What's New in JAX

Frederic Bastien at NVIDIA

Matthew Johnson at Google

JAX in one slide

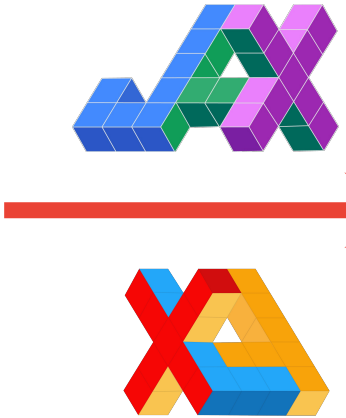
```
import jax.numpy as jnp


def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.maximum(outputs, 0)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)
```

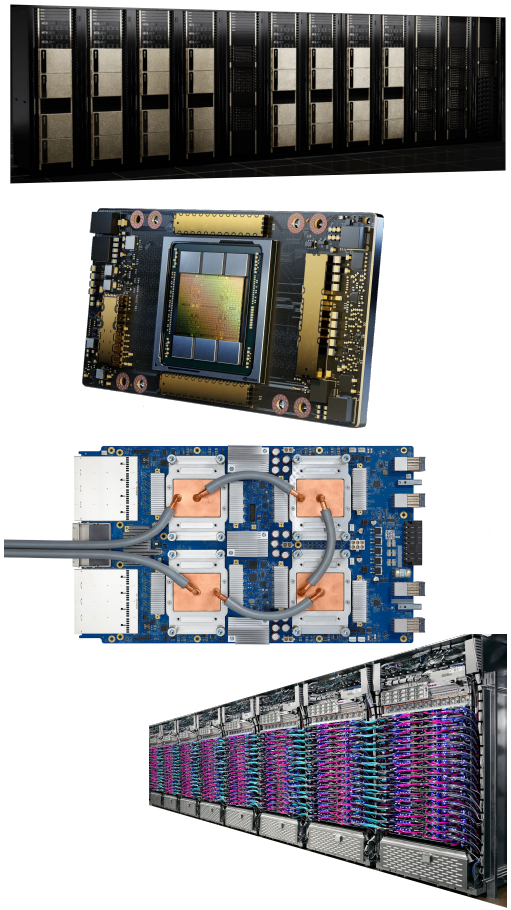
JAX in one slide

```
import jax.numpy as jnp
from jax import grad, jit, vmap

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.maximum(outputs, 0)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)


gradient_fun = jit(grad(loss))
perexample_grads = jit(vmap(grad(loss), in_axes=(None, 0)))
```

JAX in one slide

```
import jax.numpy as jnp
from jax import grad, jit, vmap

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.maximum(outputs, 0)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)


gradient_fun = jit(grad(loss))
perexample_grads = jit(vmap(grad(loss), in_axes=(None, 0)))
```

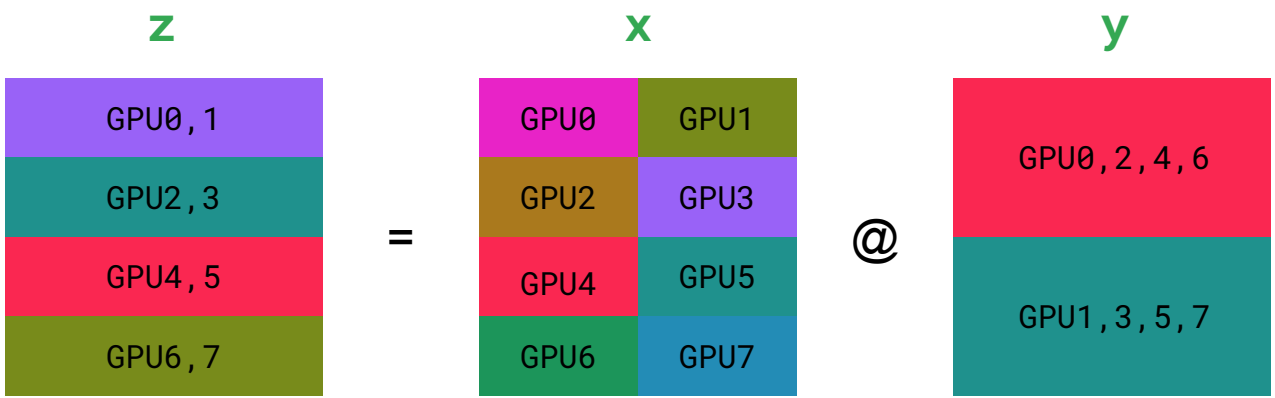




Background: `jit` optimizes, parallelizes over devices

- XLA **optimizes** fusions, layouts, schedules, rematerialization, ...
- XLA's GSPMD **automatically partitions** computation over devices
- **Unified** data / tensor / pipeline parallelism
- **Automatic parallelization tutorial** at **jax.readthedocs.io**

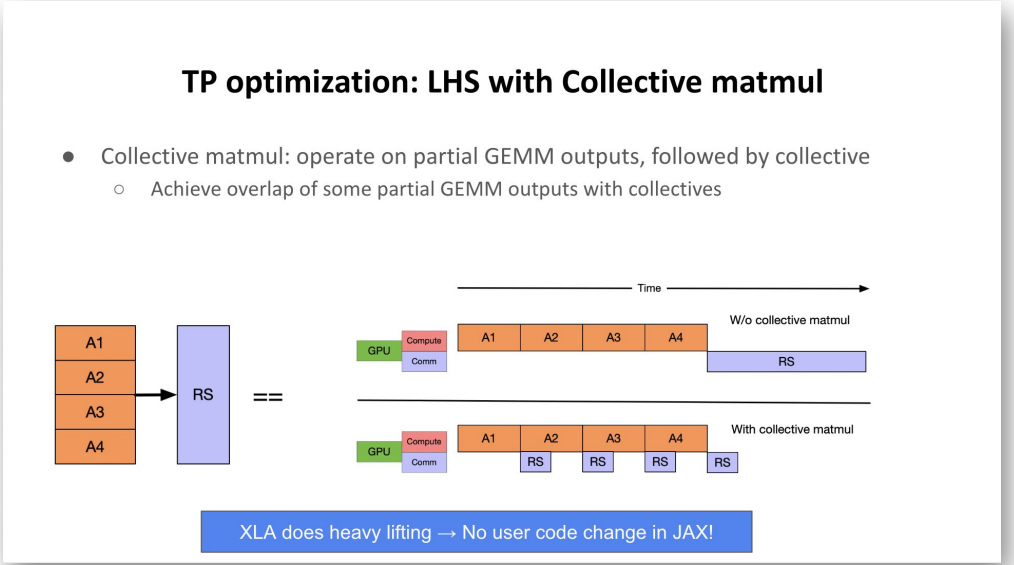
```
x = with_sharding_constraint(x, sharding1)
y = with_sharding_constraint(y, sharding2)
z = x @ y
```



