INDIA-AI IMPACT SUMMIT 2026

MINISTRY OF ELECTRONICS & INFORMATION TECHNOLOGY GOVERNMENT OF INDIA

INDIAai

Workshop on Systems for LLMs

Systems for Large Language Models (LLMs) Workshop with Hands-On
Exploring the Infrastructure Behind Intelligent Systems

Sessions Overview:
LLM building blocks
GPUs for LLM
Inference serving with vLLM
Distributed Training

Tools: Hugging Face, PyTorch

Details
Date: 6th,7th September
Time: 9:30 am-5pm
Register @
https://forms.gle/a8rQQ835jJG42i3eA

Supported by: IBM-IITB Academic Research Partnership

IBM

IIT Bombay

GPUs for LLMs

Outline

1. What does a DL Framework like Pytorch do?
2. How does Inference look like?
    a. (Defer this to a full session tomorrow)
3. What does a basic train loop look like?
4. How is the GPU used?

What does a DL Framework get you?

Different approaches to the framework

TensorFlow/JAX, Google

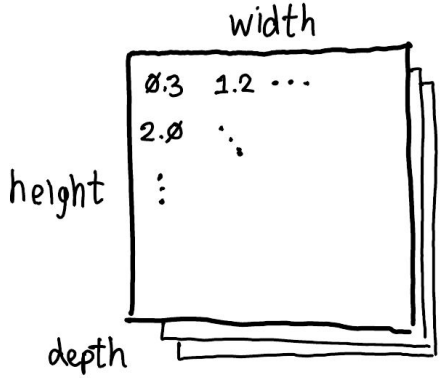
Pytorch, Meta

Philosophical difference: how to specify your task and how easy is the abstraction

Tensors

```
>>> import torch
>>> torch.ones(2, 3)
tensor([[1., 1., 1.],
        [1., 1., 1.]])
>>> torch.rand(3, 5)
tensor([[0.8594, 0.0313, 0.8472, 0.6617, 0.8709],
        [0.5300, 0.9456, 0.6548, 0.7148, 0.1227],
        [0.2620, 0.9029, 0.3826, 0.3281, 0.3003]])
```

Tensor



sizes	(D, H, W)
strides	(H*W, W, 1)
dtype	float
device	cuda:0
layout	strided

contiguous

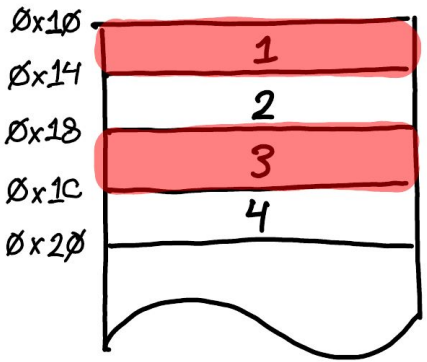
Tensor: Strided Representation

logical

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

tensor[:, 0]

physical



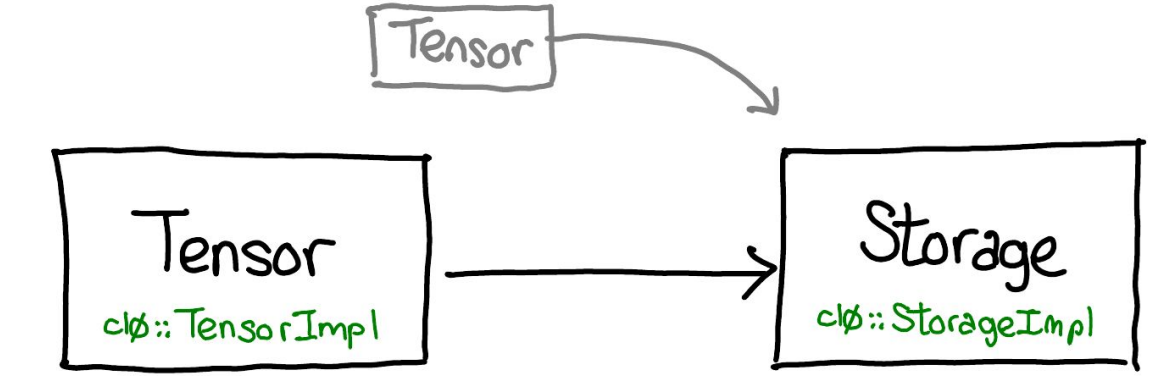
sizes [2]
strides [2]

stride ≠ 1 means we skip elements

Since tensors can be big, we need efficient ways to have views of tensors

https://ezyang.github.io/stride-visualizer

Tensor: Strided Representation



the logical view
sizes, strides, offset

the actual physical data
physical size, dtype

there is always a Tensor+Storage, even for "simple" cases

Storage and Tensor

```
>>> a = torch.ones(2,3)
tensor([[1., 1., 1.],
        [1., 1., 1.]])
>>> b = a.t()
>>> a.shape
>>> b.shape
>>> a.data_ptr()
>>> b.data_ptr()
>>> a.untyped_storage() == b.untyped_storage()
>>> b = a + 1
>>> b.data_ptr()
>>> a.untyped_storage()
```

Operations on Tensors

```
>>> import torch
>>> a = torch.ones(2,3)
tensor([[1., 1., 1.],
        [1., 1., 1.]])
>>> b = torch.ones(3,2) * 2
>>> b
tensor([[2., 2.],
        [2., 2.],
        [2., 2.]])
>>> a @ b
tensor([[6., 6.],
        [6., 6.]])
```

Scalar Multiplication

Matrix Multiplication

Some example Operators (ops)

Creating new tensors: `ones, empty, random, linspace, diag` etc

Point wise operation: `add, mult, divide` etc

View operations: `cat, unsqueeze, transpose` etc

Math: `cos, sin, arctan` etc.

Statistics, Logic, etc etc

By category:
http://blog.ezyang.com/2020/05/a-brief-taxonomy-of-pytorch-operators-by-shape-behavior/

Shape

```
>>> a = torch.arange(8)
>>> a
tensor([0, 1, 2, 3, 4, 5, 6, 7])
>>> a.view(2,4)
tensor([[0, 1, 2, 3],
        [4, 5, 6, 7]])
>>> a.view(4,2)
tensor([[0, 1],
        [2, 3],
        [4, 5],
        [6, 7]])
```

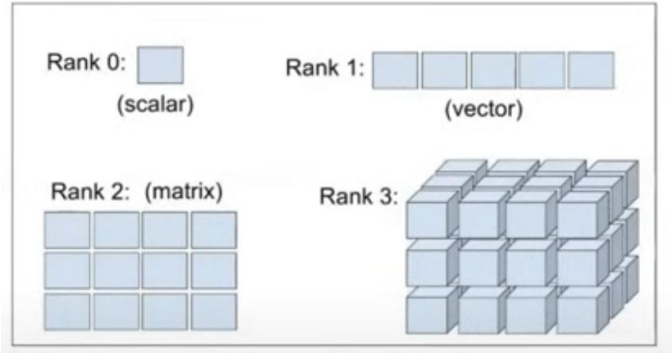
Also, look at stride of both

Unsqueeze

```
>>> b = a.view(2,4)
>>> b
tensor([[0, 1, 2, 3],
        [4, 5, 6, 7]])
>>> b.unsqueeze(0)
tensor([[[0, 1, 2, 3],
         [4, 5, 6, 7]]])
>>> b.unsqueeze(0).shape
torch.Size([1, 2, 4])
>>> b.unsqueeze(1)
tensor([[[0, 1, 2, 3]],

        [[4, 5, 6, 7]]])
```

```
>>> b.unsqueeze(1).shape
torch.Size([2, 1, 4])
>>> b.unsqueeze(2)
tensor([[[0],
         [1],
         [2],
         [3]],

        [[4],
         [5],
         [6],
         [7]]])
>>> b.unsqueeze(2).shape
torch.Size([2, 4, 1])
```

Why? The main dimensions in LLM inputs



(BS, seqlen, embed dim)

Embedding Size

Each token is expanded into embedding space
Model-dependent: like 768 (for gpt2)

BS=4

The cat climbed…
The house was painted…
The capital of India is …
The people of Mumbai …

Seqlen=5

The cat climbed the wall.

>>> tok.encode("The cat climbed the wall")
[464, 3797, 19952, 262, 3355]

Max of the batch

Some real numbers

4

64

128

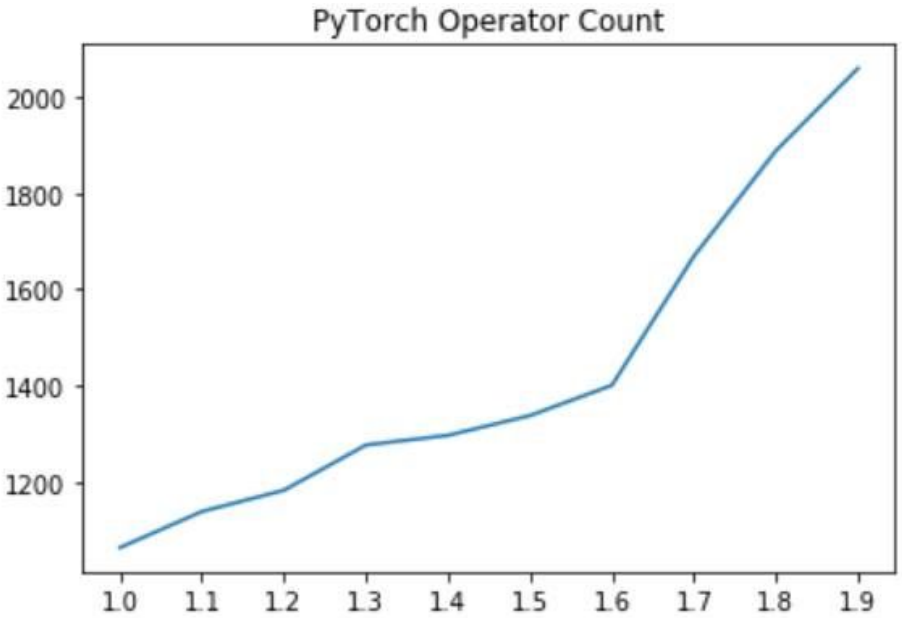
4096/8192

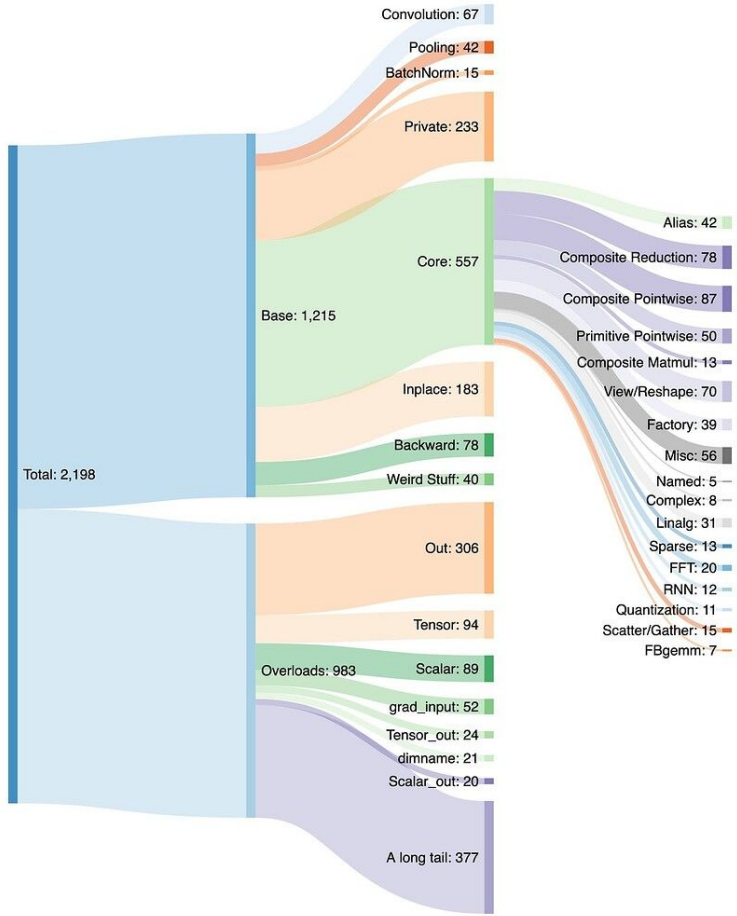
In Long Context: this can go to 128K and upwards

Computation operators

```
>>> a = torch.arange(10)
>>> a
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> torch.cos(a)
tensor([ 1.0000,  0.5403, -0.4161, -0.9900, -0.6536,  0.2837,
0.9602,  0.7539,
        -0.1455, -0.9111])
>>> b = a.view(2, 5)
>>> b
tensor([[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]])
>>> torch.cos(b)
tensor([[ 1.0000,  0.5403, -0.4161, -0.9900, -0.6536],
        [ 0.2837,  0.9602,  0.7539, -0.1455, -0.9111]])
```

How many built-in operators do you think there are in Pytorch?



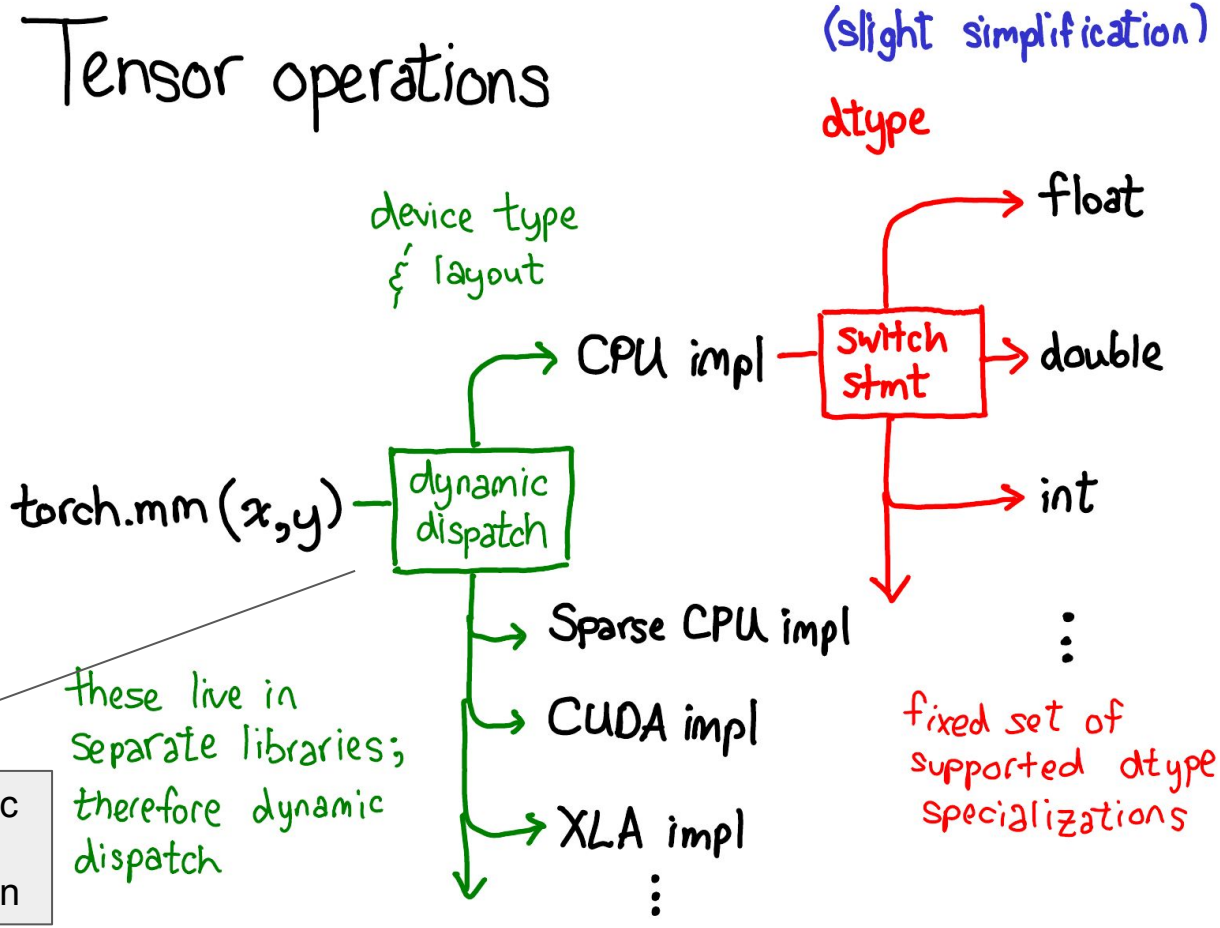


https://dev-discuss.pytorch.org/t/where-do-the-2000-pytorch-operators-come-from-more-than-you-wanted-to-know/373

