" for this)

- **Further investigation necessary:**
  - are models weights (*.o) shareable between model objects?

Backup

Model Building context

Today: Challenges during AOT compilation

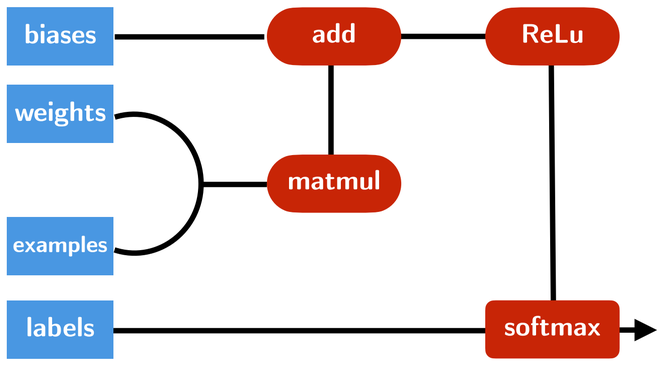
Question: When does AOT compilation does (not) work?

(to gain insights to possible constraints for models to be compiled)

Compilation workflow

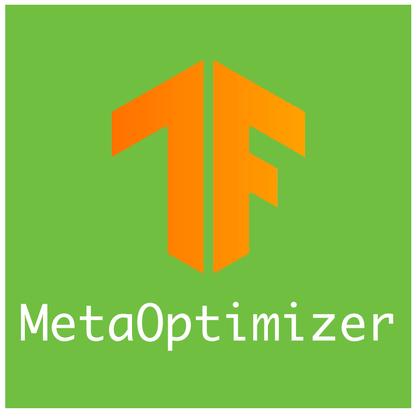
Step 0: Prepare graph

model saved with
"tf.saved_model.save"



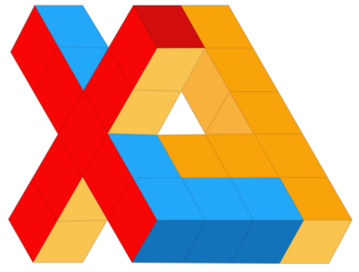
Add signatures to saved_model with static shape in "TensorSpec" (tool is already created)

Step 1: Optimization by "Grappler" (MetaOptimizer)



configuration through
"tf.config.optimizer.set_experimental_options"

Step 2: Optimization using XLA



Flags during compilation (AOT_XLA_FLAGS, XLA_DEBUG_FLAGS)

Step 1: What does Grappler do in detail

- Get graph for specific signature (e.g. `__inference_standard_lstm_17024_....`)
- Optimize graph with TF's MetaOptimizer (more details)
  - Read progress logs as following:

    `<Optimizer Method>: Graph size <AFTER> (<DIFFERENCE>), <needed_time>`

```
2022-11-21 15:36:27.616914: I tensorflow/core/grappler/optimizers/meta_optimizer.cc:1137] Optimization results for grappler item:
graph_to_optimize
  model_pruner: Graph size after: 252 nodes (0), 963 edges (0), time = 5.325ms.
  implementation_selector: Graph size after: 252 nodes (0), 963 edges (0), time = 4.803ms.
  function_optimizer: Graph size after: 9221 nodes (8969), 17930 edges (16967), time = 356.309ms.
  common_subgraph_elimination: Graph size after: 7329 nodes (-1892), 15056 edges (-2874), time = 94.834ms.
  constant_folding: Graph size after: 7085 nodes (-244), 14618 edges (-438), time = 295.155ms.
  shape_optimizer: shape_optimizer did nothing. time = 13.713ms.
  arithmetic_optimizer: Graph size after: 7085 nodes (0), 14348 edges (-270), time = 129.22ms.
  layout: Graph size after: 7085 nodes (0), 14348 edges (0), time = 170.869ms.
  remapper: Graph size after: 7085 nodes (0), 14348 edges (0), time = 42.489ms.
  loop_optimizer: Graph size after: 7085 nodes (0), 14323 edges (-25), time = 59.98ms.
  dependency_optimizer: Graph size after: 2838 nodes (-4247), 5130 edges (-9193), time = 107.603ms.
  memory_optimizer: Graph size after: 2838 nodes (0), 5130 edges (0), time = 157.728ms.
  model_pruner: Invalid argument: Graph does not contain terminal node StatefulPartitionedCall_5.
  implementation_selector: implementation_selector did nothing. time = 0.402ms.
  function_optimizer: Graph size after: 2838 nodes (0), 5130 edges (0), time = 82.992ms.
  common_subgraph_elimination: Graph size after: 2528 nodes (-310), 4342 edges (-788), time = 26.611ms.
  constant_folding: Graph size after: 2411 nodes (-117), 4109 edges (-233), time = 82.159ms.
  shape_optimizer: shape_optimizer did nothing. time = 5.279ms.
  arithmetic_optimizer: Graph size after: 2411 nodes (0), 4109 edges (0), time = 49.24ms.
  remapper: Graph size after: 2411 nodes (0), 4109 edges (0), time = 14.993ms.
  dependency_optimizer: Graph size after: 2409 nodes (-2), 4105 edges (-4), time = 27.301ms.
Optimization results for grappler item: __inference_standard_lstm_17024_specialized_for_StatefulPartitionedCall_1_StatefulPartitio
nedCall_StatefulPartitionedCall_StatefulPartitionedCall_StatefulPartitionedCall_rnn_3_StatefulPartitionedCall_StatefulPartitionedC
all_at_graph_to_optimize
```