Background: `jit` optimizes, parallelizes over devices

- XLA **optimizes** fusions, layouts, schedules, rematerialization, ...
- XLA's GSPMD **automatically partitions** computation over devices
- **Unified** data / tensor / pipeline parallelism
- **Automatic parallelization tutorial** at **jax.readthedocs.io**

JAX Supercharged on GPUs: High Performance LLMs with JAX and OpenXLA [S62246]




1,000
800
600
400
200
2024

Today

1. How to get more **control** (when you want it)
2. More ingredients for **maximal performance**
3. **[next talk]** JAX Supercharged on GPUs: High Performance LLMs

How to get more control (when you want it)

- **Compiler-oriented**: you write the math, compiler decides how to execute it
  - Great out-of-the-box performance
  - Flexible code: revise parallelism strategy or scale without total rewrites
- But to squeeze out **every last bit of performance**, you may want to:
  - Use advanced algorithms that compiler doesn't yet support (e.g. flash attention)
  - Use the hardware in different ways than are exposed by the compiler
- JAX goals:
  - Let you **take control** when you need to
  - Make it **composable** with automatic compiler approaches

How to get more control (when you want it)

- **Autodiff** memory control with **`jax.checkpoint`** policies
- **GPU kernels** with **`pallas`**
- **Inter-device parallelization** and communication with **`shard_map`**

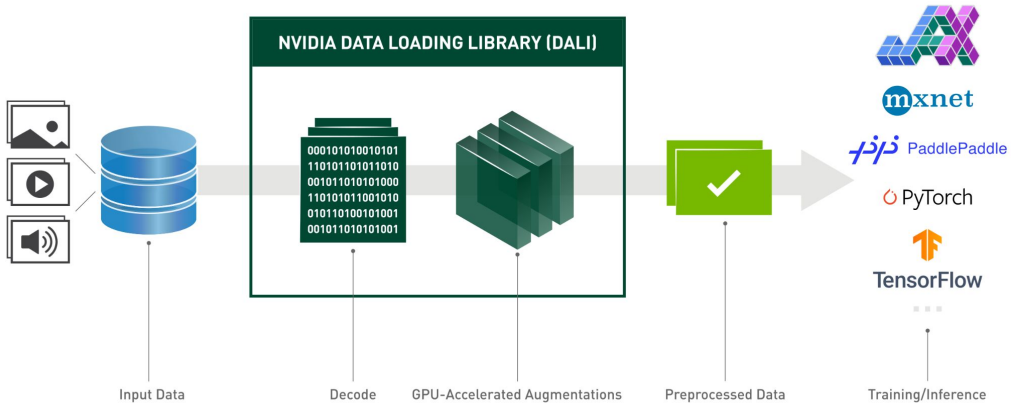
Demo

How to get good perf?

DALI

The NVIDIA Data Loading Library (DALI) is a GPU-accelerated library for **data loading** and **pre-processing** to accelerate deep learning applications.

- ~100 GPU accelerated operators
- Native python experience
- Unified workflow for training and inference via DALI Triton
- Great performance without need to understand GPU programming model



https://github.com/NVIDIA/DALI

DALI + JAX

```
import nvidia.dali.fn as fn
from nvidia.dali.plugin.jax import data_iterator
@data_iterator(output_map=["images", "labels"], reader_name="image_reader")
def iterator_fn():
    jpegs, labels = fn.readers.file(file_root=image_dir, name="image_reader")
    images = fn.decoders.image(jpegs, device="mixed")
    images = fn.resize(images, resize_x=256, resize_y=256)
    return images, labels
iterator = iterator_fn(batch_size=8)
batch = next(iterator)  # batch of data ready to be used by JAX
```

https://docs.nvidia.com/deeplearning/dali/user-guide/docs/plugins/jax_tutorials.html

DALI + JAX

```
import nvidia.dali.fn as fn
from nvidia.dali.plugin.jax import data_iterator
@data_iterator(output_map=["images", "labels"], reader_name="image_reader")
def iterator_fn():
    jpegs, labels = fn.readers.file(file_root=image_dir, name="image_reader")
    images = fn.decoders.image(jpegs, device="mixed")
    images = fn.resize(images, resize_x=256, resize_y=256)
    return images, labels
iterator = iterator_fn(batch_size=8)
batch = next(iterator)  # batch of data ready to be used by JAX
```

https://docs.nvidia.com/deeplearning/dali/user-guide/docs/plugins/jax_tutorials.html

DALI JAX Multi-GPU

```
@data_iterator(output_map=["images", "labels"], reader_name="image_reader",
               sharding=sharding)
def iterator_fn(num_shards=1, shard_id=0):
    jpegs, labels = fn.readers.file(file_root=image_dir, name="image_reader",
        num_shards=num_shards, shard_id=shard_id)
    images = fn.decoders.image(jpegs, device="mixed")
    images = fn.resize(images, resize_x=256, resize_y=256)
    return images, labels.gpu()
```

- A Flax API is also available

Now it's running and the data loading and preprocessing are fast, how to make it even faster?

FP8

Even if JAX has the fp8 dtype, the **using it manually it is complex** due to the need for **per tensor scaling**.

APIs to **automate fp8 per-tensor scaling** are now available in **Flax and Praxis.**

FP8 – Flax

```
from flax.linen import Dense
from flax.linen.fp8_ops import Fp8DotGeneralOp
import jax
import jax.numpy as jnp

random_key = jax.random.PRNGKey(0)
init_key = jax.random.PRNGKey(1)
A = jax.random.uniform(random_key, (16, 32)).astype(jnp.bfloat16)

model = Dense(features=64, dtype=jnp.bfloat16, dot_general_cls=Fp8DotGeneralOp)
params = model.init(init_key, A)

@jax.jit
def fp8matmul(var, a):
  c = model.apply(var, a) # Result in BF16
  return c

C = fp8matmul(params, A)
```

FP8 – Flax: Dtype specification

- **`param_dtypes`** parameter controls the **master weight dtype**.
- **`dtype`** parameter controls **compute dtype and output dtype**
- The **compute dtype** can be **overwritten by `dot_general_cls` in fp8** scenario.

	master weight dtype	compute dtype	output dtype
`Dense(..., param_dtypes=`**`fp32`**`, dtype=`**`fp32`**`) # default`	fp32	fp32	fp32
`Dense(..., param_dtypes=`**`fp32`**`, dtype=`**`bf16`**`)`	fp32	**bf16**	**bf16**
`Dense(..., param_dtypes=`**`fp32`**`, dtype=`**`bf16`**`, `**`dot_general_cls=Fp8DotGeneralOp`**`)`	fp32	**fp8**	bf16

FP8 - Flax: Gradient and Weight Update

All fp8 handling is hidden for the gradient and optimizer.

