⚡ Efficiency

🎯 Goals

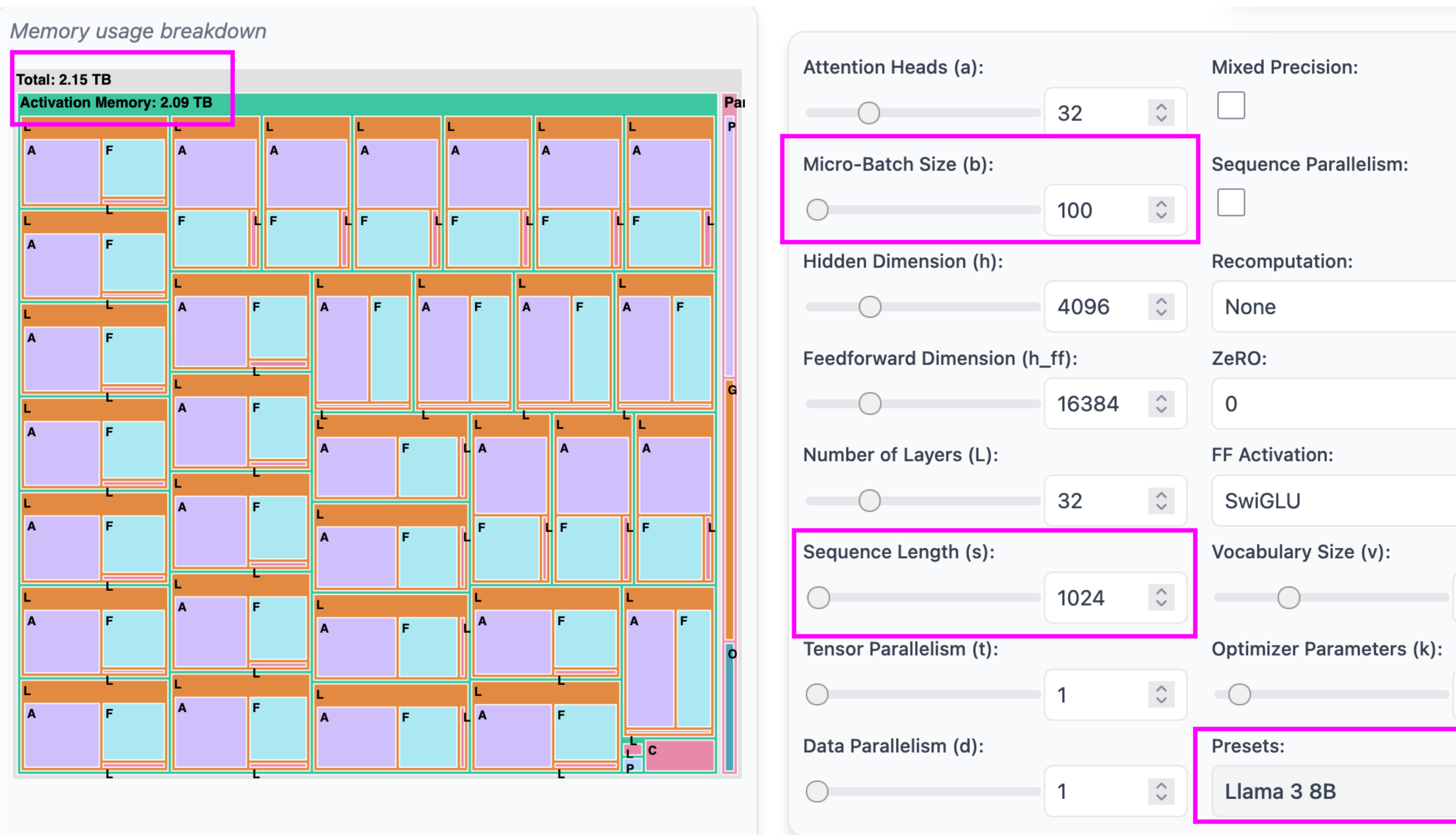
- Learn practical approaches to improve traini
  and efficiency
- Understand CUDA architecture fundamenta
  memory hierarchy
- Master GPU memory management and opti
  techniques

⚠️Problems

- **GPU Memory**
- **Training Speed** (Compute/Memory/Disk bottlenecks)

💾GPU Memory

- Good GPU memory size for NN training: **80G**
- **Llama3.1 8B: 1M** tokens is a good batch size
- 100M Batch size requires **2TB** GPU memory
  training realization) 🤔


Memory usage breakdown
Total: 2.15 TB
Activation Memory: 2.09 TB
Attention Heads (a):
32
Mixed Precision:
Micro-Batch Size (b):
100
Sequence Parallelism:
Hidden Dimension (h):
4096
Recomputation:
None
Feedforward Dimension (h_ff):
16384
ZeRO:
0
Number of Layers (L):
32
FF Activation:
SwiGLU
Sequence Length (s):
1024
Vocabulary Size (v):
Tensor Parallelism (t):
1
Optimizer Parameters (k):
Data Parallelism (d):
1
Presets:
Llama 3 8B

