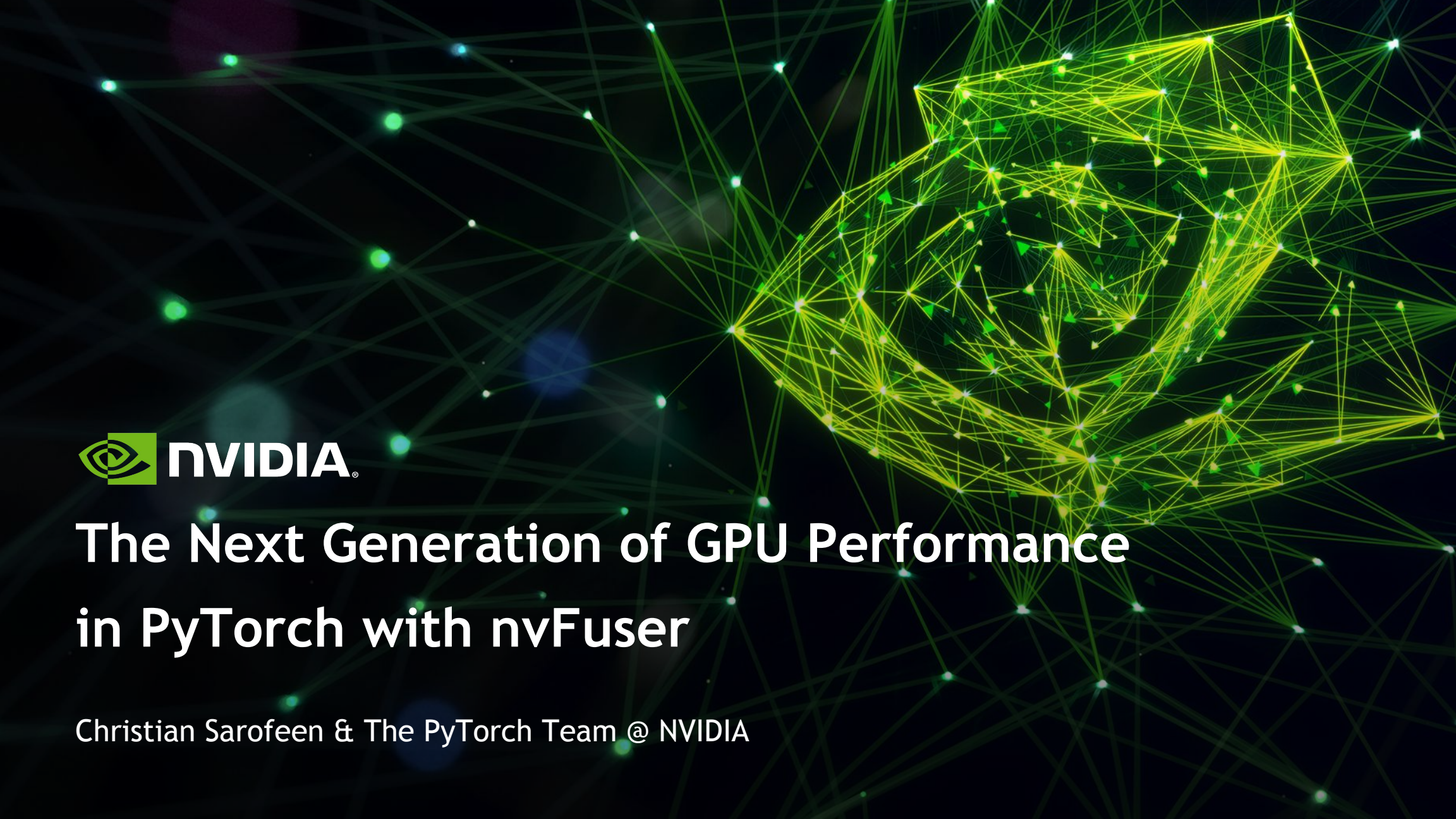


The Next Generation of GPU Performance in PyTorch with nvFuser

Christian Sarofeen & The PyTorch Team @ NVIDIA

“Fusion” is a Critical Technology for DL Compilers

“Fusion is XLA's single most important optimization.”

- TF XLA Docs

“TVM solves optimization challenges specific to deep learning, such as high-level operator fusion, …”

- TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

“Pipelines of simple map operations can be optimized by traditional loop fusion: merging multiple successive operations on each point into a single compound operation improves arithmetic intensity by maximizing producer-consumer locality, …”

- Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

“Fusion” is a Critical Technology for DL Compilers

“Fusion is XLA's single most important optimization.”

- TF XLA Docs

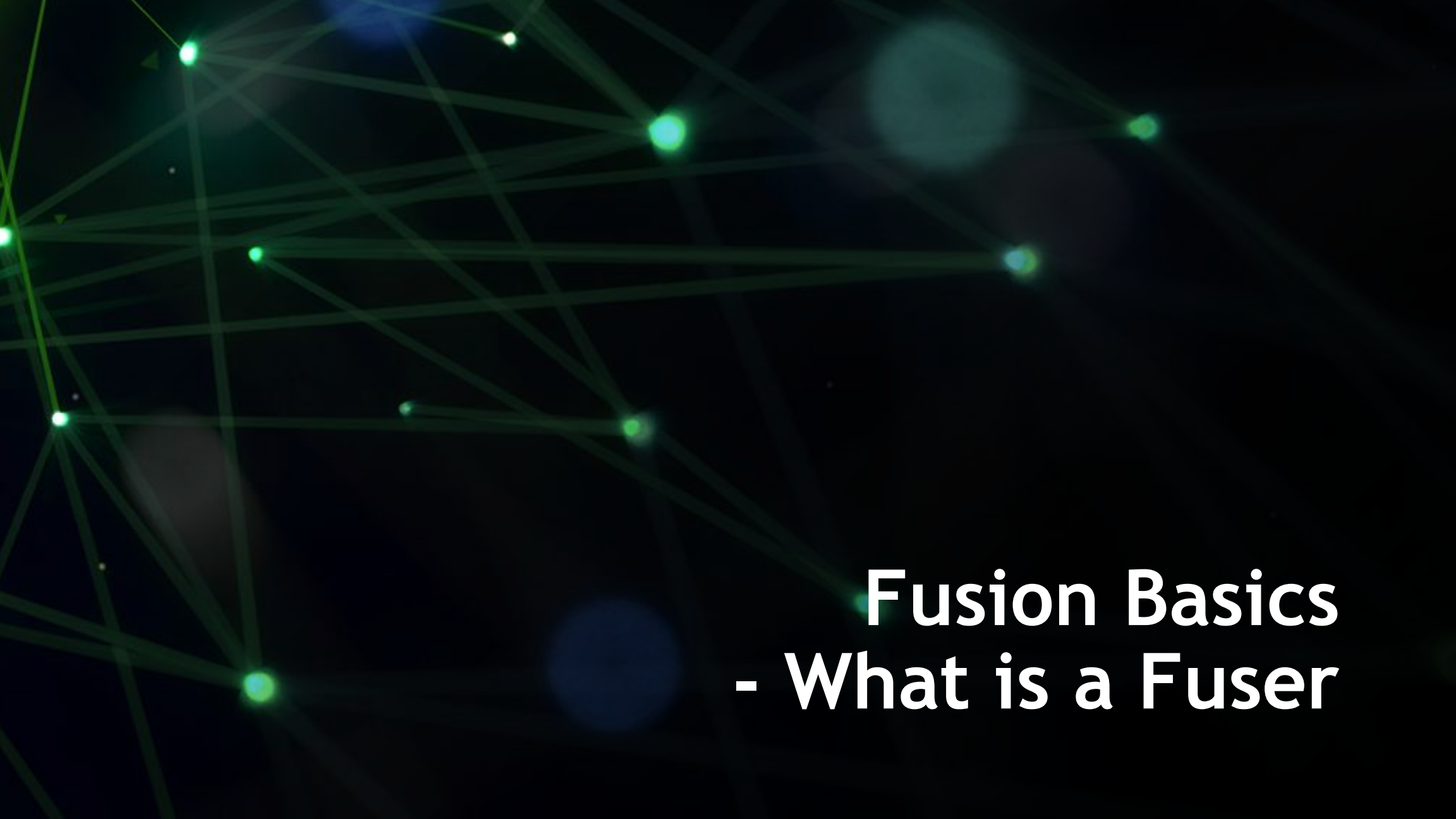
“TVM solves optimization challenges specific to deep learning, such as high-level operator fusion, …”

- TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

“Pipelines of simple map operations can be optimized by traditional loop fusion: merging multiple successive operations on each point into a single compound operation improves arithmetic intensity by maximizing producer-consumer locality, …”

- Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

**Taking multiple operations,
into a single compound operation,
to maximize locality.**



Fusion Basics
- What is a Fuser

Fusion basics

Which code snippet is faster?

```
size = 1024 * 1024

for i in range(size):
    y[i] = y[i] * y[i] * y[i]
for i in range(size):
    y[i] = y[i] * 0.044715
for i in range(size):
    y[i] = y[i] + a[i]
for i in range(size):
    y[i] = 0.5 * y[i]
for i in range(size):
    y[i] = (2./math.sqrt(math.pi)) * y[i]
for i in range(size):
    y[i] = math.sqrt(2.0) * y[i]
for i in range(size):
    y[i] = math.tanh(y[i])
for i in range(size):
    y[i] = y[i] + 1.0
for i in range(size):
    y[i] = a[i] * y[i]
```

```
size = 1024 * 1024

for i in range(size):
    y[i] = y[i] * (math.tanh(
        math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
        (y[i] * y[i] * y[i] * 0.044715 + y[i])) + 1.0)
```

Fusion basics

Which code snippet is faster?

```
size = 1024 * 1024

for i in range(size):
    y[i] = y[i] * y[i] * y[i]
for i in range(size):
    y[i] = y[i] * 0.044715
for i in range(size):
    y[i] = y[i] + a[i]
for i in range(size):
    y[i] = 0.5 * y[i]
for i in range(size):
    y[i] = (2./math.sqrt(math.pi)) * y[i]
for i in range(size):
    y[i] = math.sqrt(2.0) * y[i]
for i in range(size):
    y[i] = math.tanh(y[i])
for i in range(size):
    y[i] = y[i] + 1.0
for i in range(size):
    y[i] = a[i] * y[i]
```

77.19s

```
size = 1024 * 1024

for i in range(size):
    y[i] = y[i] * (math.tanh(
        math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
        (y[i] * y[i] * y[i] * 0.044715 + y[i])) + 1.0)
```

1.77s

Why is it so much faster?

Memory locality

Compute is lightning fast

Moving memory around is very slow (relatively speaking)

On A100 80GB

312 TFlops (312e12 FP16 operations per second)

2 TB/s (1e12 FP16 values per second from DRAM)

Memory locality == Keep data in cache or registers

Fusion is primarily the optimization of keeping intermediate values in cache or registers

Is there a difference now?

```
size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
                dtype=torch.float)
y = a.clone()

y = torch.pow(y, 3)
y = y * 0.044715
y = y + a
y = 0.5 * y
y = y * (2./math.sqrt(math.pi))
y = y * math.sqrt(2.0)
y = torch.tanh(y)
y = y + 1.0
y = a * y
```

```
size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
                dtype=torch.float)
y = a.clone()

y = y * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (y * y * y * 0.044715 + y)) + 1.0)
```

Is there a difference now?

Not really… why?

```
size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
                dtype=torch.float)
y = a.clone()

y = torch.pow(y, 3)
y = y * 0.044715
y = y + a
y = 0.5 * y
y = y * (2./math.sqrt(math.pi))
y = y * math.sqrt(2.0)
y = torch.tanh(y)
y = y + 1.0
y = a * y
```

13.6ms

```
size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
                dtype=torch.float)
y = a.clone()

y = y * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (y * y * y * 0.044715 + y)) + 1.0)
```

12.9ms

PyTorch

is a tensor library

Every operation PyTorch gets, runs eagerly

Although the code looks different, in both snippets the code is executed one tensor operation at a time

Intermediate values are pushed to GPU DRAM

This is where a fuser really helps

With nvFuser

Full example

```
size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
                dtype=torch.float)
y = a.clone()

y = y * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (y * y * y * 0.044715 + y)) + 1.0)
```

```
import torch
import math
from timeit import default_timer as timer

size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
dtype=torch.float)
y = a.clone()

@torch.jit.script
def activation(x):
  return x * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (x * x * x * 0.044715 + x)) + 1.0)


with torch.jit.fuser("fuser2"):
    # warmup iterations
    for i in range(3):
        activation(y)

    torch.cuda.synchronize()
    start = timer()
    activation(y)
    torch.cuda.synchronize()
    end = timer()
    print(end - start)
```

Let’s break down the example quickly

```
import torch
import math
from timeit import default_timer as timer

size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
dtype=torch.float)
y = a.clone()
```

```
@torch.jit.script
def activation(x):
  return x * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (x * x * x * 0.044715 + x)) + 1.0)


with torch.jit.fuser("fuser2"):
    # warmup iterations
    for i in range(3):
        activation(y)

    torch.cuda.synchronize()
    start = timer()
    activation(y)
    torch.cuda.synchronize()
    end = timer()
    print(end - start)
```

Basic problem setup

Let’s break down the example quickly

```
import torch
import math
from timeit import default_timer as timer

size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
dtype=torch.float)
y = a.clone()

@torch.jit.script
def activation(x):
  return x * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (x * x * x * 0.044715 + x)) + 1.0)


with torch.jit.fuser("fuser2"):
    # warmup iterations
    for i in range(3):
        activation(y)

    torch.cuda.synchronize()
    start = timer()
    activation(y)
    torch.cuda.synchronize()
    end = timer()
    print(end - start)
```

TorchScript decorator above target function. This turns user code into a graph that nvFuser can optimize.

Let’s break down the example quickly

```
import torch
import math
from timeit import default_timer as timer

size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
dtype=torch.float)
y = a.clone()

@torch.jit.script
def activation(x):
  return x * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (x * x * x * 0.044715 + x)) + 1.0)


with torch.jit.fuser("fuser2"):
    # warmup iterations
    for i in range(3):
        activation(y)

    torch.cuda.synchronize()
    start = timer()
    activation(y)
    torch.cuda.synchronize()
    end = timer()
    print(end - start)
```

Enable nvFuser. PyTorch ships with nvFuser today, but must be manually enabled. Soon nvFuser will be the default so this line won’t be necessary then.

Let’s break down the example quickly

```
import torch
import math
from timeit import default_timer as timer

size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
dtype=torch.float)
y = a.clone()

@torch.jit.script
def activation(x):
  return x * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (x * x * x * 0.044715 + x)) + 1.0)


with torch.jit.fuser("fuser2"):
    # warmup iterations
    for i in range(3):
        activation(y)

    torch.cuda.synchronize()
    start = timer()
    activation(y)
    torch.cuda.synchronize()
    end = timer()
    print(end - start)
```

nvFuser takes 3 iterations to “engage”.

