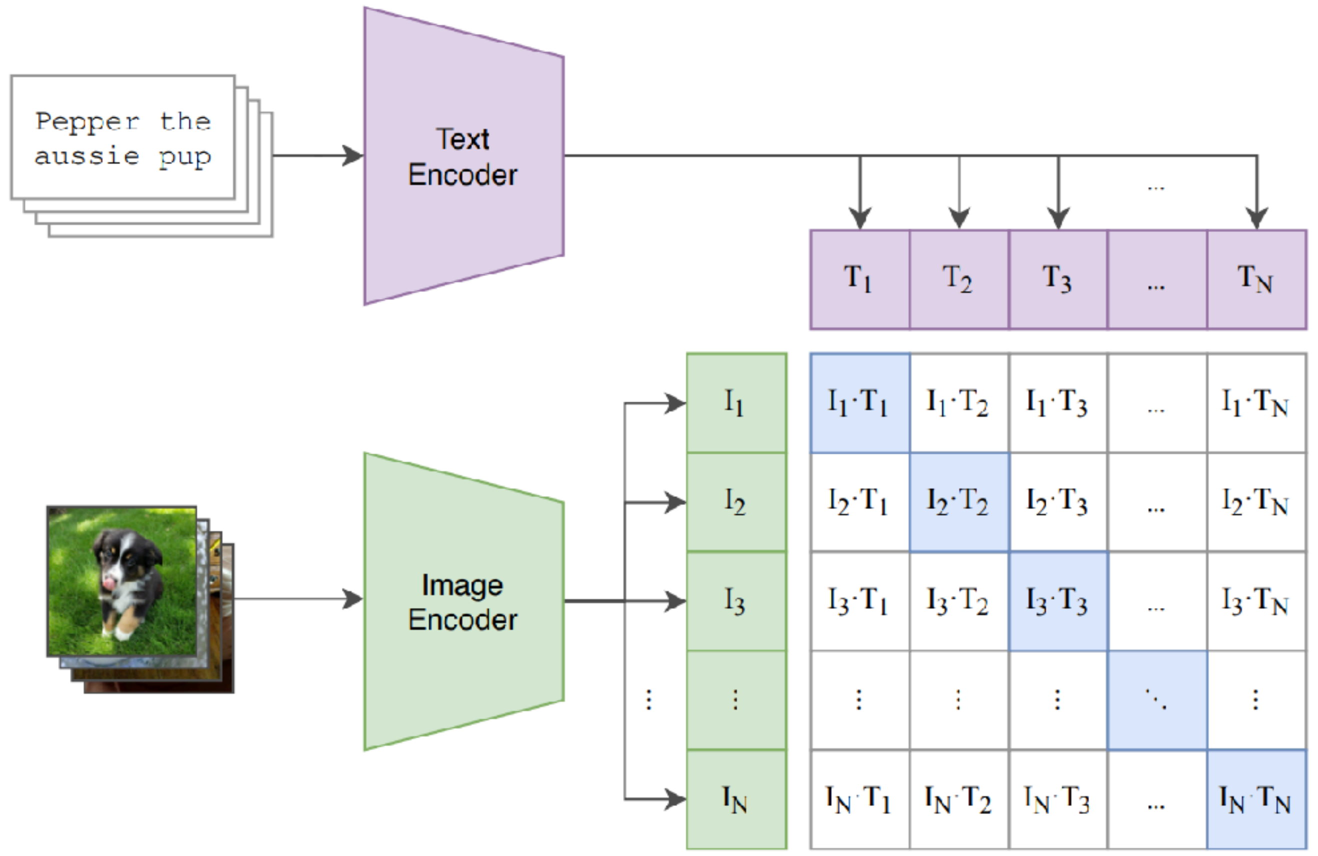
LLM Sys

Accelerating Transformer Training and Inference

Lei Li

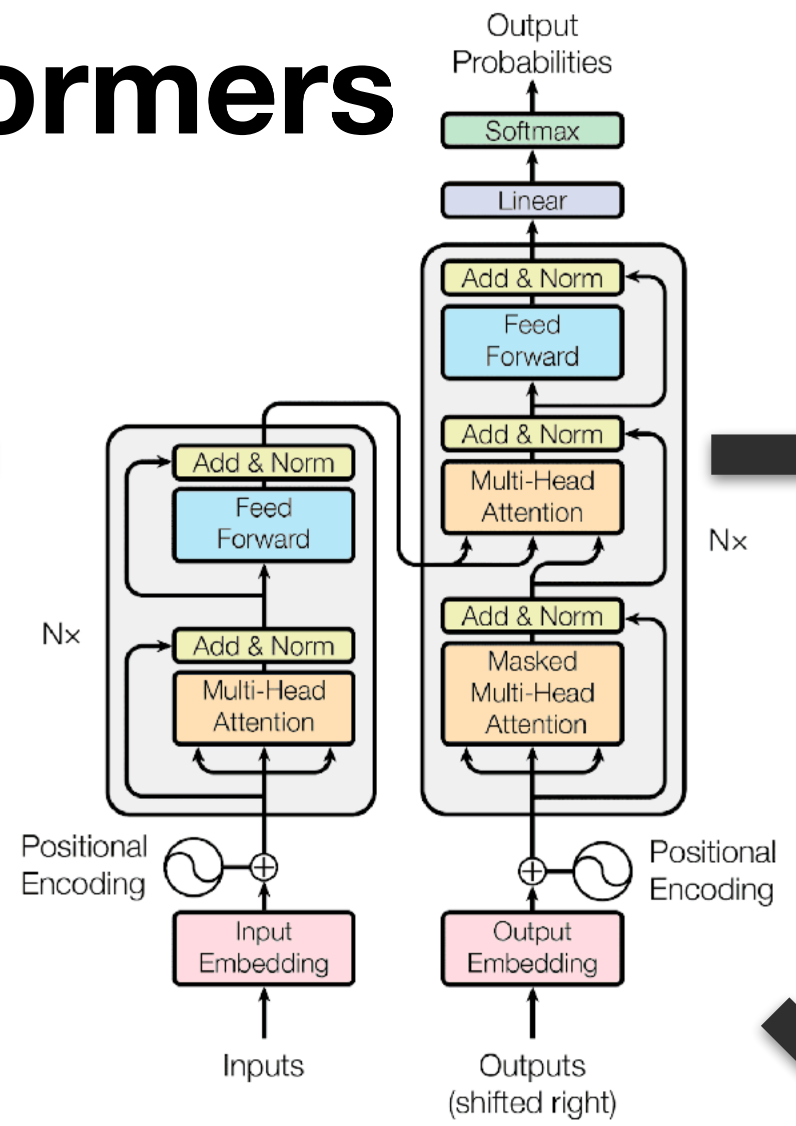
Carnegie Mellon University
Language Technologies Institute

Transformer Models as universal architecture



Natural Language Supervision for Vision (CLIP)