⏳Training Speed

💰1 GPU-hour = $2

Model	GPU-Hours
Llama3.1 8B	1.46M
Llama3.1 70B	7.0M
Llama3.1 405B	30.84M

1️⃣ Single-GPU Training

Feature	Training speed	Memory u
batch size	✅	✅
gradient accumulation	❌	✅
gradient checkpointing	❌	✅
mixed precision	✅❗	❌✅
optimizers	✅	✅
data preloading	✅	❌
torch_empty_cache_steps	❌	✅
torch.compile	✅	❌
PEFT	❌	✅
Efficient kernels	✅	✅

1️⃣ Single-GPU Training

📚 Resources:

- **HF GPU Perf Guide**

Batch size

- Why does it matter ❓ Why large batch size sa
computations ❓

Batch size

- Why does it matter ❓ Why large batch size sa
  computations ❓
  - 💡 GPUs are optimized for high parallelizat
  - 💡 Large batch size allows for more paralle
  - 💡 Small batch size requires more iteration
    converge.
  - 💡 Sometimes Model Params are larger the
    processed data. Model params loading mak
    💾 **Memory-bound** not 🧮 **Compute-bound**

Gradient Accumulatio

Idea: Split batch into smaller chunks and accum
gradients.

```
for i, batch in enumerate(dataloader):
    loss = model(batch)
    loss.backward()
    if (i + 1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

Gradient Accumulatio

Idea: Split batch into smaller chunks and accum
gradients.

HF Trainer:

```
training_args = TrainingArguments(
    ...
    gradient_accumulation_steps=4,
)
```

📸Gradient Checkpointin

Idea: Save memory by checkpointing intermedi
activations and recomputing them at backward

```
training_args = TrainingArguments(
    ...
    gradient_checkpointing=True,
)
```

❓Why do we need to save and even recompute
activations at backward pass?

🍸 Mixed Precision

Idea: Use lower precision for some model modu
speed up training.

torch.amp

```
with torch.autocast(device_type="cuda", dtype=torch.float1
    loss = model(batch)
    ...
```

HF Trainer:

fp16 / bf16

```
training_args = TrainingArguments(
    ...
    fp16=True,
    # bf16=True,
)
```

🍸 Automatic Mixed Precision

torch.amp

CUDA Ops that can autocast to float16

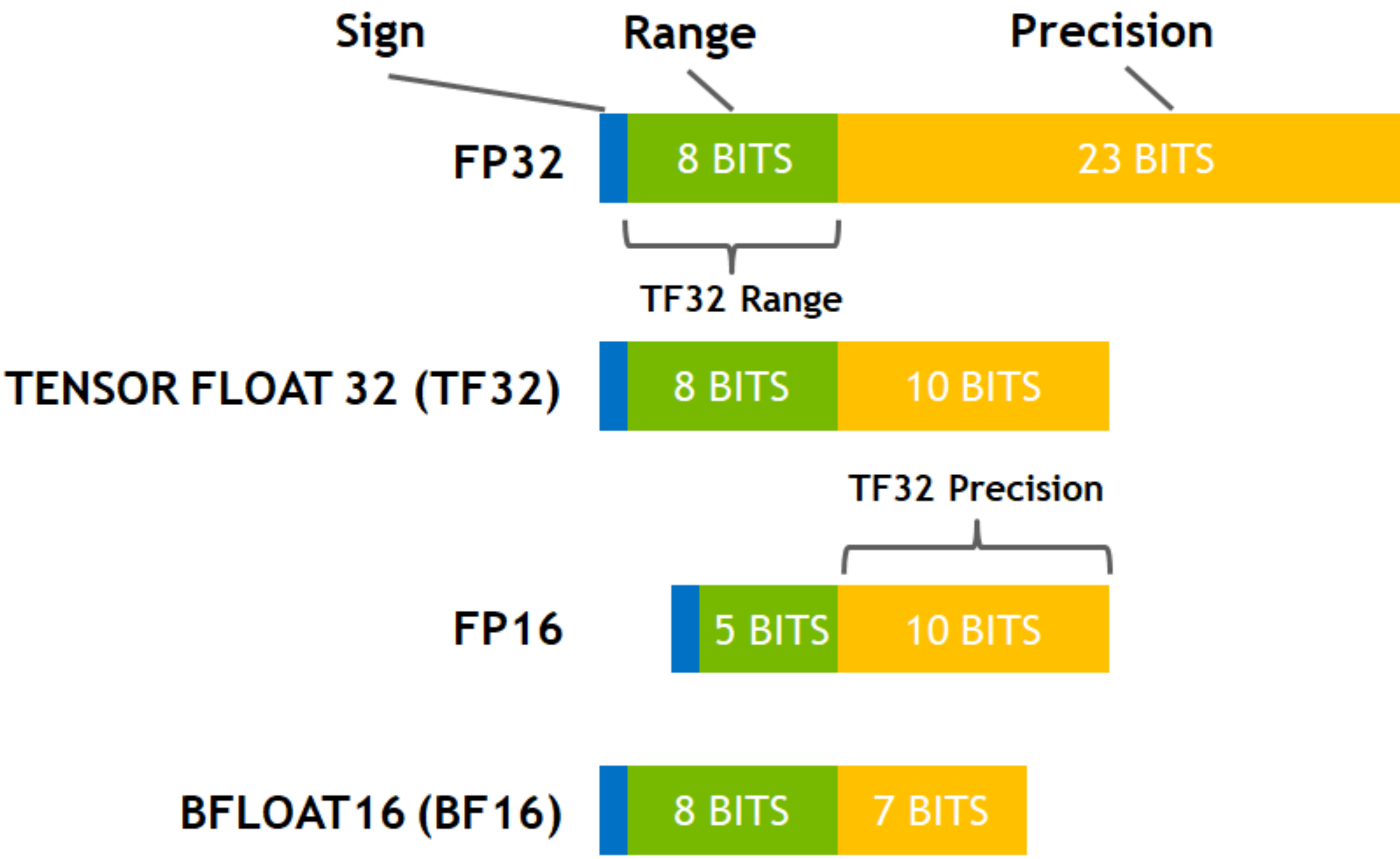
```
__matmul__, addbmm, addmm, addmv, addr,baddbmm, bmm, chair
multi_dot, conv1d, conv2d, conv3d, conv_transpose1d,
conv_transpose2d, conv_transpose3d, linear,
matmul, mm, mv,
... and more
```

CUDA Ops that can autocast to float32