1. Generate TorchScript Graph
2. Collect runtime parameters
3. nvFuser compiles the graph
4. nvFuser runs its specialized kernels

Let’s break down the example quickly

```
import torch
import math
from timeit import default_timer as timer

size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
dtype=torch.float)
y = a.clone()

@torch.jit.script
def activation(x):
  return x * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (x * x * x * 0.044715 + x)) + 1.0)


with torch.jit.fuser("fuser2"):
    # warmup iterations
    for i in range(3):
        activation(y)

    torch.cuda.synchronize()
    start = timer()
    activation(y)
    torch.cuda.synchronize()
    end = timer()
    print(end - start)
```

Timers with proper device syncs

Let’s break down the example quickly

```
import torch
import math
from timeit import default_timer as timer

size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
dtype=torch.float)
y = a.clone()

@torch.jit.script
def activation(x):
  return x * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (x * x * x * 0.044715 + x)) + 1.0)


with torch.jit.fuser("fuser2"):
    # warmup iterations
    for i in range(3):
        activation(y)

    torch.cuda.synchronize()
    start = timer()
    activation(y)
    torch.cuda.synchronize()
    end = timer()
    print(end - start)
```

Use of the scripted function

With nvFuser

Full example

```
size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
                dtype=torch.float)
y = a.clone()

y = y * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (y * y * y * 0.044715 + y)) + 1.0)
```

14.3ms

```
import torch
import math
from timeit import default_timer as timer

size = 128 * 1024 * 1024
a = torch.randn((size), device=torch.device("cuda"),
dtype=torch.float)
y = a.clone()

@torch.jit.script
def activation(x):
  return x * (torch.tanh(
    math.sqrt(2.0) * (2. / math.sqrt(math.pi)) * 0.5 *
    (x * x * x * 0.044715 + x)) + 1.0)


with torch.jit.fuser("fuser2"):
    # warmup iterations
    for i in range(3):
        activation(y)

    torch.cuda.synchronize()
    start = timer()
    activation(y)
    torch.cuda.synchronize()
    end = timer()
    print(end - start)
```

1.32ms

What is Fusion

When discussing DL Compilers, Fusion is:

User Defined Operations ⟶ Efficient Device Specific Code

Performance is determined by:

1. How many operators we can fuse into one GPU function
2. **How efficient those functions are**

The former is trivial, the latter is what we're focused on


nvFuser

What is nvFuser

nvFuser is ~~a Fuser~~

- an automated system that
  - takes PyTorch programs
  - and generates high performing GPU code
- a system designed and built by CUDA programers to generate CUDA code
- written by NVIDIA in close collaboration with Meta and the PyTorch community
- built for PyTorch
- available with PyTorch

What is nvFuser

Inspired by Halide but built from the ground up for GPUs and dynamic shapes

Analyzes the user program, plans parallelization schemes and optimization strategies, then designs and generates the CUDA program

Support for:

- Backwards pass
- Bool, Int32, Int64, FP16, BFloat16, FP32, FP64
- Pointwise ops, reductions, normalizations, view
- Dynamic shapes

For more information: GTC 2021 Dynamic Shapes First: Advanced GPU Fusion in PyTorch

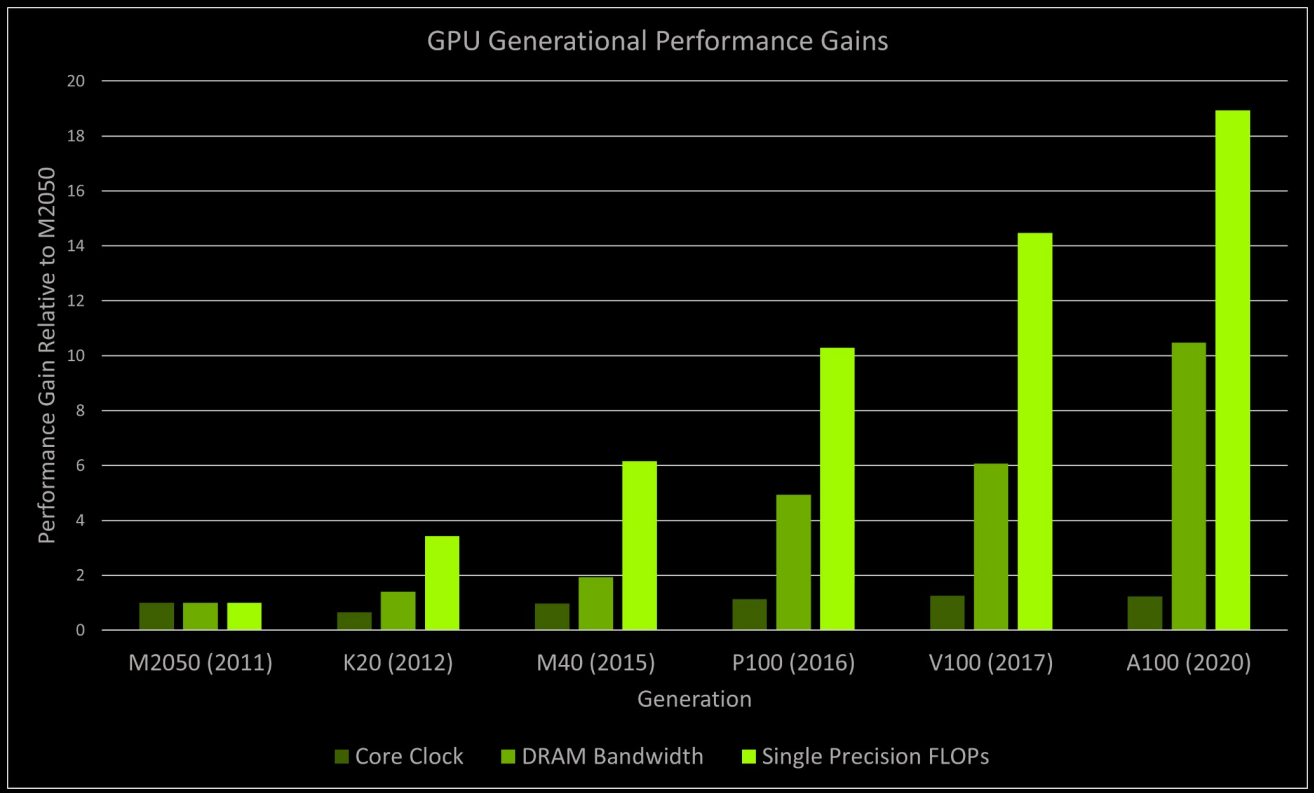
Coming soon:

- Channels Last (performance), Complex, Transpose, Pooling Layers, MatMul

Performance is Challenging

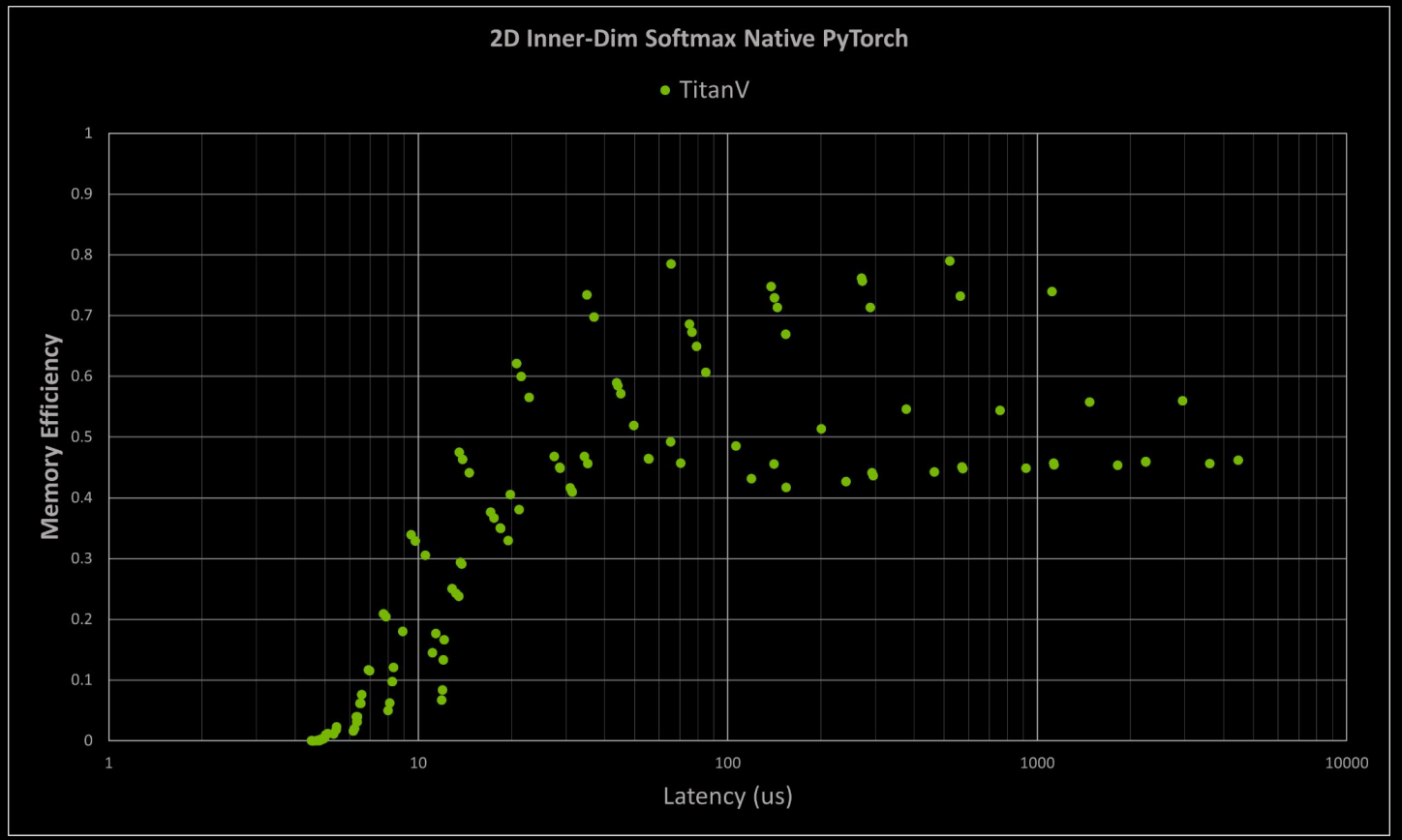
and it's only getting harder...

- GPUs are amazing latency hiding machines
- When one thread is waiting for something, another thread can execute
- However, latency is not improving at the rate of memory and compute

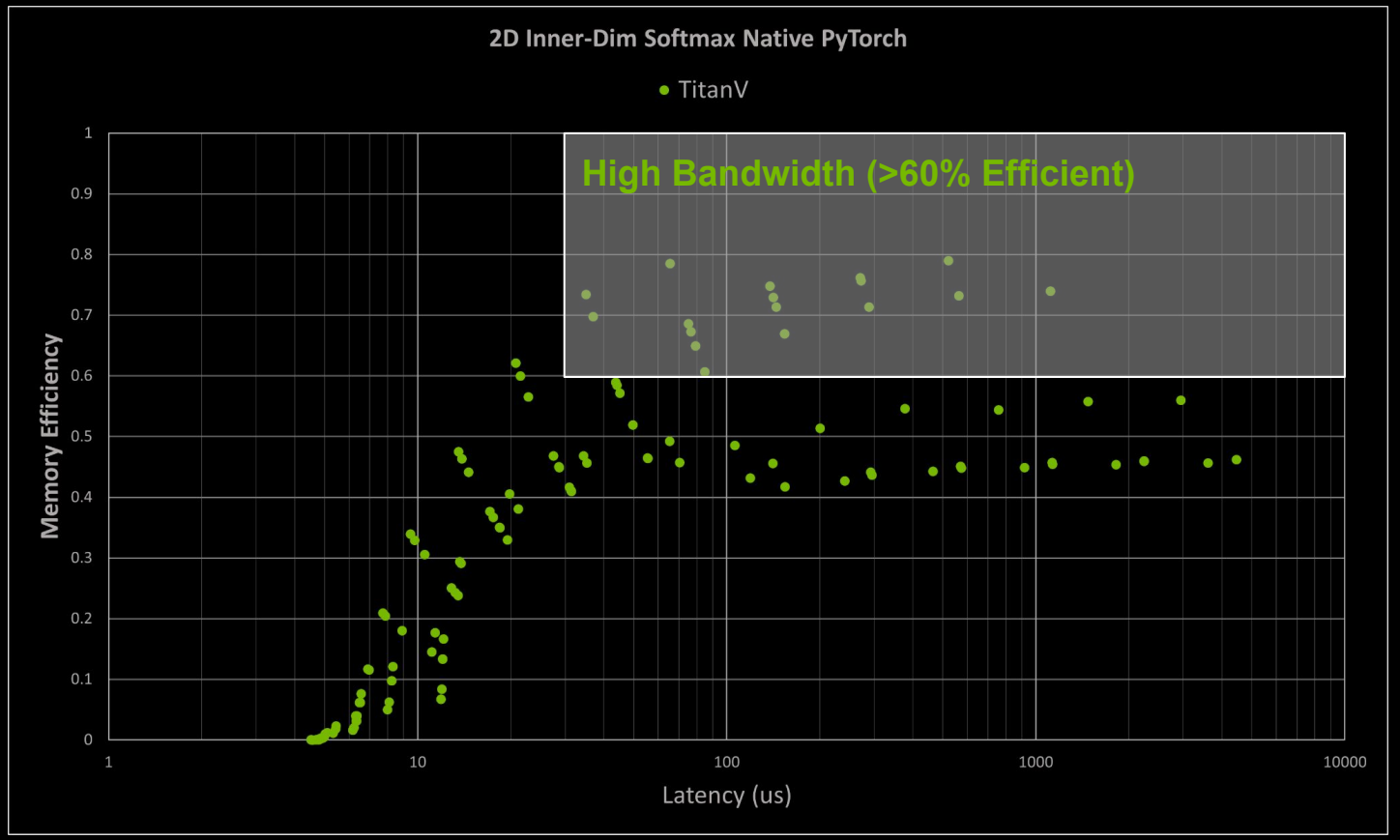


What Does This Mean Practically?