But, having 2000 ops is not the hard part…

Tensor operations

(slight simplification)



Per device, dtype specific implementation

Device specific op implementation

Softmax dtype example

```
>>> a = torch.arange(10)
>>> a
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.softmax(0)
Traceback (most recent call last):
  File "<python-input-132>", line 1, in <module>
    a.softmax(0)
    ~~~~~~~~~^^^
RuntimeError: "softmax_lastdim_kernel_impl" not implemented
for 'Long'

>>> a = torch.arange(10).to(torch.float32)
>>> a.softmax(0)
tensor([7.8013e-05, 2.1206e-04, 5.7645e-04, 1.5669e-03,
4.2594e-03, 1.1578e-02,
        3.1473e-02, 8.5552e-02, 2.3255e-01, 6.3215e-01])
```

Tensor and Op on gpu

```
>>> import torch
>>> a = torch.ones(2,3).to(“cuda:0”)
>>> b = torch.ones(2, 3).to(“cuda:0”)
>>> a + b
tensor([[2., 2., 2.],
        [2., 2., 2.]], device=’cuda:0’)
```

A Tensor on a GPU device

User-interface does not change

Op running on GPU

```
__global__ void add(float* out, float* in1, float*
in2, int numel) {
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if(id < numel)
      out[id] = in1[id] + in2[id];
  }
```

Fp32 and bf16 will have different kernels…

Example of Op

https://docs.pytorch.org/docs/stable/generated/torch.nn.GLU.html

CPU version:

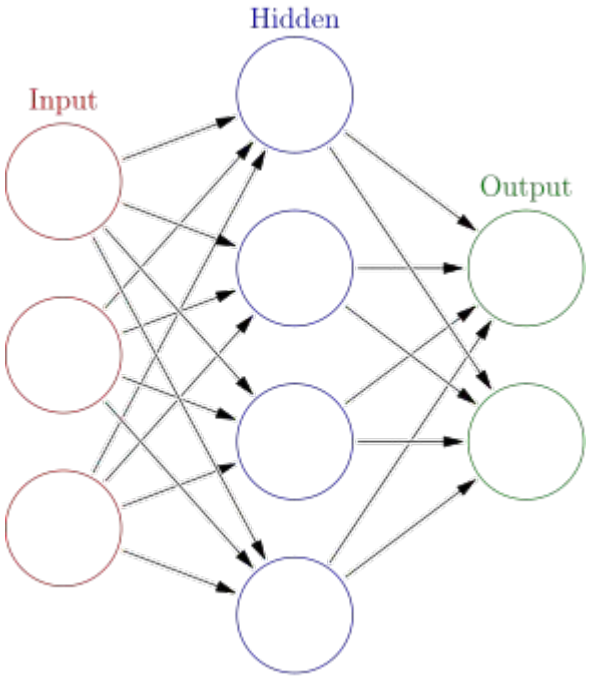
https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/GatedLinearUnit.cpp

CUDA version:

https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/cuda/ActivationGluKernel.cu

Having 2000 ops with Multiple devices, datatypes is not the hard part

Backprop!



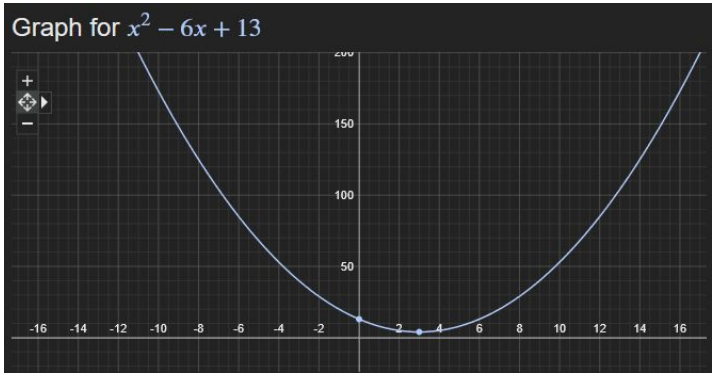
The cat is climbing up the wall.

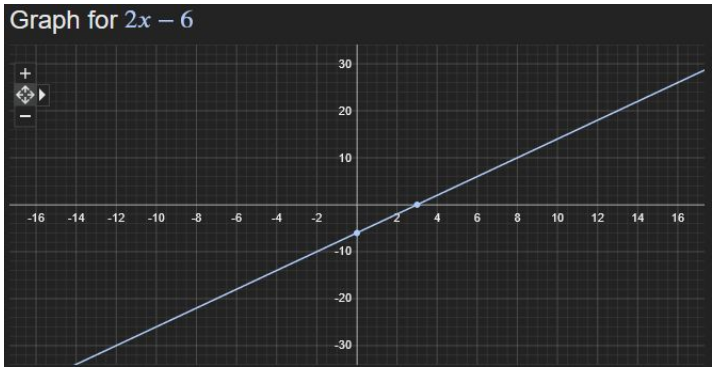
sitting

flying

green

Tangent into Derivatives





Gradients

If you have a function $w = f(x, y, z)$, a g
able at a time, leaving the other variables

$$\nabla f(x, y, z) = \begin{bmatrix} \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{bmatrix}$$

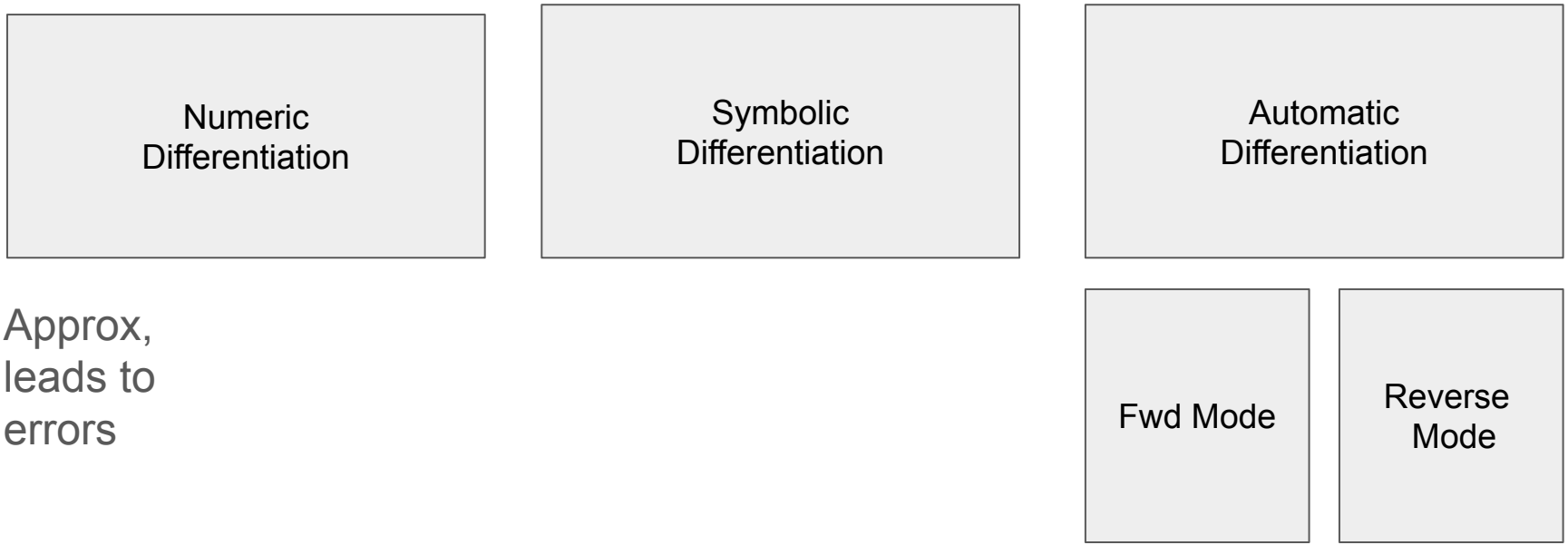
Here is the Jacobian for $v, w = f(x, y, z)$:

$$\mathbb{J} = \begin{bmatrix} \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{bmatrix}$$

https://blog.demofox.org/2025/08/16/derivatives-gradients-jacobians-and-hessians-oh-my/

3 ways of Differentiation

Build a graph of dependencies



Approx, leads to errors

Simpler, but O(input)

O(output), needs a backward pass

https://jingnanshi.com/blog/autodiff.html

Forward

```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
loss = next_h.sum()
loss.backward()
```

W_h, W_x, x, prev_h

requires grad

Automatically created backward pass

```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h2 = next_h.tanh()
loss = next_h2.sum()
```

W_h, W_x — requires grad

x, prev_h

swap inputs and outputs!

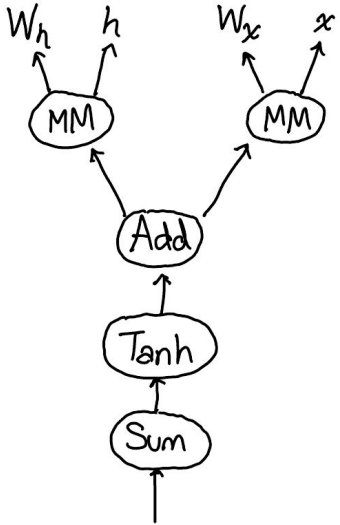
```
grad_loss = torch.tensor(1, dtype=loss.dtype())
grad_next_h2 += grad_loss.expand(next_h2.size())
grad_next_h += tanh_backward(grad_next_h2, next_h2)
grad_i2h, grad_h2h += grad_next_h, grad_next_h
W_h.grad += mm_mat1_backward(grad_h2h, prev_h.t(), W_h, 1)
W_x.grad += mm_mat1_backward(grad_i2h, x.t(), W_x, 1)
```

original inputs may be reused

How do we get here?

All grad variables are zero to start:

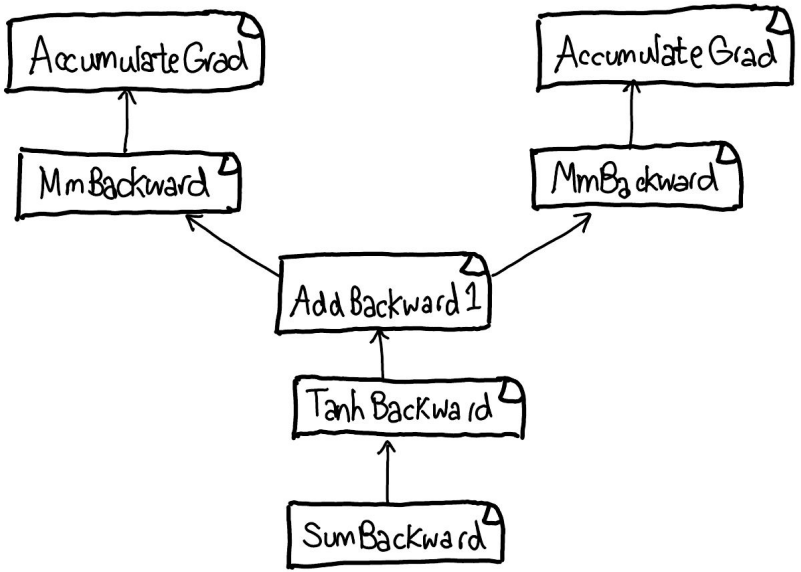
```
grad_i2h = torch.zeros_like(i2h)
```



Autograd engine

"just a parallel graph executor"

FunctionTasks:



The Autograd Function class

class **torch.autograd.Function**(**args*, ***kwargs*) [source]

Base class to create custom *autograd.Function*.

To create a custom *autograd.Function*, subclass this class and implement the `forward()` and `backward()` static methods. Then, to use your custom op in the forward pass, call the class method `apply`. Do not call `forward()` directly.

To ensure correctness and best performance, make sure you are calling the correct methods on `ctx` and validating your backward function using `torch.autograd.gradcheck()`.

https://docs.pytorch.org/docs/stable/autograd.html#function

An example Autograd function

```
>>> class Exp(Function):
>>>     @staticmethod
>>>     def forward(ctx, i):
>>>         result = i.exp()
>>>         ctx.save_for_backward(result)
>>>         return result
>>>
>>>     @staticmethod
>>>     def backward(ctx, grad_output):
>>>         result, = ctx.saved_tensors
>>>         return grad_output * result
>>>
>>> # Use it by calling the apply method:
>>> output = Exp.apply(input)
```

Actual content of forward

State needed for backward

Use state needed for backward pass

Example

```
>>> a = torch.ones(2,3, requires_grad=True)
>>> b = torch.ones(3,2, requires_grad=True)
>>> c = a @ b
>>> c
tensor([[3., 3.],
        [3., 3.]], grad_fn=<MmBackward0>)
>>> d = torch.ones(2, 2) * 4
>>> d
tensor([[4., 4.],
        [4., 4.]])
```

```
>>> lf = torch.nn.L1Loss()
>>> l = lf(c, d)
>>> l
tensor(1., grad_fn=<MeanBackward0>)
>>> l.backward()
>>> a.grad
tensor([[-0.5000, -0.5000, -0.5000],
        [-0.5000, -0.5000, -0.5000]])
>>> b.grad
tensor([[-0.5000, -0.5000],
        [-0.5000, -0.5000],
        [-0.5000, -0.5000]])
```

Having 2000 ops with Multiple devices, datatypes, and backward is the hard part

What are the supported devices?