No code changes needed!

```
# reusing fwd code in previous slide

 @jax.jit
  def step_fn(train_state, a):
      def loss_fn(vars, x):
          return jnp.mean(train_state.apply_fn(vars, x))

      # gradient includes fp8 meta variable
      grad_fn = jax.value_and_grad(loss_fn, argnums=[0])
      loss_val, vars_grads = grad_fn(train_state.params, a)

      # optimizer update is fp8 meta variable aware
      return train_state.apply_gradients(grads=vars_grads[0]),
          loss_val

  tx = optax.adam(learning_rate=0.001)

  # train_state includes fp8 meta variables
  train_state = TrainState.create(params=params, tx=tx,
      apply_fn=model.apply)
  train_state, train_loss = step_fn(train_state, A)
```

Scaled Dot Product Attention (SDPA)

Lower directly to optimized CUDNN kernel!
We are adding a public API

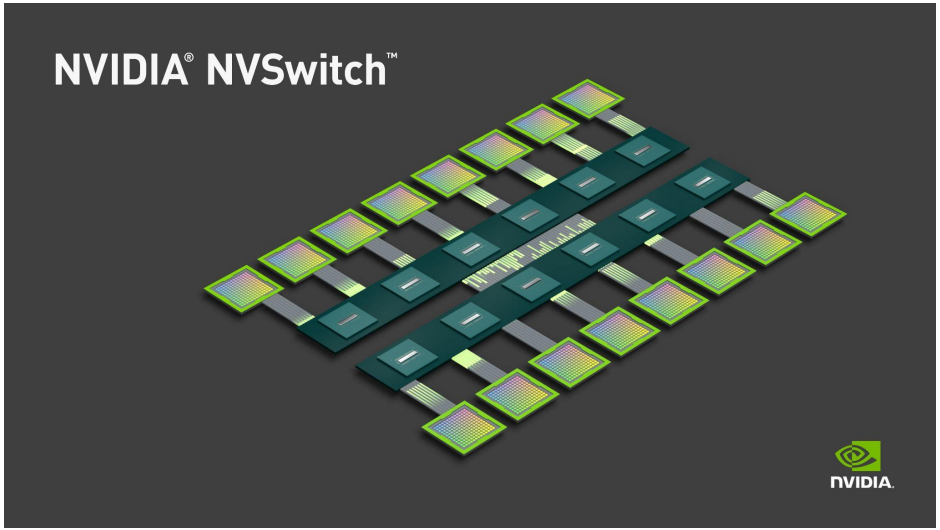
```
from jax._src.cudnn.fused_attention_stablehlo import dot_product_attention

dot_product_attention(query: Array,
                      key: Array,
                      value: Array,
                      bias: Optional[Array] = None,
                      mask: Optional[Array] = None,
                      *,
                      scale: float = 1.0,
                      is_causal_mask: bool = False,
                      seed: int = 42,
                      dropout_rate: float = 0.):

# grad, vmap, sharding are all supported!
```

Collective Broadcast

- Exist in other framework already
- MPI
  - MPI_Bcast
- Pytorch
  - pytorch.distributed.broadcast
- NCCL
  - ncclBroadcast

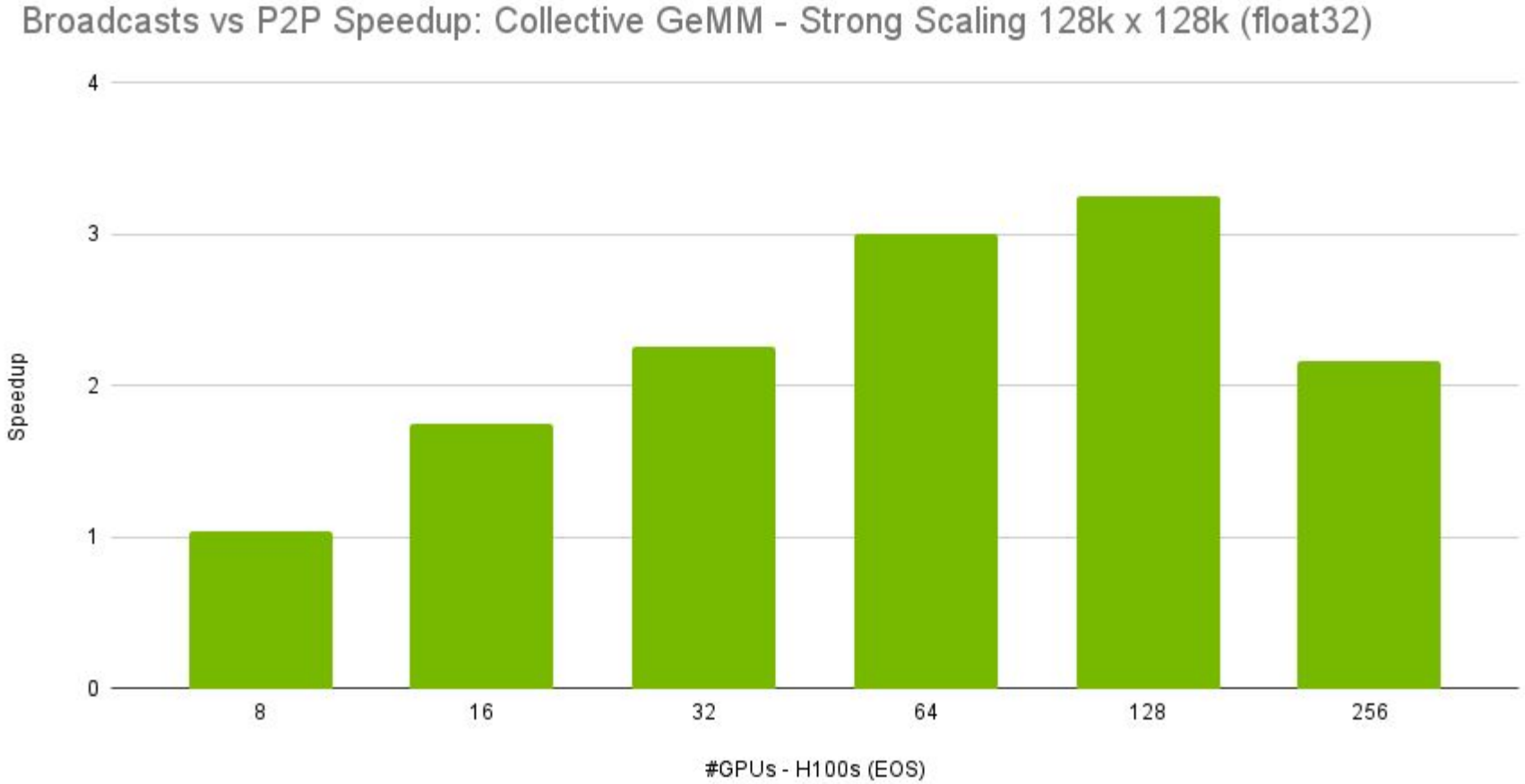
Hardware accelerated by our **network switches:** NVSwitch and InfiniBand

