Narrow GPU Acceleration With Python

What Are We Talking About?

- Focus: Narrow and practical
- Topics: Tensors, GPUs, Parallel processing, Performance limits
- Goal: Build a mental model for performance

First Things First, What Is a Tensor?

- An n-dimensional array
- Scalar (0D), Vector (1D), Matrix (2D)
- Just structured data

If You Know NumPy, You know Tensors

PyTorch is basically

- NumPy
  - GPU support
  - autograd (ignored)
- We only care about
  - Data
  - Compute

DEMO: NumPy VS PYTORCH Tensor

```
import numpy as np
import torch

x_np = np.random.randn(3, 4)

x_torch = torch.randn(3, 4)
x_gpu = x_torch.to("cuda")
print(x_gpu.device)
```

✓ 0.8s

```
cuda:0
```


Why GPUs?

CPUs: optimized for latency (Efficient for single or small number of tasks)

GPUs: optimized for throughput (Designed for many operations at once)

- Origin: Graphics
  - Pixels
  - Triangles
  - Shaders

GPUs: Many cores working in parallel

Why Does This Apply to AI?

- Most AI workloads are dominated by matrix multiplication.
  - GEMM (General Matrix Multiply)
  - Matrix multiplication is an embarrassingly parallel problem.
  - In transformer models (LLMs)
    - ~95%–99% of compute come from matrix multiplies

The Problem: How can we accelerate?

```python
import numpy as np

batch_tokens = 4096
hidden = 4096

x = np.random.randn(batch_tokens, hidden).astype(np.float32)
bias = np.random.randn(hidden).astype(np.float32)

def gelu_approx(z):
    return 0.5 * z * (
        1.0 + np.tanh(np.sqrt(2.0 / np.pi) * (z + 0.044715 * z**3))
    )

y = gelu_approx(x + bias)
print(y.shape)
```

✓ 0.9s

```
(4096, 4096)
```


What is the code doing?

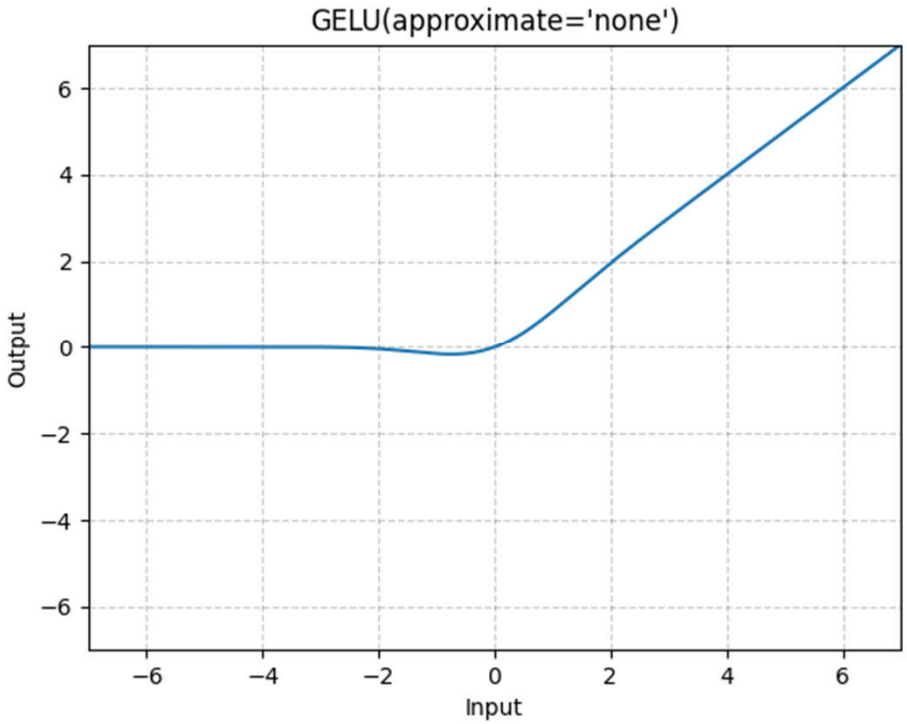
```python
import numpy as np

# (2,3)
x = np.array([[1, 2, 3],
              [4, 5, 6]])

# (3)
bias = np.array([10, 20, 30])

result = x + bias
print(result)
```

✓ 0.0s

```
[[11 22 33]
 [14 25 36]]
```

*Since the individual outputs are not dependent on each other, the process can be **parallelized**!