Transformers



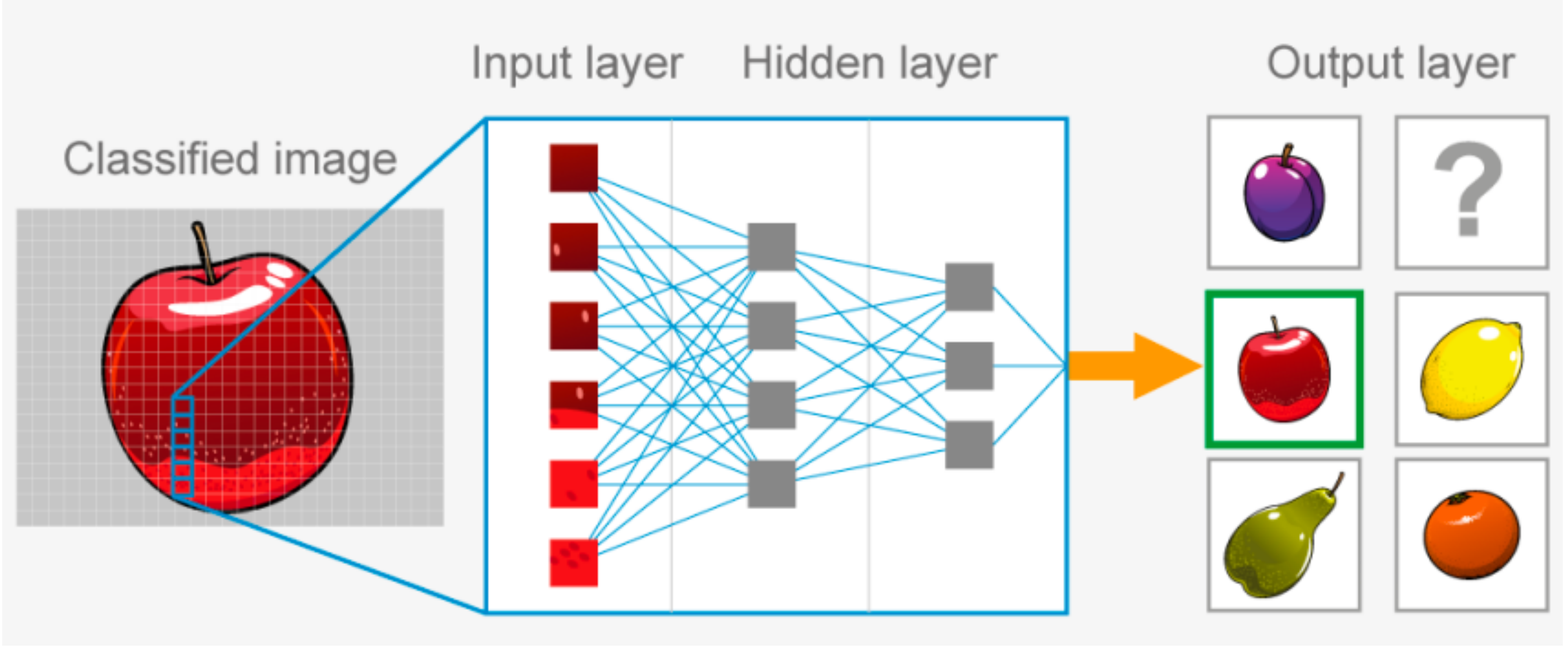
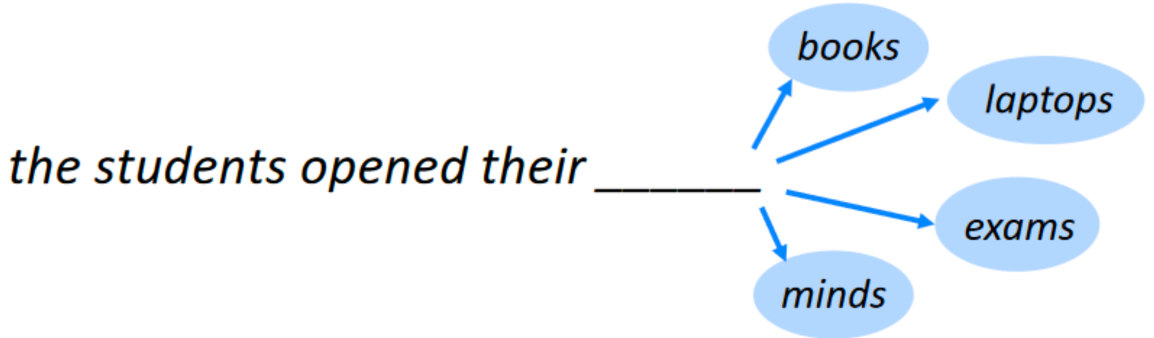


Image Classification (Vision Transformer, Swin-Transformer)