https://github.com/google/jax/pull/19602

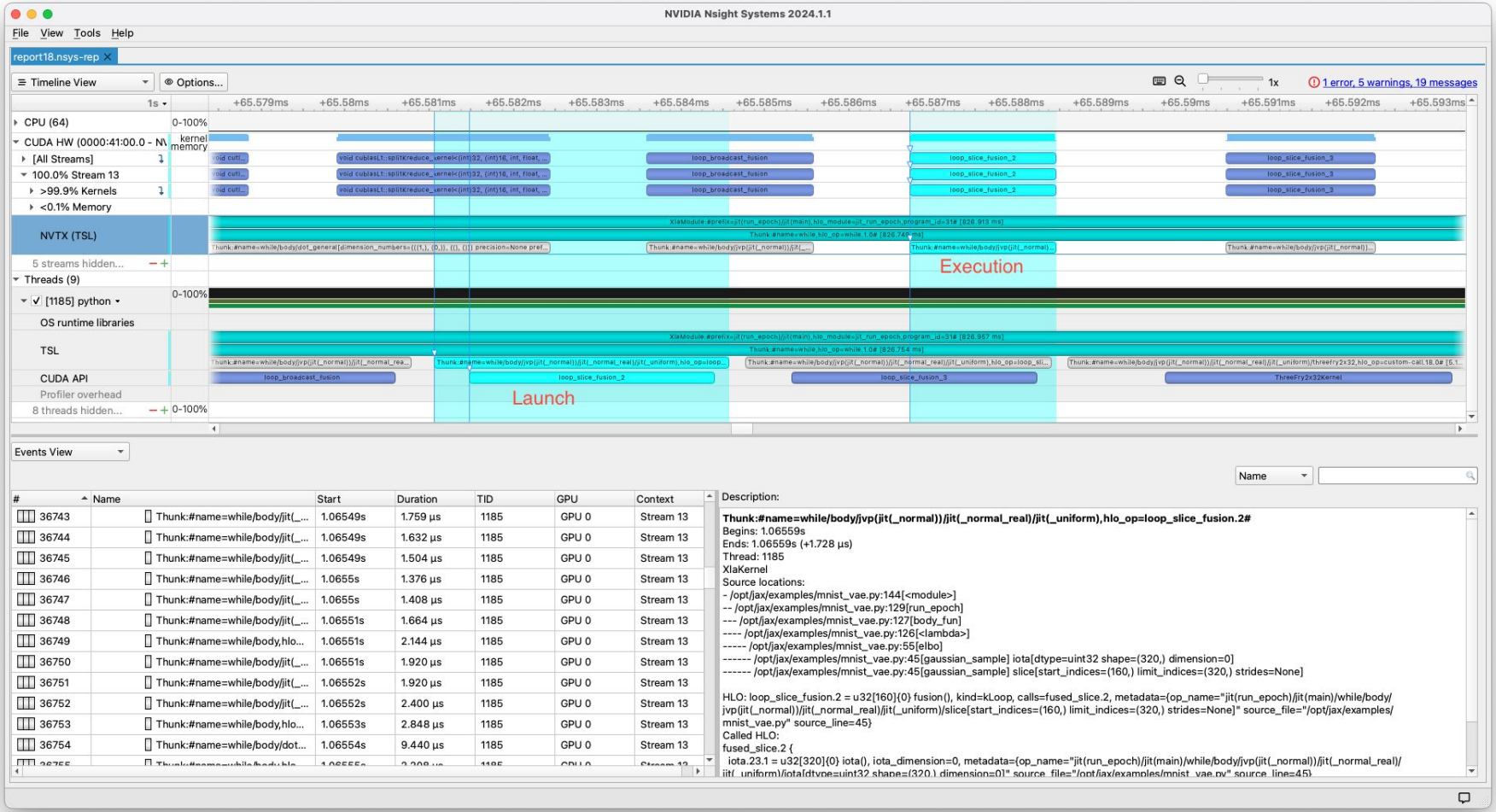


Collective Broadcast

```
@jax.jit
@partial(shard_map, mesh=...,
    in_specs=P('i', None), out_specs=P('i', None), check_rep=False)
def f(a):
  return jax.lax.pbroadcast(a, 'i', 1)

x = jnp.arange(4).reshape((2, 2))
print(x)
Array([[ 0,  1],
       [ 2,  3]], dtype=int32)
print(f(x))
Array([[2, 3],
       [2, 3]], dtype=int32)
```


Broadcasts vs P2P Speedup: Collective GeMM - Strong Scaling 128k x 128k (float32)
Speedup
0
1
2
3
4
8
16
32
64
128
256
#GPUs - H100s (EOS)

Nsys profiler



#	Name	Start	Duration	TID	GPU	Context
36743	Thunk:#name=while/body/jit(_...	1.06549s	1.759 µs	1185	GPU 0	Stream 13
36744	Thunk:#name=while/body/jit(_...	1.06549s	1.632 µs	1185	GPU 0	Stream 13
36745	Thunk:#name=while/body/jit(_...	1.06549s	1.504 µs	1185	GPU 0	Stream 13
36746	Thunk:#name=while/body/jit(_...	1.0655s	1.376 µs	1185	GPU 0	Stream 13
36747	Thunk:#name=while/body/jit(_...	1.0655s	1.408 µs	1185	GPU 0	Stream 13
36748	Thunk:#name=while/body/jit(_...	1.06551s	1.664 µs	1185	GPU 0	Stream 13
36749	Thunk:#name=while/body,hlo...	1.06551s	2.144 µs	1185	GPU 0	Stream 13
36750	Thunk:#name=while/body/jit(_...	1.06551s	1.920 µs	1185	GPU 0	Stream 13
36751	Thunk:#name=while/body/jit(_...	1.06552s	1.920 µs	1185	GPU 0	Stream 13
36752	Thunk:#name=while/body/jit(_...	1.06552s	2.400 µs	1185	GPU 0	Stream 13
36753	Thunk:#name=while/body,hlo...	1.06553s	2.848 µs	1185	GPU 0	Stream 13
36754	Thunk:#name=while/body/dot...	1.06554s	9.440 µs	1185	GPU 0	Stream 13

Thunk:#name=while/body/jvp(jit(_normal))/jit(_normal_real)/jit(_uniform),hlo_op=loop_slice_fusion.2#
Begins: 1.06559s
Ends: 1.06559s (+1.728 µs)
Thread: 1185
XlaKernel
Source locations:
- /opt/jax/examples/mnist_vae.py:144[<module>]
-- /opt/jax/examples/mnist_vae.py:129[run_epoch]
--- /opt/jax/examples/mnist_vae.py:127[body_fun]
---- /opt/jax/examples/mnist_vae.py:126[<lambda>]
----- /opt/jax/examples/mnist_vae.py:55[elbo]
------ /opt/jax/examples/mnist_vae.py:45[gaussian_sample] iota[dtype=uint32 shape=(320,) dimension=0]
------ /opt/jax/examples/mnist_vae.py:45[gaussian_sample] slice[start_indices=(160,) limit_indices=(320,) strides=None]