GELU Graph




```
y = gelu(x + bias)
```

What is Compute? (FLOPs)

- FLOP: Floating Point Operation (add, multiply…)
- FLOPs/sec: How quickly hardware can perform math.
- GPUs: Maximize FLOPs/sec

Fundamental Constraints

- Two Limits:
  - Compute → FLOPs/sec
  - Memory → bytes/sec
- Memory Wall: memory is slower than compute
- Implication: data movement dominates performance
- If GPUs are massively parallel, why isn't everything instant?
- Memory bandwidth limits performance!

Memory VS. Compute

Every operation:

- Has a maximum speed
- Has two costs
  - Time to compute
  - Time to move data
- Performance is determined by:
  - max(time_compute, time_memory)
  - Whichever is slower wins

Arithmetic Intensity

- Definition: FLOPs/bytes moved
- Arithmetic intensity tells you how much compute you do per byte of data moved
- Low intensity
  - Not enough compute to “hide” memory cost
  - → memory-bound
- High intensity
  - Enough compute to dominate runtime
  - → compute-bound

FLOPs in Our Problem?

```
y = gelu(x + bias)
```

1. Bias add

x + bias is one elementwise add:

- FLOPs per element: 1
- Total: BN

2. GELU approximation

Per element:

- z**3 -> 2 muls
- 0.044715 * z**3 -> 1 mul
- z + ... -> 1 add
- sqrt(2/pi) * ... -> 1 mul
- tanh(...) -> 1 op
- 1.0 + ... -> 1 add
- 0.5 * z -> 1 mul
- (0.5 * z) * ... -> 1 mul

```
0.5 * z * (1.0 + tanh(sqrt(2.0 / np.pi) * (z + 0.044715 * z**3)))
```

So GELU FLOPs per element:

- 2 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 9

Total GELU FLOPs:

- 9BN

3. Total

Total FLOPs per element:

- 1 + 9 = 10

Total FLOPs:

- 10BN = 10 * 16,777,216 = 167,772,160

Memory Calculations

```
y = gelu(x + bias)
```

- dtype is float32 -> 4 bytes per value
- B = 4096
- N = 4096
- we count **logical eager-mode tensor traffic**
- x: shape (4096, 4096) -> B,N elements
- bias: shape (4096,) -> B elements
- y: shape (4096, 4096) -> B,N elements

Memory Calculations

For the add:

- Reads from x and bias: BN + B
- Writes to z: BN

```
y = gelu(x + bias)
```

For GELU:

- reads from z : BN
- writes to y: BN

```
y = gelu(z)
```

Total:

- reads: BN + BN + B
- writes: BN + BN

With BN = 16,777,216 and N = 4096:

- reads: 2 * 16,777,216 + 4,096 = 33,558,528
- writes: 2 * 16,777,216 = 33,554,432

(33,558,528 + 33,554,432) * 4 = 268,451,840 total bytes moved

Arithmetic Intensity

Arithmetic_intensity =

- FLOPs / bytes_moved

```
y = gelu(x + bias)
```

For our y = gelu_approx(x + bias) example:

- FLOPs: 167,772,160

Eager

- bytes moved: 268,451,840
- Arithmetic_intensity:

  167,772,160 / 268,451,840 = 0.625 FLOPs/byte

The Ridge Point

- On i7-13700K (CPU)
  - FLOPs/sec: 1.36e12 to 1.89e12 or ~1 teraflop/sec
  - Bandwidth: 89.6e9 bytes/sec 89.6 GB/sec
  - Required arithmetic intensity:
    - 1.36e12 / 89.6e9 ≈ 15.2 FLOPs/byte
    - 1.89e12 / 89.6e9 ≈ 21.1 FLOPs/byte