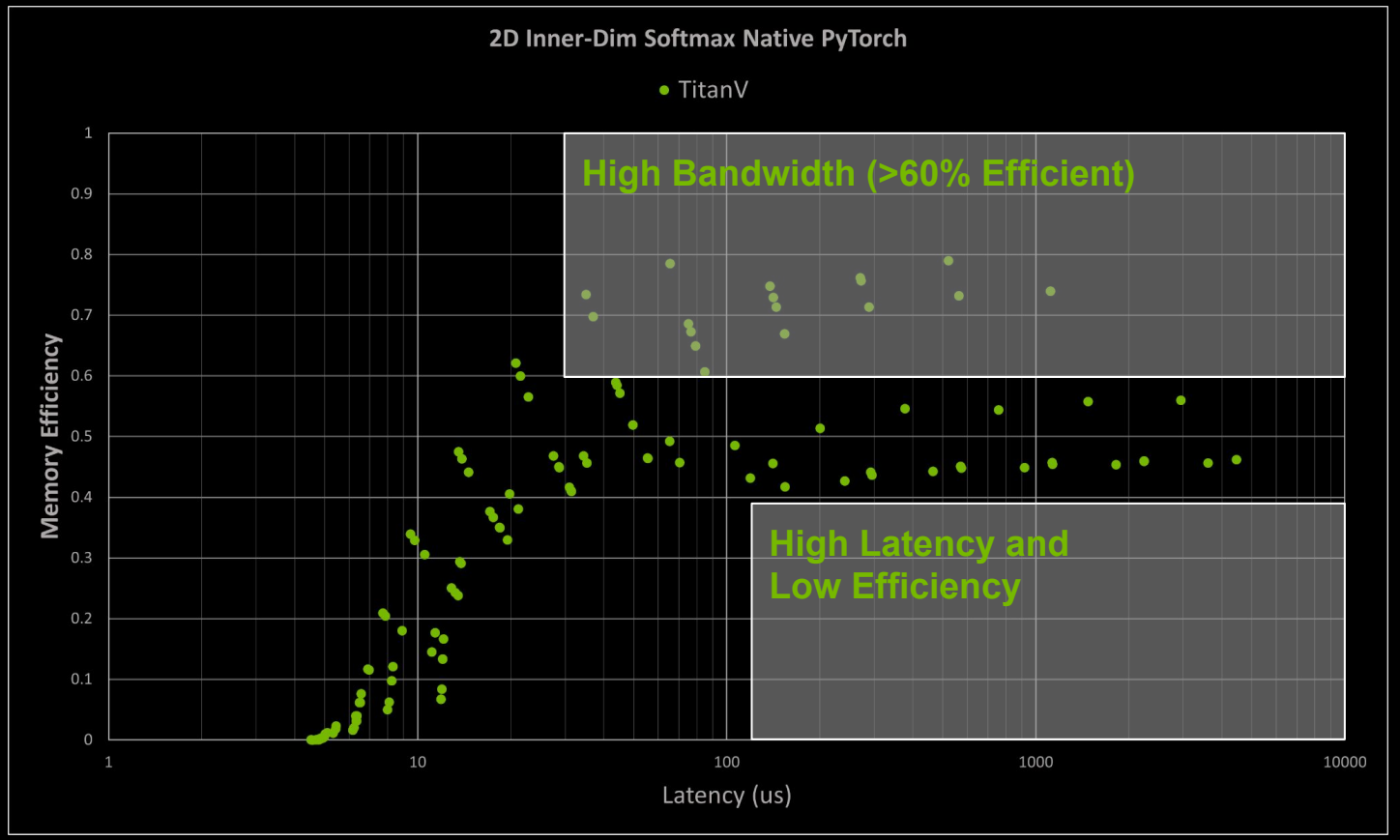
Software needs to improve, for efficiency to improve



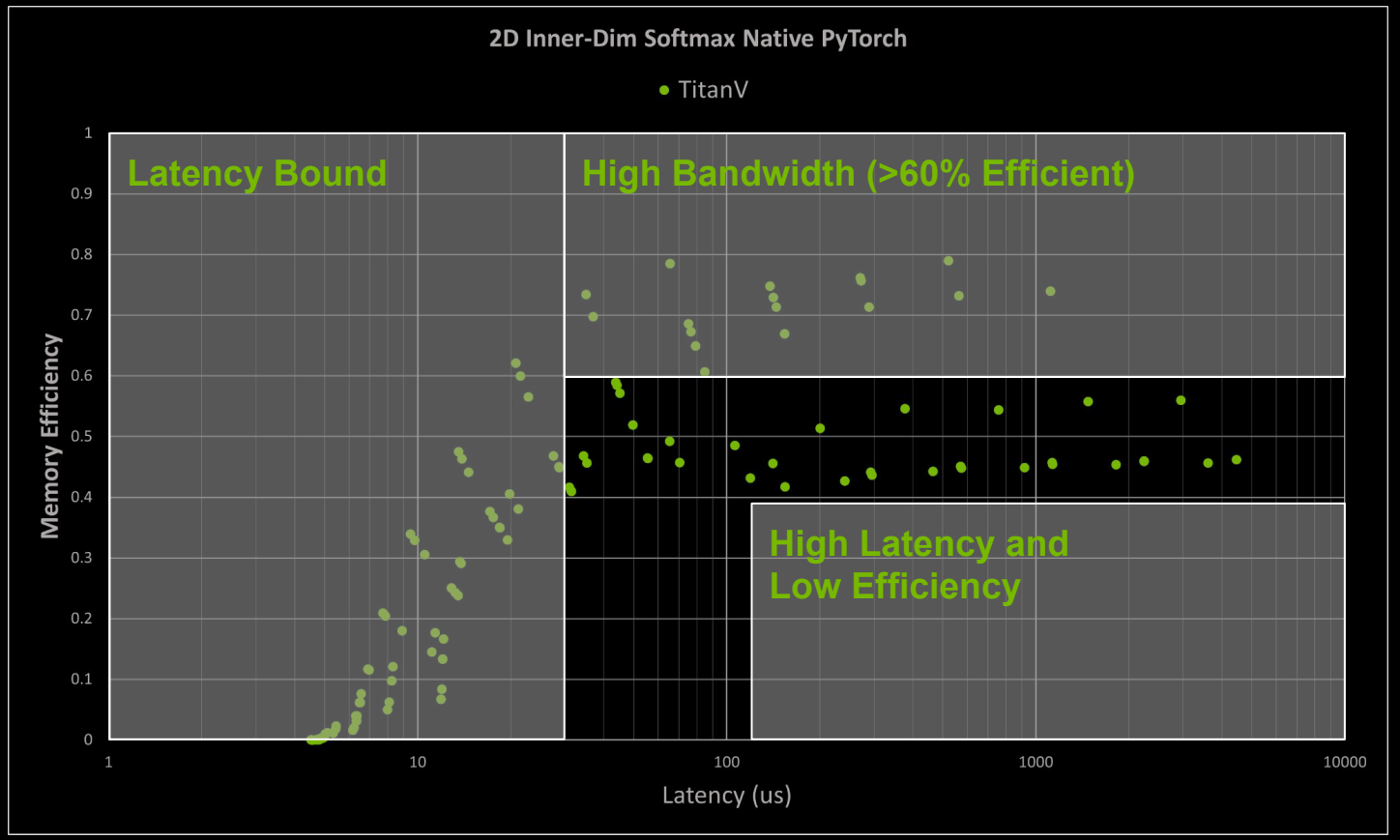
Sidebar - Perf Plot Description



Sidebar - Perf Plot Description



Sidebar - Perf Plot Description

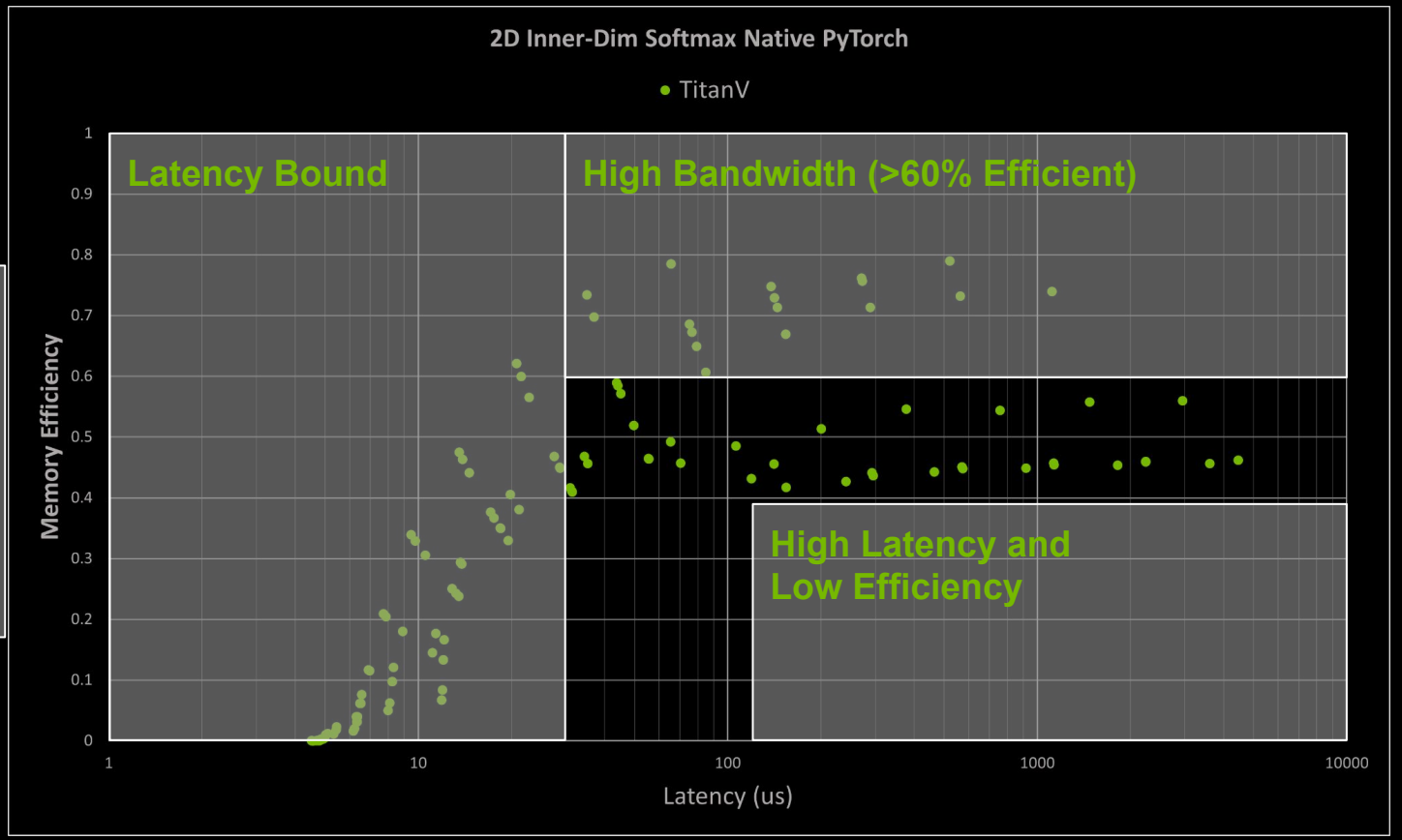


Sidebar - Perf Plot Description

"Memory Efficiency"

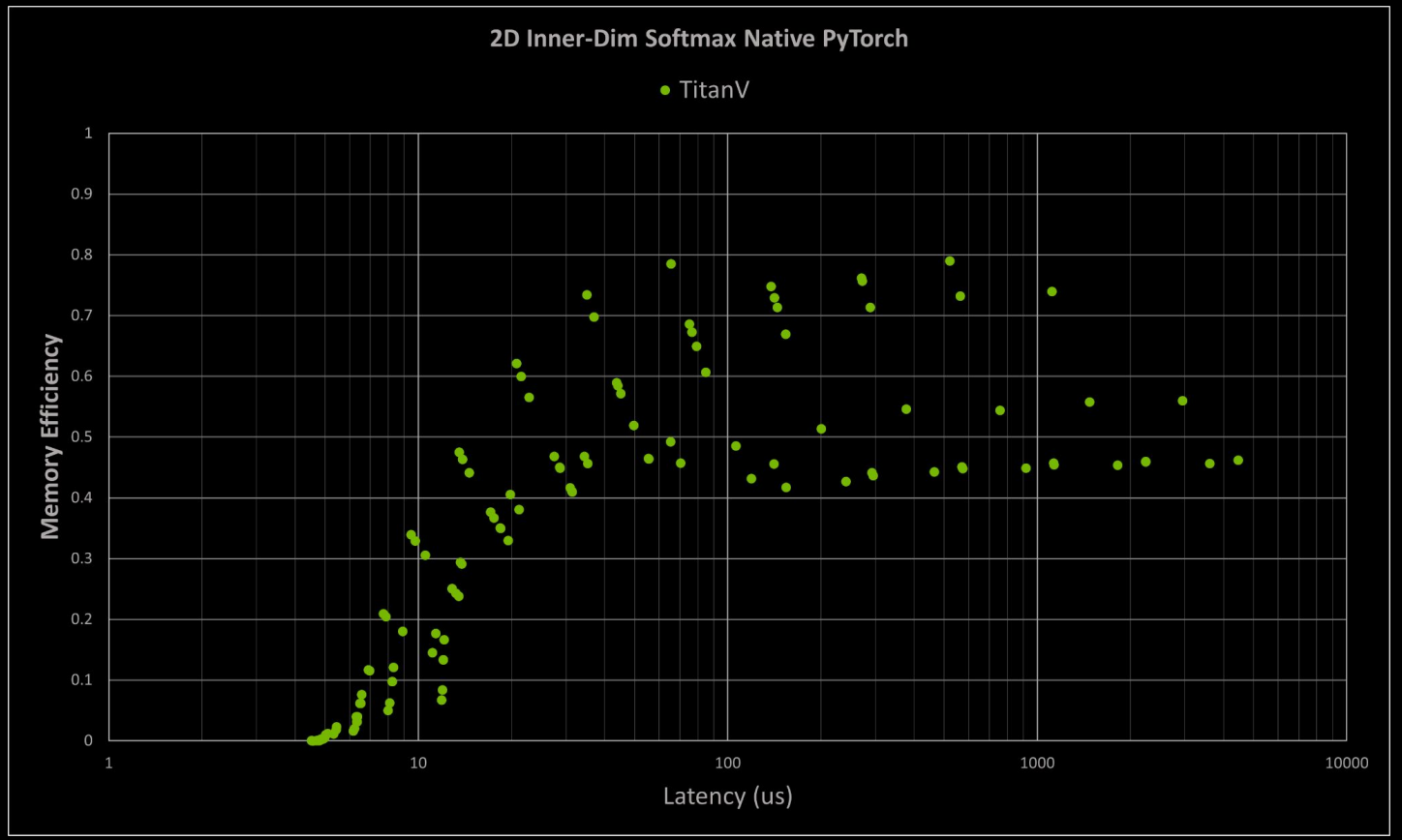
Assumes an operation can be done in a single function and fits into cache.

This is not always possible.



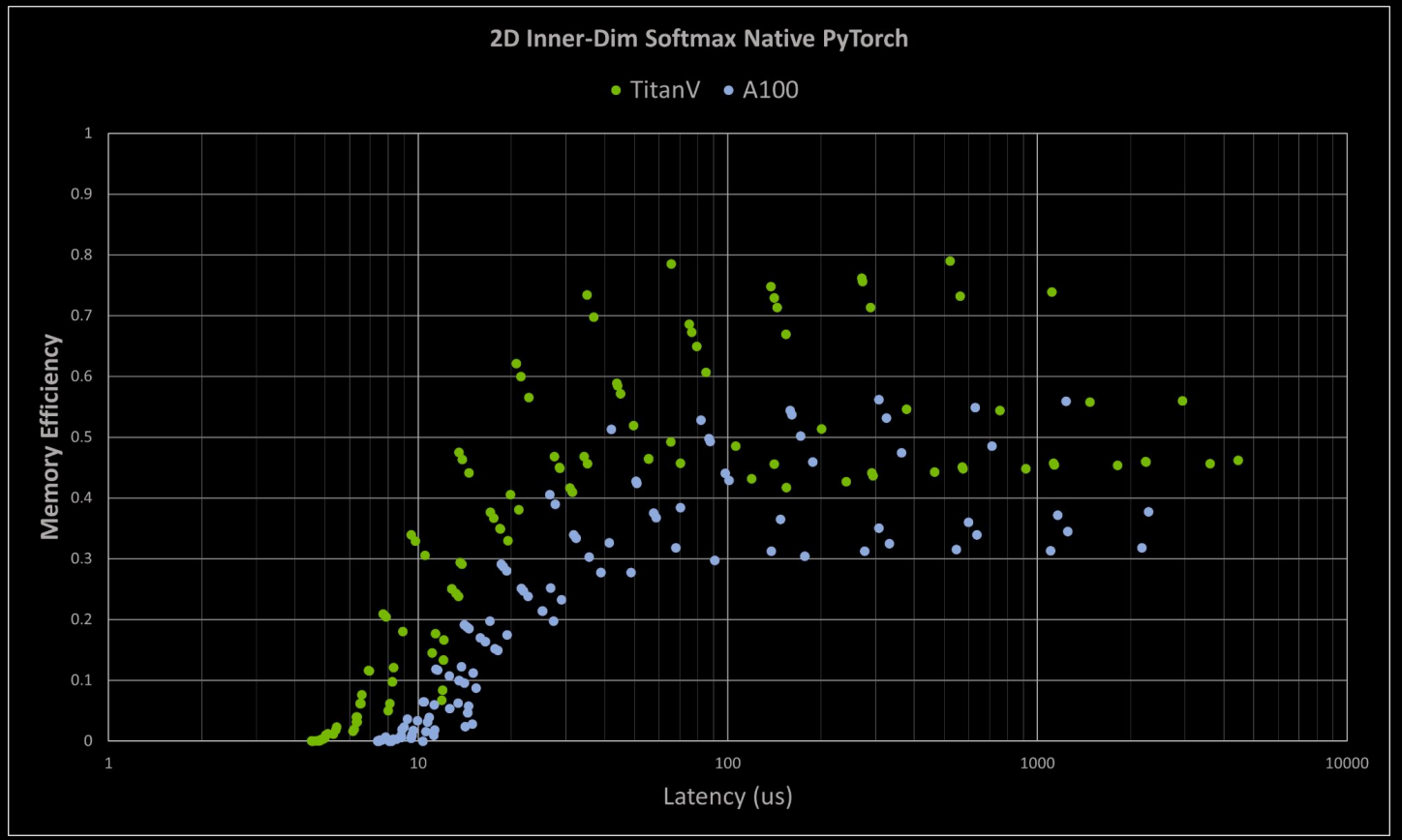
What Does This Mean Practically?