Step 2: Op compatibility with XLA

- 2 possible reasons why op-kernels might fail to AOT compile
  - Kernel produces no predictable shapes at compile time (e.g tf.where)
  - No existing XLA implementation of the kernel
    - e.g. non-unrolled LSTM might not be AOT compatible (at least not in TF2.6)

```
ValueError: Detected unsupported operations when trying to compile graph __inference_standard_lstm_67406_specialized_for_StatefulPartitionedCal
l_1_StatefulPartitionedCall_StatefulPartitionedCall_StatefulPartitionedCall_StatefulPartitionedCall_rnn_0_StatefulPartitionedCall_StatefulParti
tionedCall_at_graph_to_optimize_frozen_263[] on XLA_CPU_JIT: Enter (No registered 'Enter' OpKernel for XLA_CPU_JIT devices compatible with node
 {{node while/enter/_2}})while/enter/_2
```

→ Node with the name "while/enter/_2" is not XLA_CPU_JIT compatible

→ kernel named "Enter" hat no XLA implementation

- **Goal:** need a tool that takes any model and
  - checks if a model is AOT / XLA compatible
  - provides feedback in case it isn't (e.g. list incompatible ops and suggest known alternatives)
- Can help already during the model development phase, rather than after the fact during integration!

Two ways to check model compatibility

Brute force method

- run graph in
  - TF1 by calling "`tf.contrib.compiler.xla.compile(your_function)`"
  - TF2 by calling "`tf.xla.experimental.compile(your_function)`"
  - TF2 by using the "`tf.function(experimental_compile=True)`" decorator
  - AOT compiler
- write wrapper to catch the first op name that is not compatible

More sophisticated method

- TF knows which OP is registered by a flag
- TF to XLA kernel implementation is defined in tensorflow/compiler/tf2xla/kernels
- the following function is used to register an Op:
  REGISTER_XLA_OP(NameObject('OperationName').TypeInformation, 'OpNodeName') e.g

```
REGISTER_XLA_OP(Name("MatMul").TypeConstraint("T", kMatmulTypes), MatMulOp);
```

Sophisticated method

General idea:

create table of **all compatible ops** and find matching graph nodes

1. Get the XLA op table

- Arch. dependent, can only be created during full TF compilation in CMSDIST:
  `"bazel run -c opt -- tensorflow/compiler/tf2xla:tf2xla_supported_ops --device=XLA_(CPU or GPU)_JIT"`
- Both for CPU and GPU (example), contains
  - Compatible ops by name (unique)
  - All accepted type variations

2. Check all nodes in a graph

- Input: SavedModel or Graph (*.pb)
- Get used operations from GraphDef
  - "name" node name (arbitrary identifier)
  - "op" used operation name (unique identifier)
  - "attr" attributes (e.g shape, inputs, types)

→ Build match with table and provide feedback

- Tool already created
- To be integrated to cmsml once table is compiled in CMSDIST TF compilation (Marcel)

Example output method

Operations used by a normal feed-forward network with batch-norm

Operartion	has XLA
MatMul	True
Identity	True
Sub	True
BiasAdd	True
Mul	True
Softmax	True
NoOp	True
Rsqrt	True
Selu	True
AddV2	False
ReadVariableOp	True
Const	True

the network should not be AOT compatible

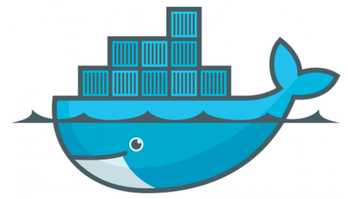
Working with IgProf

Analyse the profile

- Process profile statistics using the **“IgProf-Analyser”** tool
  - Possible report output: ASCII-text and sqlite3-database
  - Preferred way: sqlite3 (need command line sqlite3 app)

```
igprof-analyse --sqlite -d -v -g -r "MEM_TOTAL" "profile_dir.mp" | sqlite3 "dst_report.sql3"
```

    - Enables post-processing and plotting (pythons sqlite3)
    - Enables web-navigation using **“IgProf-Navigator”**, need CGI-open area
- Easiest way to navigate is Docker image (docker pull igprof/igprof) with port forwarding



Inspect logs with the browser: 127.0.0.1:PORT

Sorted by cumulative cost

(Sort by self cost)

Rank	Total %	Cumulative	Calls	Symbol name
795	0.79	28,413,804	8,042	_PyObject_MakeTpCall
796	0.78	28,397,192	222,976	std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, s
797	0.78	28,354,666	23,778	ClingMemberIterInternal::DCIter::DCIter(clang::DeclContext*, cling::Interpreter*)
798	0.78	28,052,796	302,319	PerfTesterTF::loadSessions()
800	0.78	28,052,788	302,318	tensorflow::createSession(tensorflow::GraphDef const*, tensorflow::SessionOptions&)
799	0.78	28,052,788	302,318	tensorflow::createSession(tensorflow::GraphDef const*, int)
801	0.77	28,046,280	26,956	TClass::CallShowMembers(void const*, TMemberInspector&, bool) const'2
802	0.77	28,007,926	302,218	tensorflow::DirectSession::Create(tensorflow::GraphDef const&)
1698	0.12	4,329,002	56,530	tensorflow::loadGraphDef(std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
1697	0.12	4,329,002	56,530	PerfTesterTF::loadGraph()