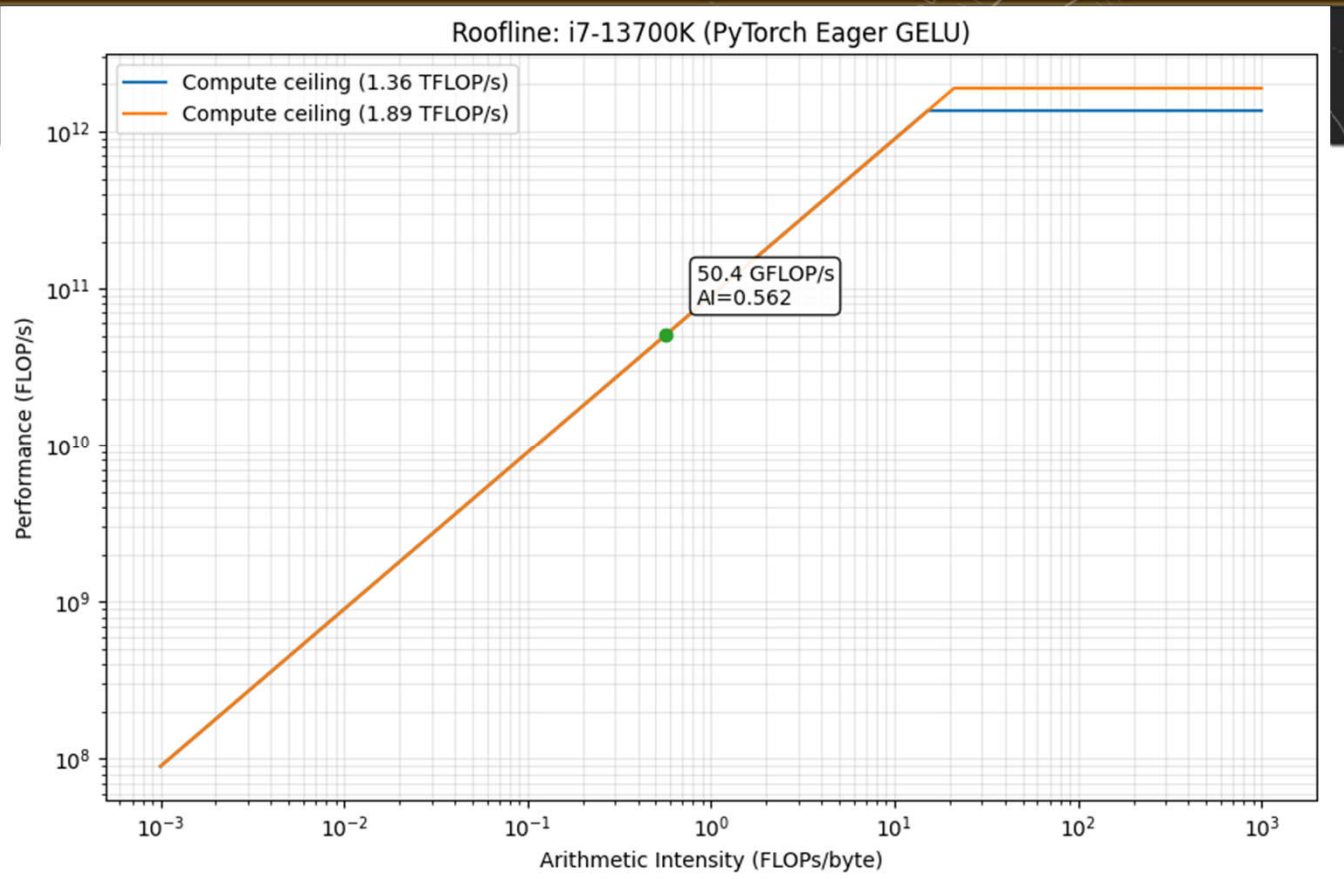
Time to Compute

- Time_compute =
  - FLOPs / FLOPs_per_second
- On i7-13700K
  - ~1.36–1.89e12 FLOPs/sec
  - For our y = gelu_approx(x + bias) example:
    - 167,772,160 / 1.36e12 ≈ 0.123 ms