Software needs to improve, for efficiency to improve



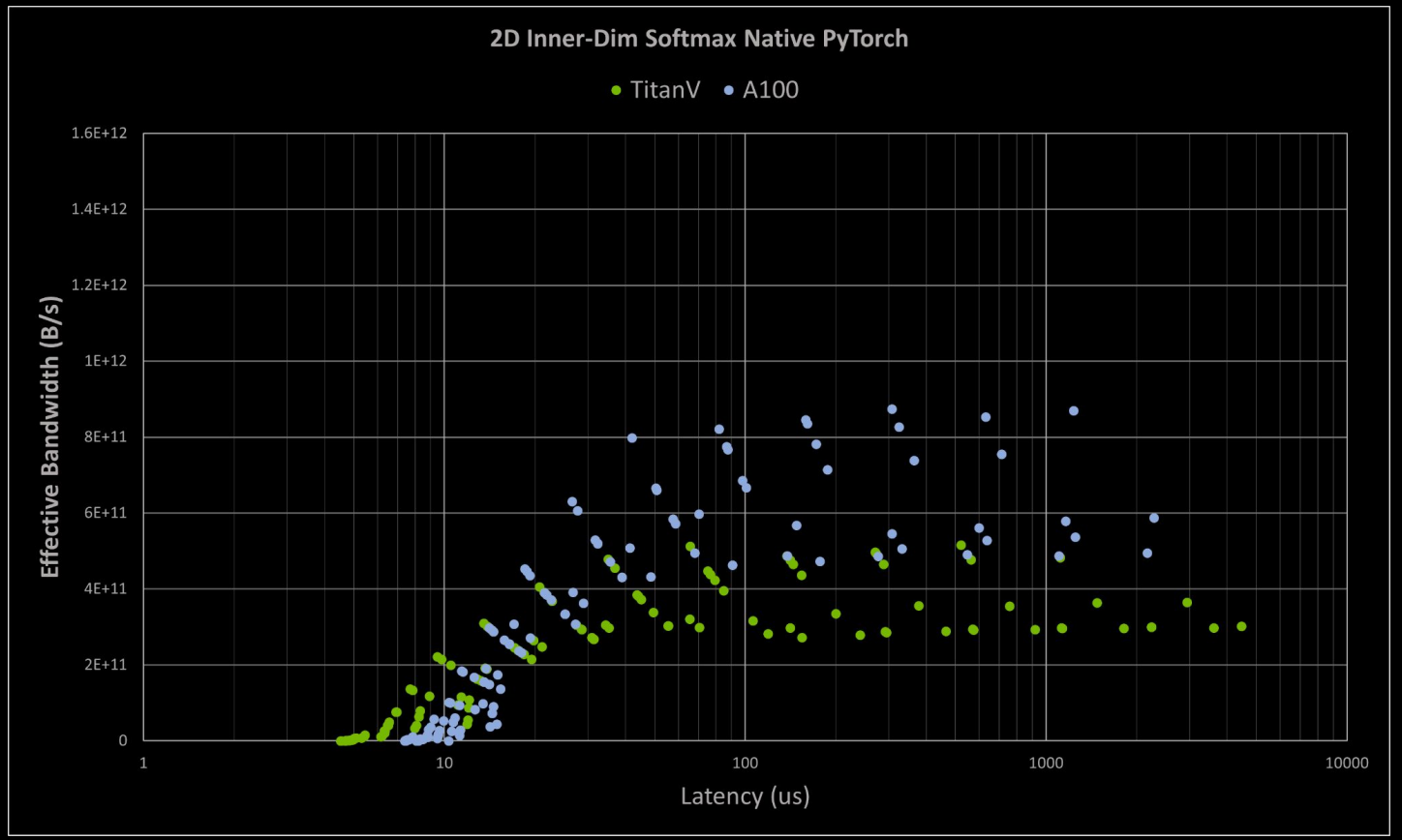
What Does This Mean Practically?

Software needs to improve, for efficiency to improve



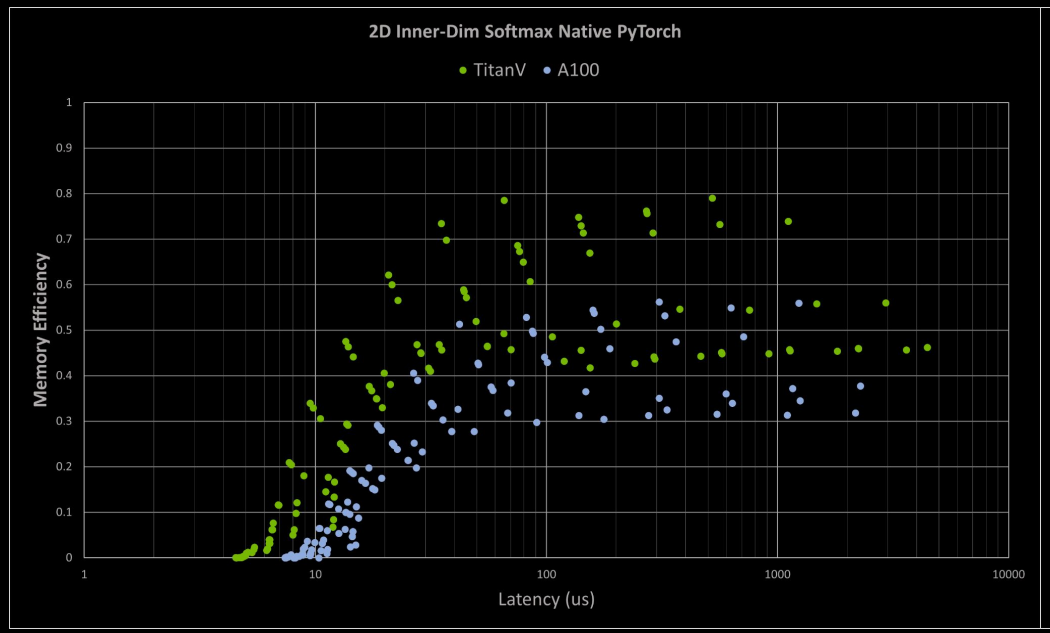
What Does This Mean Practically?

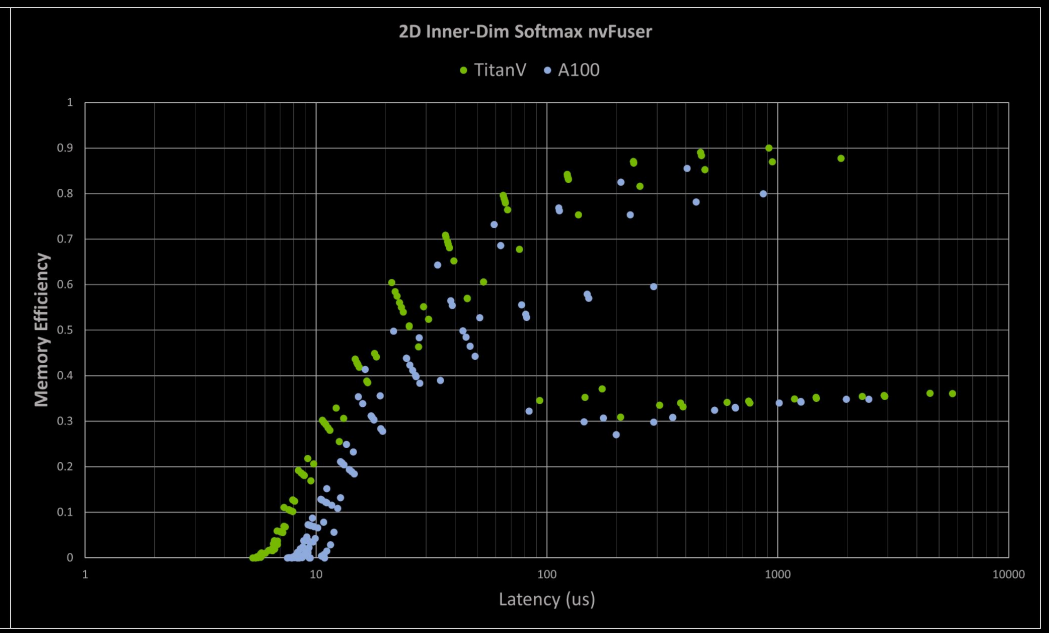
It’s still faster, just less efficient



What Does This Mean Practically?

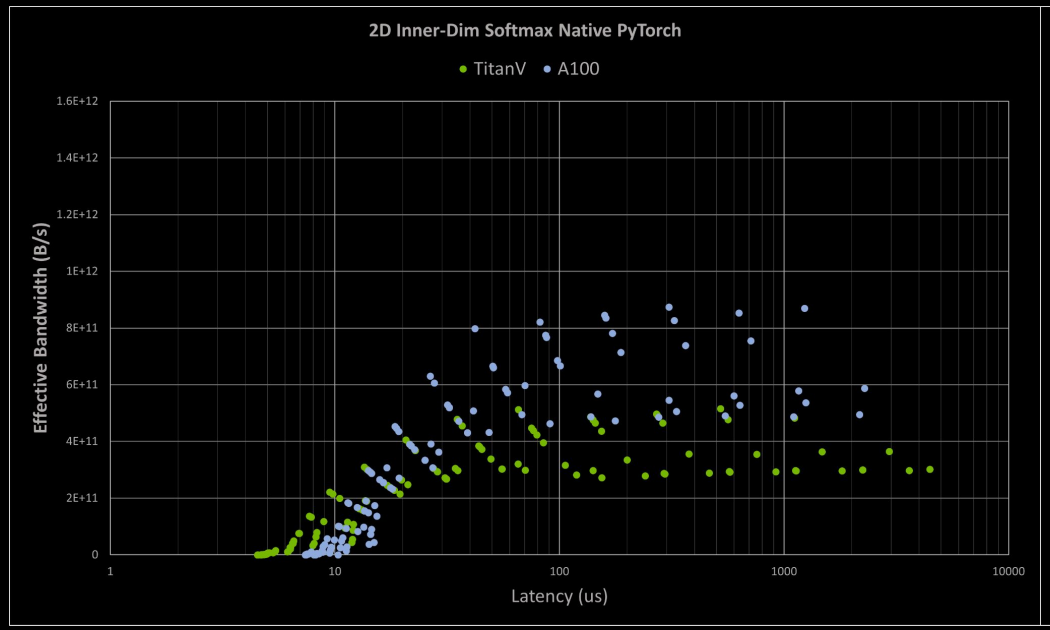
We Know Software Can Make it Better

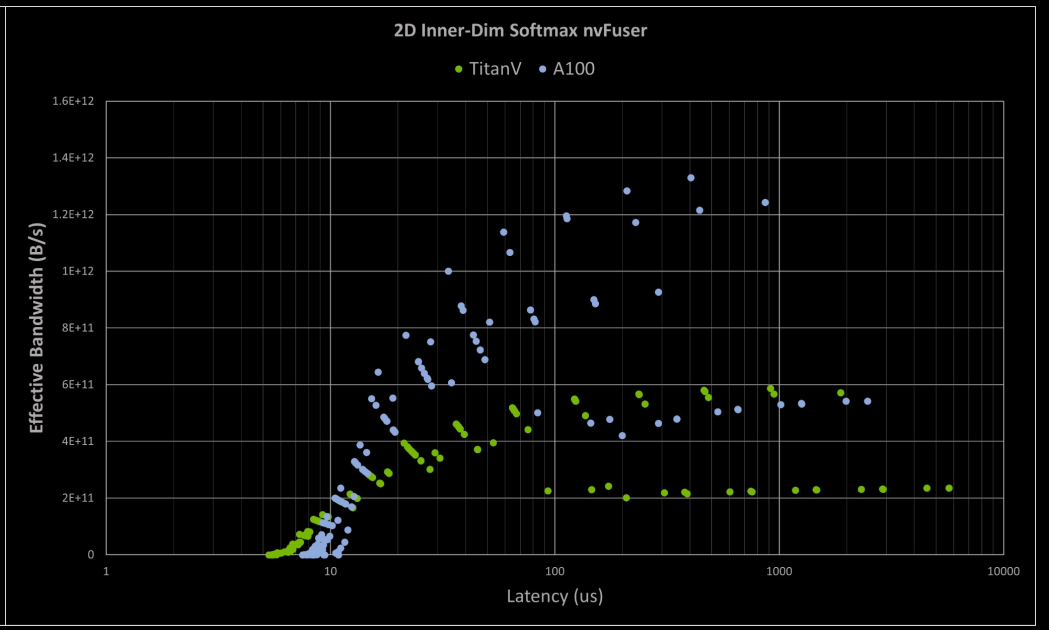




What Does This Mean Practically?

We Know Software Can Make it Better





NVIDIA

We can do better

…than just fusing existing kernels

Performance is determined by:

1. How many operators we can fuse into one GPU function
2. **How efficient those functions are**

Many DL approaches today focus on fusing existing kernels with “other things”

Or mapping operators to pre-compiled fusions (think Conv-BN-Activation)

nvFuser builds kernels from the ground up, allowing it to target novel operators

How is it done?