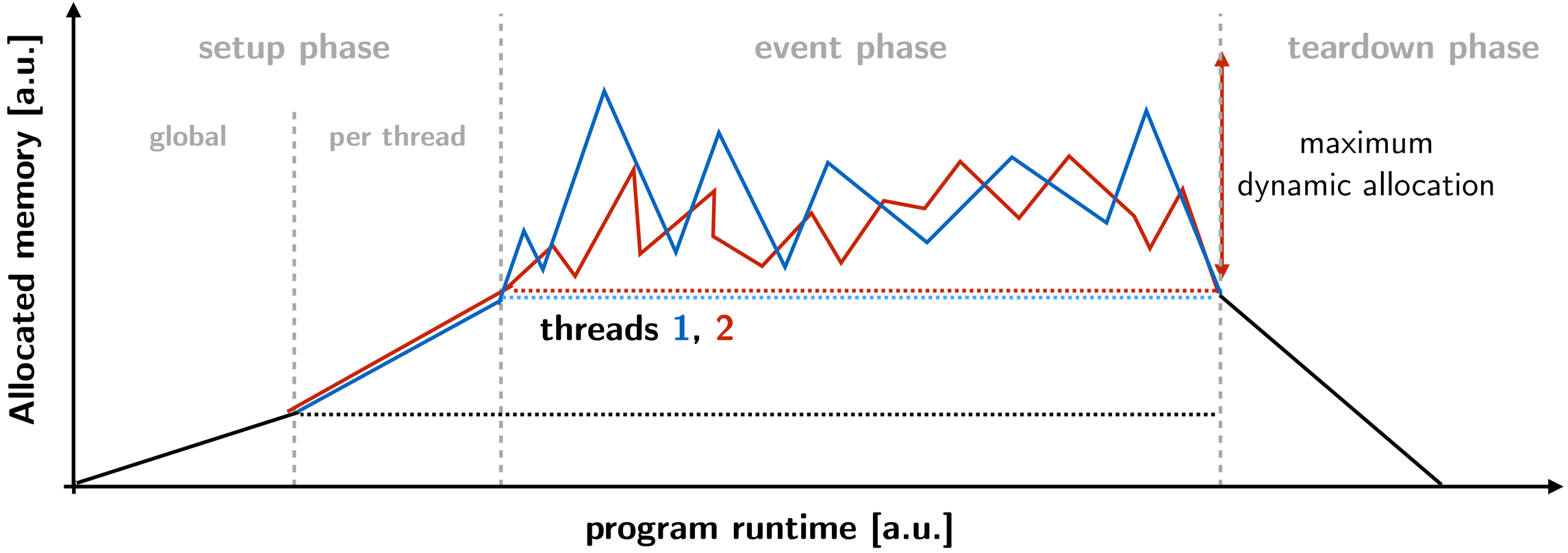
Create 1 or more Session

Load 1 or more Graphs

- We inspected memory usage of Tensorflow regarding sessions and graphs

IgProf memory measurement: **Tensorflow** event phase

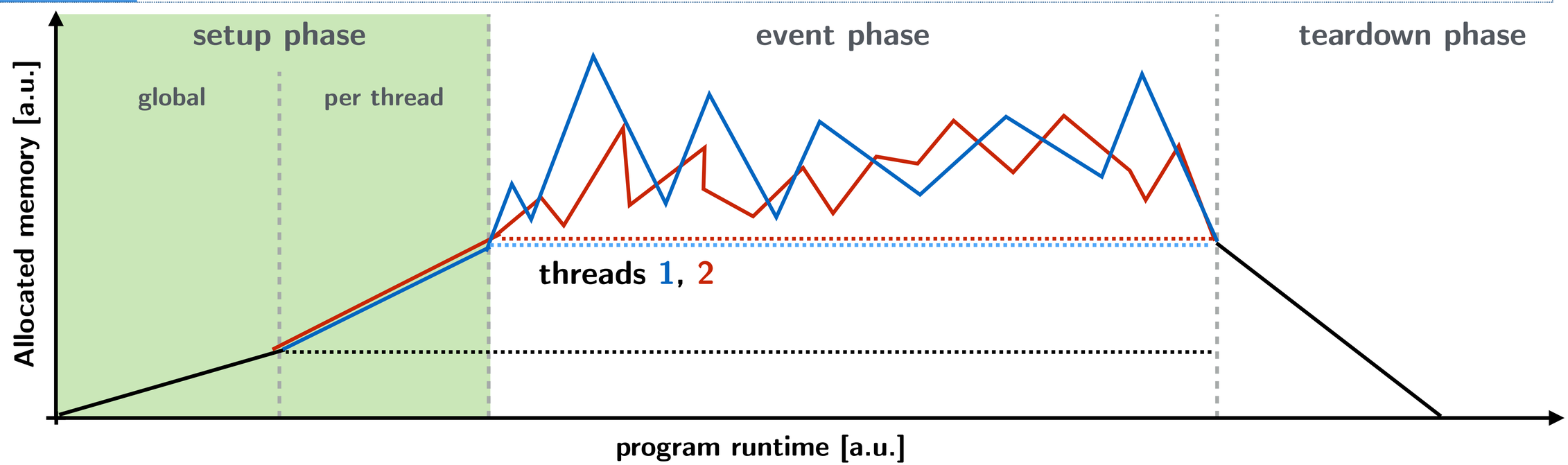
- **Event phase measurement not there yet**
- Problem 1: C++ TF interference has 2 ways to handle input and output tensor
  1. define input + output tensors in <u>each</u> new analyze call ⟶ allocate memory every time in event phase
  2. create instance variables ⟶ allocate memory 1x in setup phase
- Problem 2: need proper memory definition to classify event memory (maximum allocation?)
  - depends on the batch size ⟶ which batch size to look at?