Time to Move Data

- Time_memory =
  - bytes / memory bandwidth
- On i7-13700K
  - ~89.6e9 bytes/sec
  - For our y = gelu_approx(x + bias) example:
    - Eager: 268 MB / 89.6 GB/s ≈ 2.99 ms

Roofline Model



This is the Whole Game

- If time_memory > time_compute
  - GPU is waiting on data
  - → memory-bound
- If time_compute > time_memory
  - GPU is doing math the whole time
  - → compute-bound

Everything else we talk about is just rewriting this comparison

DEMO: PyTorch GPU VERSION

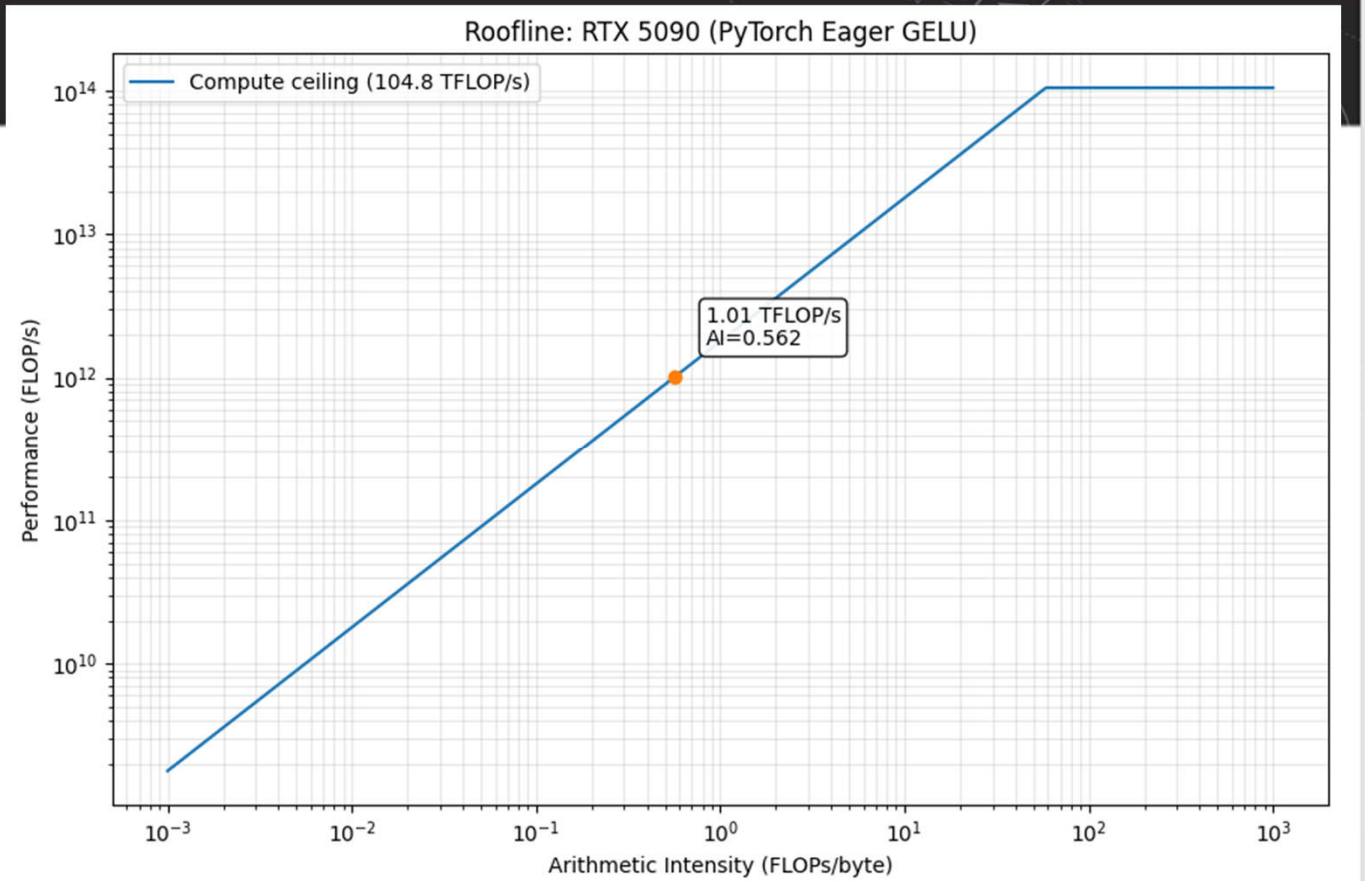
```
import torch

x = torch.randn(4096, 4096, device="cuda", dtype=torch.float32)
bias = torch.randn(4096, device="cuda", dtype=torch.float32)

y = F.gelu(x + bias)
```