```
__pow__, __rdiv__, __rpow__, __rtruediv__,
acos, asin, cosh,
binary_cross_entropy_with_logits, cosine_embedding_loss,
log, log_softmax, log10, log1p, log2,
mse_loss, multilabel_margin_loss, multi_margin_loss, l1_lo
norm, normalize,
... and more
```

🍸 Mixed Precision

- `float32` vs `float16` vs `bfloat16` vs `int8`
- Trade-offs: Precision, speed



🍸 Mixed Precision

❗ **Note:**

- bfloat16 issues - **bf16** works worse than **fp16**
  Llama3.1 models

🍸 Mixed Precision

Task 🤔

> *There is a set of floating-point numbers. We need to calculate the sum of these values.*

Naive approach

```
sum = 0
for value in values:
    sum += value
```

Any Problems ❓

🍸Mixed Precision

Task 🤔

Sort Values

```
sum = 0
for value in sorted(values):
    sum += value
```

🍸 Mixed Precision

Task 🤔

Maintain Sorted Order

```
import bisect

sum_keep_sorted_optim = 0
sorted_floats_optim = float_array.tolist()
for _ in tqdm(range(len(sorted_floats_optim) - 1)):
    sum_least_elements = sorted_floats_optim.pop(0) + sort
    bisect.insort_left(sorted_floats_optim, sum_least_elem
    if len(sorted_floats_optim) == 1:
        sum_keep_sorted_optim = sorted_floats_optim[0]
```

📈 Optimizers

- Adafactor - memory efficient optimizer
- 8-bit Adam - Keeps Quantized Weights in me

📦DataLoaders

- `num_workers`
  - Disk IO bottlenecks
- `prefetch_factor`
- `Data collators`
- Dataset caching and lazy loading
- Memory pinning

∅torch_empty_cache_st

- 💾Saves memory by clearing CUDA cache
- ❗ Slows down training (up to 2x)
- Why it slows down ❓

```
training_args = TrainingArguments(
    ...
    torch_empty_cache_steps=4,
)
```

⚒️torch.compile

JIT - Just-In-Time compilation

How it works:

- 💡Python-level tracing mechanism (TorchDy
- 💡Compiles captured operations to **fused CU kernels**
- 💡Caches compiled graphs
- 💡Reuses compiled graphs
  - If input shapes/types are the same

⚒️torch.compile

Pros:

- ✅Reduces GPU memory usage (why ❓)
- ✅Speeds up training

Cons:

- ❗ Graph compilation - first steps could be slo
- ❗ Graph recomputation if input shape chang

⚒️torch.compile

torch.compile

```
model = torch.compile(model)
```

HF Trainer:

```
training_args = TrainingArguments(
    ...
    torch_compile=True,
)
```

⚒️torch.compile

Resources:

- Look Ma, No Bubbles - single-kernel GPT
- torch.compile docs

PEFT

- Soft Prompts
- LoRA

🏷️PEFT: Soft Prompts

Feature	Prompt Tuning	Prefix Tuning	P-Tu
Affects All Transformer Layers?	❌No	✅Yes	✅Y
Learnable parameters type	Soft token embeddings	MLP over soft embeddings	MLI ove Tok

🏷️PEFT: Soft Prompts

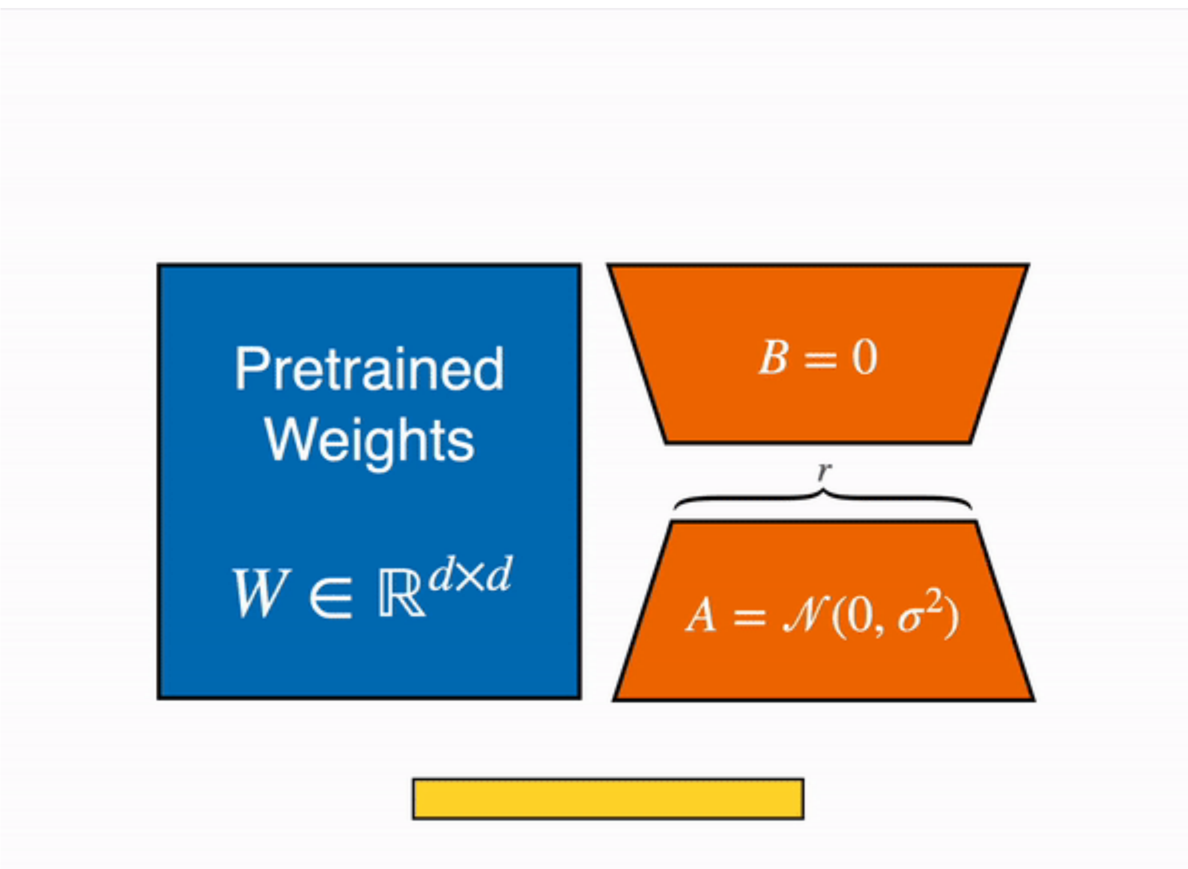
- What is the difference between Soft Prompt
  Hard Prompt ❓
- Why do we need MLP or LSTM over Soft Tok