Understanding of memory for different phases

- Talk: **Focus on setup phase**, since we can not measure event phase yet.
- global allocate memory once, per thread for each thread
- Comparison: **TF multiple sessions vs. loading multiple AOT models**

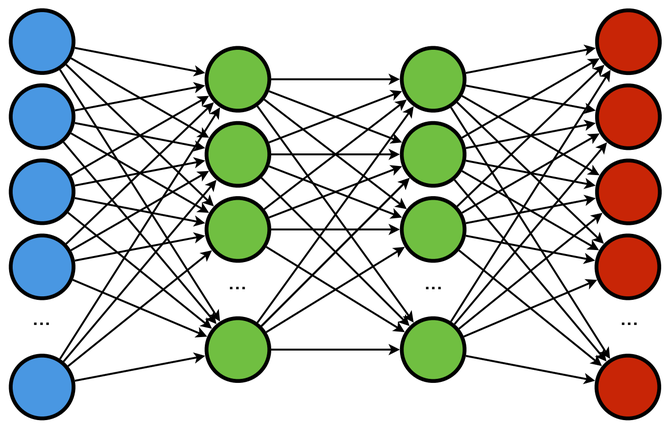
Phase	Description	Action in TF	Action in AOT
global setup	Before threads (stream modules) are launched	`tf::loadGraph();` (load model and weights into memory)	loading compiled model (external c-function in *.o file)
setup per thread	Footprint per thread, but before events are analyzed	`tf::loadSession();` (device placement & caching per thread)	`CppWrapper w;` (access to c-function & reserve buffers for input, output)
event phase	Resource consumption during event processing	`tf::run(session, ...);` (book inputs/outputs & evaluate model)	`w.run();` (evaluate model)



First performance study

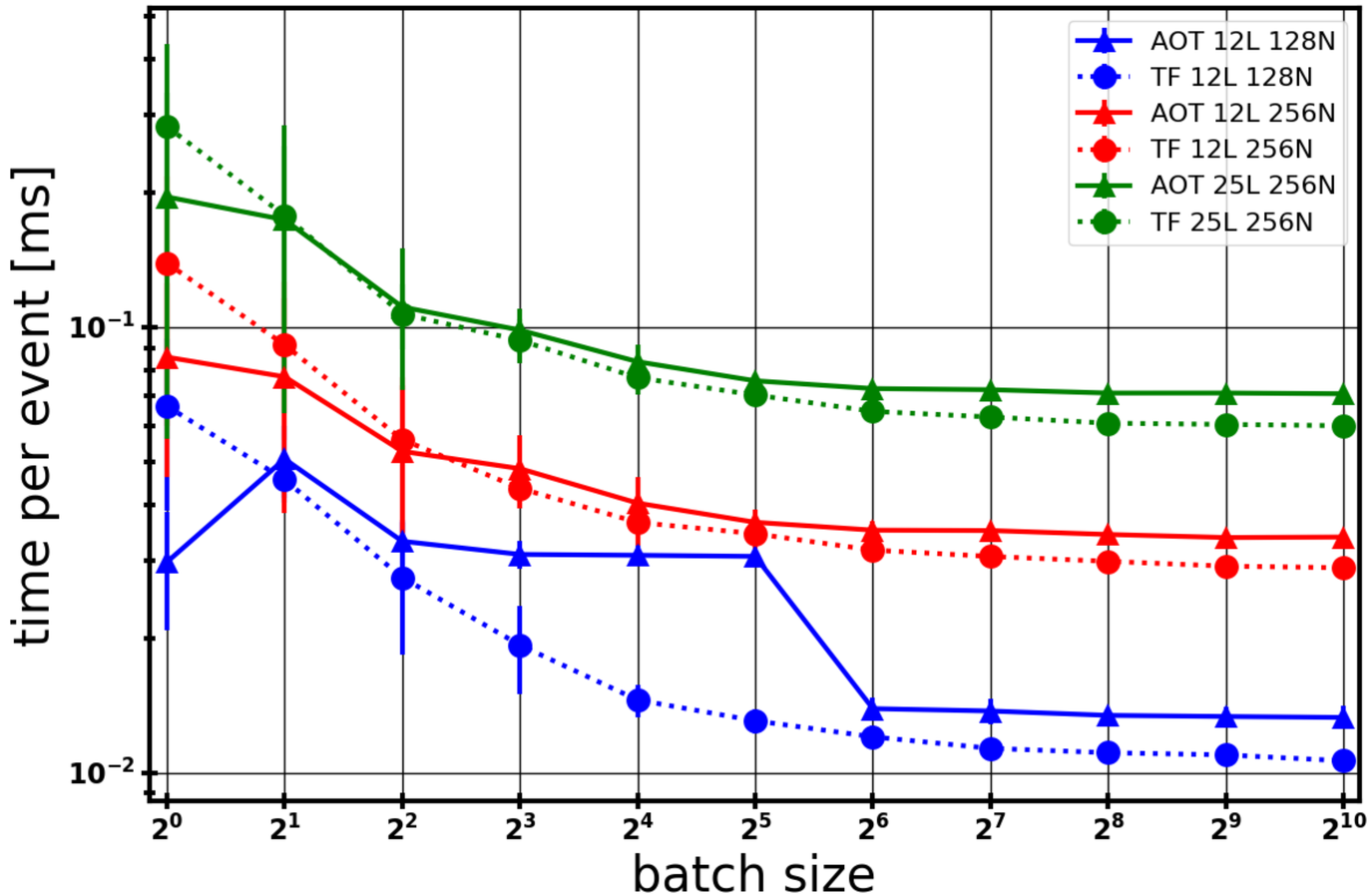
Study outline

- **Software stack**
  - `slc7_amd64_gcc10` with `CMSSW_12_4_0`
  - Using custom CMSDIST stack with TF XLA enabled and patched "eigen" library
- **Network and payload**
  - Created several feed-forward toy models
    - ▻ Up to 25 layers with 256 units, SELU activation
  - **CPU runtime tests performed on login-node:**
    - ▻ **Compared forward pass runtime of TF and AOT in CMSSW**
    - ▻ Averaged runtimes over 500 calls, after ~100 "warm up" calls (measured with plain `std::chrono`)
    - ▻ Tested **event forward time** for different batch sizes from 2^0 to 2^{10} (saturation)
    - ▻ **No XLA** optimizations applied so far
  - **Memory tests using IgProf:**
    - ▻ **Measure memory consumption for setup phase in multithreading scenario**
    - ▻ **Compare TF multiple sessions vs. multiple AOT models**