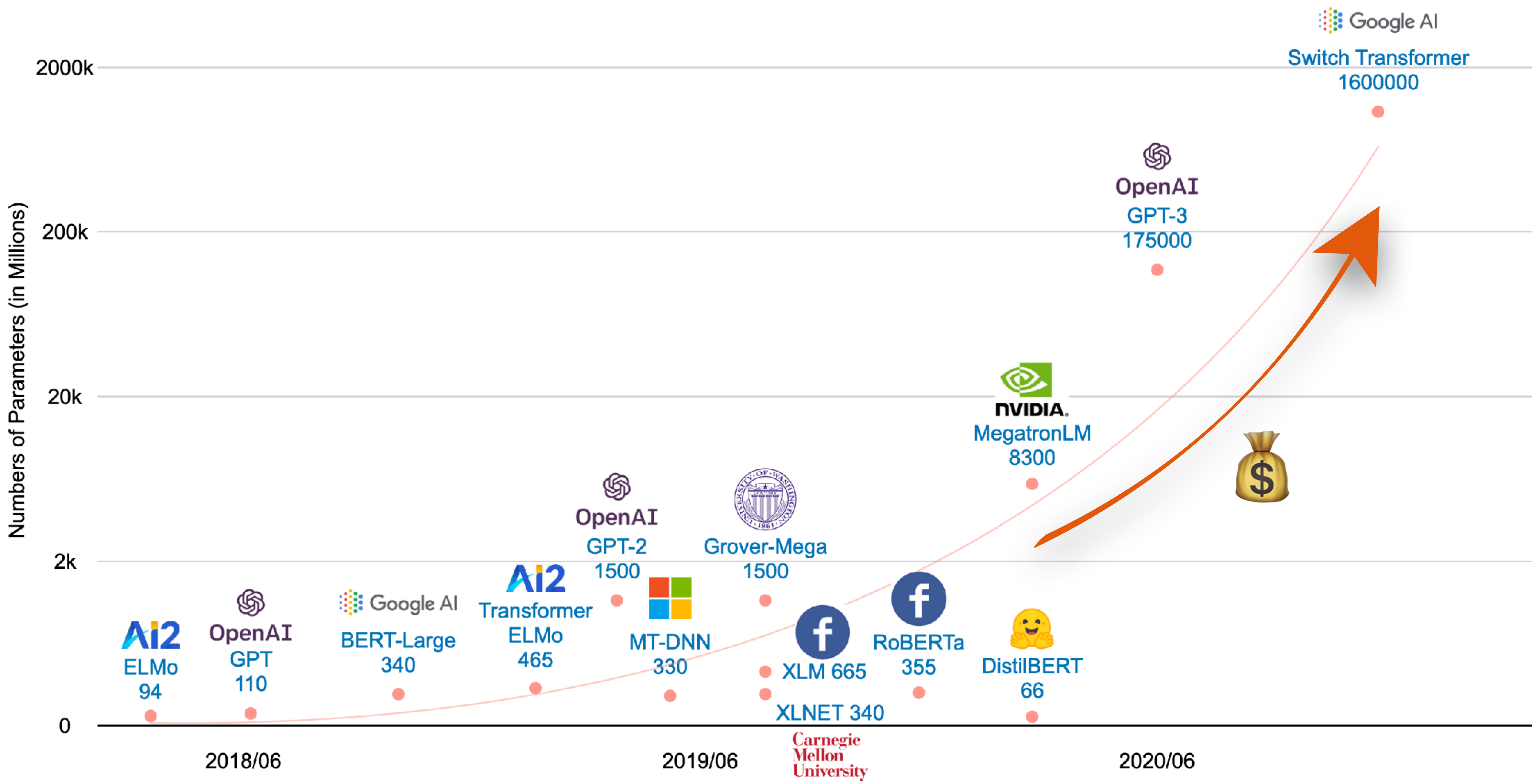