PEFT: LoRA

Idea:

- Add trainable shift for MLPs output
- Trainable params should be small

PEFT: LoRA



Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$

🛑Implementation Efficie

The first rule of optimization: ***Don't do it***

🛑Implementation Efficie

The first rule of optimization: ***Don't do it***

- Are you sure your task was never solved befo

🛑Implementation Efficie

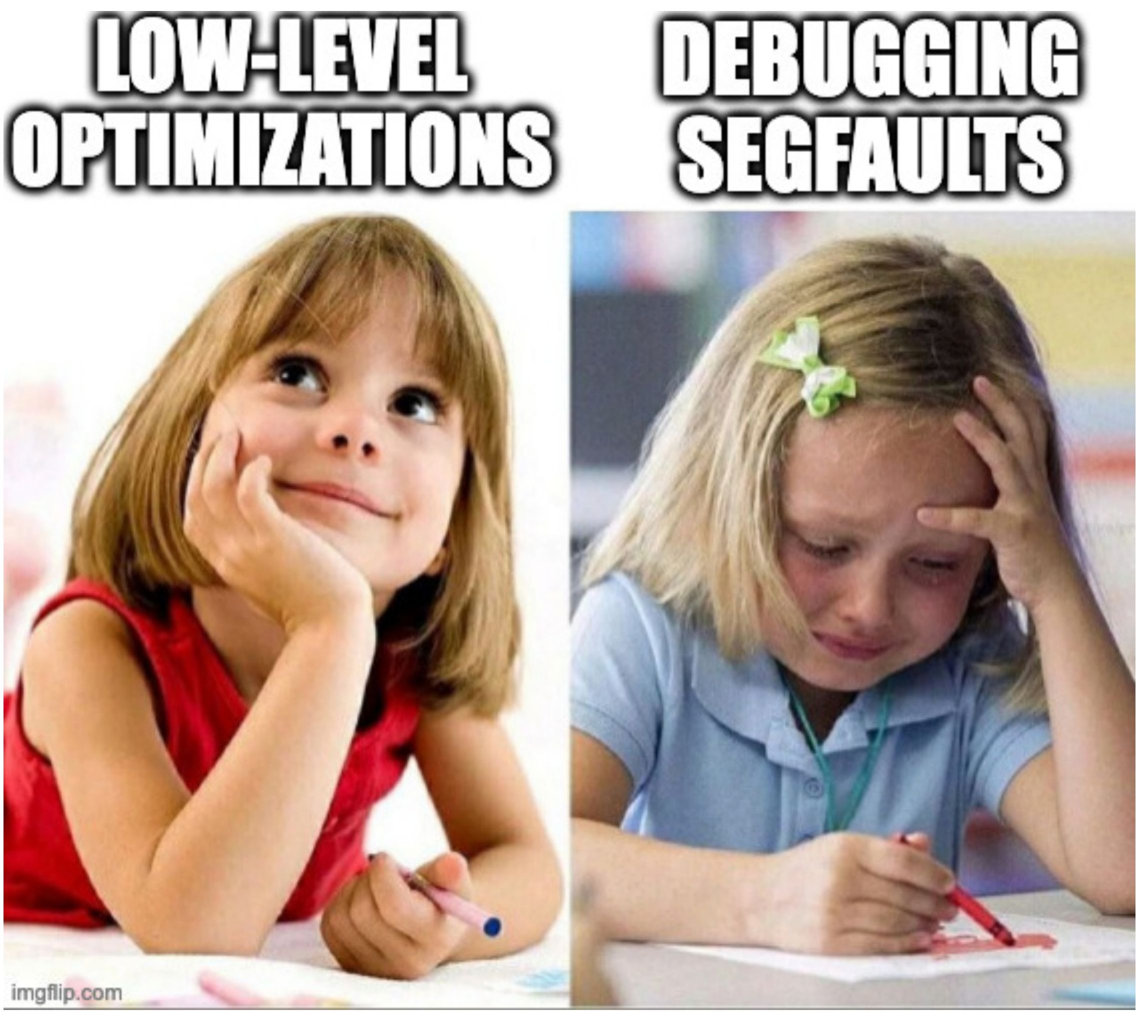
Low-level optimizations checklist:

- ✅Write dummy PyTorch implementation
- ✅Use `torch.compile`