Apply the Same Method

- On RTX 5090
  - FLOPs/sec: 1.048e14 or 104.8 teraflops/sec
  - Bandwidth: 1.792e12 bytes/sec or 1.792 TB/sec
  - Required arithmetic intensity:
    - 1.048e14 / 1.792e12 ≈ 58.5 FLOPs/byte
- On i7-13700K
  - FLOPs/sec: 1.36e12 to 1.89e12 or ~1 teraflop/sec
  - Bandwidth: 89.6e9 bytes/sec 89.6 GB/sec
  - Required arithmetic intensity:
    - 1.36e12 / 89.6e9 ≈ 15.2 FLOPs/byte
    - 1.89e12 / 89.6e9 ≈ 21.1 FLOPs/byte

New Roofline Model

- On i7-13700K 50 GFlops/s
- 20x difference!



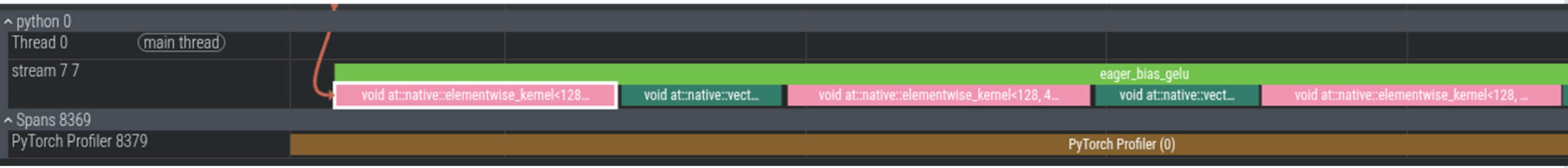
Time to Compute

- Time_compute =
  - FLOPs / FLOPs_per_second
- On RTX 5090 (GPU)
  - ~1.048e14 FLOPs/sec
  - For our y = gelu_approx(x + bias) example:
    - 167,772,160 / 1.048e14 ≈ 0.00160 ms
- On i7-13700K (CPU)
  - ~1.36–1.89e12 FLOPs/sec
  - For our y = gelu_approx(x + bias) example:
    - 167,772,160 / 1.36e12 ≈ 0.123 ms

Time to Move Data

- Time_memory =
  - bytes / memory bandwidth
- On RTX 5090 (GPU)
  - ~1.792e12 bytes/sec
  - For our y = gelu_approx(x + bias) example:
    - Eager: 268 MB / 1792 GB/s ≈ 0.150 ms
- On i7-13700K (CPU)
  - ~89.6e9 bytes/sec
  - For our y = gelu_approx(x + bias) example:
    - Eager: 268 MB / 89.6 GB/s ≈ 2.99 ms