CPU performance

1. **Runtimes**
   - **Equal performance!**
     (comparison same color plots)
   - **AOT without XLA optimizations**
     - ▻ **see bare minimum**
2. **Multi-threading**
   - CMSSW restrict program to 1 thread
   - Weights and series of compute kernels exist **once** in memory (`extern "C"`)
   - accessible by TF through lightweight wrapper
     - ▻ can be loaded into multiple threads
     - ▻ **negligible** multi-threading overhead
       (see next slides)



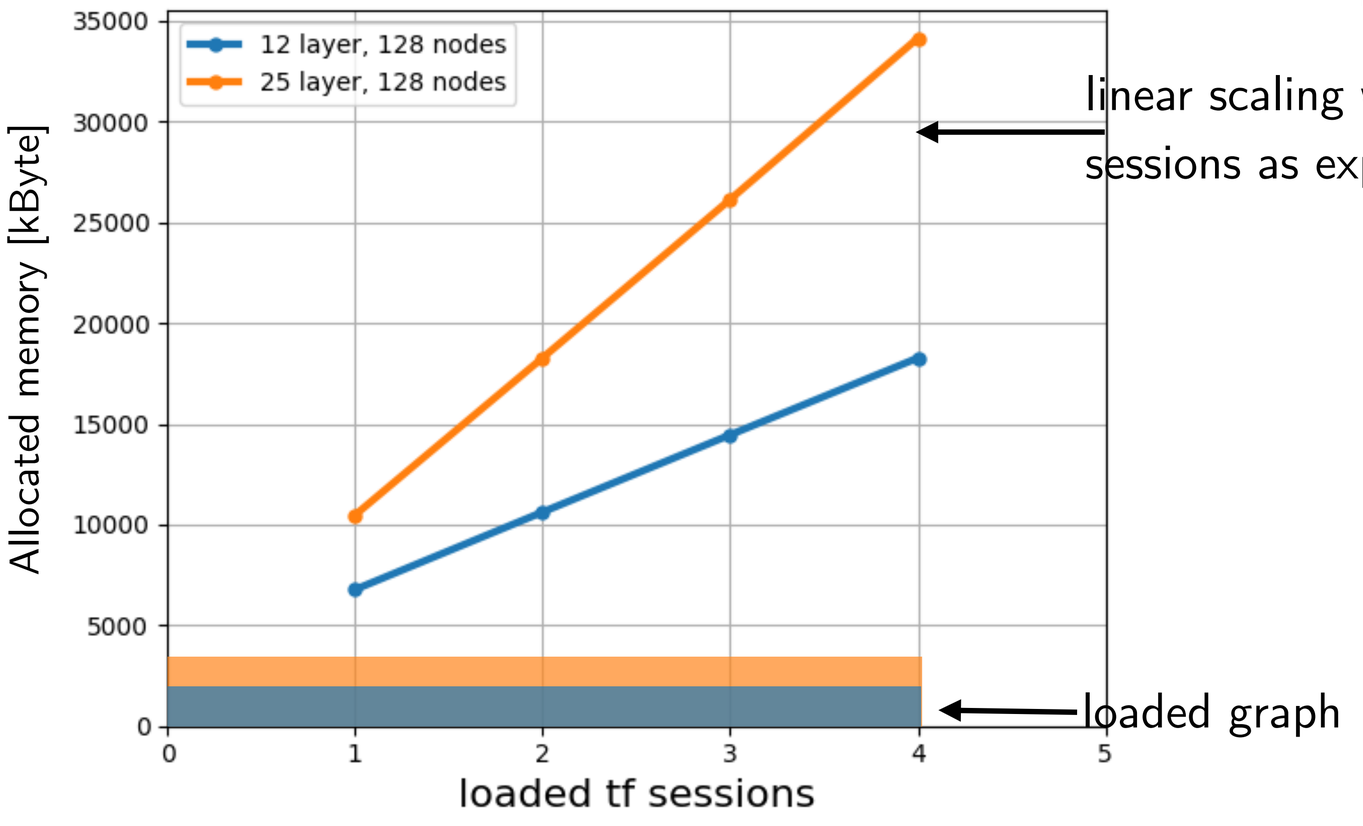
- Launch multiple sessions with same frozen graph
  - Load graph once (global)
  - Copy graph into session (per-thread)
- Does not include creation of tensors yet!

```cpp
void PerfTesterTF::loadGraph() {
  graph_def = tensorflow::loadGraphDef(graph_path);
}
```

```cpp
void PerfTesterTF::loadSessions() {
  std::vector< tensorflow::Session* > loaded_sessions;
  loaded_sessions.reserve(number_of_loaded_sessions);
  for (int i = 0; i < number_of_loaded_sessions; i++) {
    session = tensorflow::createSession(graph_def);
    loaded_sessions.push_back(session);
    }
}
```