HLO: loop_slice_fusion.2 = u32[160]{0} fusion(), kind=kLoop, calls=fused_slice.2, metadata={op_name="jit(run_epoch)/jit(main)/while/body/jvp(jit(_normal))/jit(_normal_real)/jit(_uniform)/slice[start_indices=(160,) limit_indices=(320,) strides=None]" source_file="/opt/jax/examples/mnist_vae.py" source_line=45}
Called HLO:
fused_slice.2 {
  iota.23.1 = u32[320]{0} iota(), iota_dimension=0, metadata={op_name="jit(run_epoch)/jit(main)/while/body/jvp(jit(_normal))/jit(_normal_real)/jit(_uniform)/iota[dtype=uint32 shape=(320,) dimension=0]" source_file="/opt/jax/examples/mnist_vae.py" source_line=45}

From https://github.com/NVIDIA/JAX-Toolbox/blob/main/docs/profiling.md

Nsys profiler

- **NVTX** connects **XLA** to **NSys**.
- Example kernel-level annotation in **NSys**: Shown in the **tooltip**.
- Show the fields (Source locations, HLO, Called HLO).
- Python source lines and HLO IR shown alongside kernels.

```
Thunk:#name=while/body,hlo_op=loop_maximum_fusion.7#

Source locations:
- /opt/jax/examples/mnist_vae.py:144[<module>]
-- /opt/jax/examples/mnist_vae.py:128[run_epoch]
--- /opt/jax/examples/mnist_vae.py:126[body_fun]
---- /opt/jax/examples/mnist_vae.py:125[<lambda>]
----- /opt/jax/examples/mnist_vae.py:53[elbo] add
----- /opt/jax/examples/mnist_vae.py:123[body_fun] jvp(jit(relu))/max

HLO:
loop_maximum_fusion.7 = f32[32,512]{1,0} fusion(get-tuple-element.656,
get-tuple-element.657), kind=kLoop, calls=fused_maximum, metadata=...

Called HLO:
fused_maximum {
  param_0.134 = f32[32,512]{1,0} parameter(0)
  param_1.170 = f32[512]{0} parameter(1)
  broadcast.969 = f32[32,512]{1,0} broadcast(param_1.170), dimensions={1}
  add.454 = f32[32,512]{1,0} add(param_0.134, broadcast.969)
  constant_413 = f32[] constant(0)
  broadcast.758 = f32[32,512]{1,0} broadcast(constant_413), dimensions={}
  ROOT maximum.63 = f32[32,512]{1,0} maximum(add.454, broadcast.758)
} // fused_maximum
```

JAX Custom Primitive (Instruction)

You have a super fast custom kernel and want to use it with JAX?
You can do it with a JAX custom Primitive!

The kernel can have many forms: CUDA, CUTLASS, Warp, existing library (cufft, cusolver, ...), ...

JAX Custom Primitive

You need to teach JAX everything about this new Primitive.
In Python:

- Define a new JAX primitive.
- Define a function to infer the output shapes and dtypes for a giving inputs (called abstract evaluation).
- Define its lowering to StableHLO.
- [Optional] Define the gradient.
- [Optional] vmap implementation.
- [Optional] shard_map or custom_partitioning for fast multi-GPU.

.

JAX Custom Primitive

In C++: You need to follow these steps for each new JAX primitive:

- Have CUDA kernel(s).
- Create and register a C++ function that dispatches the CUDA kernel.
- Create a descriptor to convey information needed by this function.
    - The dtypes, the shapes and other attributes.
- Bind C++ functions to Python
    - To create the descriptor and to call the primitive during execution.

Heavily used in TransformerEngine with 18 custom primitives

From: https://jax.readthedocs.io/en/latest/Custom_Operation_for_GPUs.html

`shard_map` or `custom_partitioning`

`shard_map`: great when you want to hardcode one sharding pattern in your graph.

But what if you want to support many sharding for that primitive? Like DP, TP, DP+TP, FSDP, … any combination of the above?

With `shard_map`, you will end up writing many versions and you will need to create a new API for the user.

JAX already has a sharding API, can we support it?

`custom_partitioning` is exactly what you need for that.

Sharding API

Basic JAX sharding API:

- GPUs:
  - A **Mesh** that represent an nd-array of devices.
  - Each dimension has a name.
- Data sharding representation:
  - **PartitionSpec and Sharding** specify how a tensor is sharded (split) on the mesh (devices)
- Data transfer:
  - **jax.device_put**( ) to transfer and shard data on the GPUs.
  - **jax.lax.p***, like **pbroadcast**( )

```
import jax, numpy as np

from jax.sharding import Mesh,
NamedSharding, PartitionSpec as P

devices =
np.array(jax.devices()).reshape(4, 2)

mesh = Mesh(devices, ('x', 'y'))

spec = P('x', 'y')

sharding = NamedSharding(mesh, spec)
x = np.zeros((8,8), dtype='float32')
y = jax.device_put(x, sharding)
jax.debug.visualize_array_sharding(y)
```

GPU **0**	GPU **1**
GPU **2**	GPU **3**
GPU **4**	GPU **5**
GPU **6**	GPU **7**

Sharding API

JAX sharding API:

- The user specifies the inputs and outputs sharding
  - **`jax.jit(`*`fun`*`, `*`in_shardings`*`=..., `*`out_shardings`*`=...)`**
- **XLA will infer the intermediate sharding** during compilation.
- The user can specify a custom sharding constraint manually:
  - **`x = jax.lax.with_sharding_constraint(`*`x`*`, `*`shardings`*`)`**

```
out_sharding = NamedSharding(
    mesh, P('x', None))
jitted = jax.jit(
    lambda x: x+1,
    in_shardings=sharding,
    out_shardings=out_sharding
)
z = jitted(y)
jax.debug.visualize_array_sharding(z)
```

GPU **0,1**
GPU **2,3**
GPU **4,5**
GPU **6,7**

Sharding Inference

Sharding inference works in 2 steps:

- **Planning**: propagate the sharding from the inputs to the outputs.
- **Execute the plan**: partition the graph to represent per-GPU computations.