Demo: First Profiling Pass



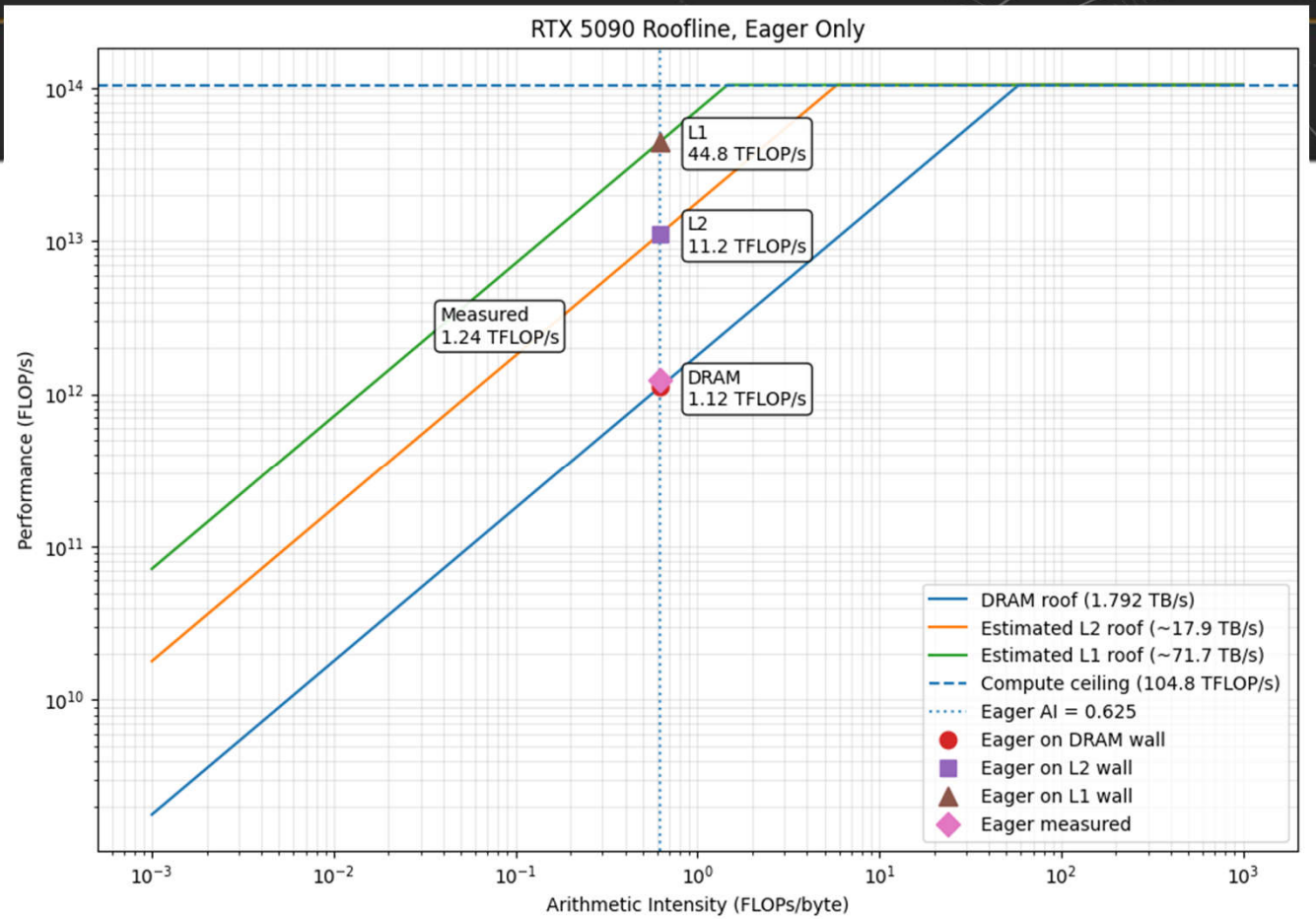
- Eager: ~1.5 TFLOPs/s
- Eager : 0.135 ms
  - But you said my theoretical maximum was 0.150ms... what gives?

Memory Hierarchy

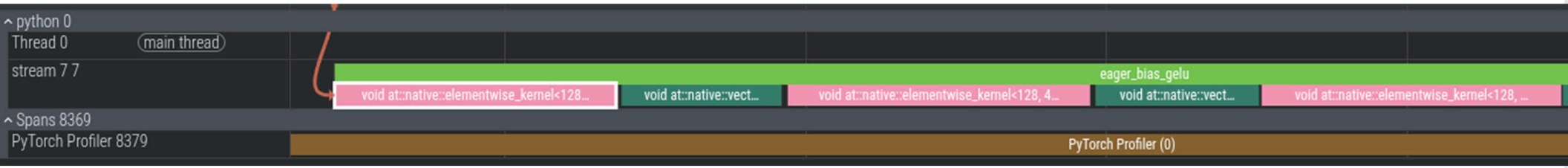
- GPU memory levels
  - Global (slow, large)
  - On-chip (fast, small)
- Performance depends on using fast memory effectively

Shading Units:	21760
TMUs:	680
ROPs:	176
SM Count:	170
Tensor Cores:	680
RT Cores:	170
L1 Cache:	128 KB (per SM)
L2 Cache:	96 MB

Roofline Model



Ok Back to The Profile... Is there a better way?