CPU

GPU:

- Nvidia via CUDA
- Intel
- AMD (ROCM)

Accelerators:

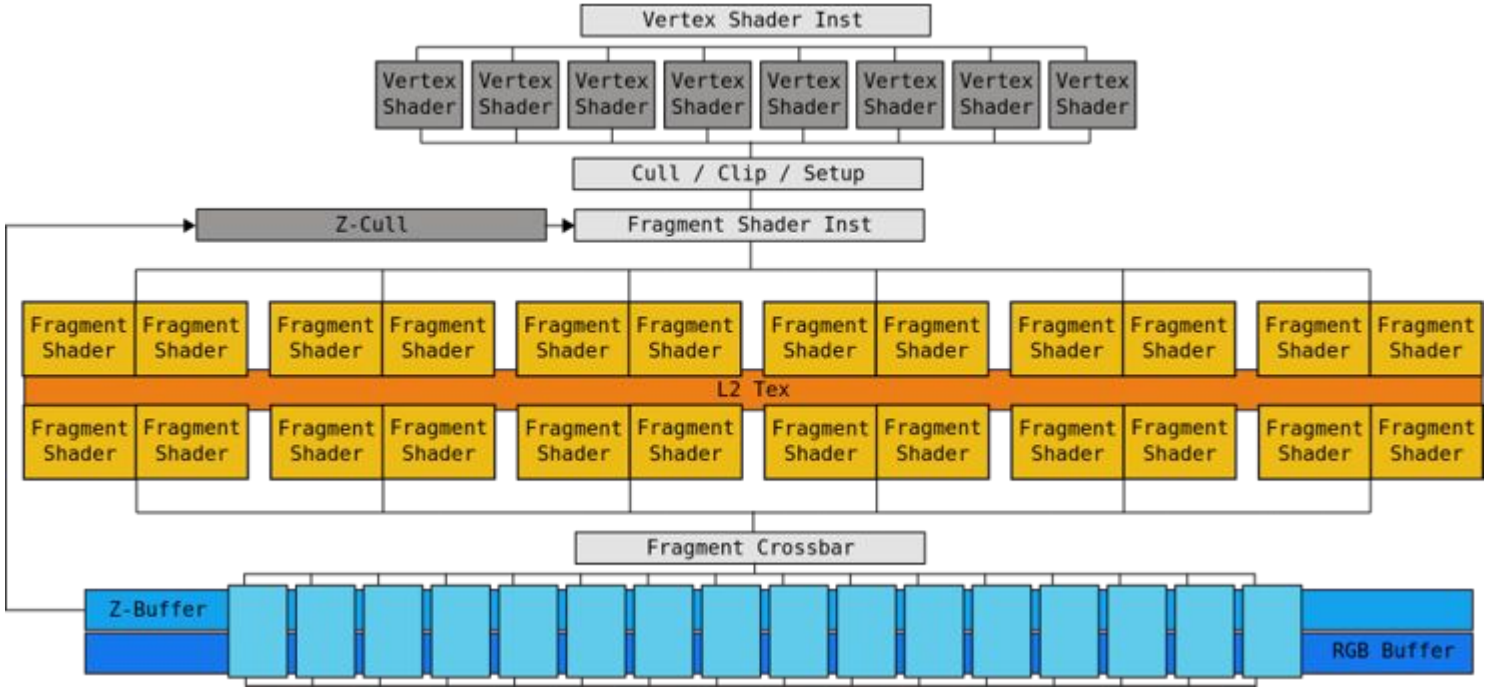
- TPU via XLA

Others:

- Mac M series, Apple Silicon

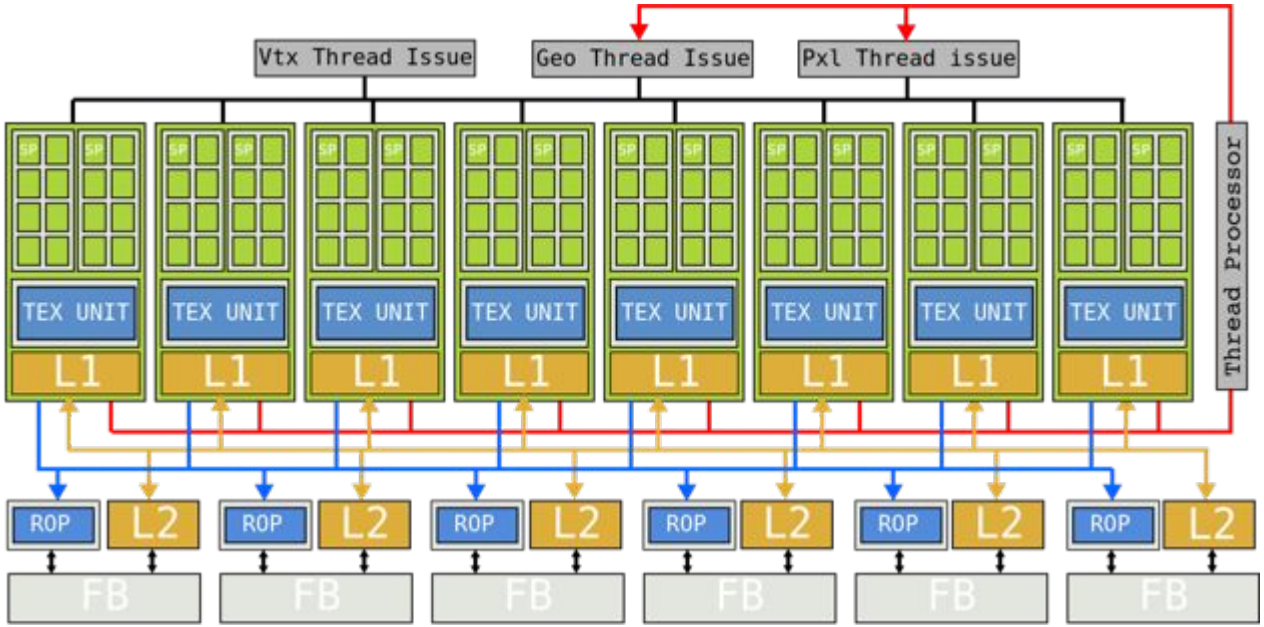
A detour into GPUs

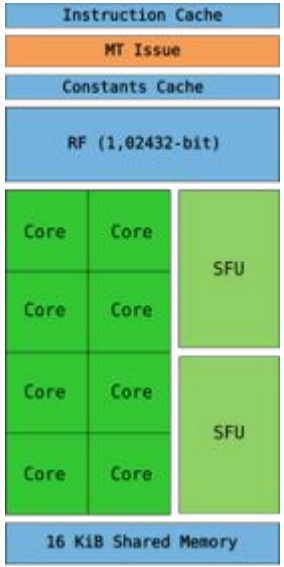
GeForce 7900 GTX (G71 die) - circa 2006



https://fabiensanglard.net/cuda/index.html

GeForce 8800 GTX (G80 die) - circa 2007/8 (Tesla)





Compute Unified Device Architecture

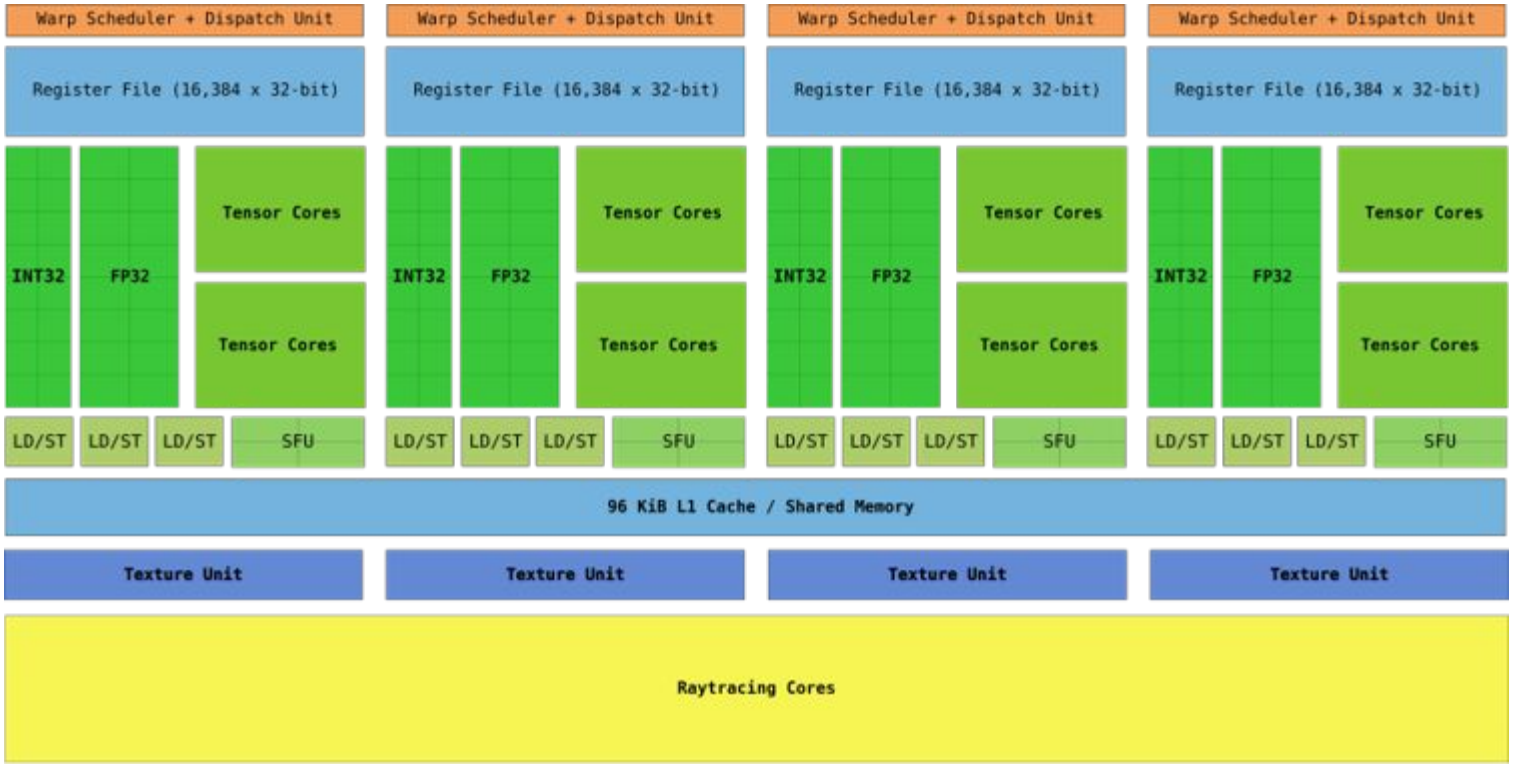
https://fabiensanglard.net/cuda/index.html

Evolution of Nvidia GPU cards

Tesla	~2008	~90nm
Fermi	~2010	~40nm
Kepler	~2012	~28nm
Maxwell	~2014	~28nm
Pascal	~2016	~16nm
Turing	~2018	~12nm
Ampere	~2020	~7nm
Hopper	~2022	~4nm
Blackwell	~2024	~4nm

Intel 10 nm ~~ TSMC 7 nm

The new TensorCore (introduced in Volta ~2017)



Turing (2018)

Tensor Cores

Specialized GEMM (Generalized Matrix Multiplication) unit

Available from CUDA 9.0

Needs a different API to program…

See examples:

https://github.com/leimao/CUDA-GEMM-Optimization/tree/main

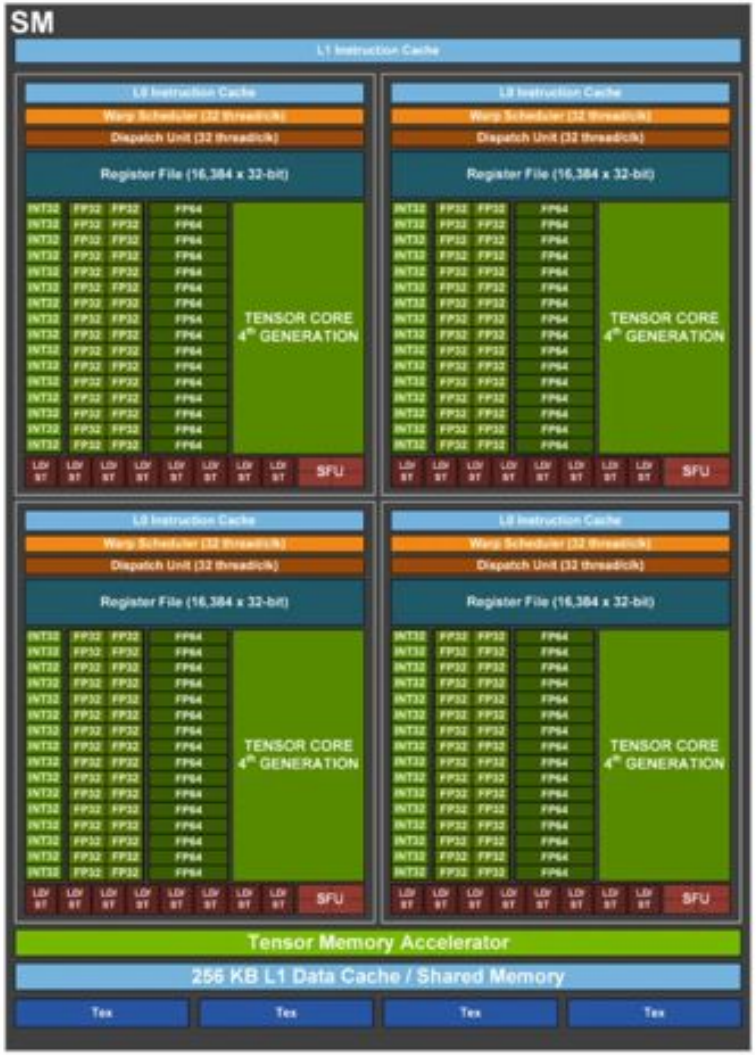
2026 lineup

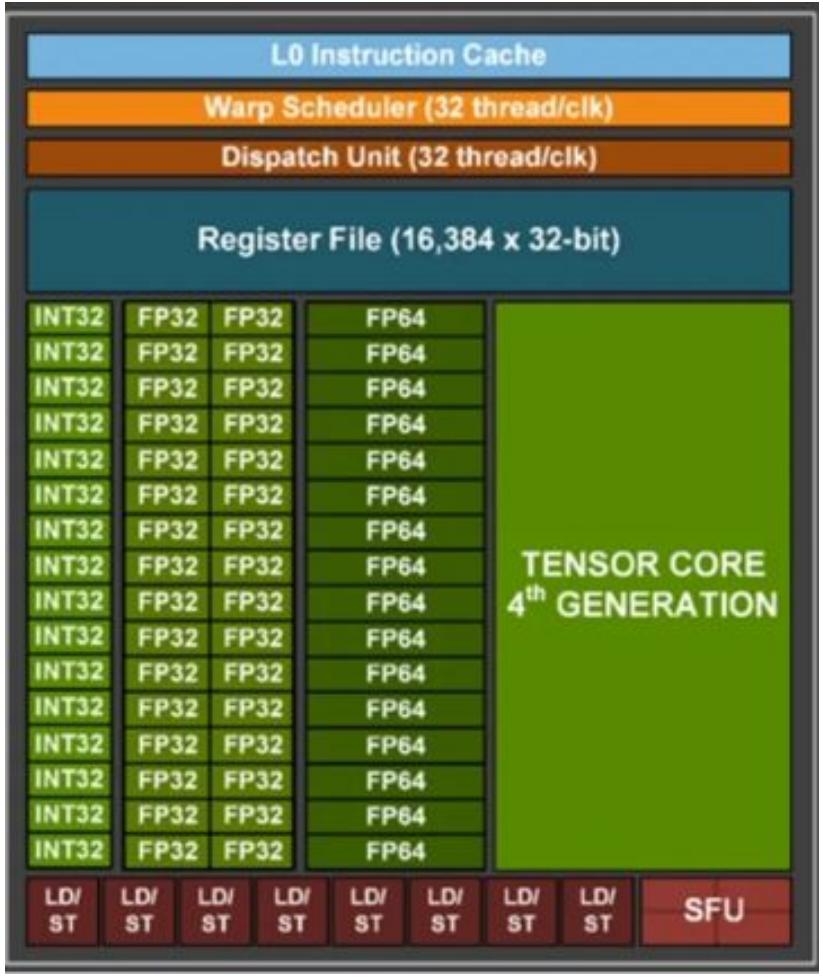
TSMC 2nm

Intel 20A (cancelled), 18A

H100


PCI Express 5.0 Host Interface
GigaThread Engine with MIG Control
GPC
L2 Cache
L2 Cache
Memory Controller
HBM3
High-Speed Hub
NVLink

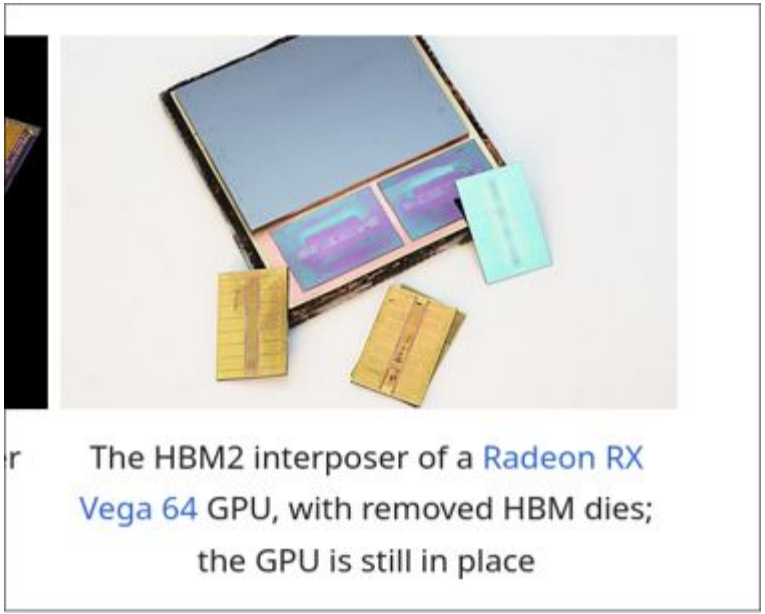

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
4th GENERATION
LD/ ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex


L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
4th GENERATION
LD/ ST
SFU

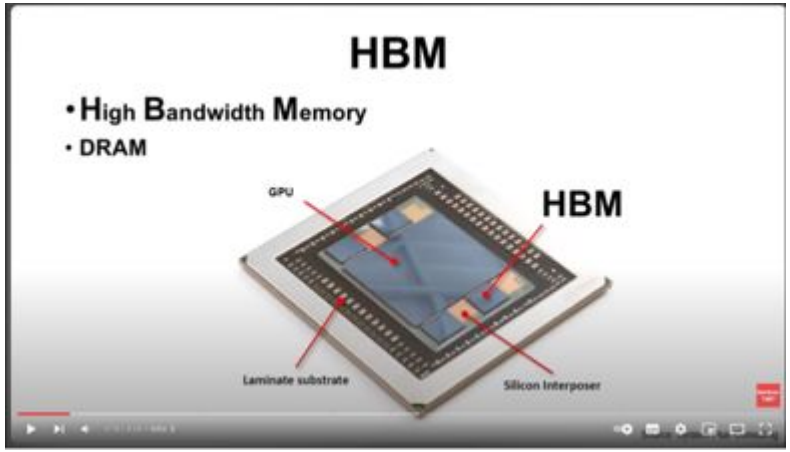

NVIDIA Tesla A100 80GB


A100 80G PCie vs H100 80G PCIe
3:43 / 4:27

High Bandwidth Memory



r The HBM2 interposer of a Radeon RX Vega 64 GPU, with removed HBM dies; the GPU is still in place

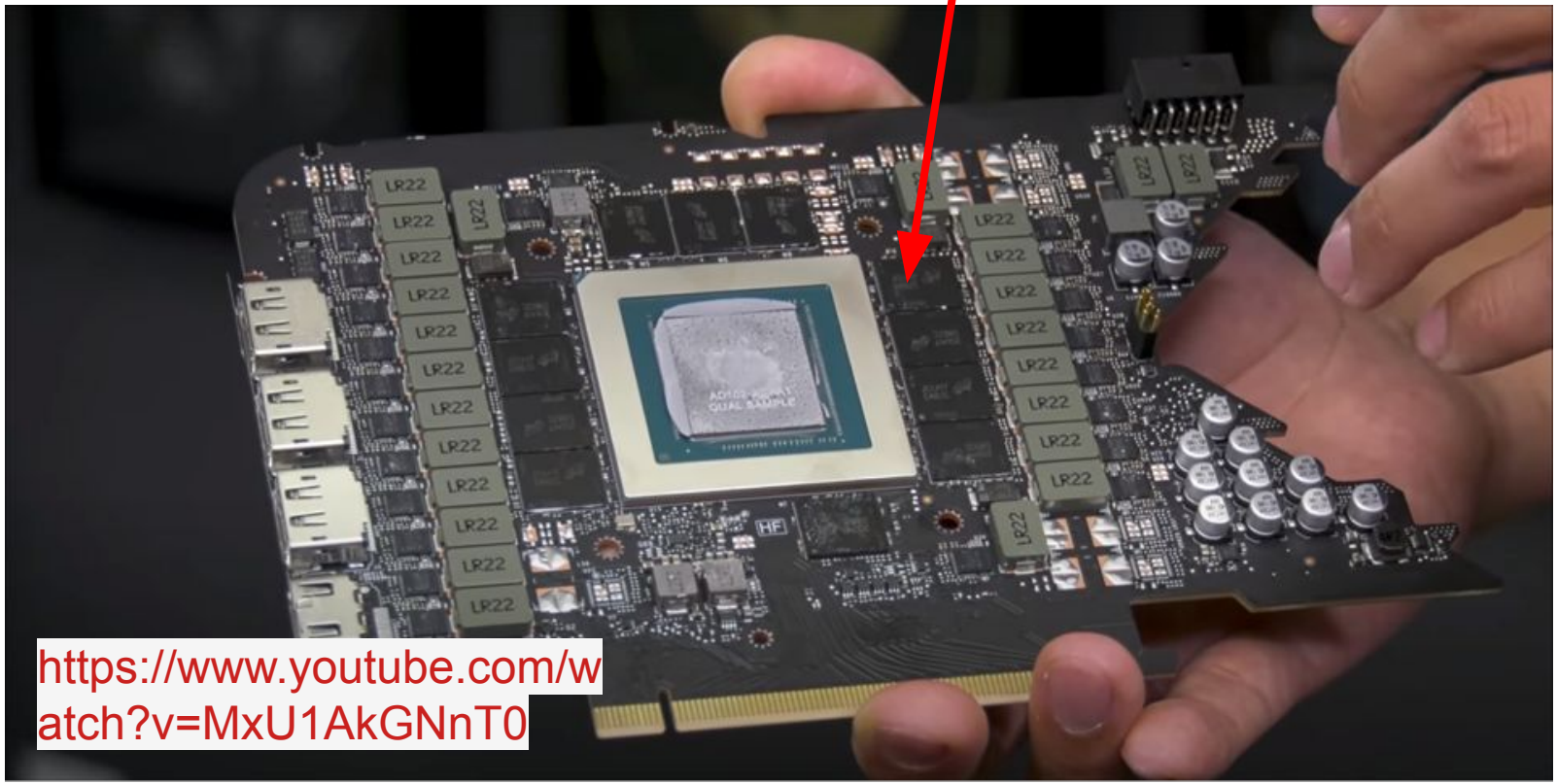


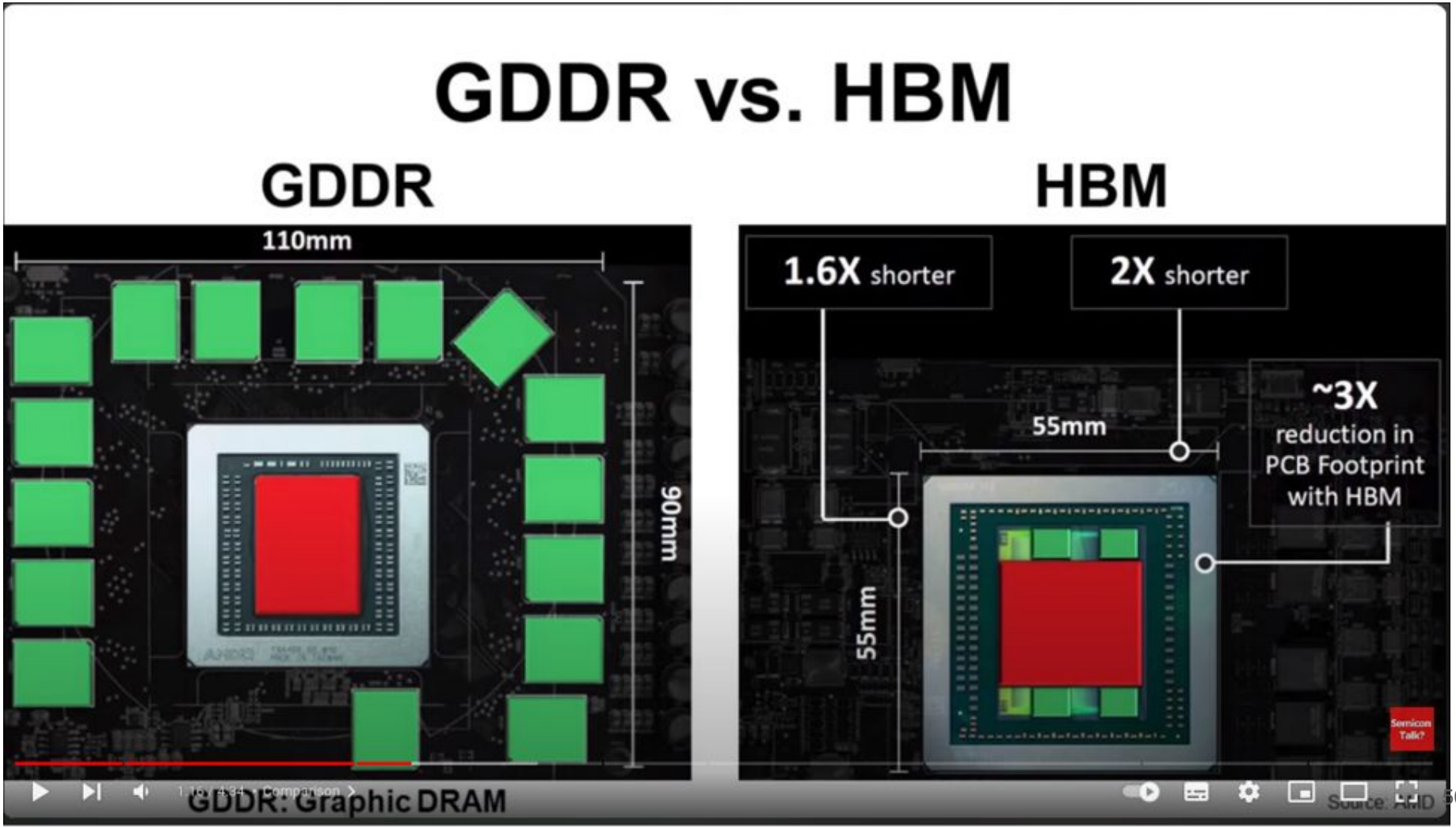
https://www.youtube.com/watch?v=KyKwiziPmD4

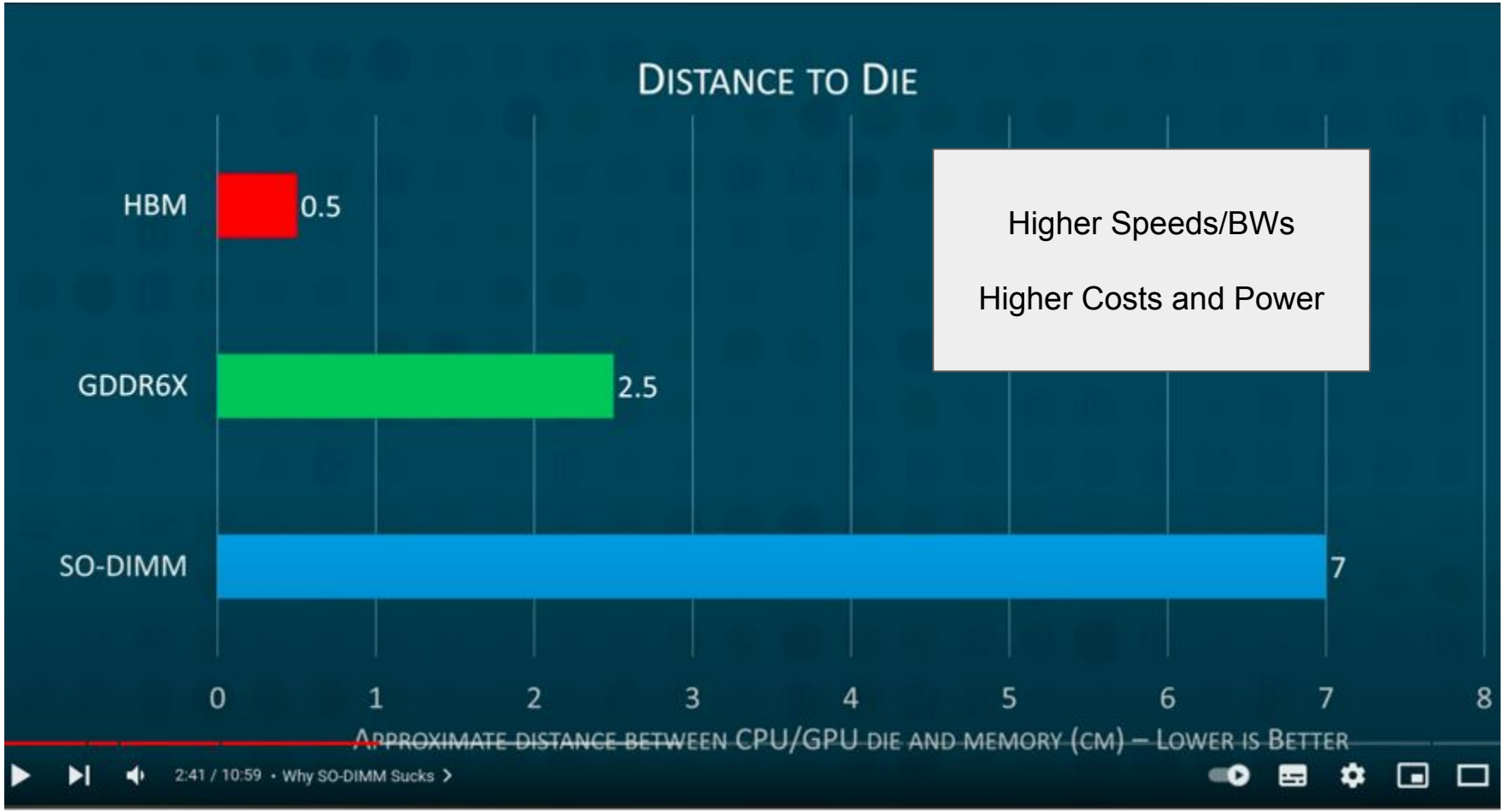
HBM3 supplied by SKHynix for Nvidia (esp Hopper)

RTX 4090 with the AD102 (same as L40)

GDDR6 memory (48 GB)

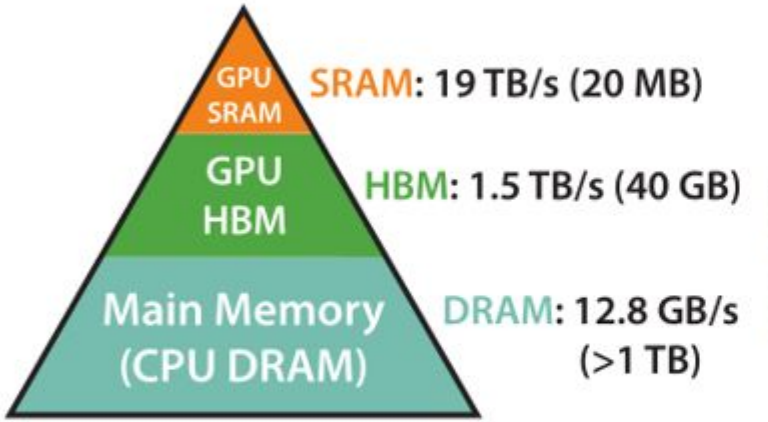



GDDR vs. HBM
GDDR
HBM
110mm
90mm
1.6X shorter
2X shorter
55mm
~3X
reduction in
PCB Footprint
with HBM
55mm
GDDR: Graphic DRAM
Source: AMD