code to load graph and sessions

Memory allocation of `loadSessions()`



Graph contains plenty of meta data

N layers	`loadGraph()` [kB]	trainable weights [kB]
12	1997	768
25	3539	1619

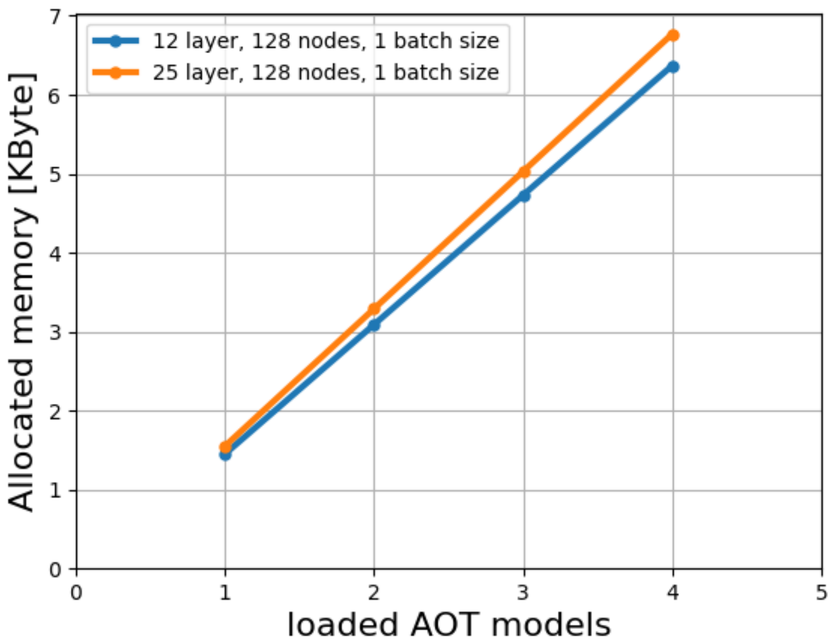
AOT setup phase

- Load multiple AOT models of same batch size
  - Cpp wrapper created to call c-function (per thread)
    - ▻ Reserves **buffers** for **inputs**, **outputs per model**
    - ▻ off-set = model depending **(slope)**
  - Reserves buffer for **intermediate layers once (*.o)**
- Important: each Cpp wrapper handles only one batch size
  - c-function with model weights might be shared between different batch sizes (check ongoing, important for stitching)
  - cpp wrapper is neglect able small compared to layers

c-function consists mainly of model weights
difference is result of pruning (bias and batch norm)

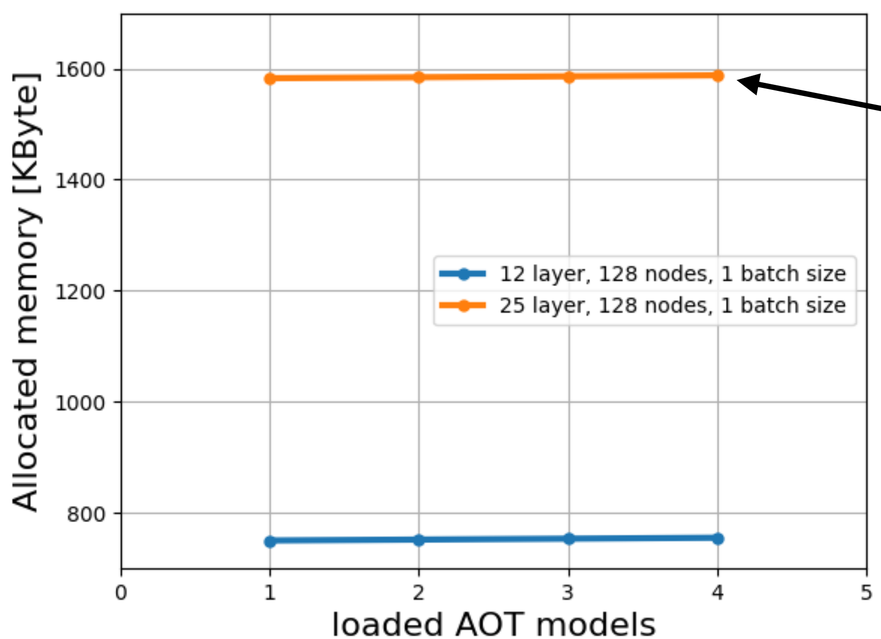
N layers	AOT model *.o [kB]	trainable weights [kB]
12	749	768
25	1581	1619

Memory allocation of Cpp wrapper



add model size
as offset

total Memory allocation

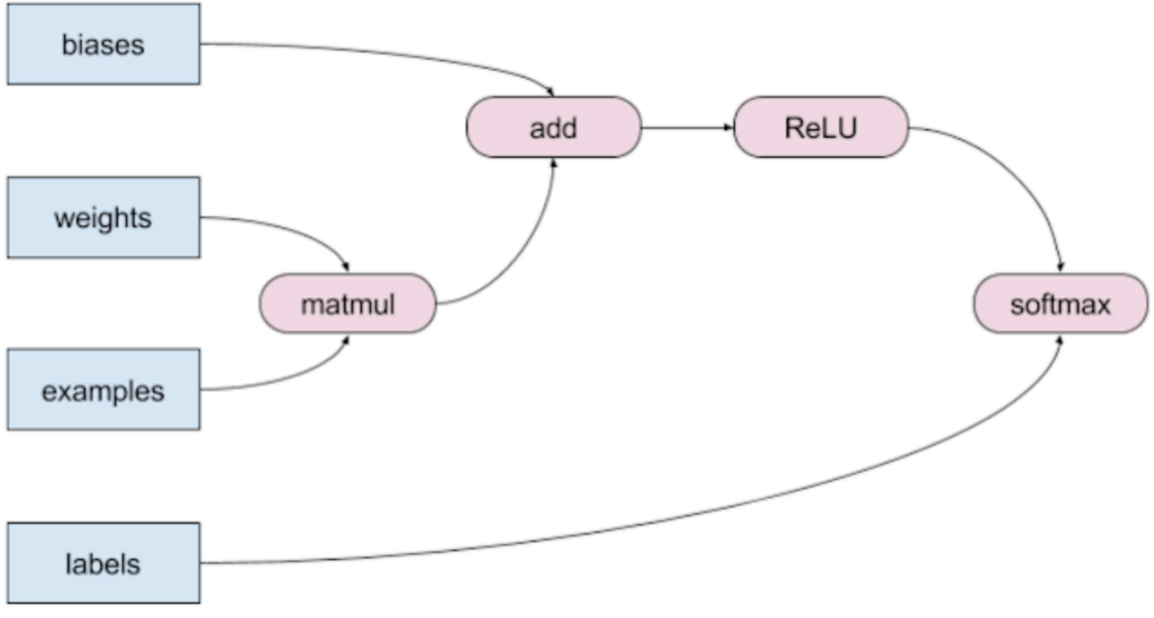


slope
barely seeable

TF and XLA insights

How TensorFlow operates

- TF generates a data flow graph representing the ML algorithm (model)
- Graphs consist out of nodes/kernel and edges
  - Nodes represent operations (Add, MatMul, Conv2D, ...)
    - ▻ TF runtime execute graph nodes
    - ▻ Operation kernels are written in C++ for CPU or GPU (e.g. with Cuda)
    - ▻ Execution runtime depends on the number of calls and complexity of the kernel
  - Edges represent data flowing (Tensors, control dependencies, resource handles, ...)