JAX has **callbacks** to extend this mechanism to our primitive via 2 functions:

- Planning: **`infer_sharding_from_operands()`**
- Execute the plan: **`partition()`**

Custom Partitioning: `infer_sharding_from_operands()`

`infer_sharding_from_operands()` example for **RMS norm**

```
def infer_sharding_from_operands(
                                mesh : jax.sharding.Mesh,
                                arg_infos : tuple[jax._src.api.ShapeDtypeStruct],
                                result_infos : tuple[jax._src.core.ShapedArray]):
  # ShapeDtypeStruct.{shape, dtype, named_shape, sharding}
  # ShapedArray.{shape, dtype, weak_type, named_shape}
  x_info, weight_info = arg_infos
  assert len(x_info.shape) == 3 and len(weight_info.shape) == 2
  x_spec = x_info.sharding.spec
  # We only support sharding on the batch dimensions.
  # Force non-sharding on all others dimensions with None.
  output_sharding = NamedSharding(mesh, PartitionSpec(x_spec[0], None, None))
  invvar_sharding = NamedSharding(mesh, PartitionSpec(x_spec[0]))
  return (output_sharding, invvar_sharding)
```

Custom Partitioning: partition()

```
def partition(
        mesh : jax.sharding.Mesh,
        arg_infos : tuple[jax._src.api.ShapeDtypeStruct],
        result_infos : tuple[jax._src.api.ShapeDtypeStruct]):

  impl = RmsNormFwdClass.impl

  # We only support sharding on the batch dimensions.
  # Force non-sharding on all others dimensions with None.
  output_shardings = (NamedSharding(...), ...)
  arg_shardings = (NamedSharding(...), …)

  return mesh, impl, output_shardings, arg_shardings
```

Full JAX primitive + custom_partitioning example:
https://jax.readthedocs.io/en/latest/Custom_Operation_for_GPUs.html

Frameworks

Now, let’s talk about frameworks that use JAX.

Transformer Engine

Transformer Engine (TE) is a library for **accelerating Transformer models on NVIDIA GPUs** using highly-optimized, fused operations with support for **8-bit floating point (FP8)** precision on Hopper GPUs.

- Framework-agnostic C++ API for optimized implementations of common Transformer operations.
- TE/JAX integrate TE C++ API into JAX with custom primitive.

  E.g.: Fused FP8-cast+transpose, scaled+masked softmax, FP8 GEMM, cuDNN fused attention.
- Flax- and Praxis-like Transformer architecture building blocks based on TE/JAX custom ops:
  - Normalizations: LayerNorm, RMSNorm
  - Fused Layers: LayerNormDenseGeneral, LayerNormMLP
  - Attention Block: MultiheadAttention (w/ cuDNN backend)
  - Encoder/Decoder: TransformerLayer (fused layers + attention)
- Automatic mixed-precision FP8 scaling.

Warp

Warp is a **Python framework** for writing high-performance **simulation and graphics code**. Warp takes regular Python functions and **JIT compiles them to efficient kernel code** that can run on the CPU or GPU.

Why use with JAX?

Warp is designed for **spatial computing** and comes with a rich set of primitives that make it easy to write programs for **physics simulation, perception, robotics, and geometry processing**.

Warp kernels are **differentiable**, so can be used for machine-learning.

Good for **thread based computation**.

Warp/JAX API

Exchange data without copy via DLPack:

```
warp.jax.to_jax(wp_array)
warp.jax.from_jax(jax_array, dtype=None)
```

Convert the **device** representation:

```
warp.jax.device_to_jax(wp_device)
warp.jax.device_from_jax(jax_device)
```

Convert the **dtype** representation:

```
warp.jax.dtype_to_jax(wp_dtype)
warp.jax.dtype_from_jax(jax_dtype)
```

Tight JAX/Warp integration

Warp kernels as JAX primitives

Experimental feature https://nvidia.github.io/warp/modules/interoperability.html#jax

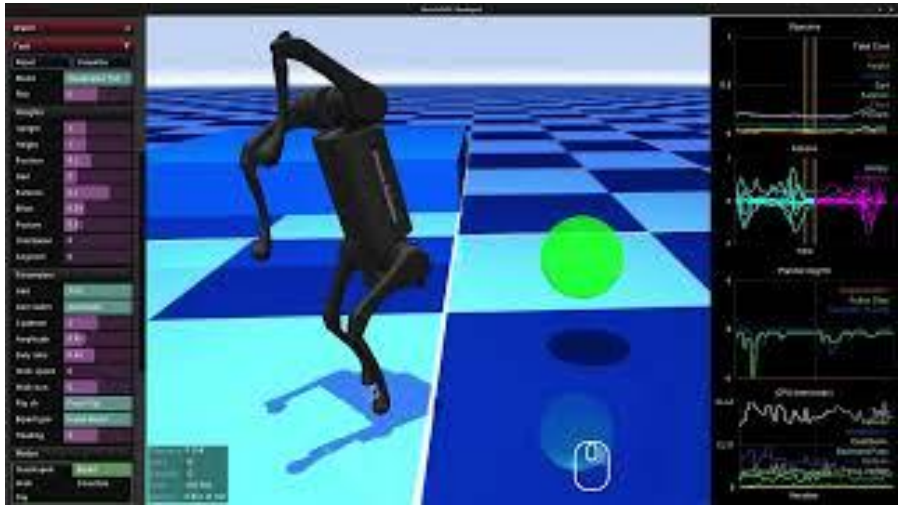
Does not support `grad` or `vmap`, requires manual `shard_map` for multi-GPU..

```
import warp as wp
import jax
import jax.numpy as jp

# import experimental feature
from warp.jax_experimental import jax_kernel

@wp.kernel
def triple_kernel(input: wp.array(dtype=float), output: wp.array(dtype=float)):
    tid = wp.tid()
    output[tid] = 3.0 * input[tid]

wp.init()

# create a Jax primitive from a Warp kernel
jax_triple = jax_kernel(triple_kernel)

# use the Warp kernel in a Jax jitted function
@jax.jit
def f():
    x = jp.arange(0, 64, dtype=jp.float32)
    return jax_triple(x)

print(f())
```

MuJoCo/MjX

MuJoCo is a general purpose physics engine that aims to facilitate research and development in **robotics**, **biomechanics**, **graphics**, **animation** and **machine learning** that demand **fast and accurate simulation of articulated structures interacting with their environment**.

MjX allows MuJoCo to run on hardware supported by the XLA compiler via the JAX framework.

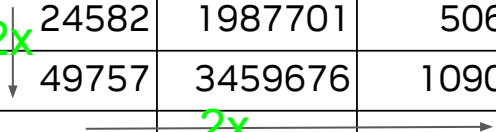


MjX

Contributed optimizations:

- Small matmul optimization
- Parallelized solvers.
- capsule-capsule interaction
  - **via tight JAX/Warp integration**
  - 35ms -> 7ms for 1024x18000 collision pairs.
  - End-end speed up of 20%
- 2x speed up on newer GPUs
- 2x speed up since we started optimizing

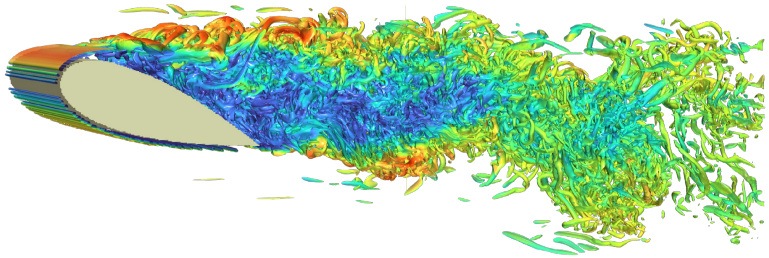
Units: steps per second	Started at		Where we are	
	1 Humanoid	10 Humanoids	1 Humanoid	10 Humanoids
A100	1120181	24582	1987701	50682
H100	2536900	49757	3459676	109090
Hopper on GH200	2688842	55845	3703446	128755