https://www.youtube.com/watch?v=WXp4g-KzdAI

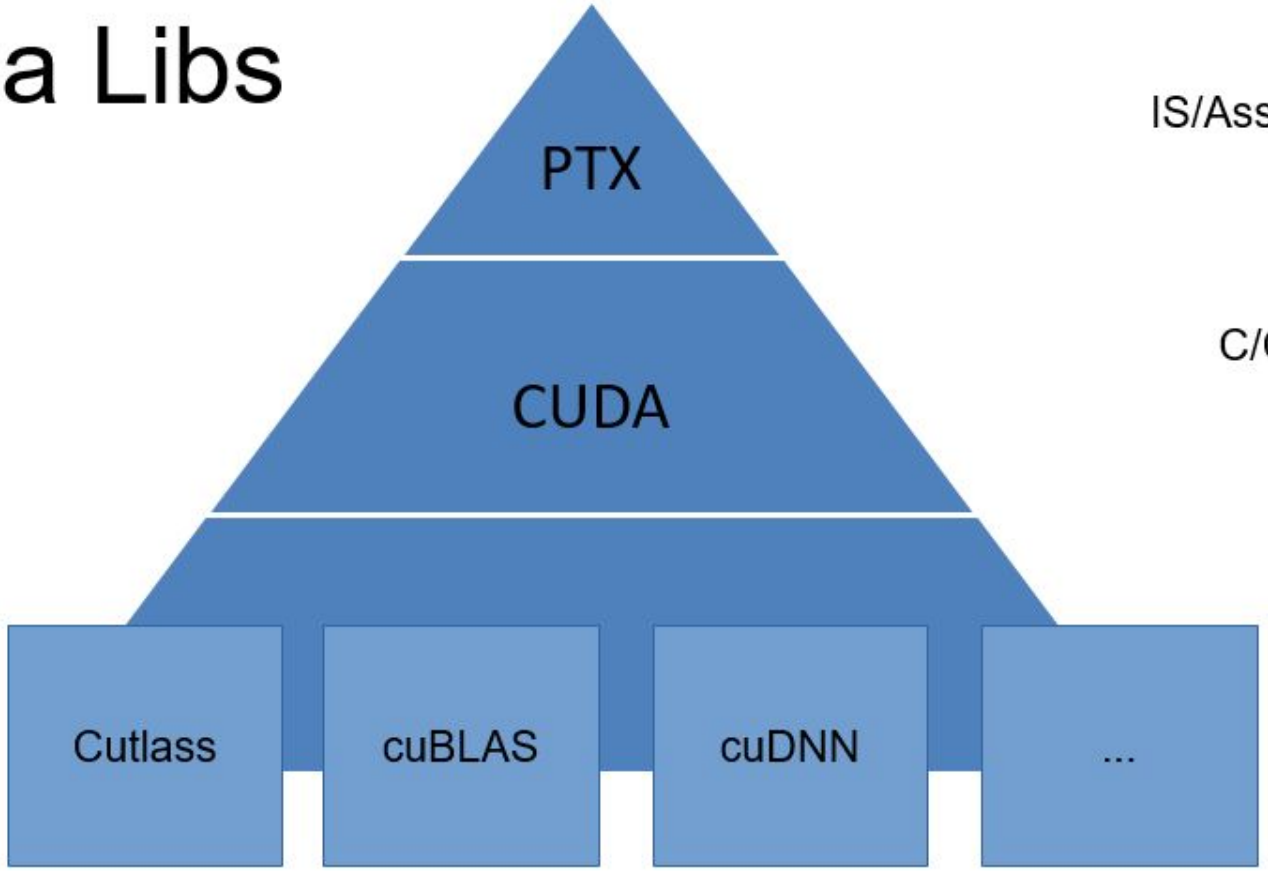
Memory hierarchy



Memory Hierarchy with Bandwidth & Memory Size

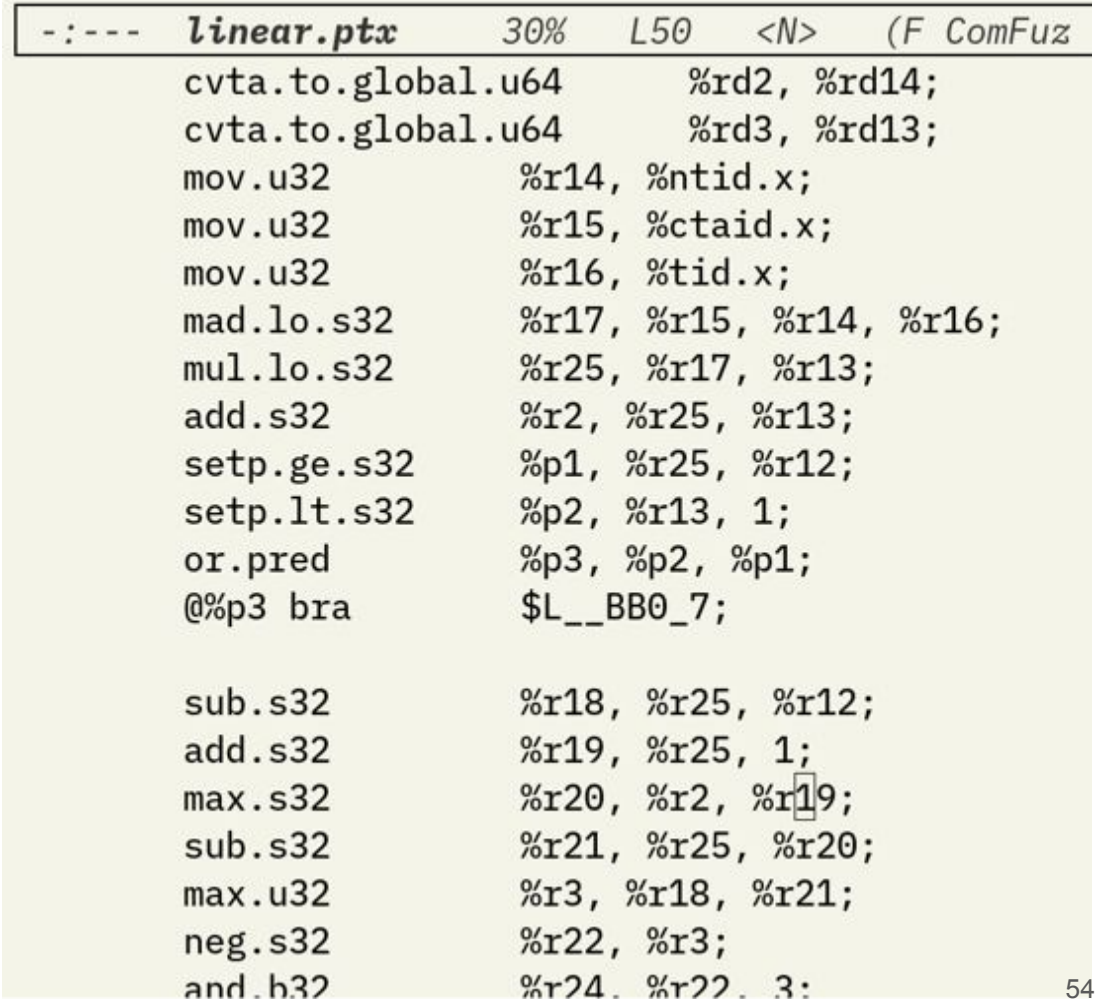
From the Flash Attention Paper

Nvidia Libs


PTX
IS/Assembly
CUDA
C/C++
Cutlass
cuBLAS
cuDNN
...
Libs (closed-source)
Templates for Linear Algebra Subroutines
Basic Linear Algebra Subprograms

PTX: Parallel Thread Execution

nvcc -ptx main.cu



https://docs.nvidia.com/cuda/parallel-thread-execution

Examining Profiles

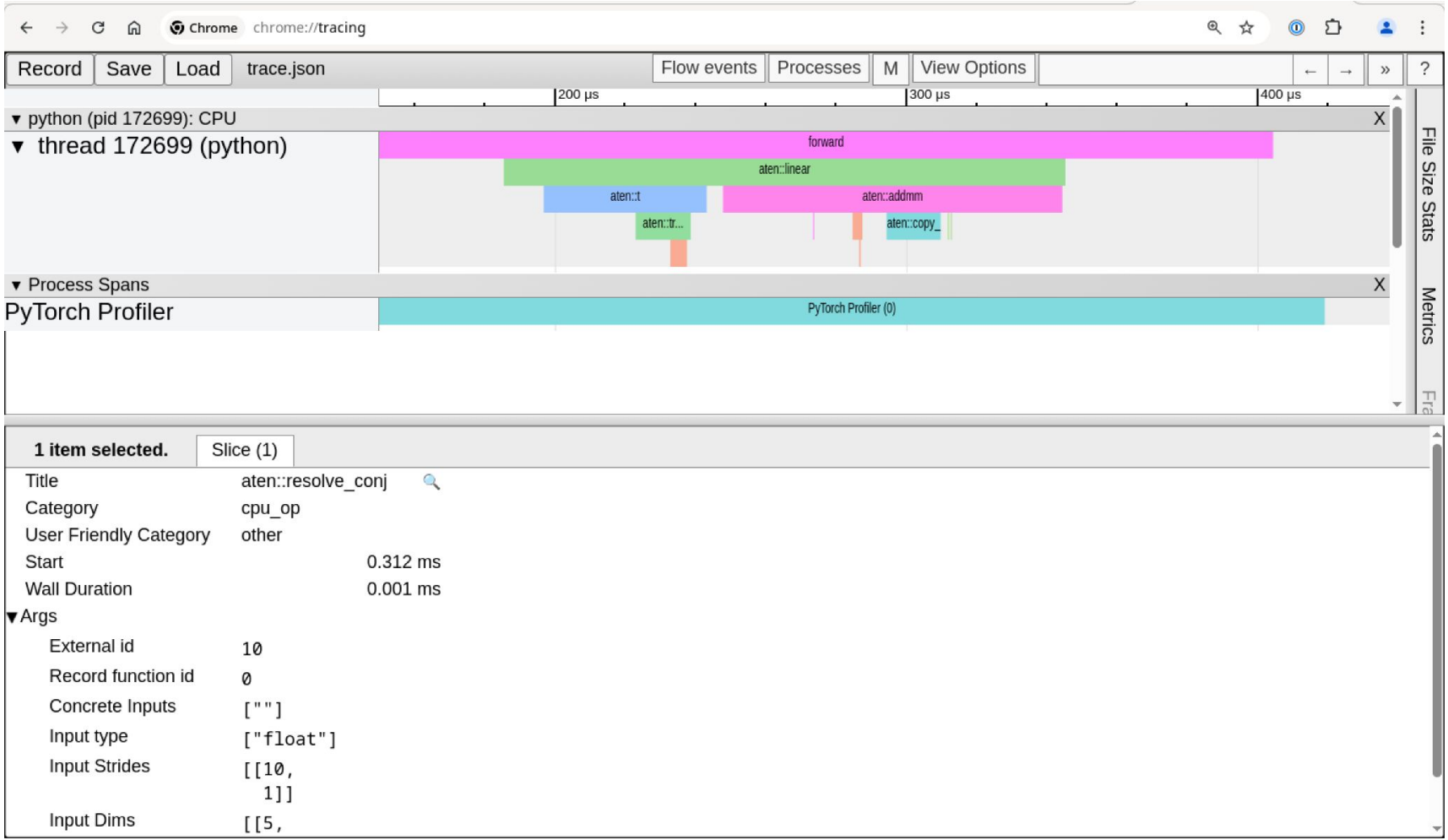
Running on Google Collab

Use the in-built files module to download files:

```

```
from google.colab import files
files.download('example.txt')
```
```

Simple pytorch profiling

```
import torch
from torch.profiler import profile, ProfilerActivity,
record_function

model = torch.nn.Linear(20, 10)

i = torch.randn(5, 20)

with profile(activities=[ProfilerActivity.CPU],
             record_shapes=True) as prof:
    with record_function("forward"):
        o = model(i)

print(o.shape)
prof.export_chrome_trace("trace.json")
```


Chrome
chrome://tracing
Record
Save
Load
trace.json
Flow events
Processes
M
View Options
200 µs
300 µs
400 µs
▼ python (pid 172699): CPU
▼ thread 172699 (python)
forward
aten::linear
aten::t
aten::addmm
aten::tr...
aten::copy_
▼ Process Spans
PyTorch Profiler
PyTorch Profiler (0)
File Size Stats
Metrics
1 item selected.
Slice (1)
Title aten::resolve_conj
Category cpu_op
User Friendly Category other
Start 0.312 ms
Wall Duration 0.001 ms
▼Args
External id 10
Record function id 0
Concrete Inputs [""]
Input type ["float"]
Input Strides [[10, 1]]
Input Dims [[5,

Memory Profiling

```
import torch
from torch.profiler import profile, ProfilerActivity, record_function

model = torch.nn.Linear(20, 10)
i = torch.randn(5, 20)

with profile(activities=[ProfilerActivity.CPU],
             profile_memory=True,
             record_shapes=True) as prof:
    with record_function("forward"):
        o = model(i)

print(prof.key_averages().table(sort_by="cpu_memory_usage", row_limit=10))
```

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	CPU Mem	Self CPU Mem	# of Calls
forward	37.00%	94.739us	100.00%	256.056us	256.056us	200 B	0 B	1
aten::linear	6.82%	17.468us	63.00%	161.317us	161.317us	200 B	0 B	1
aten::addmm	30.71%	78.646us	38.33%	98.159us	98.159us	200 B	200 B	1
aten::t	11.92%	30.527us	17.84%	45.690us	45.690us	0 B	0 B	1
aten::transpose	4.11%	10.528us	5.92%	15.163us	15.163us	0 B	0 B	1
aten::as_strided	1.98%	5.073us	1.98%	5.073us	2.537us	0 B	0 B	2
aten::expand	0.94%	2.409us	1.11%	2.847us	2.847us	0 B	0 B	1
aten::copy_	6.35%	16.265us	6.35%	16.265us	16.265us	0 B	0 B	1
aten::resolve_conj	0.16%	0.401us	0.16%	0.401us	0.201us	0 B	0 B	2

Self CPU time total: 256.056us

Profiling with CUDA

```
import torch
from torch.profiler import profile, ProfilerActivity,record_function

model = torch.nn.Linear(20, 10).to(“cuda:0”)

i = torch.randn(5, 20).to(“cuda:0”)

with profile(activities=[ProfilerActivity.CPU,ProfilerActivity.CUDA],
             record_shapes=True) as prof:
    with record_function("forward"):
        o = model(i)

print(o.shape)
prof.export_chrome_trace("trace.json")
```

Pre-prepared GPT2 profile Exploration

Open trace.json using chrome://tracing

See cpu-gpu interactions


Chrome
chrome://tracing
Record
Save
Load
trace.json
Flow events
Processes
M
View Options
5 ms
10 ms
15 ms
20 ms
25 ms
▾ python (pid 5115): CPU
▾ thread 5115 (pt_main_thread)
simple.py(50): <module>
trl/trainer/sft_trainer.py(451): train
transformers/trainer.py(1932): train
transformers/trainer.py(2343): _inner_training_loop
ProfilerStep#2
transformers/trainer.py(3282): training_step
transformers/trainer.py(3328): compute_loss
accelerate/accelerator.py(2...
nn.Module: GPT2LMHeadModel_0
torch/_tensor.py(466): backward
torch/nn/modules/module.py(1534): _call_impl
torch/autograd/__init__.py(...
transformers/models/gpt2/modeling_gpt2.py(1390): forward
torch/autograd/graph.py(...
nn.Module: GPT2Model_0
<built-in method run_...
transformers/models/gpt2/modeling_gpt2.py(1076): forward
File Size Stats
Metrics
Frame Data
Input Latency
Alerts

Case study: Flash Attention

FLASHATTENTION: Fast and Memory-Efficient Exact Attention with IO-Awareness

Tri Dao[†], Daniel Y. Fu[†], Stefano Ermon[†], Atri Rudra[‡], and Christopher Ré[†]

[†]Department of Computer Science, Stanford University
[‡]Department of Computer Science and Engineering, University at Buffalo, SUNY

{trid,danfu}@cs.stanford.edu, ermon@stanford.edu, atri@buffalo.edu, chrismre@cs.stanford.edu