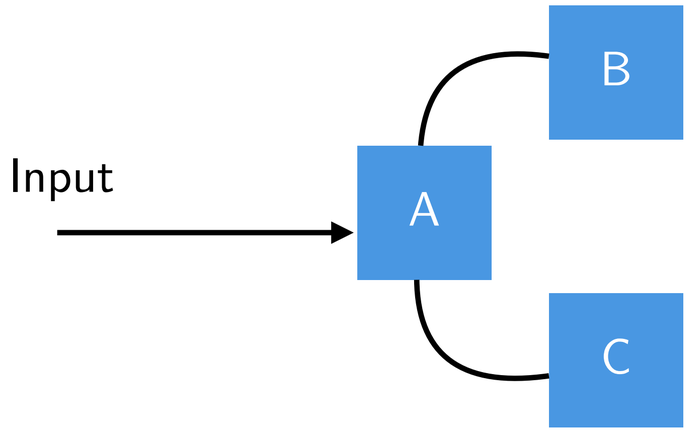


Example of a 1-layer network
Nodes are math operations and placeholder variables, connecting lines are edges

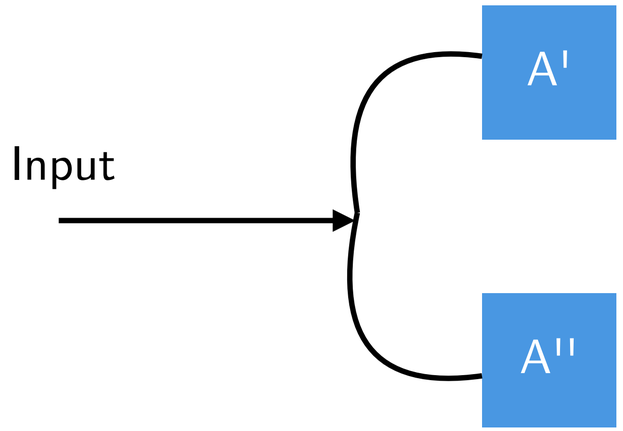
- This how tensorflow currently operates! The new part is: **optimizations (XLA)** and **independence (AOT)**

Node Fusion example

reduction of Ops overhead from 3 to 2



1 read of input in A
1 write of A to memory
2 reads of output A from memory



2 reads of input to A' and A''

A' = A->B
A'' = A->C
A is cloned and fused with B and C

Rules when to fuse:

- no increase in byte transfer
- producer ops is fused with "_all_" consumers
  - if one op is not fuseable, no fusion happens

What is a SavedModel?

- SavedModel is the prefered way to save Models in TF2.X
- A SavedModel is a directory containing
  - trained parameters (weights, and variables
  - The MetaGraphDef container, which contains:
    - the GraphDef (*.pb file) ⟶ does not require the original model code to run
      - ‣ low-level definition of the graph (including list of nodes, input and output connections)
    - the SaverDef ⟶ A class to save and restore variables from checkpoints
  - multiple signatures ⟶ can contain multiple variants of the model (multiple v1.MetaGraphDefs, identified with the --tag_set flag to saved_model_cli).

136 Files used by Grappler

```
tensorflow/compiler/jit/build_xla_ops_pass.cc
tensorflow/compiler/jit/compilability_check_util.cc
tensorflow/compiler/jit/deadness_analysis.cc
tensorflow/compiler/jit/encapsulate_subgraphs_pass.cc
tensorflow/compiler/jit/encapsulate_xla_computations_pass.cc
tensorflow/compiler/jit/introduce_floating_point_jitter_pass.cc
tensorflow/compiler/jit/mark_for_compilation_pass.cc
tensorflow/compiler/jit/resource_operation_safety_analysis.cc
tensorflow/compiler/jit/xla_activity_logging_listener.cc
tensorflow/compiler/jit/xla_cluster_util.cc
tensorflow/compiler/jit/xla_cpu_device.cc
tensorflow/compiler/jit/xla_gpu_device.cctensorflow/compiler/tf2xla/const_analysis.cc
tensorflow/compiler/tf2xla/xla_op_registry.cctensorflow/compiler/xla/parse_flags_from_env.cctensorflow/
compiler/mlir/mlir_graph_optimization_pass.cc
```