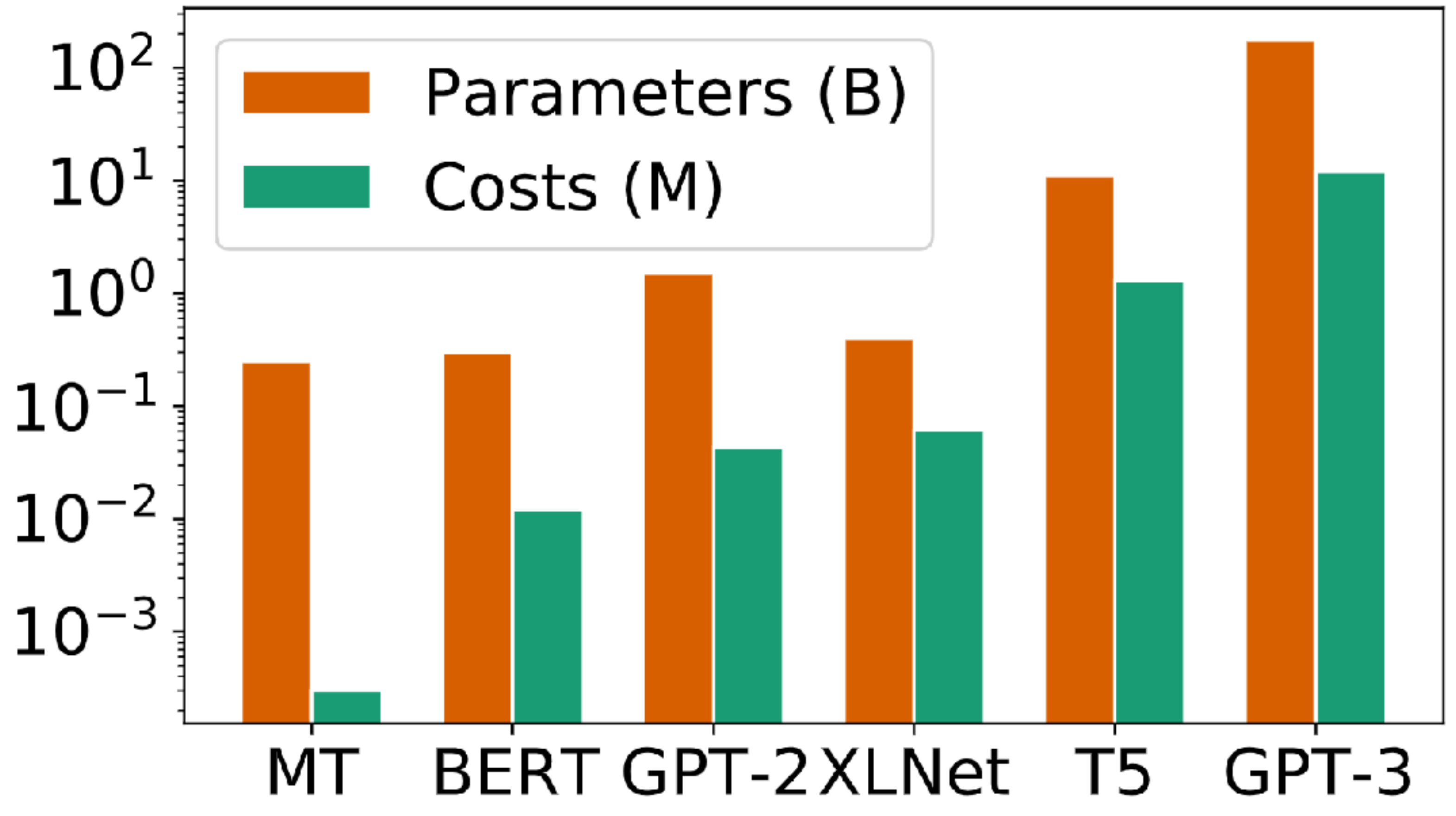
Language Model (BERT, T5, GPT3/4)