- There's so many kernel calls, can we just... do one for the entire operation?

Calculus Changes when Operations are Fused

Fused Version:
A fused kernel can do, per output element:

- read x[i, j] -> 1
- read bias[j] -> 1
- compute (x[i, j] + bias[j]), then GELU, entirely in registers
- write y[i, j] -> 1

So:

- reads: BN + B
- writes: BN

Numerically:

- reads: 16,777,216+4,096=16,781,312
- writes: 16,777,216

Bytes:

- reads: 67,125,248 bytes
- writes: 67,108,864 bytes
- total: 134,234,112 bytes
- reads: 2BN + B -> BN + B
- writes: 2BN -> BN
- **Eager:** 268,451,840 bytes
- **Fused:** 134,234,112 bytes

```
y = gelu(x + bias)
```

So kernel fusion reduces memory traffic:
~ 50**% total bytes moved**

Arithmetic Intensity

Arithmetic_intensity =

- FLOPs / bytes_moved
- For our y = gelu_approx(x + bias) example:
- FLOPs: 167,772,160

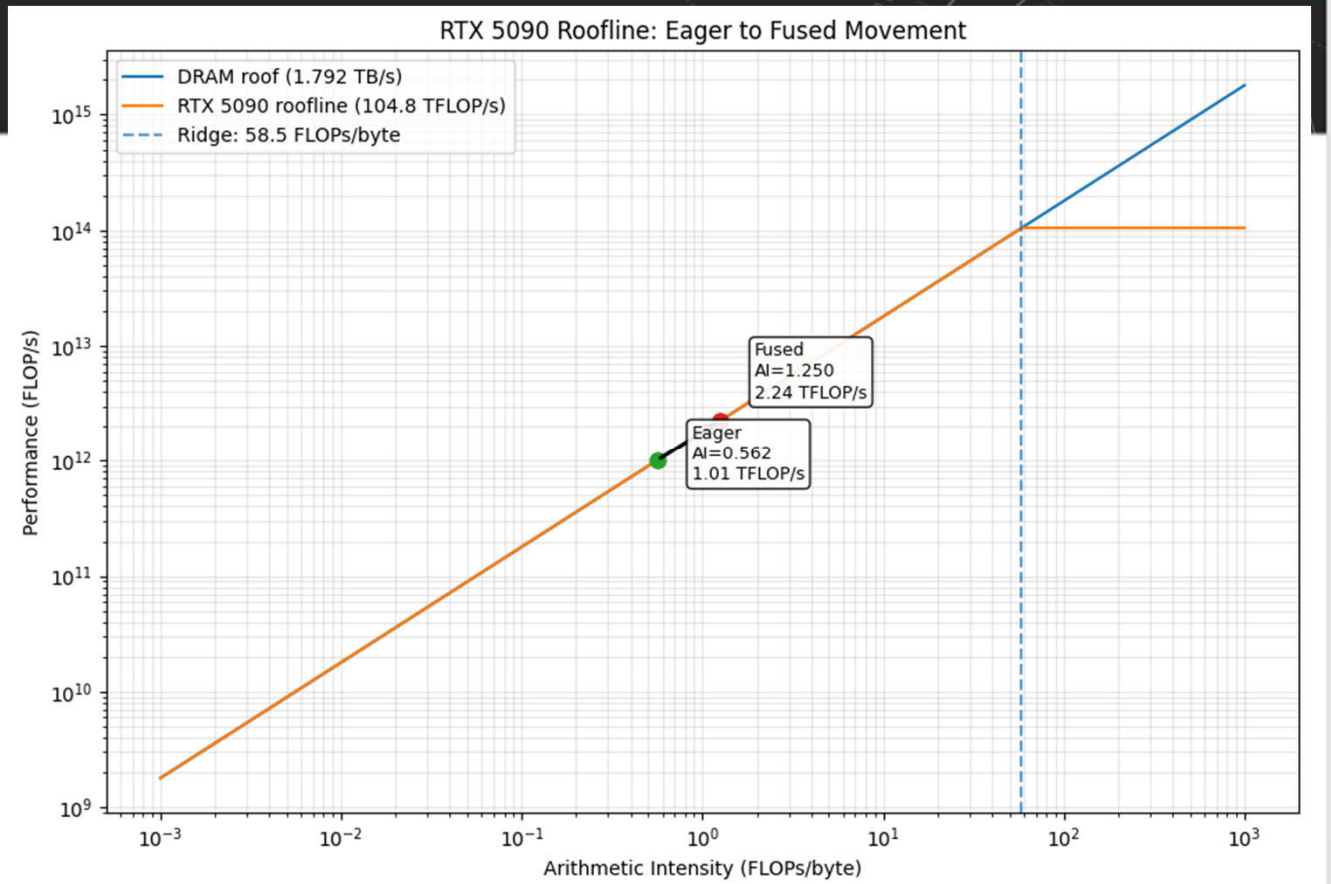
Eager

- bytes moved: 268,451,840
- Arithmetic_intensity:
  - 167,772,160 / 268,451,840 = 0.625 FLOPs/byte

Fused

- bytes moved: 134,234,112
- Arithmetic_intensity:
  - 167,772,160 / 134,234,112 ≈ 1.298 FLOPs/byte

Moving up the roofline



Time to Compute

- Time_compute =
  - FLOPs / FLOPs_per_second
- On RTX 5090
  - ~1.048e14 FLOPs/sec