- ✅Ask ChatGPT to optimize it
- 🛑Use Triton
- 🛑Use C++/CUDA

🛑Implementation Efficie

Still think you need to write C++/CUDA code ❓

🛑Implementation Efficie

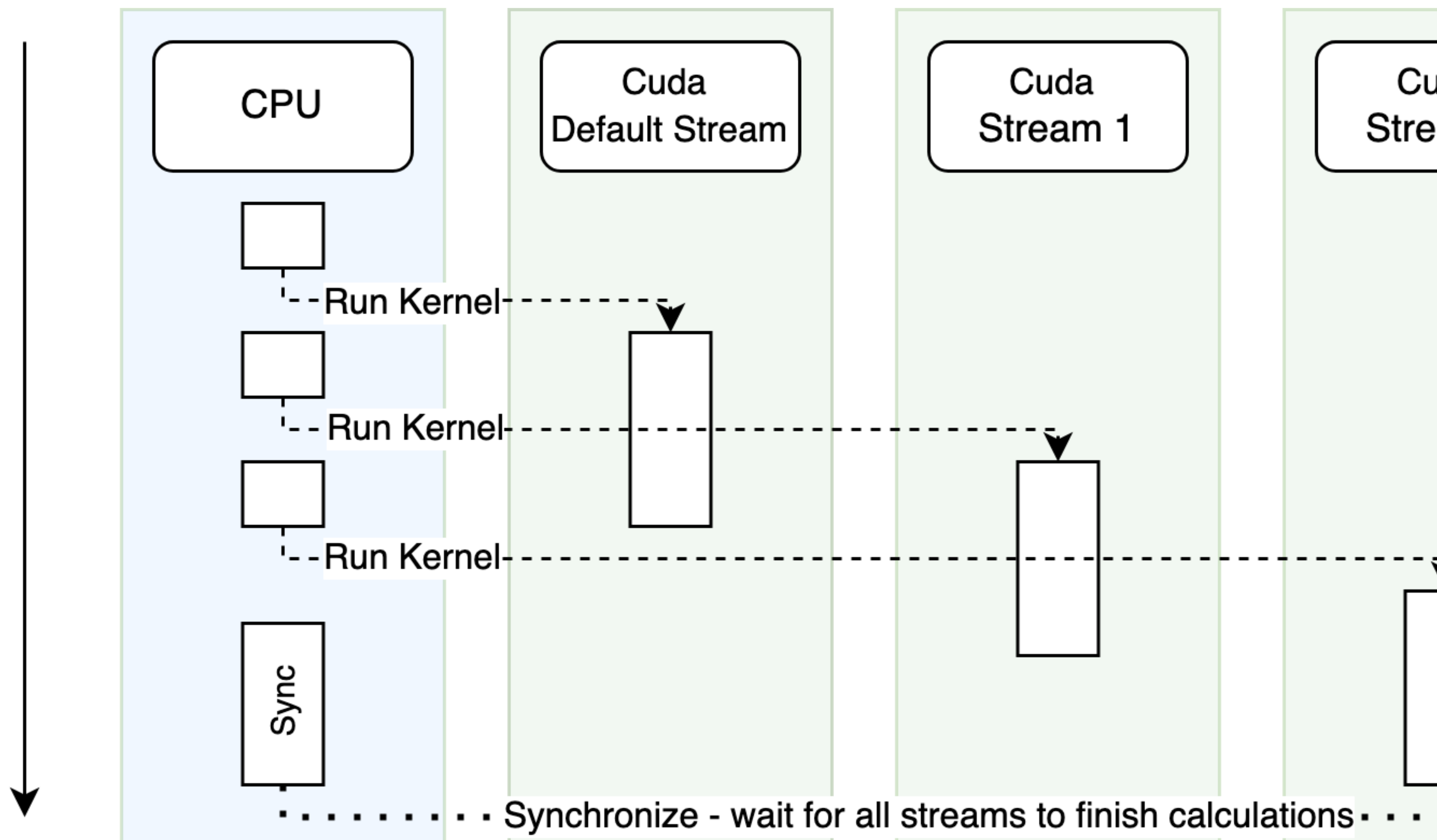


🛑Implementation Efficie

Topics:

- CUDA Async Nature
- CUDA Architecture
- CUDA Memory Hierarchy
- CUDA Kernel Life Cycle
- CUDA Kernel fusion
- CUDA kernels with Triton
- CUDA kernels with C++/CUDA

Cuda Async Nature



Resources:

- Asynchronous Execution (Torch Docs)

Cuda Async Nature

❗ This code measures only CPU time for kerne