XLB: Differentiable CFD for Physics-Based ML

Differentiability + Scalability + Accessibility

- **Developed in Python with JAX**
- **Differentiable** Lattice Boltzmann library for ML
- **Distributed scalability** on multi-node systems
- **Numpy-like interface** with parallelization mechanics under the hood)
- **Easy integration** with JAX ML libraries, such as Flax and Equinox



Collaborators:

Mehdi Ataei, Autodesk Research

Hesam Salehipour, Autodesk Research

Massimiliano Meneghin, Autodesk Research

Oliver Hennigh, NVIDIA

Frédéric Bastien, NVIDIA

Olli Lupton, NVIDIA

Repository:

https://github.com/Autodesk/XLB

Paper:

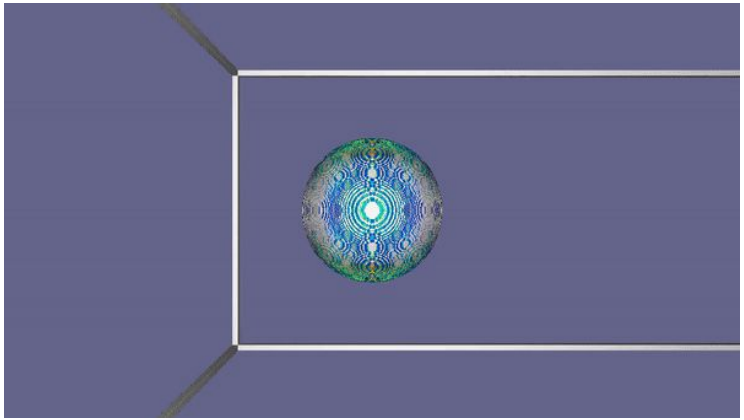
https://arxiv.org/abs/2311.16080

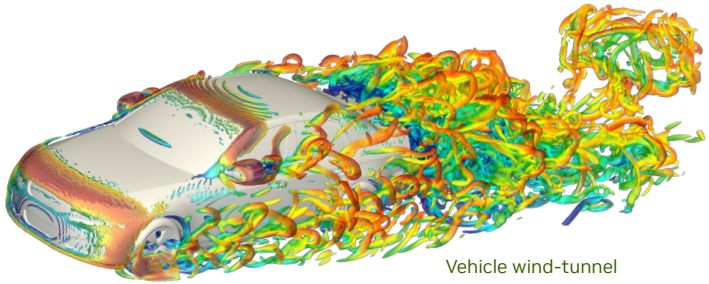
XLB for Large-Scale Simulations

State-of-the-Art simulation models with exceptional performance

- **JAX Numpy-like interface** allows the same piece of code to be executed on a desktop CPU, or a distributed multi-GPU cluster for simulations with tens of billions of cells
- **In-situ rendering** enabled by data communication using DLPack (sharing of tensors among frameworks)
- Out-of-core features targeting **Grace Hopper GPU system for larger simulations** is a work in progress

Out-of-Core Computation + in-situ rendering in XLB (courtesy of Oliver Hennigh, NVIDIA)





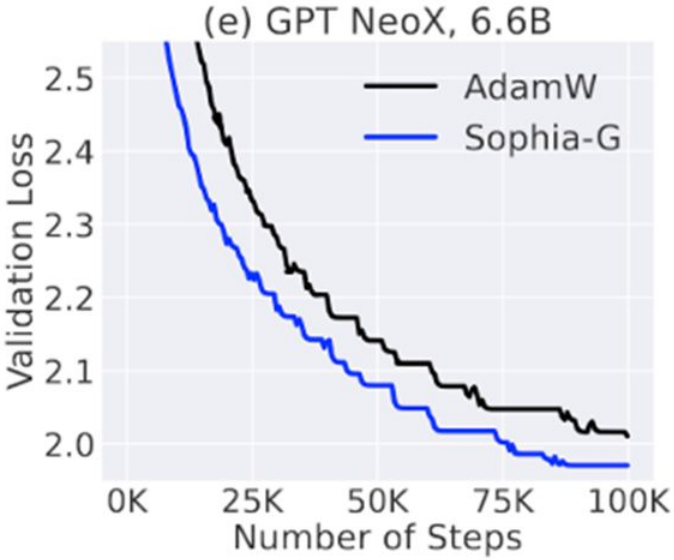
Vehicle wind-tunnel

(DrivAer Model)

Levanter

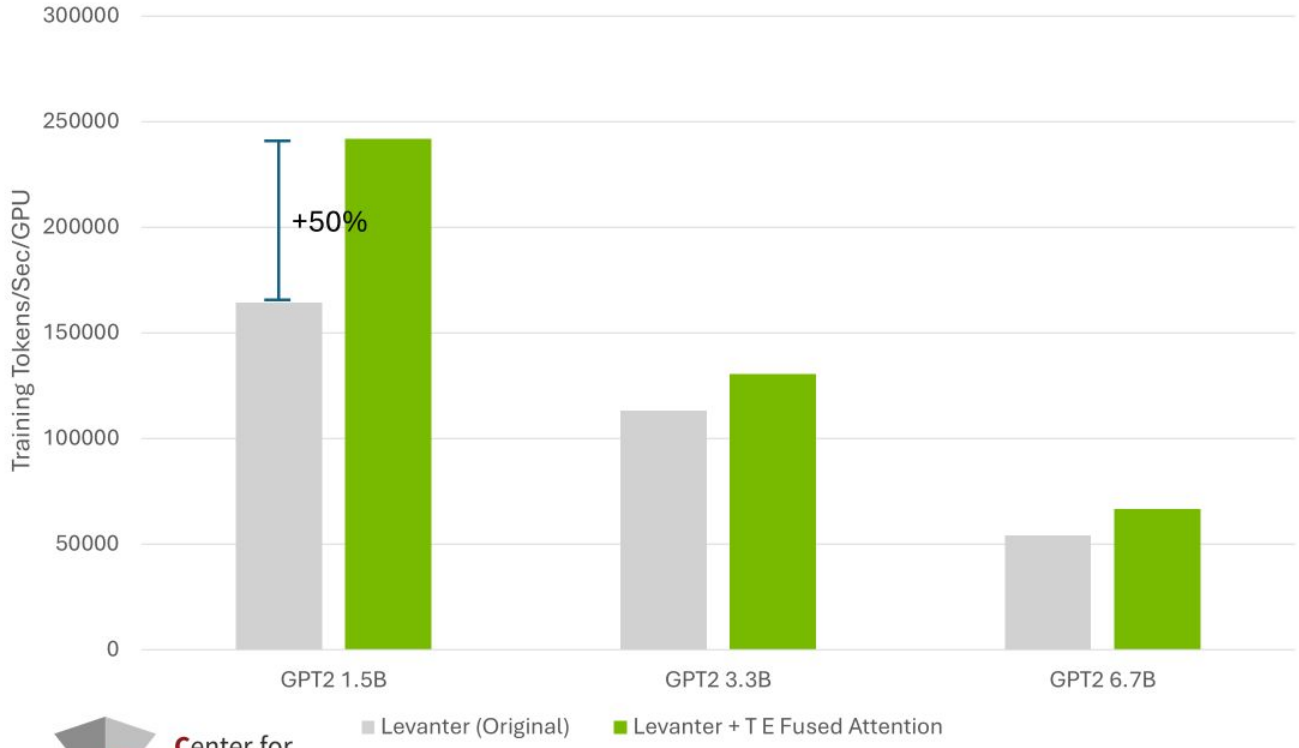
- **Legible**: Levanter uses named tensors to improve the readability and flexibility of code
- **Scalable:** Thanks to JAX and TransformerEngine, throughput numbers competitive with best frameworks
- **Advanced:** Incorporates state-of-the-art advances from Stanford including Flash Attention, Sophia, and DoReMi, giving 2x speedups over default stack.
- **Multi-Modal:** Used for training LLMs, Music FMs, ASR models, Biomedical models, and more!
- **More:** Support for LoRA, HuggingFace Hub and PEFT compatibility!

```
# Attention
scores = dot(q, k, axis="key")
scores /= sqrt(q.axis_size("key"))
scores = softmax(scores, axis="pos")
y = dot(scores, value, axis="pos")
```