  - For our y = gelu_approx(x + bias) example:
    - 167,772,160 / 1.048e14 ≈ 0.00160 ms

Time to Move Data

- Time_memory =
  - bytes / memory bandwidth
- On RTX 5090
  - ~1.792e12 bytes/sec
  - For our y = gelu_approx(x + bias) example:
    - Eager: 268 MB / 1792 GB/s ≈ 0.150 ms
    - Fused: 134 MB / 1792 GB/s ≈ 0.075 ms

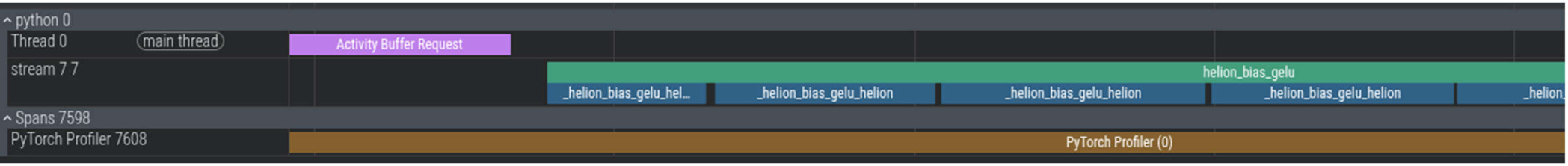
Tiling = Locality + Parallelism

- Locality: Reuse data
- Parallelism: Independent Tiles
- Core abstraction of GPU kernels

Helion Code

```python
def bias_gelu_eager(x: torch.Tensor, bias: torch.Tensor) -> torch.Tensor:
    return F.gelu(x + bias)

# Helion fused kernel
@helion.kernel()
def bias_gelu_helion(x: torch.Tensor, bias: torch.Tensor) -> torch.Tensor:
    # Host code: normal PyTorch, runs on CPU
    m, n = x.shape
    out = torch.empty_like(x)

    # Device code: compiled into one GPU kernel
    for tile_m, tile_n in hl.tile([m, n]):
        z = x[tile_m, tile_n] + bias[tile_n]
        out[tile_m, tile_n] = F.gelu(z)

    return out
```

Profile It



- eager : 0.137 ms
  - ~1.5 TFLOPs/s
- helion (fused): 0.083 ms
  - ~2.4 TFLOPs/s
- speedup : 1.64x

Key Takeaways

- GPUs are powerful but not magic
- Memory matters more than you think
- Optimize data movement
- Use fusion and tiling