Analysis + Optimizations

Starts with analysis of the program, is this a fusion with a reduction, broadcast pattern, normalization

Heuristics take that information and map it to an optimization scheme

Schedulers applies a “bag of optimizations” to generate a CUDA kernel

This approach is applied independently of how operations are composed:

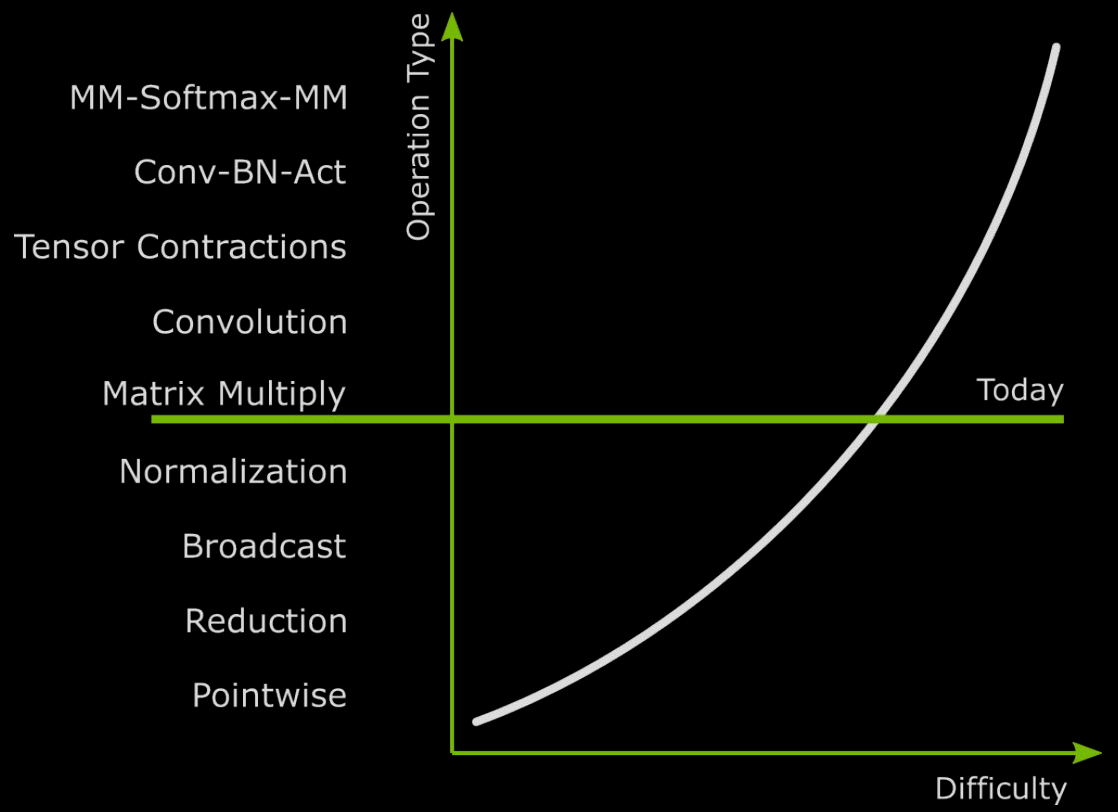
> Doesn’t matter if it’s a Softmax, LayerNorm, RMSNorm, InstanceNorm3D channels last, BatchNorm, or EvoNorm. The same process applies.

Allows us to program our knowledge into the system

More information: GTC 2021 Dynamic Shapes First: Advanced GPU Fusion in PyTorch

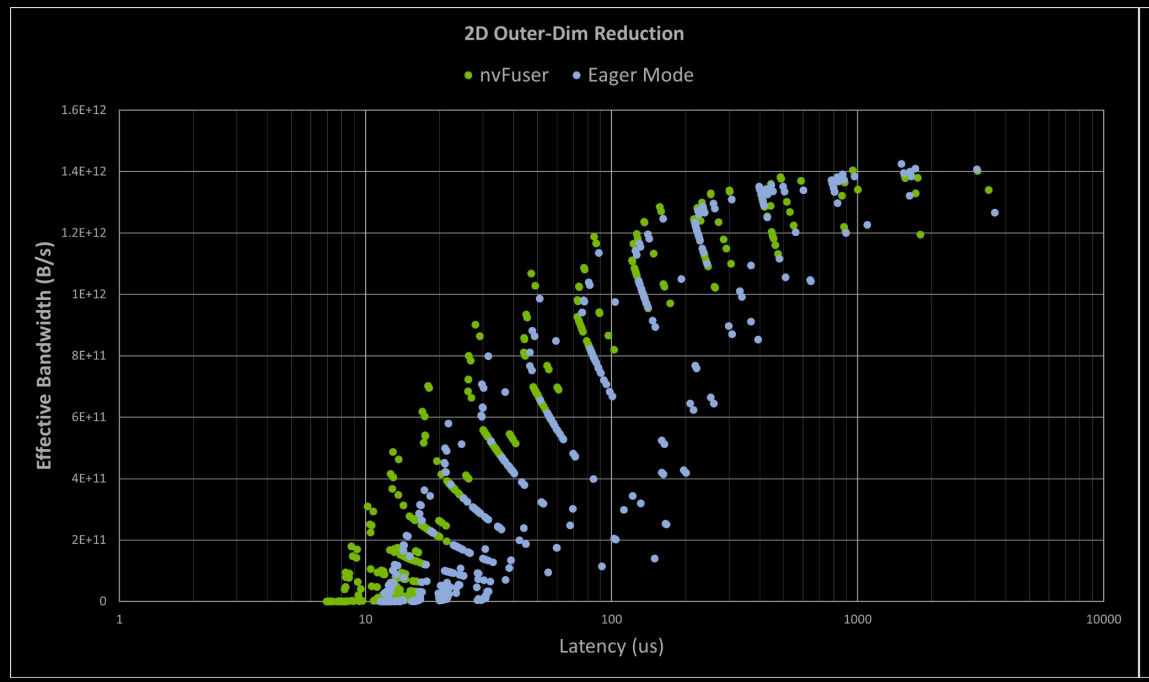
Problem solved?

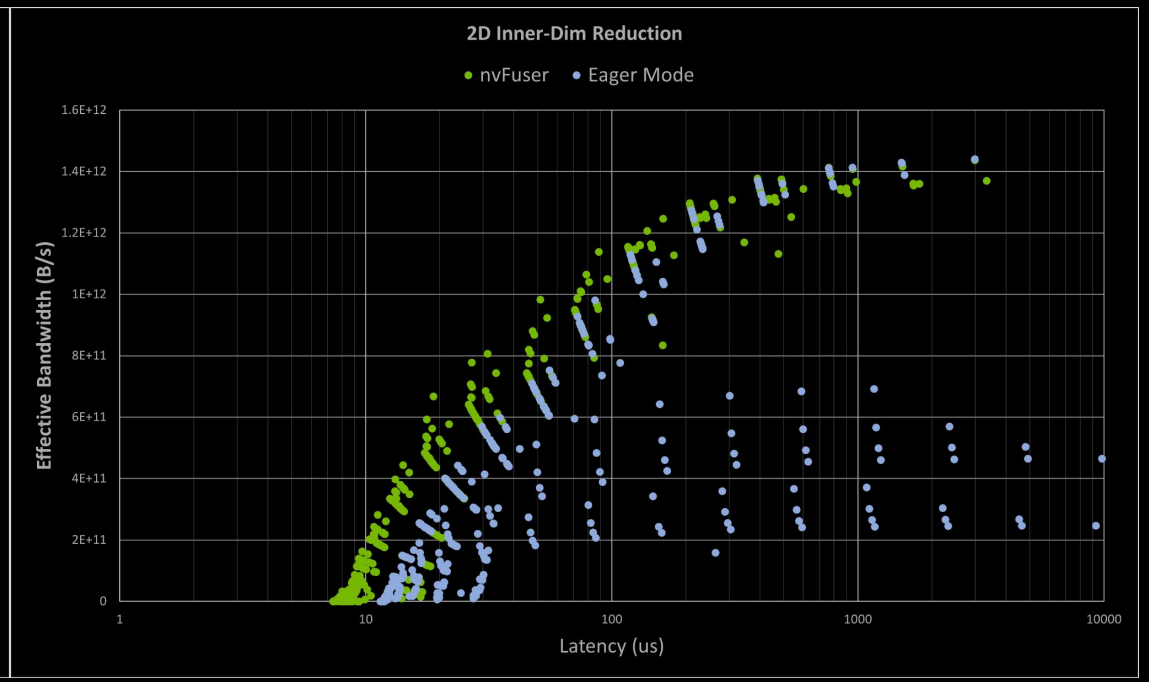
How challenging is it to write a fast kernel



Reductions

2D Reduce one axis (892 cases)

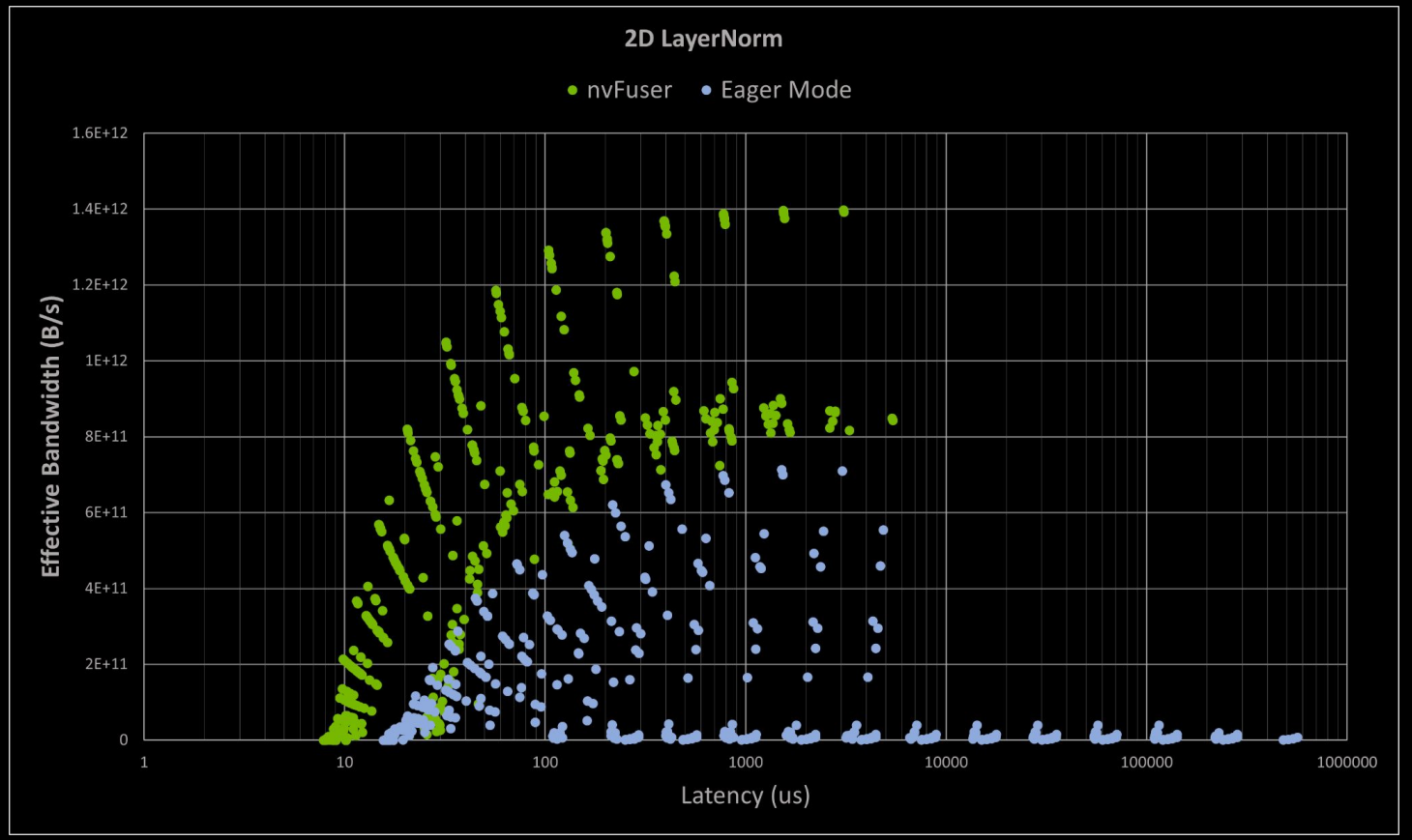




NVIDIA.

Layer Norm

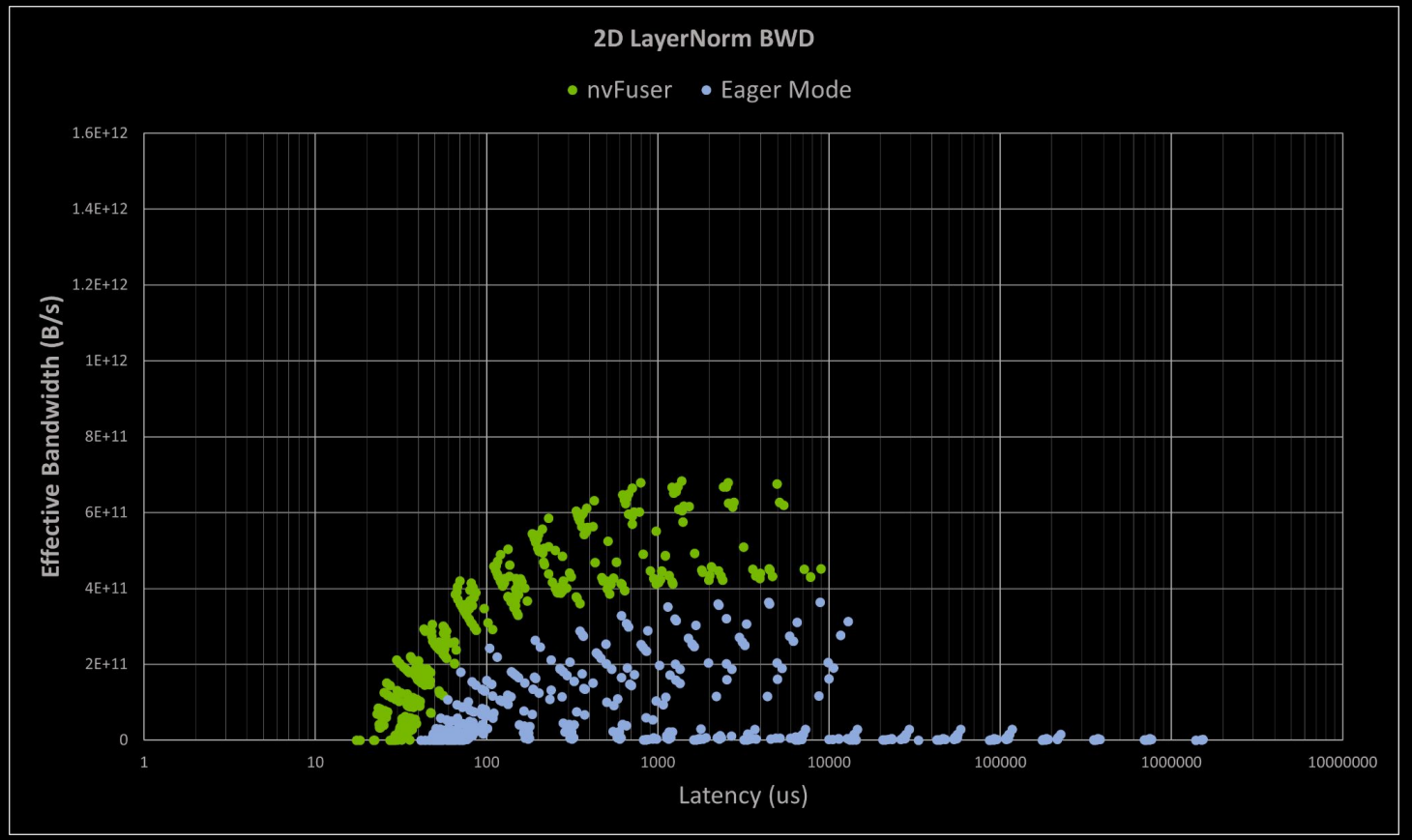
2D normalize inner dimension (432 cases)