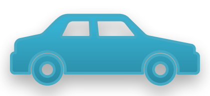
Text to Image (Stable Diffision)

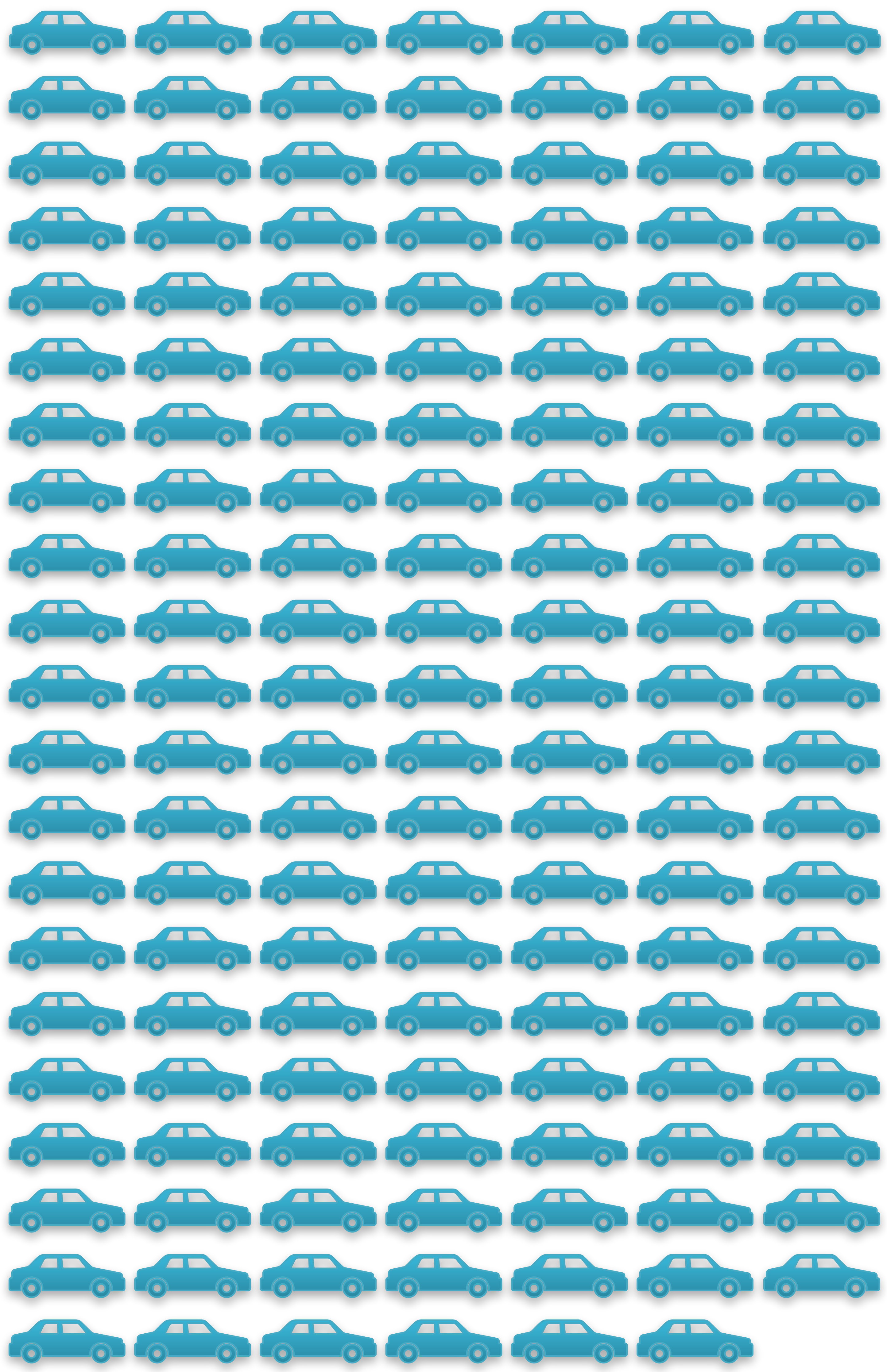
Speech Recognition (wav2vec, HuBERT)