```
start_time = time.time()
module.forward(x)
end_time = time.time()
print(f"Time taken: {end_time - start_time} seconds")
```

Cuda Async Nature

❗ use `torch.cuda.synchronize()`

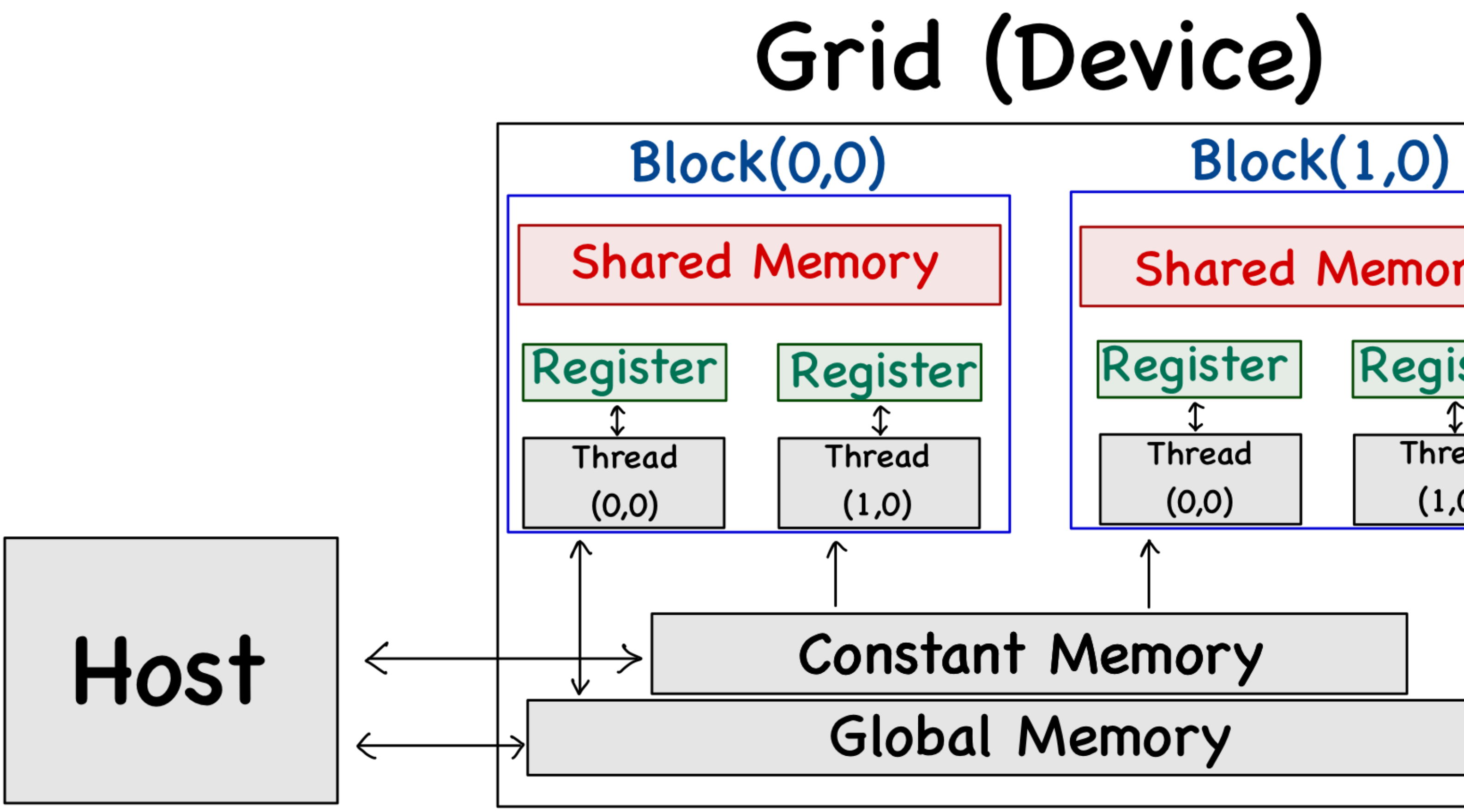
```
start_time = time.time()
module.forward(x)
torch.cuda.synchronize() # ❗NEW LINE❗
end_time = time.time()
print(f"Time taken: {end_time - start_time} seconds")
```

Cuda Async Nature

- When do we need to run async streams ❓
- Could we benefit from async streams in case parallel compute-bound tasks that utilizes all cores ❓

CUDA Architecture

- Grid > Block > Thread



CUDA Architecture

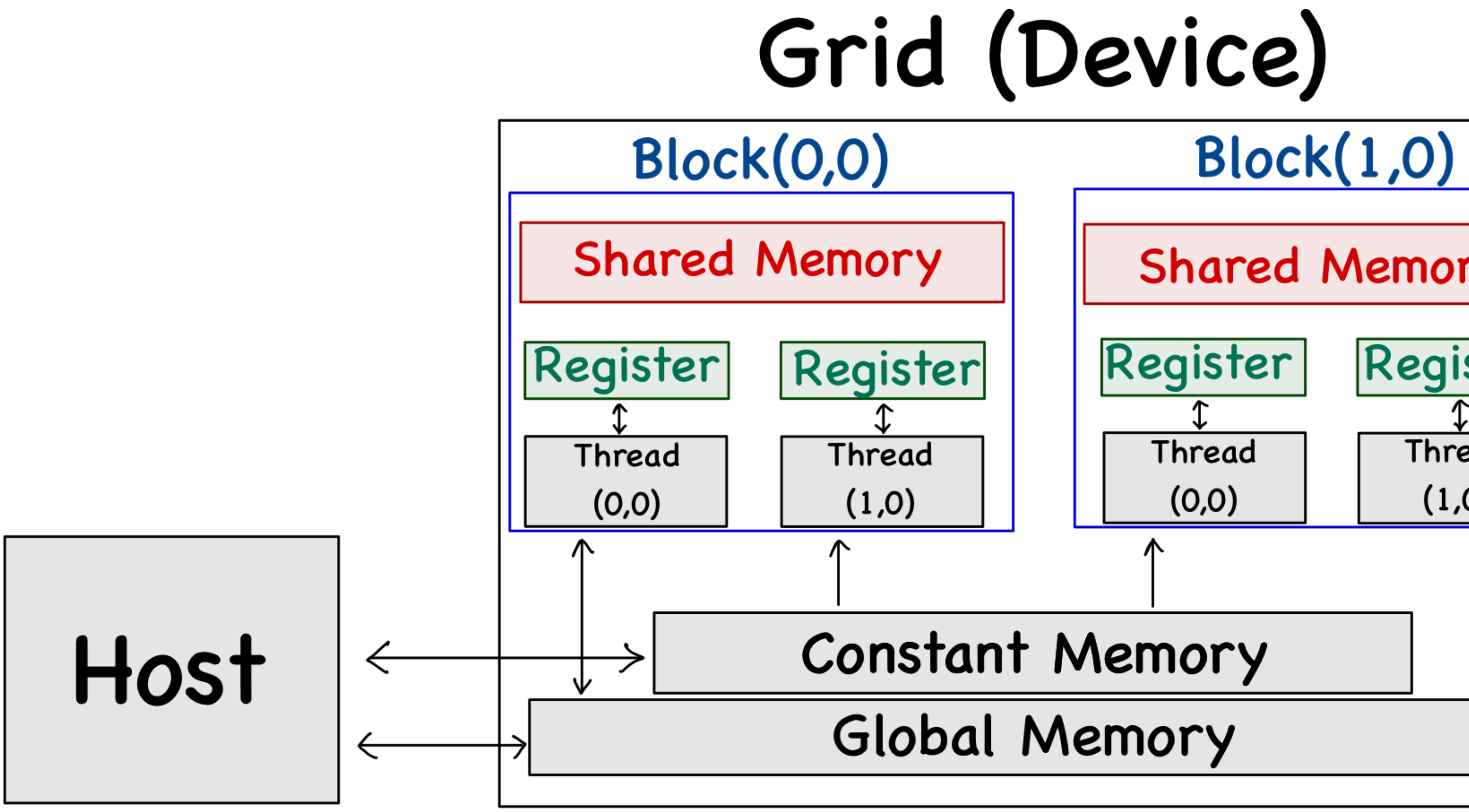
Level	What It Is	Can Communicate?
Thread	Basic execution unit	With other threads in b
Block	Group of threads	Within the block only
Grid	Group of blocks	No direct communicati

Resources:

- **CUDA Programming Guide**
- **GPU Compute and Memory Architecture**

CUDA Memory Hierarchy

- Off-chip memory
- On-chip memory



Grid (Device)
Block(0,0)
Shared Memory
Register
Register
Thread
(0,0)
Thread
(1,0)
Block(1,0)
Shared Memor
Register
Regis
Thread
(0,0)
Thre
(1,0
Host
Constant Memory
Global Memory

CUDA Memory Hierarchy

Off-chip memory

Memory Type	Size	Latency / Bandwidth	Notes
Global Memory	~80 GB	High latency, low bandwidth	Main memory fo data; read/write
Local Memory	Per-thread	High latency	Private memory a thread; stored i memory
Constant Memory	~64 KB	Low latency, high bandwidth	Read-only on GP optimized via cac

CUDA Memory Hierarchy