NVIDIA

Layer Norm Backward

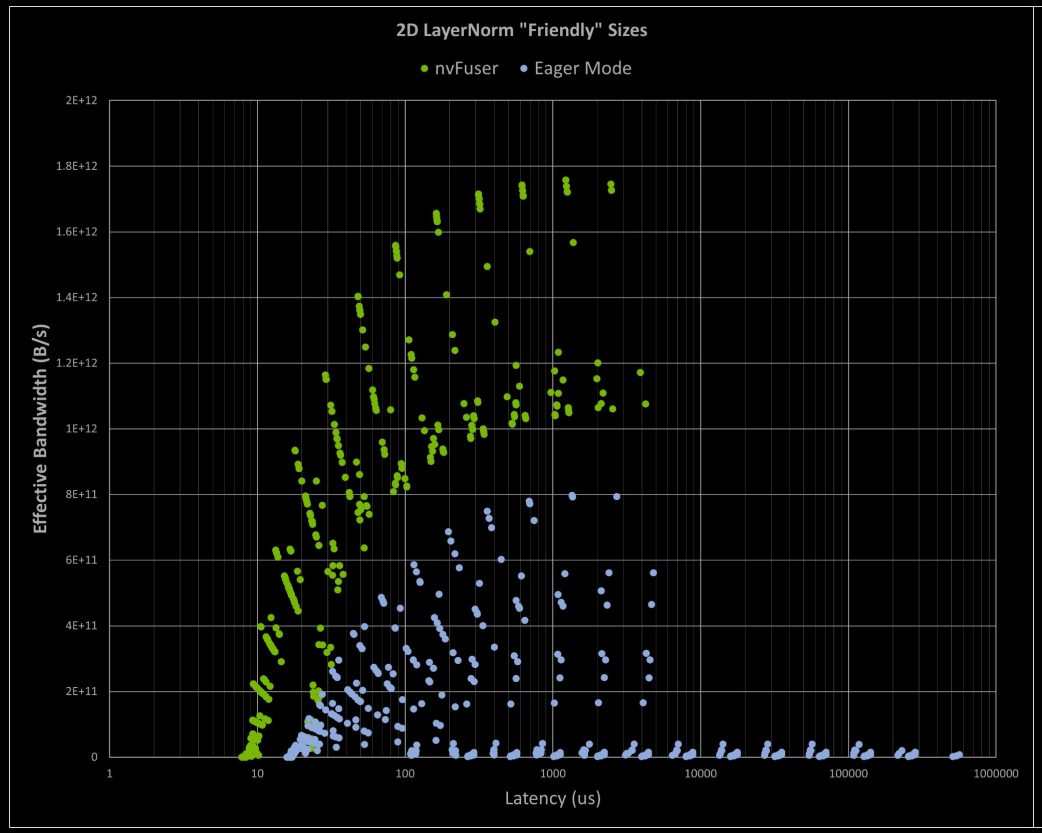
2D normalize inner dimension (432 cases)

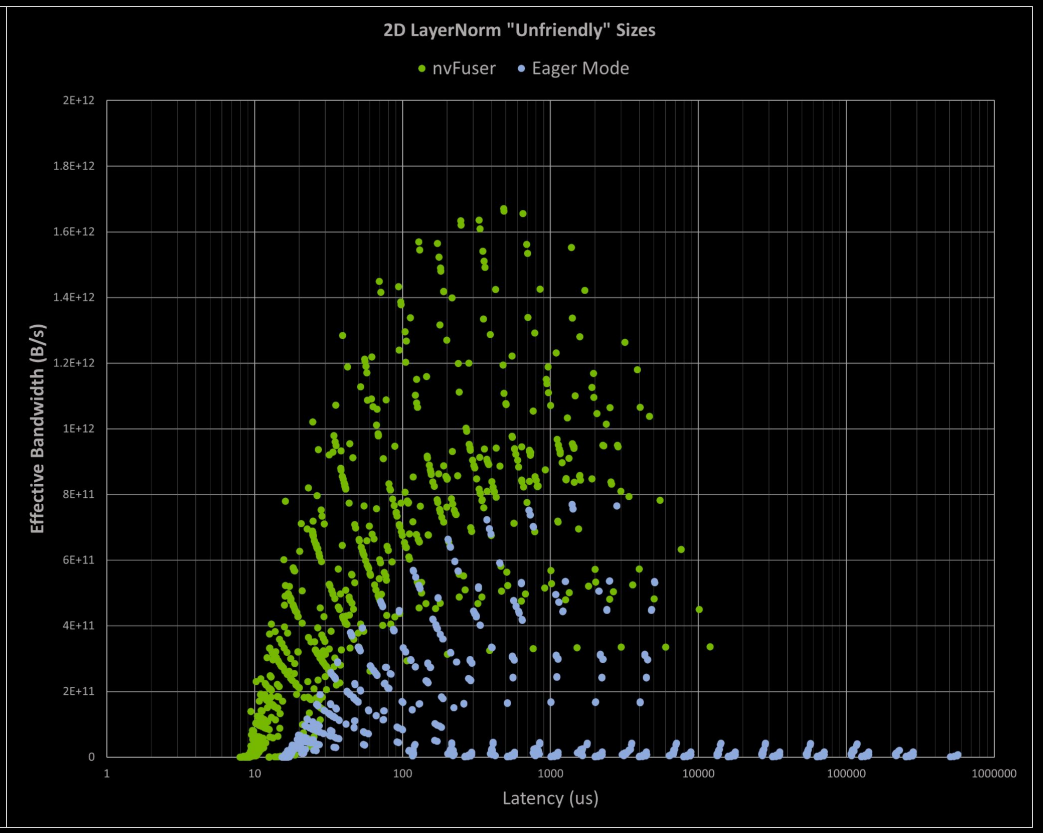


NVIDIA.

“Unfriendly” Layer Norm A100 80GB

Power of 2 Sizes vs Power 2 Sizes + or - 1



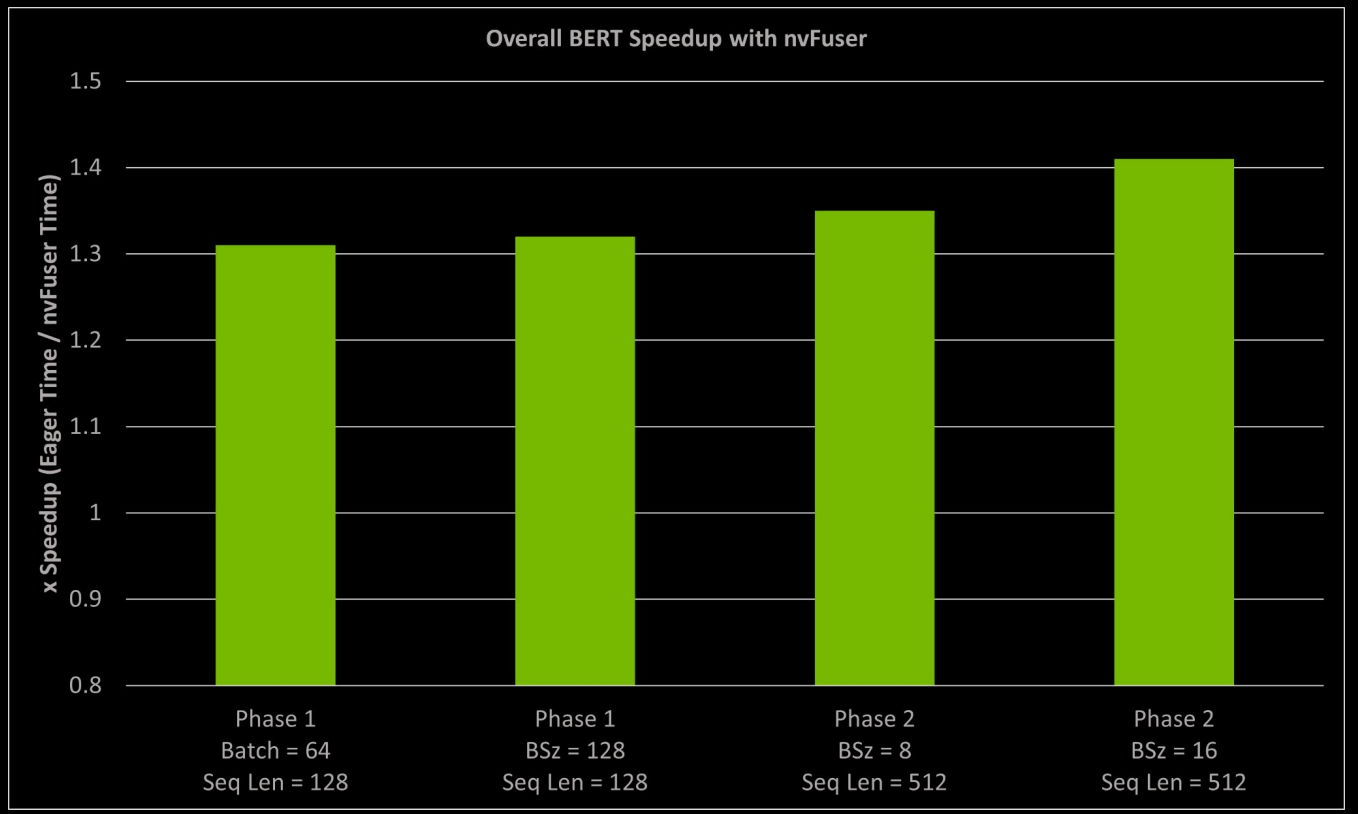


NVIDIA


nvFuser - E2E
Performance

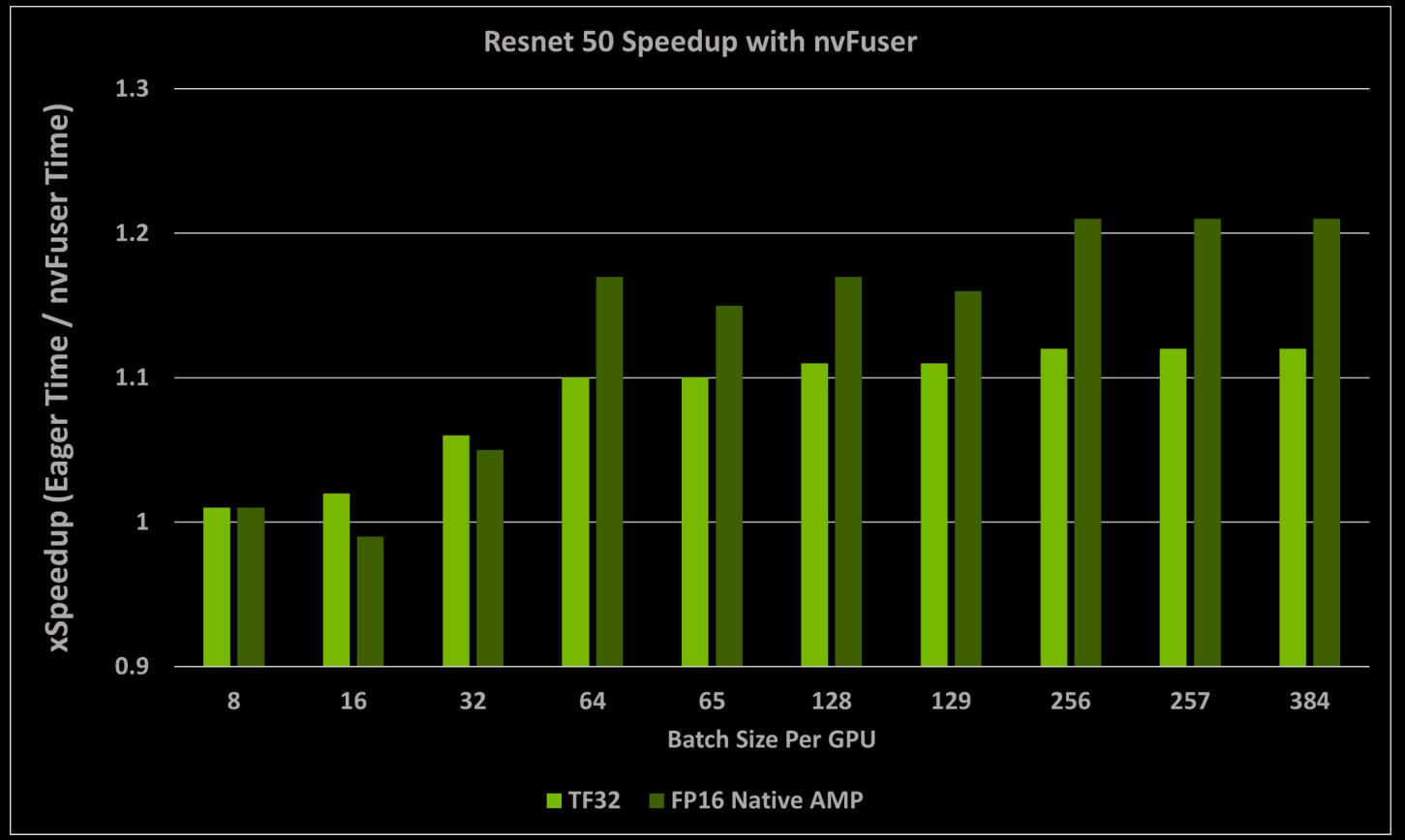
JOC BERT E2E Speedup 8xA100 80GB (With Dynamic Batching)