Training Large Models Are Expensive!



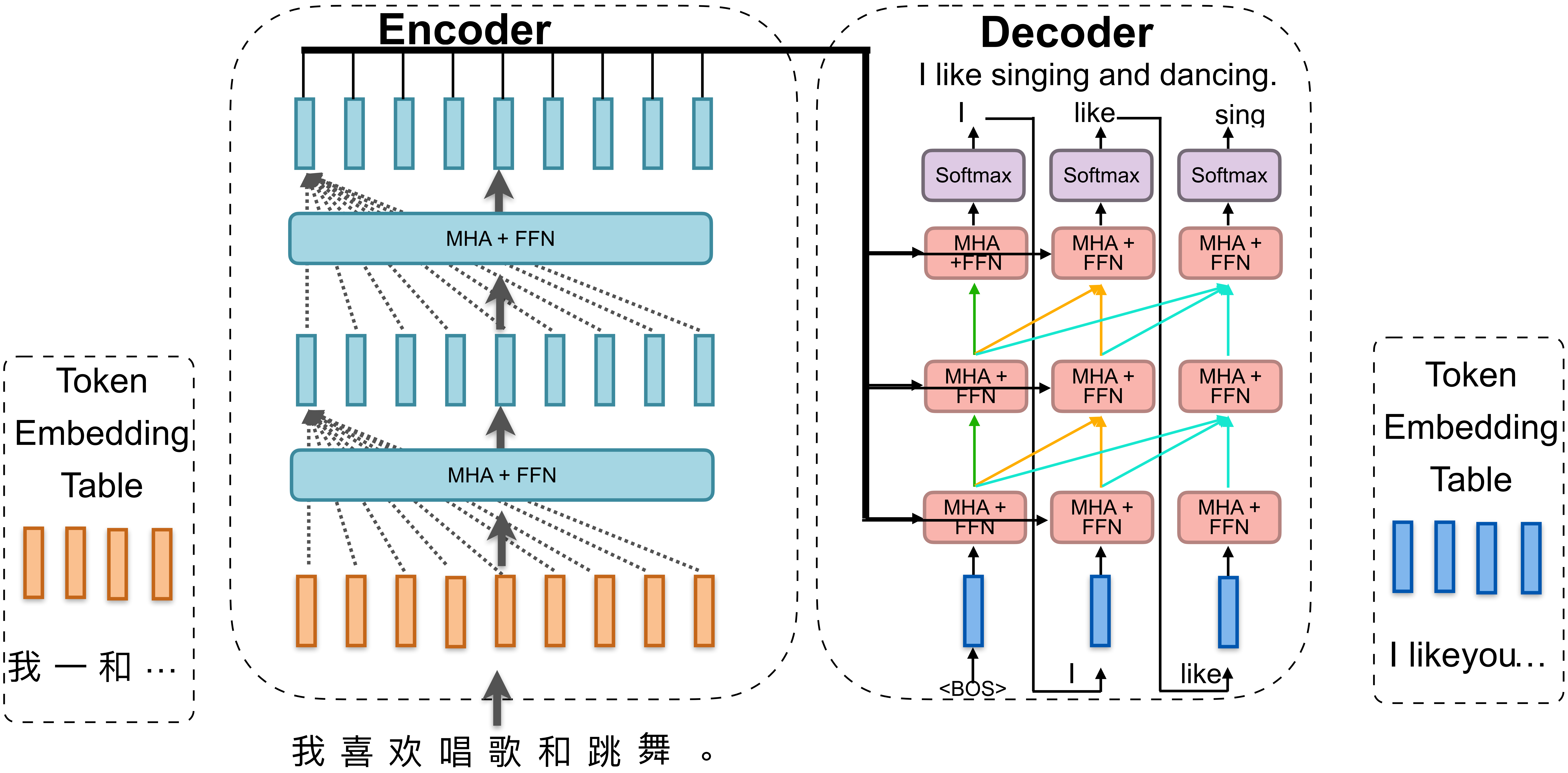


 = 1 Car Year CO2



Carbon footprint: Training GPT3 = driving a car for 146 years!