Stanford Center for Research on Foundation Models

Levanter + Transformer Engine: Up to 50% faster



Stanford Center for Research on Foundation Models

16xA100 on Radium's Cloud

Ra radium

JAX-Toolbox

The JAX ecosystem is based on many different repositories.

To help JAX-Toolbox have:

- **Nightly optimized containers** to be able to use the most up to date version of everything.
  - ghcr.io/nvidia/jax:{**jax,pax,t5x,maxtext,levanter**,...}-2024-03-04
  - Some in-progress PR are included to get the best perf earlier
- **Nightly CI** to detect ecosystem breakage rapidly.
- **Optimized models**
  - GTP-3, LLAMA2, T5, ViT, Imagen
- Links on how to use **JAX in different CSP**.

https://github.com/NVIDIA/JAX-Toolbox

Other JAX presentations

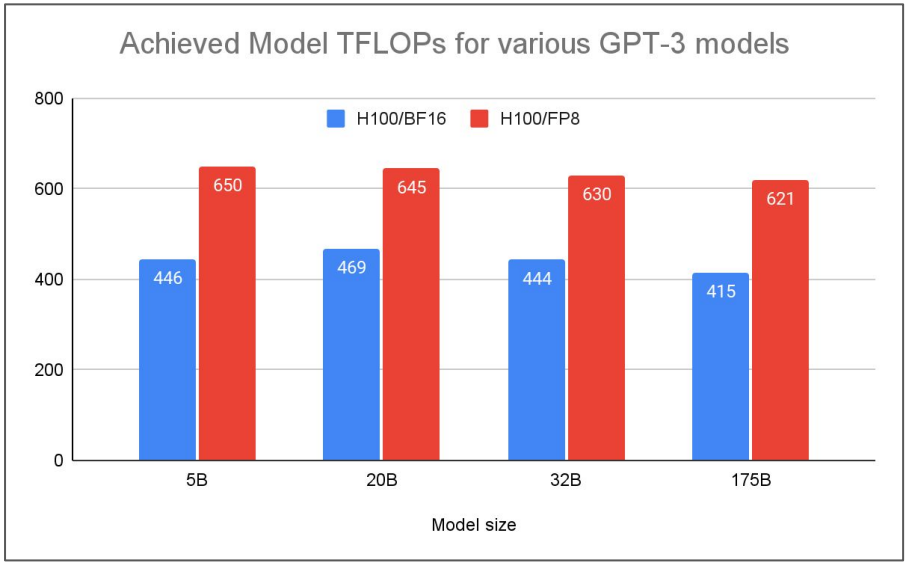
- High-Performance LLMs Made Easy with JAX and OpenXLA on NVIDIA Hopper [S62246] **(Just after this presentation)**
- Introduction to **Equinox** for JAX: Functionality and GPU Optimizations [S62668]
- Nucleotide Transformer: Advancing **Genomic Analysis** with Large Language Models [S62438]
- **Scaling Grok** with JAX and H100 [S63257]
- Deep Dive Into Training and Inferencing **Large Language Models on Azure's** leading AI Infrastructure Powered by NVIDIA H100 Tensor Core GPUs [S63273]
- Scientific Computing with **NVIDIA Grace** and the Arm Software Ecosystem [S61598]

JAX Supercharged: High-Performance LLMs with JAX and XLA [S62246]

Qiao Zhang (Google), Nitin (NVIDIA), Chang Lan (Google)

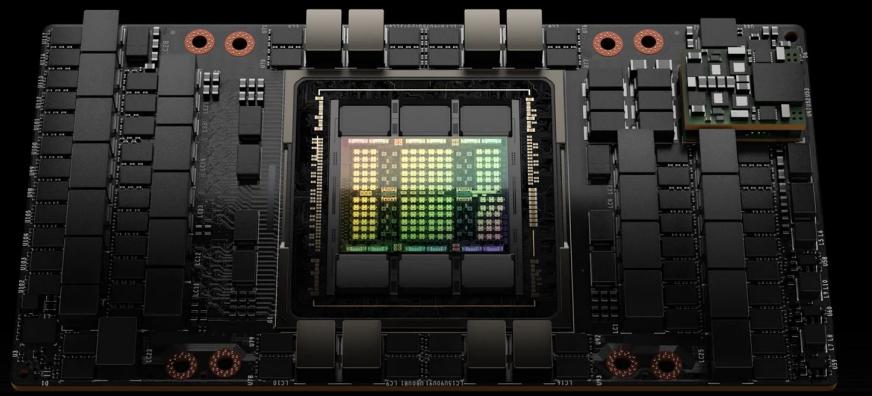
Just after this presentation in the same room.

- Age of generative AI: assistants powered by compute-hungry LLMs
  - NVIDIA H100 is a leading engine of AI compute with massive FLOPs
  - JAX with XLA based automatic parallelization scales LLMs to 1000s of GPUs
- Learn about winning strategies for JAX / XLA to maximizing H100 efficiency



The future is bright!

- Join us on Discord: https://discord.gg/nvidiadeveloper
- We want your input on what we should be working on, share at https://developer.nvidia.com/jax
- Get access to GPU-optimized JAX containers:
  - NGC: https://catalog.ngc.nvidia.com/orgs/nvidia/containers/jax
  - Nightly: https://github.com/NVIDIA/JAX-Toolbox
- Join us and contribute to the open source at: https://github.com/google/jax




NVIDIA
Thank you
ank you