June 24, 2022

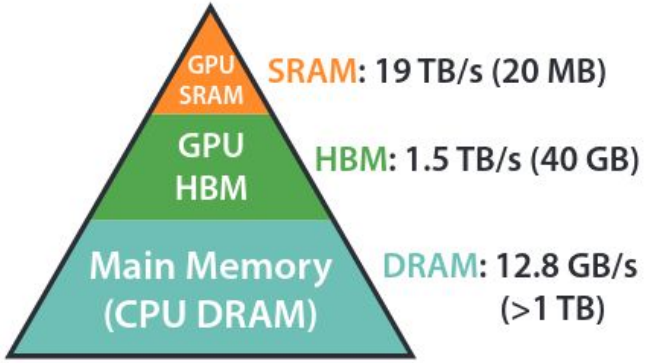
Standard Attention

Algorithm 0 Standard Attention Implementation

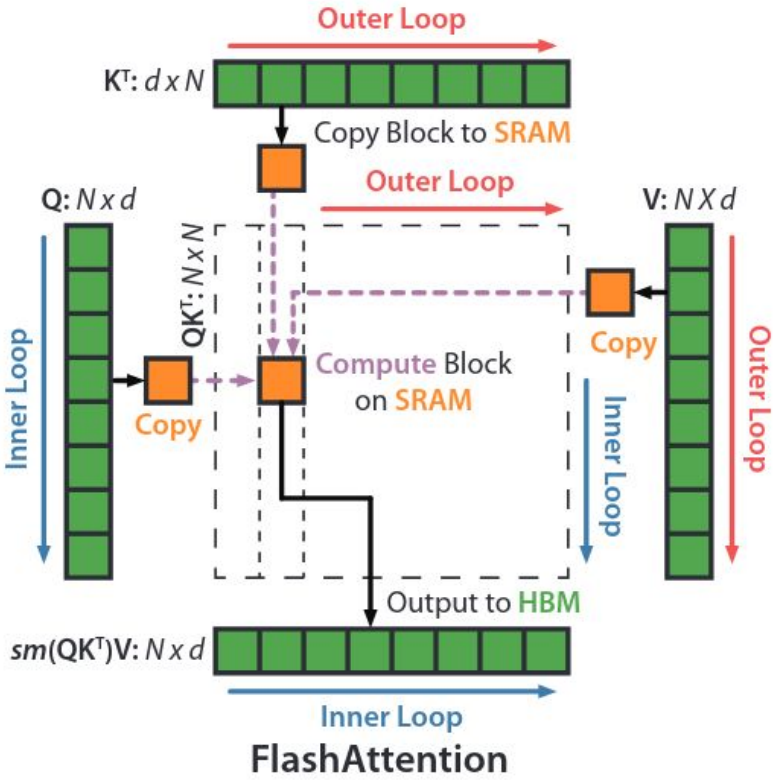
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

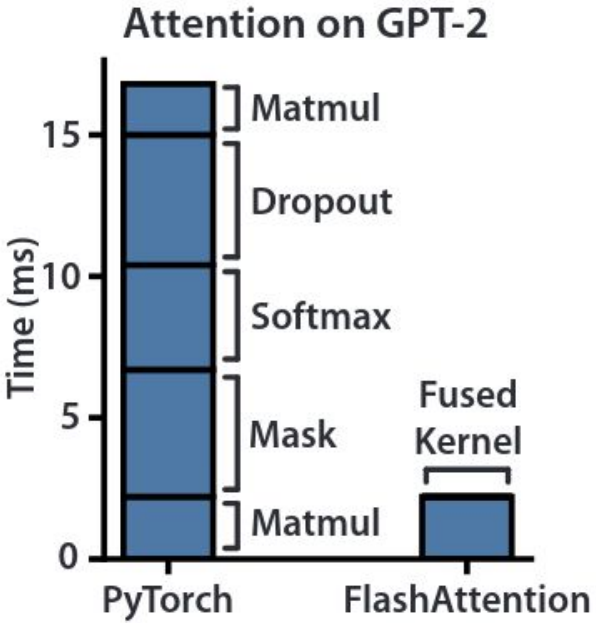
Flash Attention



Memory Hierarchy with Bandwidth & Memory Size



Fusion and Time-savings



Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

Impact on Backward Pass

We would need a custom backward pass.
Autograd cannot solve this for us.

These special purpose ops are defined as custom-ops

extra statistics (Algorithm 1 line 10), and combine the results (Algorithm 1 line 12).

Recomputation. One of our goals is to not store $O(N^2)$ intermediate values for the backward pass. The backward pass typically requires the matrices $\mathbf{S}, \mathbf{P} \in \mathbb{R}^{N \times N}$ to compute the gradients with respect to $\mathbf{Q}, \mathbf{K}, \mathbf{V}$. However, by storing the output $\mathbf{O}$ and the softmax normalization statistics (m, ℓ), we can recompute the attention matrix $\mathbf{S}$ and $\mathbf{P}$ easily in the backward pass from blocks of $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ in SRAM. This can be seen as a form of selective gradient checkpointing [10, 34]. While gradient checkpointing has been suggested to reduce the maximum amount of memory required [66], all implementations (that we know off) have to trade speed for memory. In contrast, even with more FLOPs, our recomputation speeds up the backward pass due to reduced HBM accesses (Fig. 2). The full backward pass description is in Appendix B.

Implementation details: Kernel fusion. Tiling enables us to implement our algorithm in one

https://github.com/Dao-AILab/flash-attention/tree/main/csrc/flash_attn/src

Check the profiles: bf16 and bf16_fa

Training the Model

So far we have seen…

model = …

input = …

model.forward(input)

This is a single forward pass

For Inference

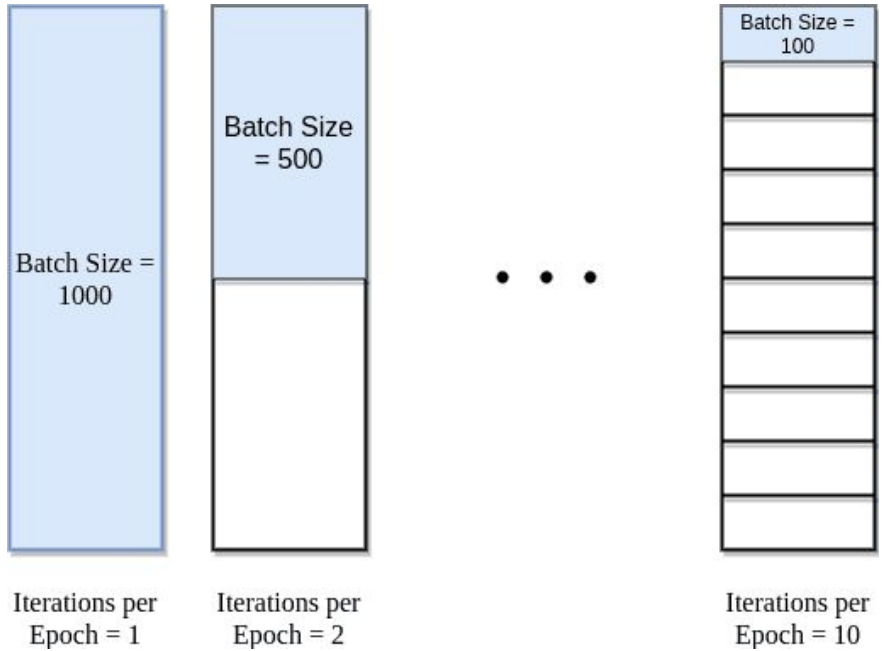
```
model = …

input = …


for i in num_output_tokens:
  output = model.forward(input)
  decode(output)
```

We will cover Inference in more details in Session 3

The Train Loop

```
For e in epochs:
  For b in batches:
      loss = model.forward(b)
      loss.backward()

      optimizer.step()
```

1 epoch = all data
Batch size is number of iterations to complete an epoch



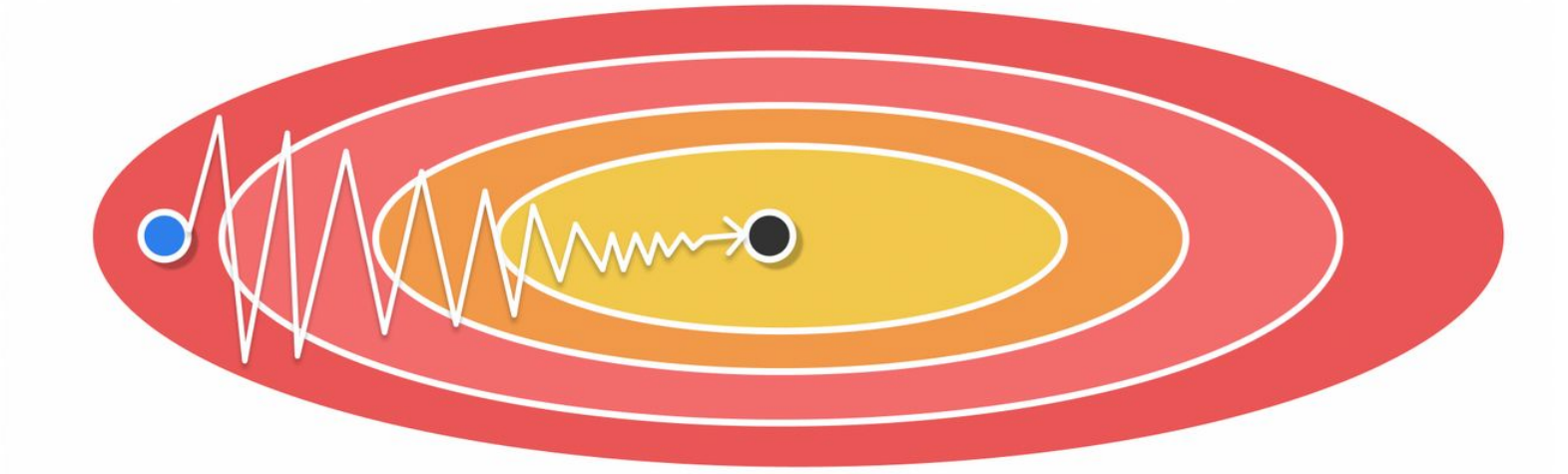
https://www.baeldung.com/cs/epoch-neural-networks

What is the optimizer?

$$w_t = w_{t-1} - \alpha \cdot dw_t$$

Gradient descent equation. w is the weight vector, dw is the gradient of w, α is the learning rate, t is the iteration number

https://towardsdatascience.com/understanding-deep-learning-optimizers-momentum-adagrad-rmsprop-adam-e311e377e9c2/



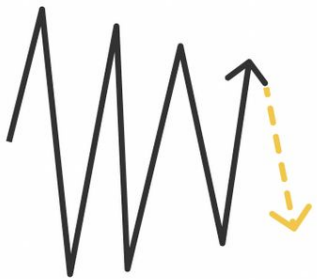
Example of an optimization problem with gradient descent in a ravine area. The starting point is depicted in blue and the local minimum is shown in black.

$$v_t = \beta v_{t-1} + (1 - \beta) dw_t$$

$$w_t = w_{t-1} - \alpha v_t$$

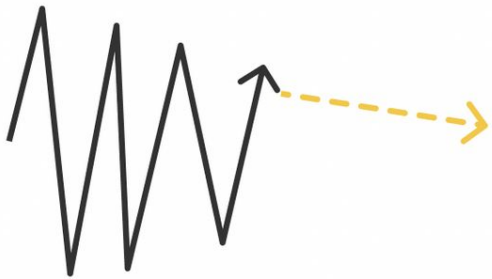
Momentum equations

Gradient descent



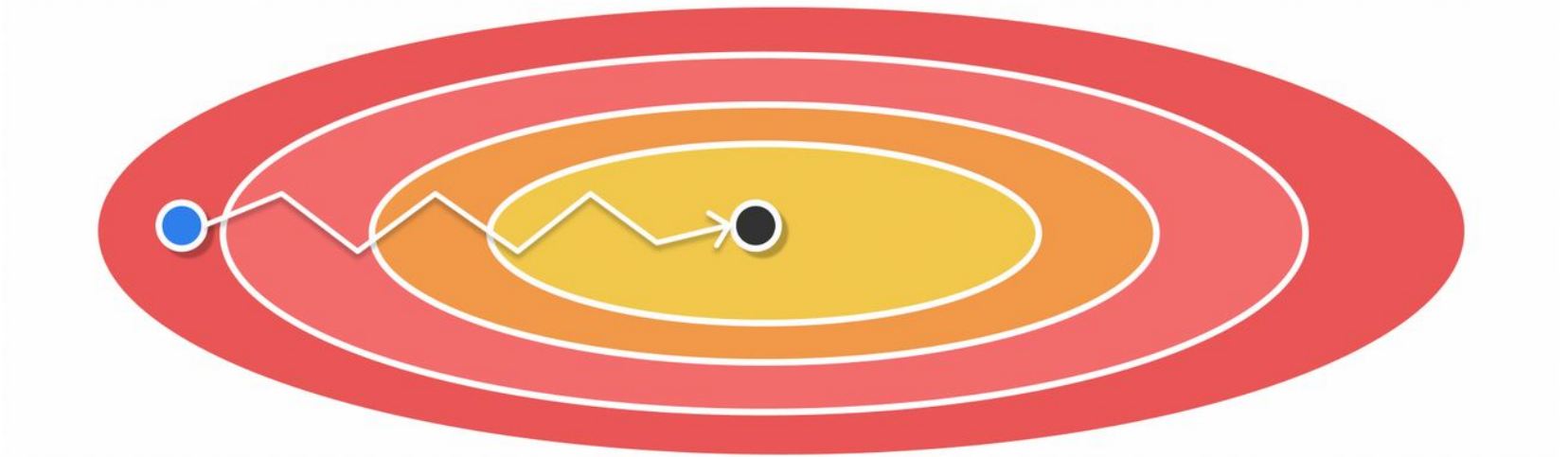
the gradient computed on the current iteration does not prevent gradient descent from oscillating in the vertical direction

Momentum



the average vector of the horizontal component is aligned towards the minimum

the average vector of the vertical component is close to 0



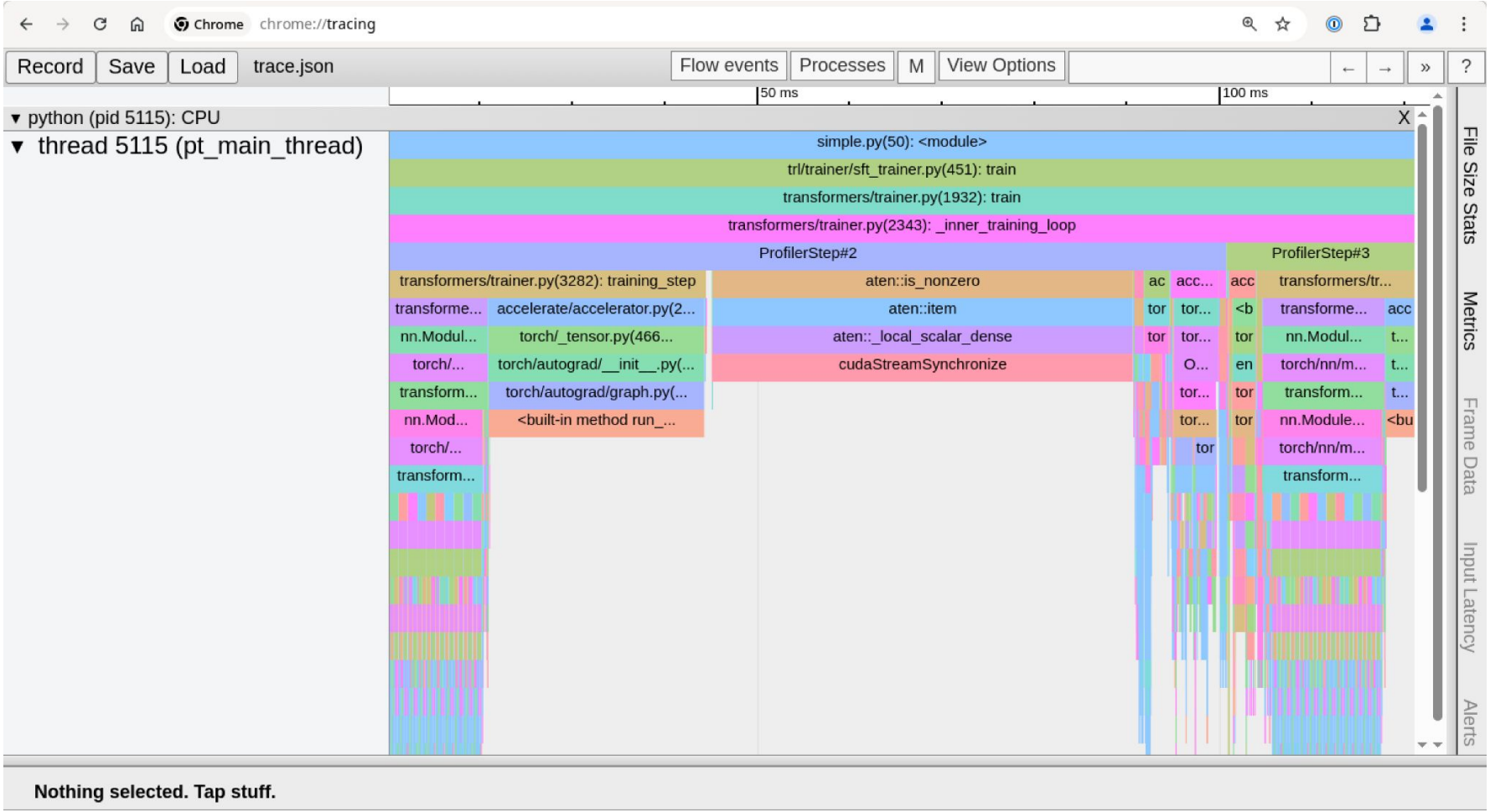
Optimization with Momentum

Run this train loop on CPU/Colab

```
import torch
from transformers import AutoModelForCausalLM
from torch.profiler import profile, ProfilerActivity, record_function

model = AutoModelForCausalLM.from_pretrained("gpt2")
optimizer = torch.optim.AdamW(model.parameters(), lr=0.001, betas=(0.9,
0.95), weight_decay=0.0,)

def inpgen(model, size):
    inp = {}
    inp["input_ids"] = torch.randint(0, model.config.vocab_size, size)
    inp["labels"] = torch.randint(0, model.config.vocab_size, size)
    inp["position_ids"] = torch.arange(0, size[1]).repeat(size[0], 1)
    return inp
```