Recap Transformer Architecture



Transformer Training Stages

Synchronization

Aggregated Gradients

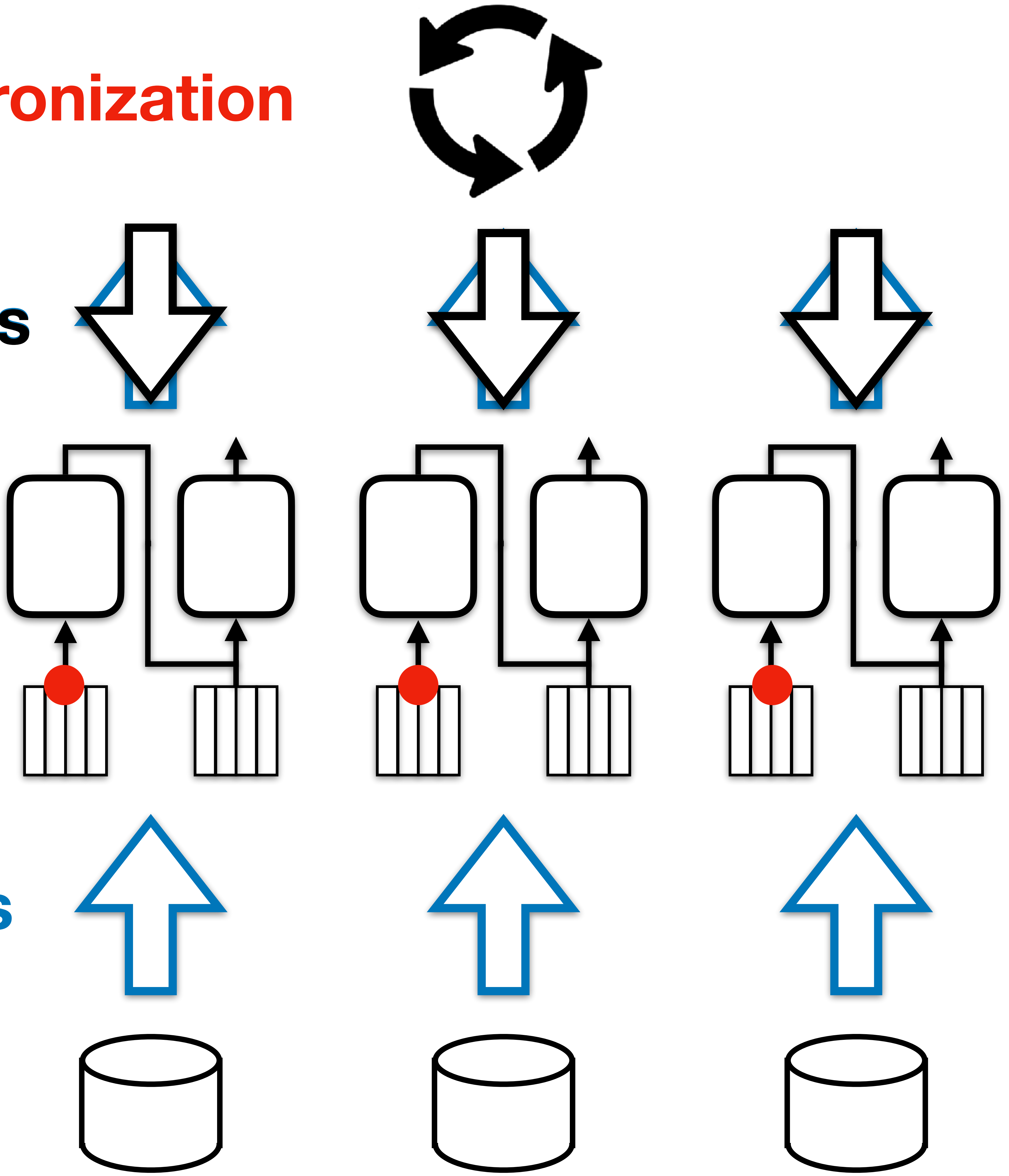
Model Replicas

Backward Forward

Update Parameters

Samples

Data Shards



This Lecture
Accelerated GPU Computation for Transformer Training

for moderate model size (< GPU memory)

based on LightSeq library

LightSeq: A High Performance Inference Library for Transformers. Wang et al 2021.
LightSeq2: Accelerated Training for Transformer-based Models on GPUs. Wang et al 2022.
TensorRT-LLM (FasterTransformer), only for inference