137 Files used additionally by XLA

```
xla/service/all_gather_combiner.cc
xla/service/all_reduce_combiner.cc
xla/service/batchnorm_expander.cc
xla/service/bfloat16_normalization.cc
xla/service/buffer_assignment.cc
xla/service/call_graph.cc
xla/service/call_inliner.cc
xla/service/conditional_canonicalizer.cc
xla/service/conditional_simplifier.cc
xla/service/copy_insertion.cc
xla/service/dfs_hlo_visitor.cc
xla/service/dfs_hlo_visitor_with_default.h
xla/service/dot_merger.cc
xla/service/dump.cc
xla/service/dynamic_dimension_inference.cc
xla/service/dynamic_dimension_simplifier.cc
xla/service/dynamic_padder.cc
xla/service/executable.cc
xla/service/flatten_call_graph.cc
xla/service/generic_transfer_manager.cc
xla/service/gpu/buffer_comparator.cc
xla/service/gpu/cudnn_fused_conv_rewriter.cc
xla/service/gpu/fusion_merger.cc
xla/service/gpu/gemm_algorithm_picker.cc
xla/service/gpu/gemm_thunk.cc
xla/service/gpu/gpu_compiler.cc
xla/service/gpu/gpu_conv_algorithm_picker.cc
xla/service/gpu/gpu_conv_rewriter.cc
xla/service/gpu/gpu_conv_runner.cc
xla/service/gpu/gpu_executable.cc
xla/service/gpu/hlo_to_ir_bindings.cc
xla/service/gpu/horizontal_input_fusion.cc
xla/service/gpu/horizontal_loop_fusion.cc
xla/service/gpu/ir_emitter_unnested.cc
xla/service/gpu/kernel_thunk.cc
xla/service/gpu/launch_dimensions.cc
xla/service/gpu/llvm_gpu_backend/gpu_backend_lib.cc
xla/service/gpu/multi_output_fusion.cc
xla/service/gpu/nvptx_compiler.cc
xla/service/gpu/nvptx_helper.cc
xla/service/gpu/parallel_loop_emitter.cc
xla/service/gpu/
reduction_dimension_grouper.cc
xla/service/gpu/
reduction_layout_normalizer.cc
xla/service/gpu/reduction_splitter.cc
xla/service/gpu/stream_assignment.cc
xla/service/gpu/tree_reduction_rewriter.cc
xla/service/heap_simulator.cc
xla/service/hlo_alias_analysis.cc
xla/service/hlo_computation.cc
xla/service/hlo_computation.h
xla/service/hlo_constant_folding.cc
xla/service/hlo_cse.cc
xla/service/hlo_dataflow_analysis.cc
xla/service/hlo_dce.cc
xla/service/hlo_evaluator.cc
xla/service/hlo_graph_dumper.cc
xla/service/hlo_instruction.cc
xla/service/hlo_instructions.cc
xla/service/hlo_instructions.h
xla/service/hlo_memory_scheduler.cc
xla/service/hlo_module.cc
xla/service/hlo_parser.cc
xla/service/hlo_pass_fix.h
xla/service/hlo_pass_pipeline.cc
xla/service/hlo_phi_graph.cc
xla/service/hlo_proto_util.cc
xla/service/hlo_schedule.cc
xla/service/hlo_verifier.cc
xla/service/instruction_fusion.cc
xla/service/layout_assignment.cc
xla/service/llvm_ir/fused_ir_emitter.cc
xla/service/local_service.cc
xla/service/platform_util.cc
xla/service/reduce_scatter_combiner.cc
xla/service/reshape_mover.cc
xla/service/service.cc
xla/service/shape_inference.cc
xla/service/slow_operation_alarm.cc
xla/service/sort_simplifier.cc
xla/service/stream_pool.cc
xla/service/tuple_points_to_analysis.cc
xla/service/while_loop_constant_sinking.cc
xla/service/while_loop_simplifier.cc
xla/shape.cc
xla/shape_util.cc
xla/util.cc
```

jit/clone_constants_for_better_clustering.cc
jit/kernels/xla_ops.cc
jit/xla_compilation_cache.cc
jit/xla_launch_util.cctf2xla/graph_compiler.cc
tf2xla/kernels/reduction_ops_common.cc
tf2xla/kernels/reshape_op.cc
tf2xla/xla_compilation_device.cc
tf2xla/xla_compiler.cc
tf2xla/xla_context.ccmlir/mlir_graph_optimization_pass.cc
mlir/tensorflow/translate/import_model.cc
mlir/tensorflow/utils/bridge_logger.cc
mlir/tensorflow/utils/dump_mlir_util.cc
mlir/xla/transforms/xla_legalize_tf.ccxla/literal_comparison.cc
xla/parse_flags_from_env.cc

Layer weights

- Formula to calculate number of parameters:
- 25 Hidden Layer network:
- $\underbrace{32 * 64 + 3 * 64}_{\text{Input Layer}} + \underbrace{64 * 128 + 128 * 3 + 24*(128^2 + 128 * 3)}_{\text{Hidden Layer}} + \underbrace{128 * 10 + 10}_{\text{Output Layer}} = 414538$ --> 1658152 Bytes

  12 Hidden Layer network:
- $\underbrace{32 * 64 + 3 * 64}_{\text{Input Layer}} + \underbrace{64 * 128 + 128 * 3 + 11*(128^2 + 128 * 3)}_{\text{Hidden Layer}} + \underbrace{128 * 10 + 10}_{\text{Output Layer}} = 196554$ --> 786216 Bytes

- 414538 / 196554 = 2,109

Allocated Buffers

- First 2 bits are used to define "kind", else is value
- kind = 0 ⟶ layers, 1 ⟶ temp (output),
  2 ⟶ entrypoint (input), 3 ⟶ stackbuffer (intern)
- sum of 0 = *.o file (loaded once)

```
// Number of buffers for the compiled computation.
static constexpr size_t kNumBuffers = 25;

static const ::xla::cpu_function_runtime::BufferInfo* BufferInfos() {
  static const ::xla::cpu_function_runtime::BufferInfo
    kBufferInfos[kNumBuffers] = {
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({262144ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({131072ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({32768ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({20480ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({514ULL, 0ULL}),
::xla::cpu_function_runtime::BufferInfo({161ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({33ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({16ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({19ULL, ~0ULL}),
::xla::cpu_function_runtime::BufferInfo({4097ULL, ~0ULL})
    };
  return kBufferInfos;
}
```

Buffer value	Buffer size (B)	Buffer size (kB)	kind
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
262144	65536	64.0	0
131072	32768	32.0	0
32768	8192	8.0	0
20480	5120	5.0	0
514	128	0.125	2
161	40	0.0390625	1
33	8	0.0078125	1
16	4	0.00390625	0
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
19	4	0.00390625	3
4097	1024	1.0	1