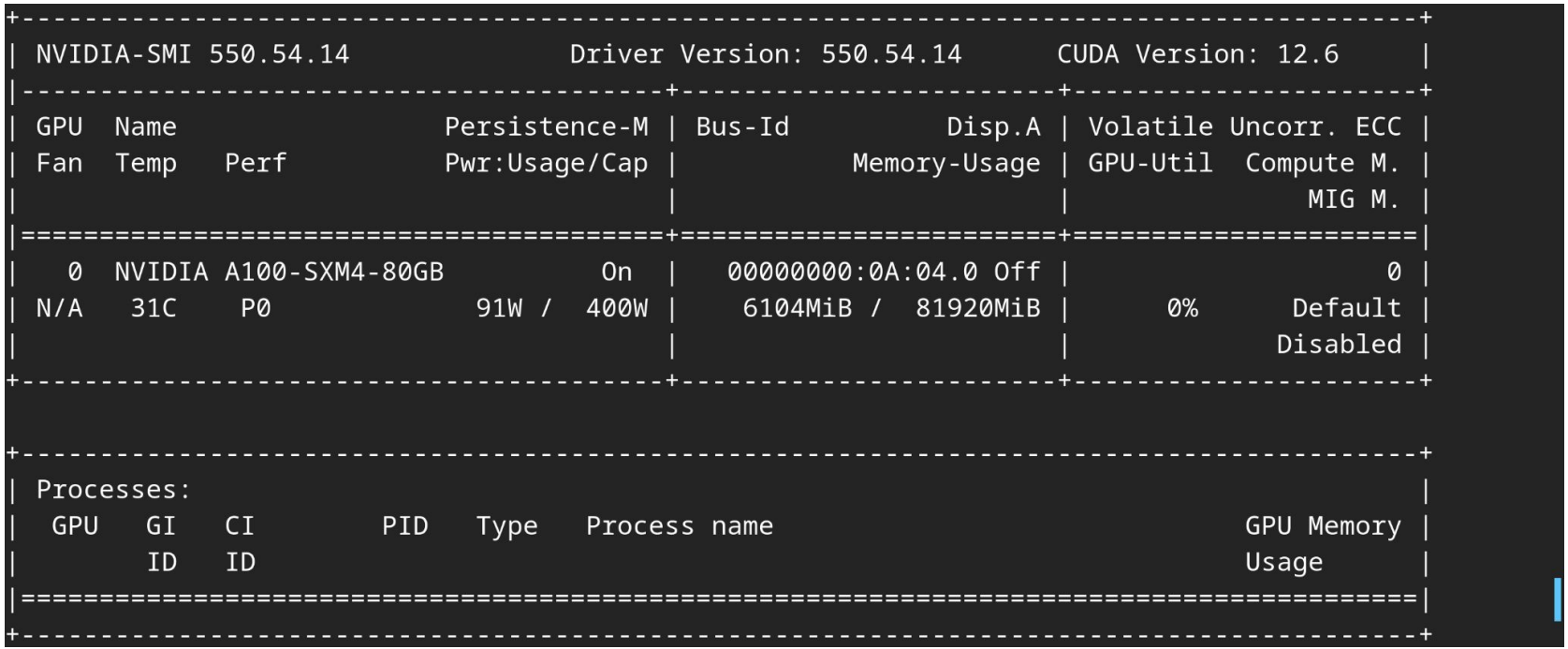
```
def step(inp, model, optimizer):
    outputs = model(**inp)
    loss = outputs.loss
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

size = (4, 512)
with profile(activities=[ProfilerActivity.CPU],
             record_shapes=True) as prof:
    for i in range(3):
        step(inpgen(model, size), model, optimizer)

prof.export_chrome_trace("trace.json")
```

Examine Training components
On GPT2 profile (CUDA)


Chrome
chrome://tracing
Record
Save
Load
trace.json
Flow events
Processes
M
View Options
50 ms
100 ms
▼ python (pid 5115): CPU
▼ thread 5115 (pt_main_thread)
simple.py(50): <module>
trl/trainer/sft_trainer.py(451): train
transformers/trainer.py(1932): train
transformers/trainer.py(2343): _inner_training_loop
ProfilerStep#2
ProfilerStep#3
transformers/trainer.py(3282): training_step
aten::is_nonzero
aten::item
aten::_local_scalar_dense
cudaStreamSynchronize
accelerate/accelerator.py(2...
torch/_tensor.py(466...
torch/autograd/__init__.py(...
torch/autograd/graph.py(...
<built-in method run_...
File Size Stats
Metrics
Frame Data
Input Latency
Alerts
Nothing selected. Tap stuff.

Examining Memory

nvidia-smi


NVIDIA-SMI 550.54.14 Driver Version: 550.54.14 CUDA Version: 12.6
GPU Name Persistence-M Bus-Id Disp.A Volatile Uncorr. ECC
Fan Temp Perf Pwr:Usage/Cap Memory-Usage GPU-Util Compute M.
MIG M.
0 NVIDIA A100-SXM4-80GB On 00000000:0A:04.0 Off 0
N/A 31C P0 91W / 400W 6104MiB / 81920MiB 0% Default
Disabled
Processes:
GPU GI CI PID Type Process name GPU Memory
ID ID Usage

Cuda Memory snapshot: look at mem.pickle

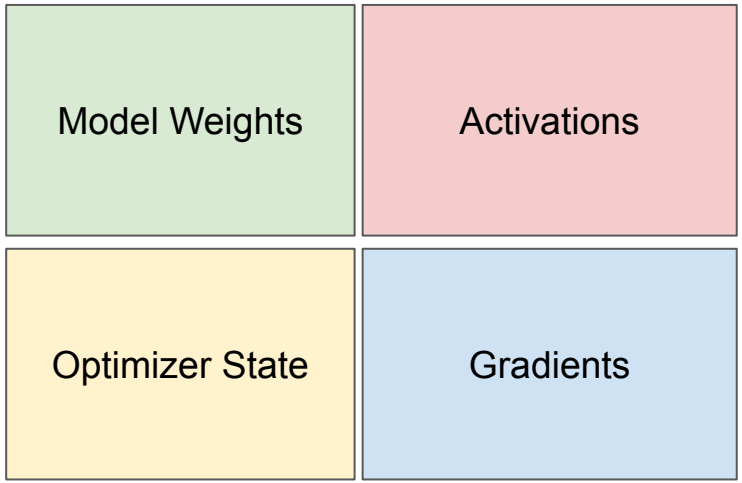
torch.cuda.memory._record_memory_history()

…

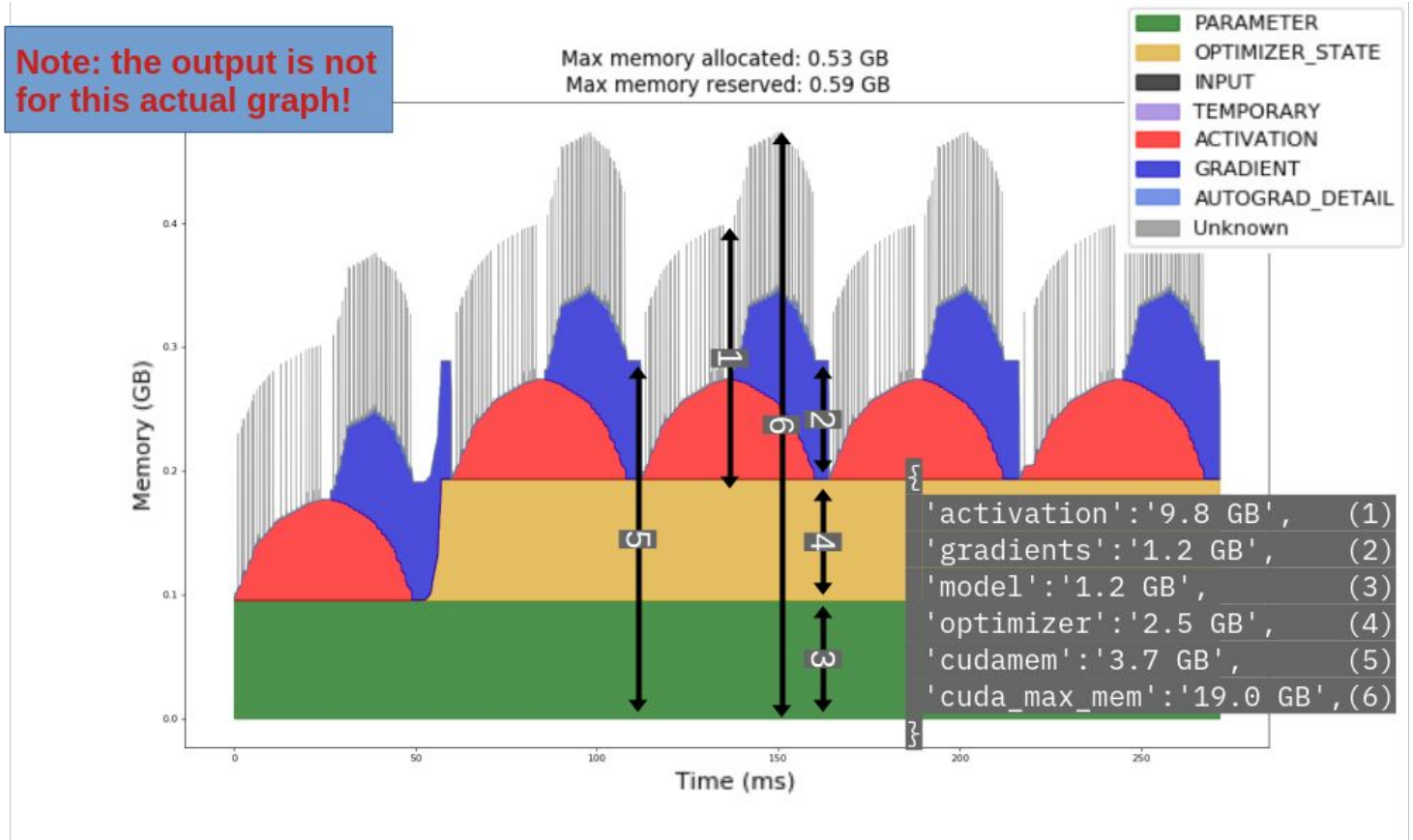
torch.cuda.memory._dump_snapshot("mem.pickle")

Load it into: https://docs.pytorch.org/memory_viz

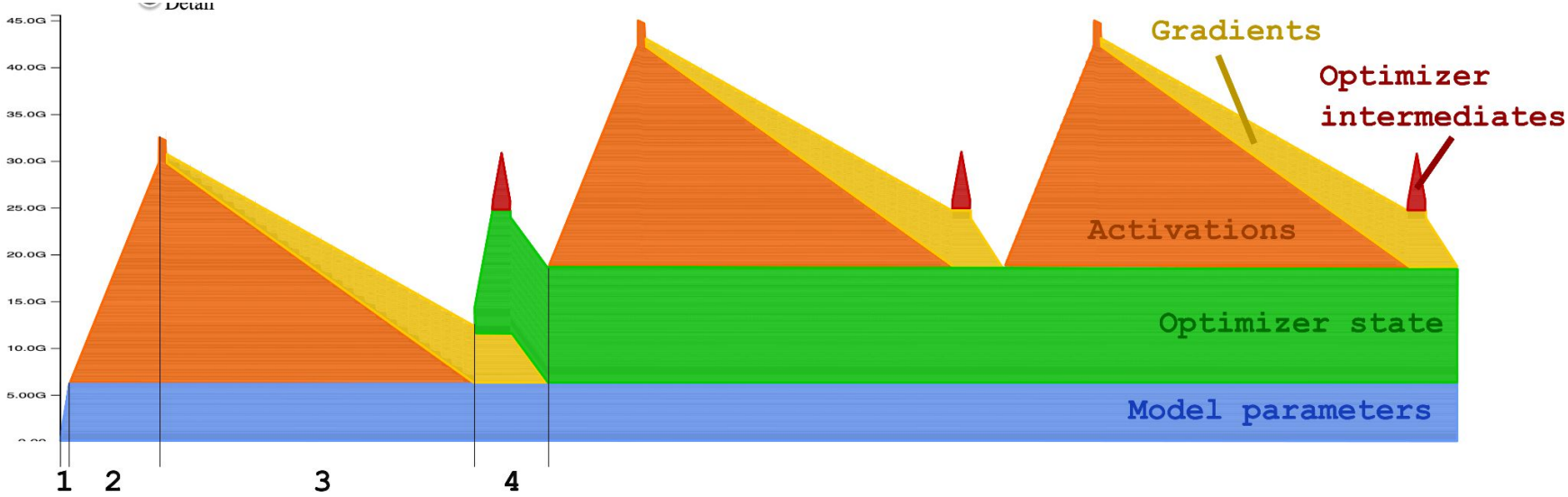
Key Components wrt Memory



Profiler output wrt Memory



Memory explained



https://huggingface.co/blog/train_memory

Using the profiler for memory: output.html

Profiling Code used

```
with profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
             record_shapes=True,
             profile_memory=True,
             with_stack=True
             ) as prof:
    for i in range(3):
        step(inpgen(model, size), model, optimizer)

from torch.cuda._memory_viz import profile_plot
with open('output.html', 'w') as f:
    f.write(profile_plot(prof))
```