https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling



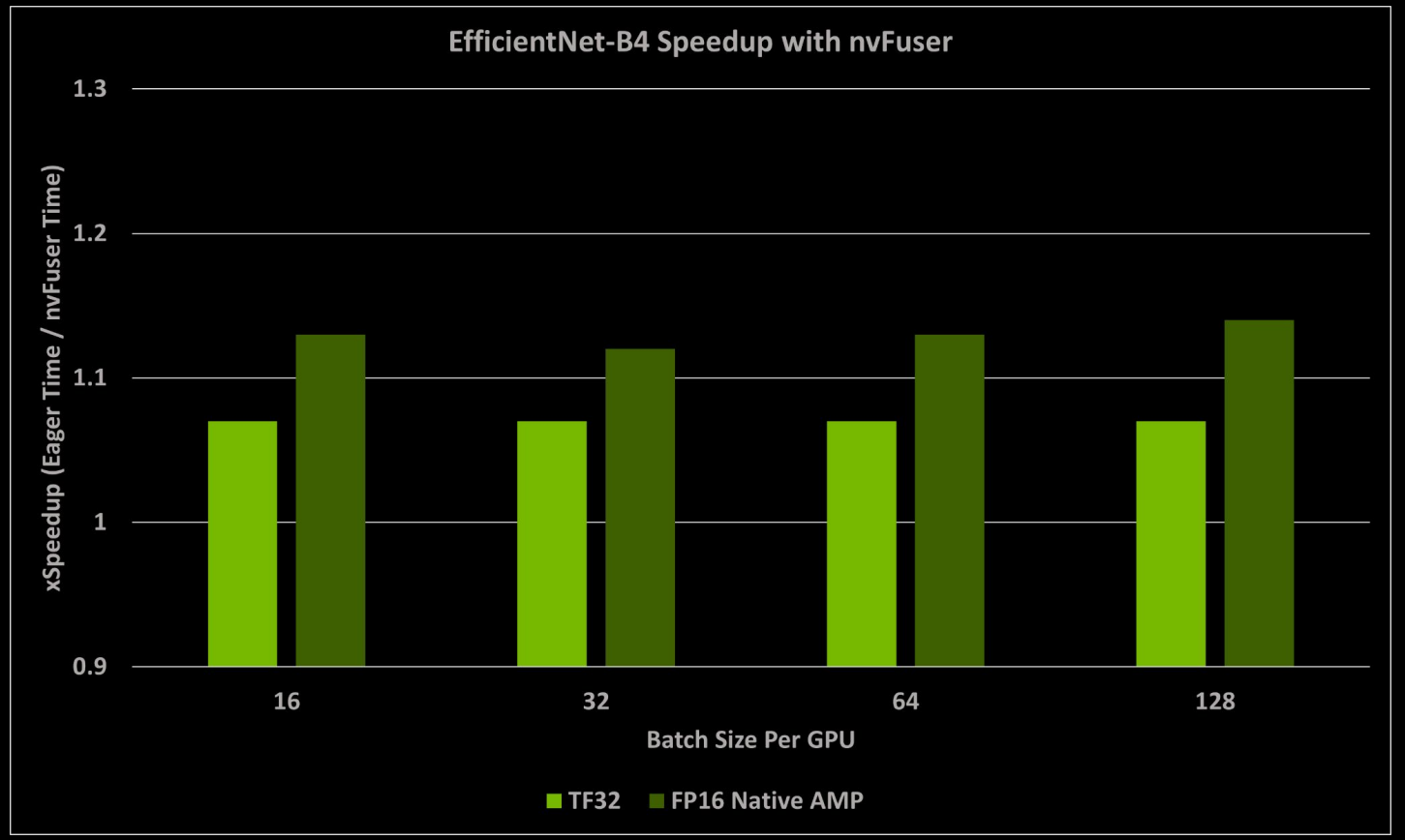
Resnet 50

8xA100 80GB NCHW



EfficientNet-B4

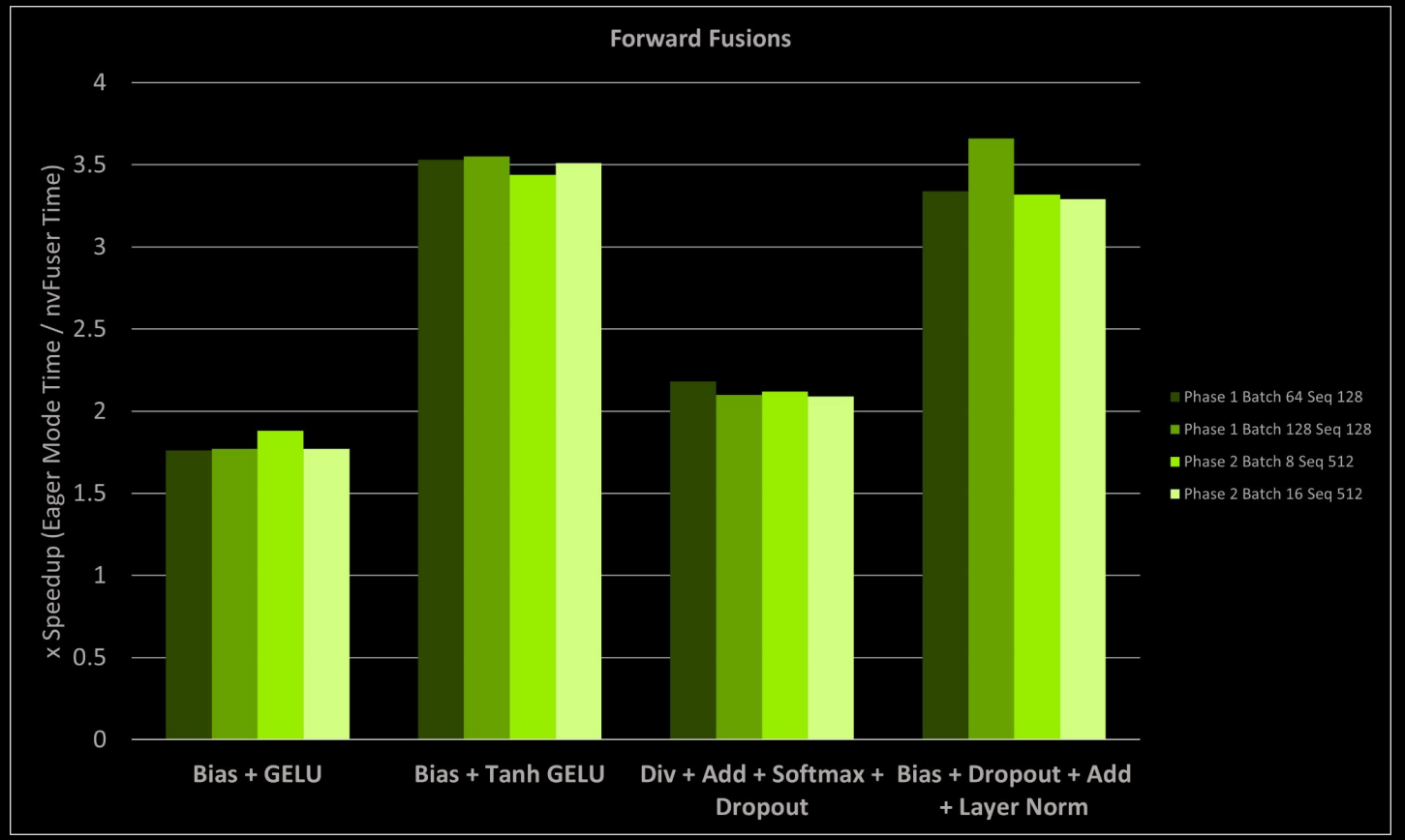
8xA100 80GB NCHW




nvFuser - “Fused”
Operator Performance

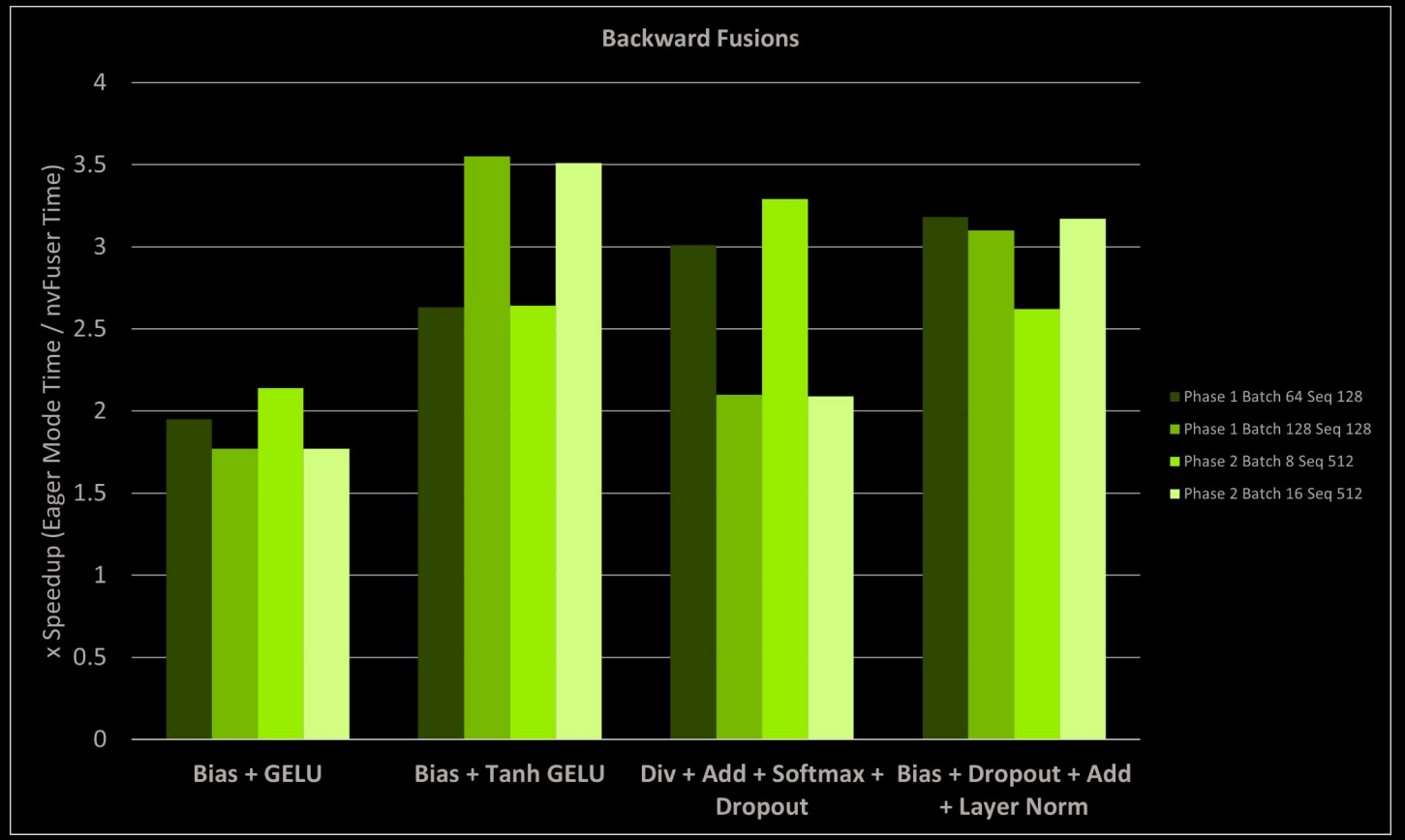
DL Examples BERT Fusions

The Important Fusions A100 FP16



DL Examples BERT Fusions

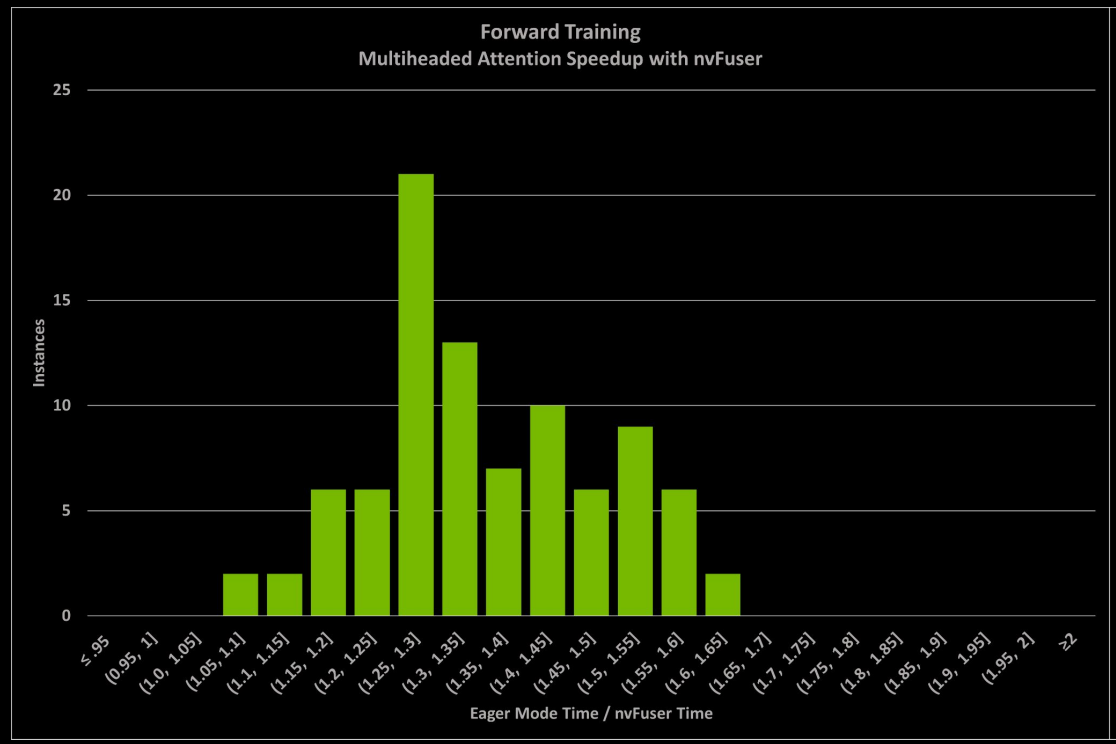
The Important Backward Fusions A100 FP16

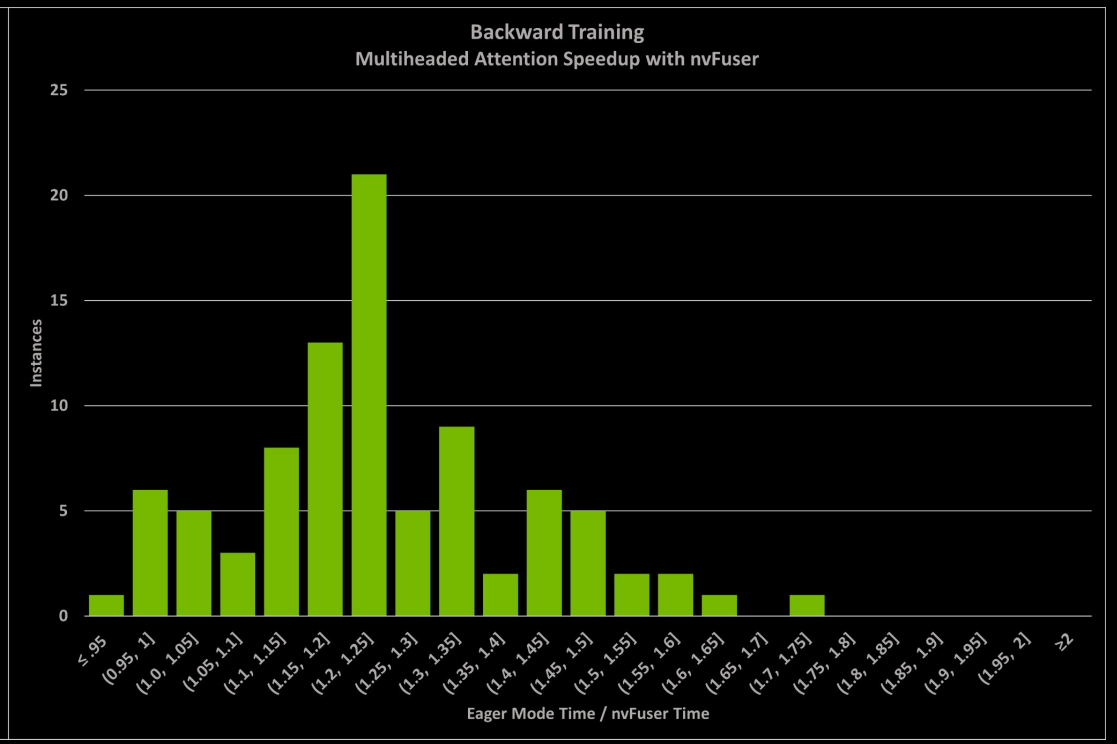


Multi-Headed Attention

xPerf Gain, product of:

batch size [10, 20, 40]
number of heads [12, 16, 24, 32, 64]
seq len [256, 512],
head size [26, 32, 64]