On-chip memory

Memory Type	Size (typical)	Latency / Bandwidth	Notes
Shared Memory	~16 KB / SM	Low latency, high bandwidth	Enables inter-th communication
Registers	~8 KB / SM	Very low latency	Fastest memory thread-local var

CUDA Memory Hierarchy

- How should we use this knowledge ❓
- Is there any parallels with CPU memory hiera

CUDA Memory Hierarchy

Outlines

- **Off-chip memory** → larger but slower.
- **On-chip memory** → faster but very limited.
- **Shared memory** allows collaboration within a
  while **global memory** is accessible across blo

CUDA Kernel Life Cycle

Stage	Where it Happens	Output For
Writing Kernel Code	Developer (CPU)	.cu file
Compiling	CPU	Host obj + P
Loading	CPU → GPU	PTX/SASS
Execution	GPU	Running ke
Data Handling	CPU ↔ GPU	Raw memo

CUDA Kernel Fusion

What is kernel fusion ❓

Original:

```
__global__ void kernelA(...) {  }
__global__ void kernelB(...) { ... }
```

Fused:

```
__global__ void fusedKernel(...) {
    // do A's work
    // do B's work
}
```

CUDA Kernel Fusion

Benefit	Explanation
Less kernel launch overhead	Reduces CPU-GPU sync and s
Better memory locality	Intermediate data kept in fast
Less global memory traffic	Avoids slow reads/writes to D

😢Out of scope:

- **Extreme low-bit quantization**
  - **AWQ**
  - **Quantization-aware training**

- **Distributed training**
  - **HF Distributed Training**
  - **HF Perf Training Many GPUs**
  - **Ultrascale Playbook**

- **Pytorch internals**
  - **Pytorch Internals**
  - **A Tour of Pytorch Internals**