Advanced
Research Opportunities

Triton - Job opportunity

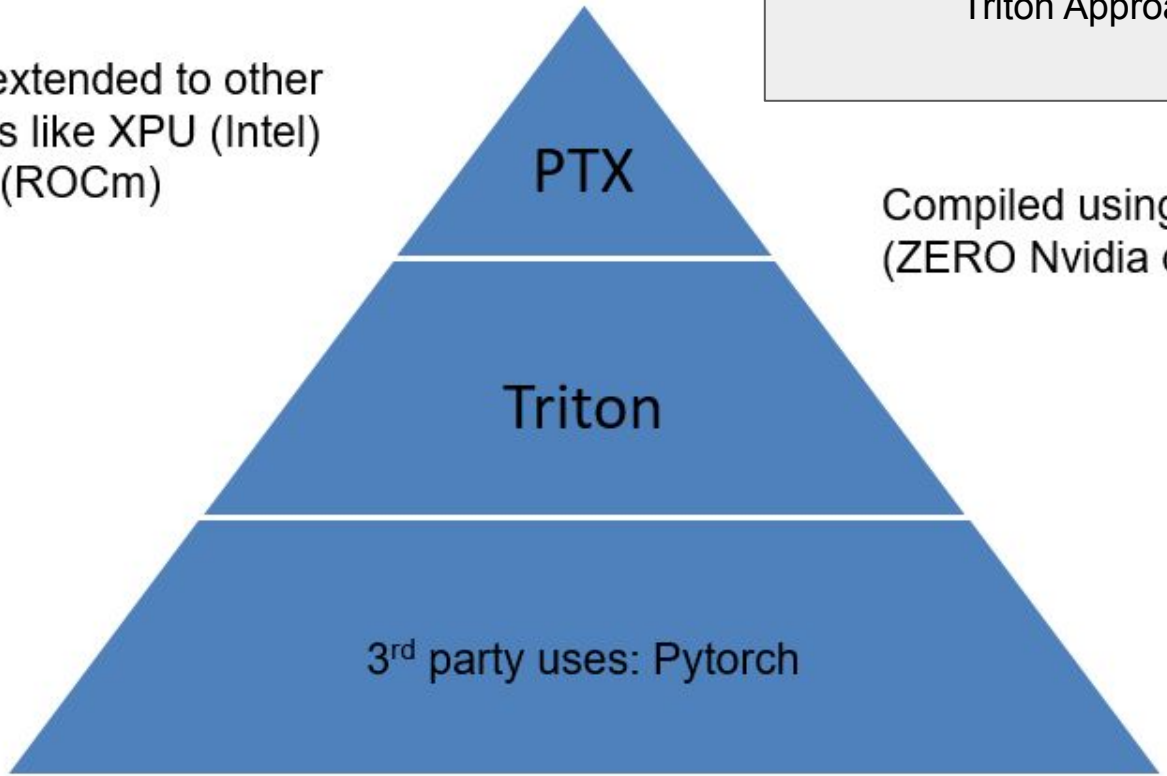
Compile - Open Research area

Llama.cpp - quantization on cpu

Triton: reducing CUDA dependency

Triton Approach

Can be extended to other backends like XPU (Intel) Or AMD (ROCm)



Compiled using LLVM (ZERO Nvidia dependency)

https://triton-lang.org/main/index.html

```python
@triton.jit
def add_kernel(x_ptr,  # *Pointer* to first input vector.
               y_ptr,  # *Pointer* to second input vector.
               output_ptr,  # *Pointer* to output vector.
               n_elements,  # Size of the vector.
               BLOCK_SIZE: tl.constexpr,  # Number of elements each program should process.
               # NOTE: `constexpr` so it can be used as a shape value.
               ):
    # There are multiple 'programs' processing different data. We identify which program
    # we are here:
    pid = tl.program_id(axis=0)  # We use a 1D launch grid so axis is 0.
    # This program will process inputs that are offset from the initial data.
    # For instance, if you had a vector of length 256 and block_size of 64, the programs
    # would each access the elements [0:64, 64:128, 128:192, 192:256].
    # Note that offsets is a list of pointers:
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Create a mask to guard memory operations against out-of-bounds accesses.
    mask = offsets < n_elements
    # Load x and y from DRAM, masking out any extra elements in case the input is not a
    # multiple of the block size.
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    # Write x + y back to DRAM.
    tl.store(output_ptr + offsets, output, mask=mask)
```

To try out CUDA and Triton programs

leetgpu.com

Compile

Let’s start with the simplest possible example:

```
@torch.compile
def foo(x: torch.Tensor, y: torch.Tensor):
    z = x * y
    w = z + x
    return w
```

Let’s start without Compile and examine the Profile output. The relevant section is show below, before we zoom into the part we are interested in.

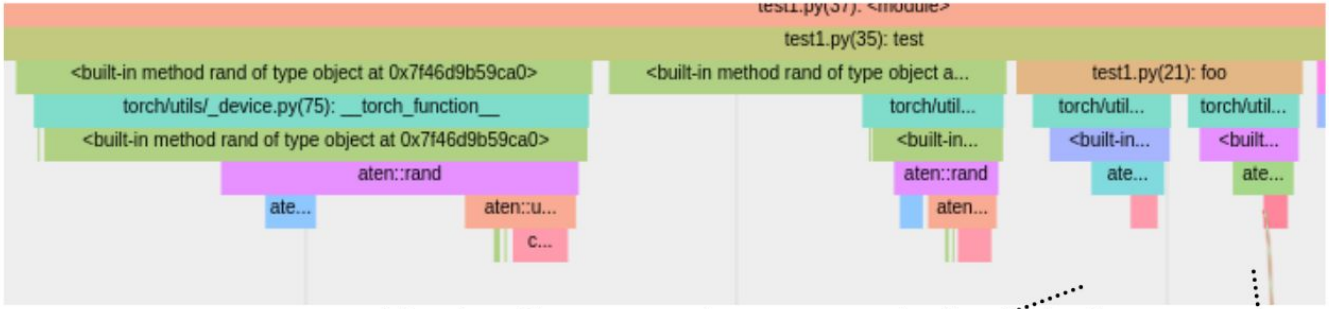


Figure 1: High level Profile Capture. The 2 aten::rand calls are the data preparation done before the function foo is called.

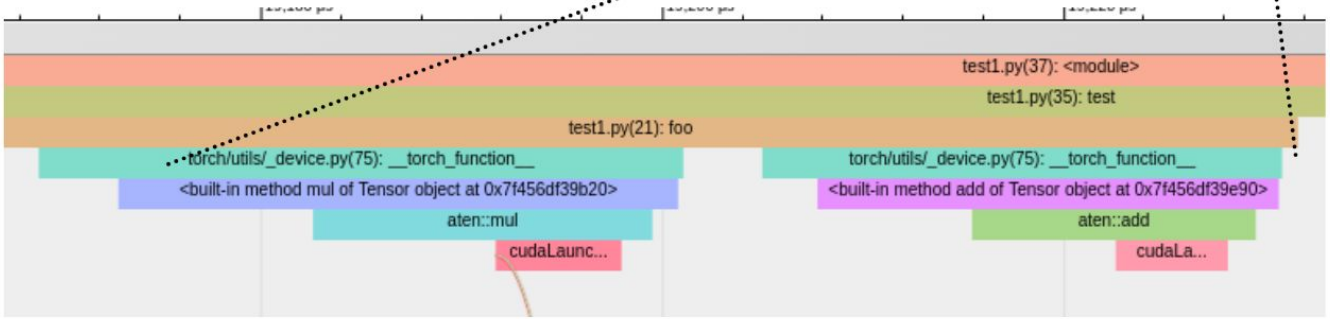


Figure 2: Function foo zoomed in. We have 2 cudaLaunchKernel calls.

Figure 3: Same view from the GPU pov. 2 rand calls, followed by a mult (currently selected) and add.

Now, the same foo, with torch.compile.

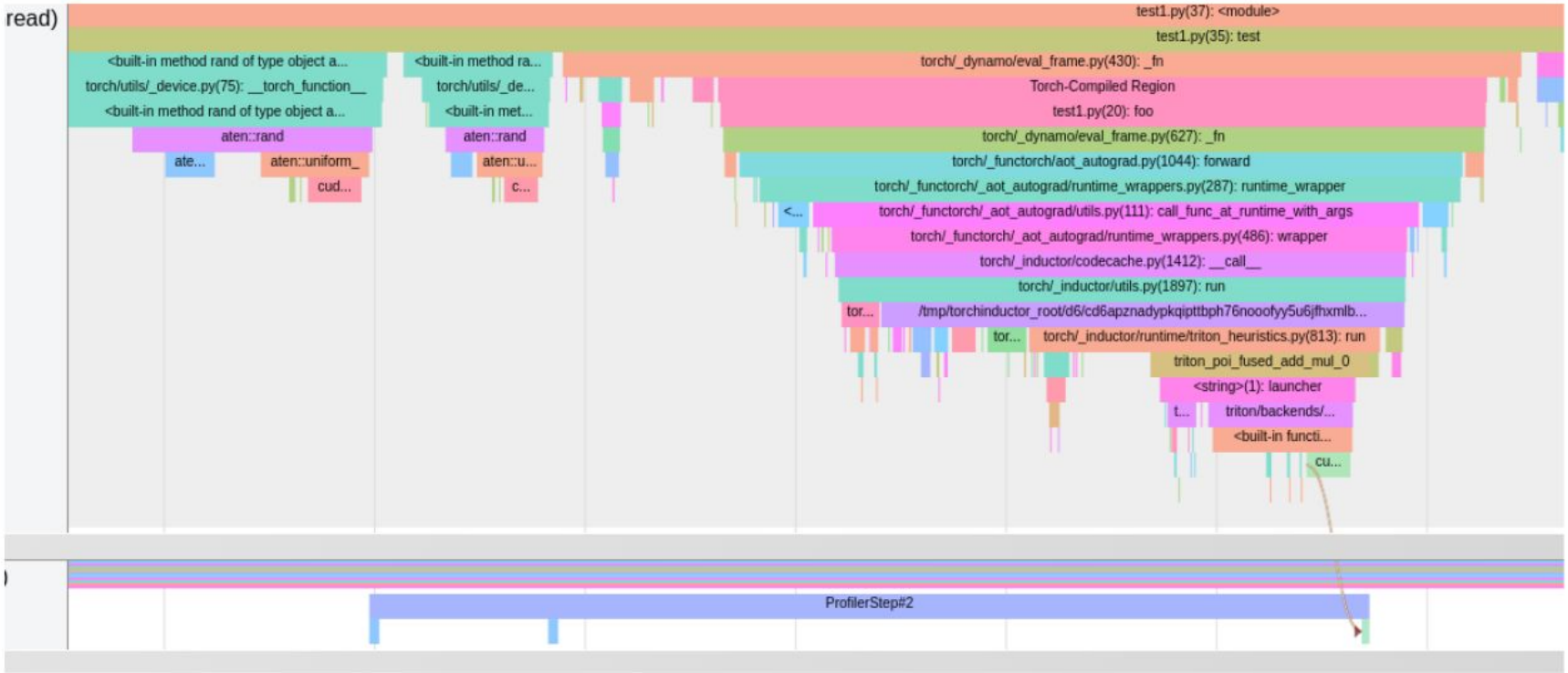


Figure 4: With torch.compile, there are only 3 kernels now on the GPU. The add and mult have been fused into a single Triton kernel.

We can use the tlparse command to examine torch.compile stack trace information. The generated code from Inductor looks like this: showing the fusion clearly.

```
triton_poi_fused_add_mul_0 = async_compile.triton('triton_', '''
…
@triton.jit
def triton_(in_ptr0, in_ptr1, out_ptr0, xnumel, XBLOCK : tl.constexpr):
    xnumel = 25
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + (x0), xmask)
    tmp1 = tl.load(in_ptr1 + (x0), xmask)
    tmp2 = tmp0 * tmp1
    tmp3 = tmp2 + tmp0
    tl.store(out_ptr0 + (x0), tmp3, xmask)
…


stream0 = get_raw_stream(0)
triton_poi_fused_add_mul_0.run(arg0_1, arg1_1, buf0, 25, grid=grid(25), stream=stream0)
del arg0_1
```

Example of compile: on CPU

```
import torch
import time

def foo1(x1, x2):
    a = torch.neg(x1)
    b = torch.maximum(x2, a)
    y = torch.cat([b], dim=0)
    return y

x1 = torch.randint(256, (1, 8),
dtype=torch.uint8)
x2 = torch.randint(256, (8390, 8),
dtype=torch.uint8)

compiled_foo1 = torch.compile(foo1)
```

```
start = time.time()
result = compiled_foo1(x1, x2)
end = time.time()
print("With compile: ", result, end
- start)

start = time.time()
result = foo1(x1, x2)
end = time.time()
print("Without compile: ", result,
end - start)
```

TORCH_TRACE=./logs/
CUDA_VISIBLE_DEVICES="" python main.py

tlparse ./logs/*.log
Open the html file in tl_out

```
cpp_fused_maximum_neg_0 = async_compile.cpp_pybinding(['const uint8_t*', 'const uint8_t*', 'uint8_t*'], '''
#include "/tmp/torchinductor_chandergovind/pi/cpicxudqmdsjh5cm4klbtbrvy2cxwr7whxl3md2zzdjdf3orvfdf.h"
extern "C"  void kernel(const uint8_t* in_ptr0,
                        const uint8_t* in_ptr1,
                        uint8_t* out_ptr0)
{
    #pragma omp parallel num_threads(14)
    {
        int tid = omp_get_thread_num();
        {
            #pragma omp for
            for(int64_t x0=static_cast<int64_t>(0L); x0<static_cast<int64_t>(8390L); x0+=static_cast<int64_t>(1L))
            {
                for(int64_t x1=static_cast<int64_t>(0L); x1<static_cast<int64_t>(8L); x1+=static_cast<int64_t>(8L))
                {
                    {
                        if(C10_LIKELY(x1 >= static_cast<int64_t>(0) && x1 < static_cast<int64_t>(8L)))
                        {
                            auto tmp0 = at::vec::Vectorized<uint8_t>::loadu(in_ptr0 + static_cast<int64_t>(x1 + 8L*x0), static_cast<int64_t>(8));
                            auto tmp1 = at::vec::Vectorized<uint8_t>::loadu(in_ptr1 + static_cast<int64_t>(x1), static_cast<int64_t>(8));
                            auto tmp2 = tmp1.neg();
                            auto tmp3 = at::vec::maximum(tmp0, tmp2);
                            tmp3.store(out_ptr0 + static_cast<int64_t>(x1 + 8L*x0), static_cast<int64_t>(8));
                        }
                    }
                }
            }
        }
    }
}
''')
```

Ollama and llama.cpp

Two options for optimizing local runs

Ollama

Run models locally for use

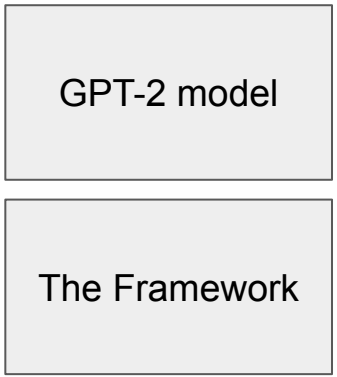
Uses llama.cpp underneath for some parts

https://ollama.com/

Llama.cpp / ggml

To understand lower level details

https://github.com/ggml-org/ggml/tree/master/examples/gpt-2



https://github.com/ggml-org/ggml/blob/master/examples/gpt-2/main-ctx.cpp

https://github.com/ggml-org/ggml/blob/master/include/ggml.h

Build

```
git clone https://github.com/ggml-org/ggml
cd ggml

# install python dependencies in a virtual environment
python3.10 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt

# build the examples
mkdir build && cd build
cmake ..
cmake --build . --config Release -j 8
```

GPT inference (example)

```
# run the GPT-2 small 117M model
../examples/gpt-2/download-ggml-model.sh 117M
./bin/gpt-2-backend -m models/gpt-2-117M/ggml-model.bin -p "This is an example"
```

```
(llm-productivity) ~/D/t/i/g/build$ bash ../examples/gpt-2/download-ggml-model.sh 117M^C
(llm-productivity) ~/D/t/i/g/build$ ./bin/gpt-2-ctx -m models/gpt-2-117M/ggml-model.bin -p "This is
 an example"
main: seed = 1756643706
gpt2_model_load: loading model from 'models/gpt-2-117M/ggml-model.bin'
gpt2_model_load: n_vocab = 50257
gpt2_model_load: n_ctx   = 1024
gpt2_model_load: n_embd  = 768
gpt2_model_load: n_head  = 12
gpt2_model_load: n_layer = 12
gpt2_model_load: ftype   = 1
gpt2_model_load: qntvr   = 0
gpt2_model_load: ggml tensor size = 336 bytes
gpt2_model_load: ggml ctx size = 384.88 MB
gpt2_model_load: memory size =    72.00 MB, n_mem = 12288
gpt2_model_load: model size  =   239.08 MB
extract_tests_from_file : No test file found.
test_gpt_tokenizer : 0 tests failed out of 0 tests.
main: prompt: 'This is an example'
main: number of tokens in prompt = 4, first 8 tokens: 1212 318 281 1672

This is an example of the way that our approach to business development can work.

We often have to make decisions based on what we believe are the best fit for a particular person's
 needs and interests. Our approach is to consider all possible options and make our own choices.
```

Summary

- What does a framework like Pytorch do?
- What does a GPU give you?
- How to profile compute and memory
- Compile and llama.cpp