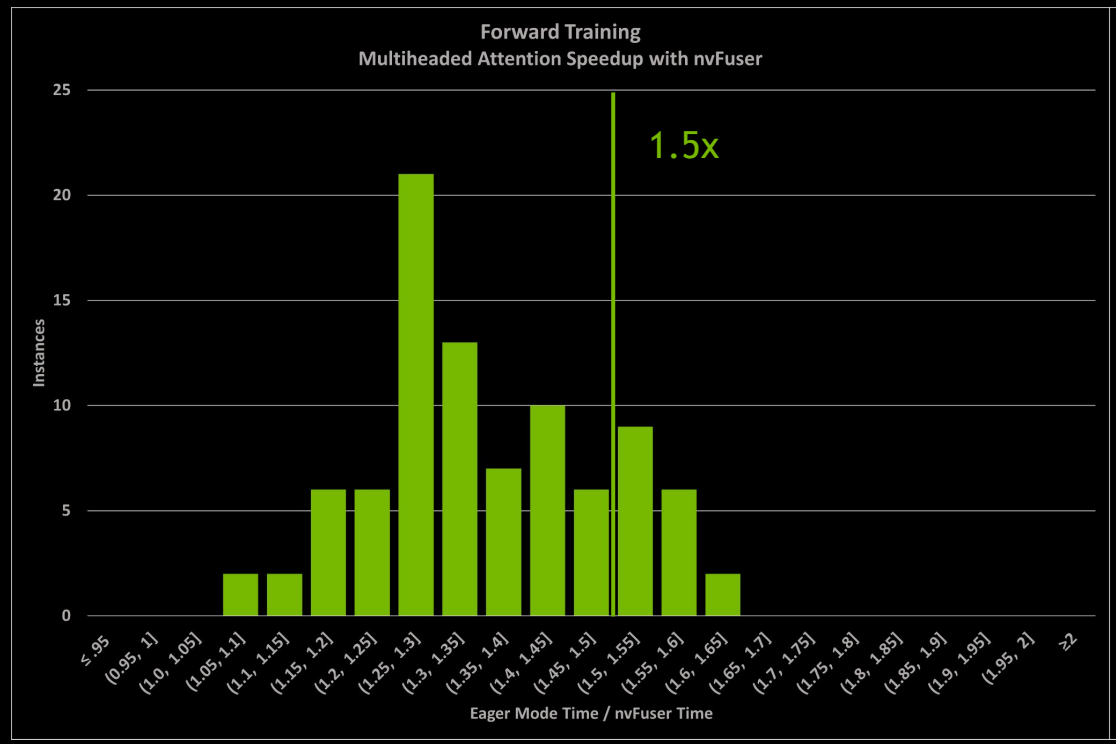


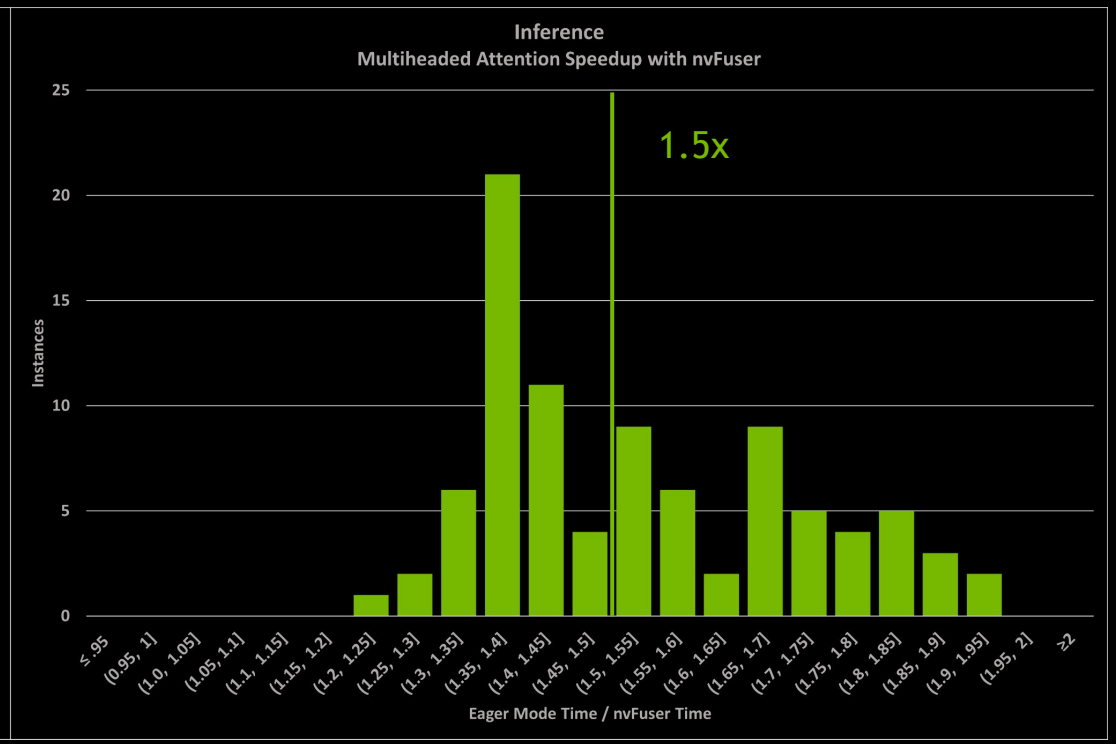


NVIDIA.

Multi-Headed Attention

What helps in Training helps in Inference
Same sizes as previous slide





NVIDIA.

Saved Tensors

Inference Perf Gains > Training Perf Gains

- TorchScript has no mechanism to decide
  - which tensors to save from forward
  - to compute backwards

- This results in more tensors being saved in DRAM than necessary

memory_efficient_fusion mode in FuncTorch is a prototype which reduces the number of tensors saved from forward and used in backwards

https://pytorch.org/functorch/nightly/notebooks/aot_autograd_optimizations.html


Composability with
Performance

Popular Operations get Fast Kernels

Traditional approach is backwards

Researchers are always experimenting with new operations

- Normalization, attention, convolutions, activations, loss functions…

Out of the box implementations in Python typically have disappointing performance

- Python defined LayerNorm can be 9x slower than the c++ composite implementation

This can severely impede research

Python Defined LayerNorm

definition

```
def layerNorm(self, x, weight, bias, eps):
    u = x.mean(-1, keepdim=True)
    s = (x - u)
    s = s * s
    s = s.mean(-1, keepdim=True)
    x = (x - u) / torch.sqrt(s + eps)
    x = weight * x + bias
    return x
```

Python Defined LayerNorm

without fusion

```
def layerNorm(self, x, weight, bias, eps):
    u = x.mean(-1, keepdim=True)
    s = (x - u)
    s = s * s
    s = s.mean(-1, keepdim=True)
    x = (x - u) / torch.sqrt(s + eps)
    x = weight * x + bias
    return x
```

Intermediate

Saved For Backwards

12 intermediate tensors saved

6 tensors saved for backwards (only 3 additional on intermediates)

Python Defined LayerNorm

with fusion

```
def layerNorm(self, x, weight, bias, eps):
    u = x.mean(-1, keepdim=True)
    s = (x - u)
    s = s * s
    s = s.mean(-1, keepdim=True)
    x = (x - u) / torch.sqrt(s + eps)
    x = weight * x + bias
    return x
```

Saved For Backwards

~~12 intermediate tensors saved~~

6 tensors saved for backwards

Python Defined LayerNorm

with Forward-Backward tensor optimization

```
def layerNorm(self, x, weight, bias, eps):
    u = x.mean(-1, keepdim=True)
    s = (x - u)
    s = s * s
    s = s.mean(-1, keepdim=True)
    x = (x - u) / torch.sqrt(s + eps)
    x = weight * x + bias
    return x
```

~~12 intermediate tensors saved~~

~~6 tensors saved for backwards~~

Popular Operations get Fast Kernels

Traditional approach is backwards

Fusion with forward to backward optimization enables research

Out of box performance can improve greatly, encouraging the development of novel operations

Example LayerNorm -> RMSNorm:

A100 80GB

FP16

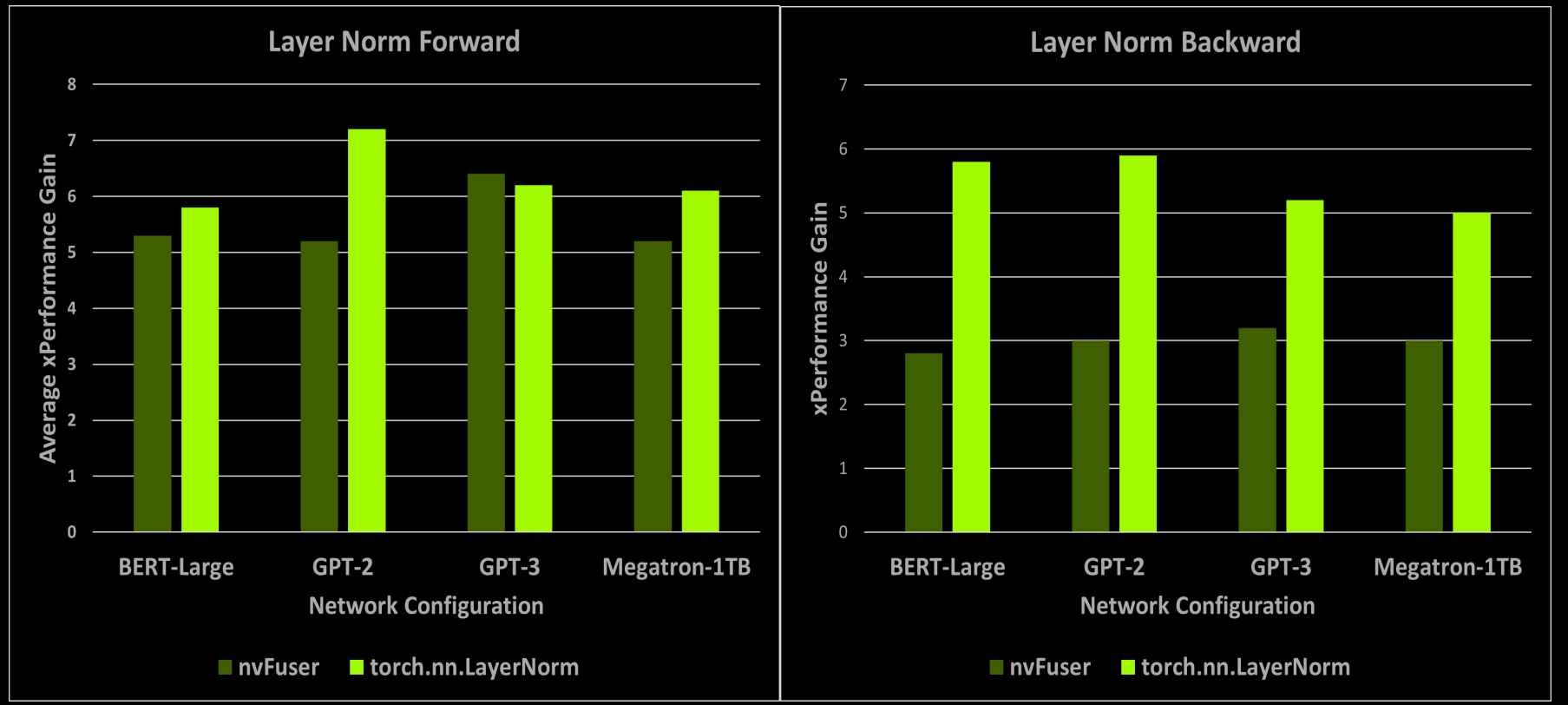
Using memory_efficient_fusion mode in FuncTorch

4 Network configurations:

- BERT-Large [(1-256), 128, 1024]
- GPT-3 [(1-16), 2048, 16384]
- GPT-2[(1-256), 1024, 2048]
- Megatron-1TB [(1-8), 2048, 25600]

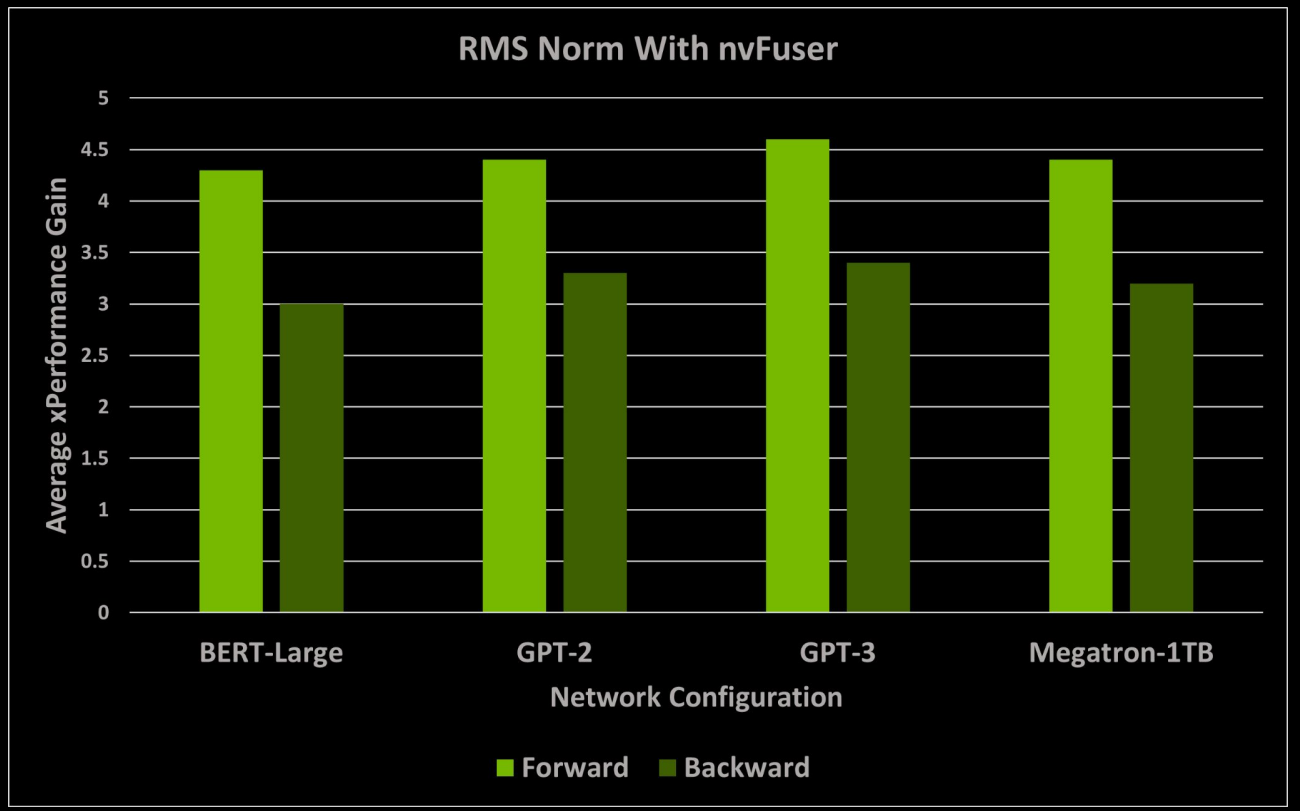
LayerNorm

If there's a composite operation, still best



RMSNorm

When there's no other option, nvFuser can provide benefit



Conclusion

nvFuser

A Halide-like system reimagined for GPUs and dynamic shapes

Allows us to program our expertise into PyTorch

Can generate fast operations, especially Normalizations

Climbing the optimization curve to get to Matrix Multiplications and beyond

Supports novel operations

Performance will continue to improve

Operation support will continue to grow

What are we working towards?

What's the DL compiler promise?

DL Compilers promise to improve network performance with minimal changes

- However, they follow the traditional support model operations are optimized independently

nvFuser is designed for the future

- composable PyTorch
- software automation
- better performance today and on new generations

User Interface

Many, still in flux, nvFuser is everywhere

TorchScript

https://pytorch.org/docs/stable/jit.html

FuncTorch

https://pytorch.org/functorch/nightly/notebooks/aot_autograd_optimizations.html
https://dev-discuss.pytorch.org/t/min-cut-optimal-recomputation-i-e-activation-checkpointing-with-aotautograd/467

Lazy Tensors

https://dev-discuss.pytorch.org/t/lazy-tensor-core/232

TorchDynamo

https://dev-discuss.pytorch.org/t/torchdynamo-update-4-lazytensor-nvfuser-experiments/496


NVIDIA