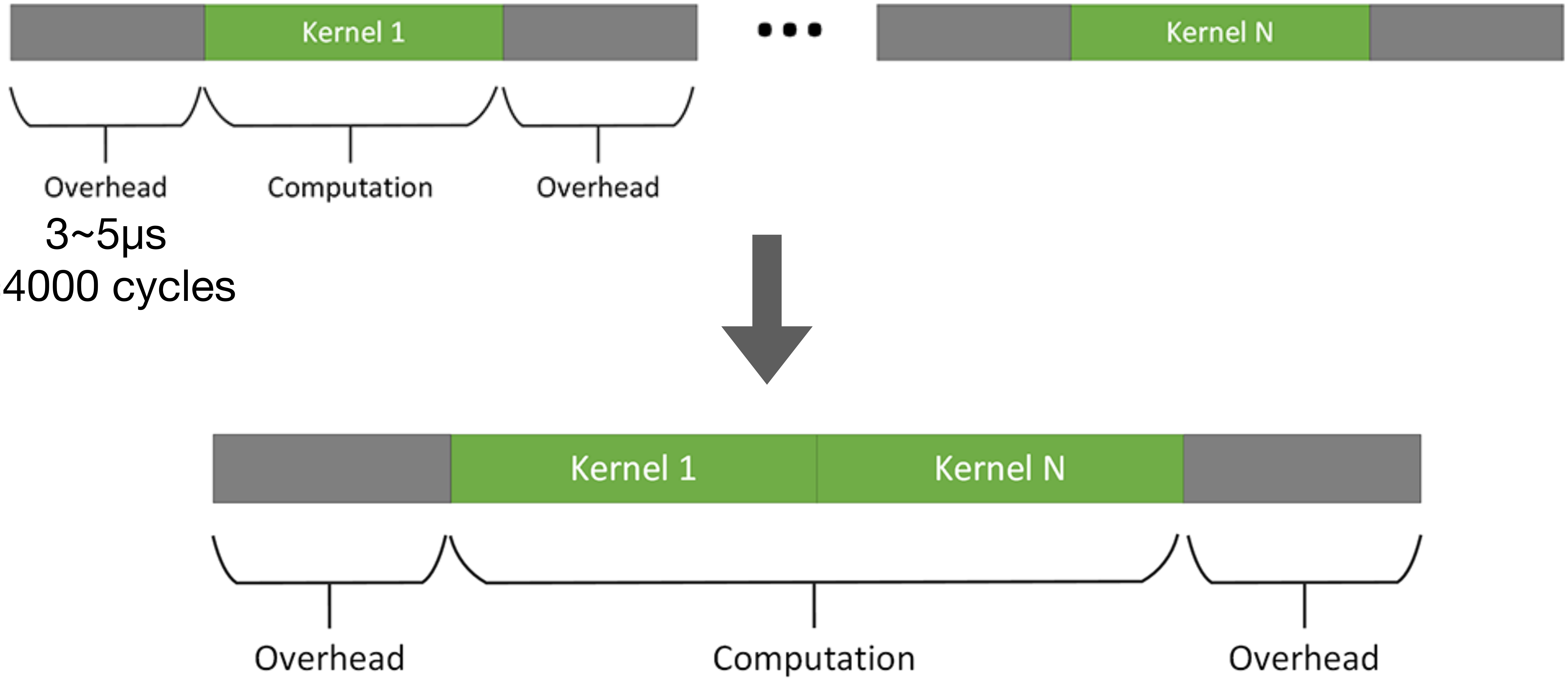
Comparison of Acceleration Libraries for Transformers

	Full Transformer	Training	Inference	PyTorch	Tensorflow
TensorRT-LLM	✔	✘	✔	✔	✔
TurboTransformers	✔	✘	✔	✔	?
DeepSpeed	✔*	✔	✔	✔	✘
LightSeq	✔	✔	✔	✔	✔

*DeepSpeed implemented Transformer Kernel in Oct 2022

LightSeq/LightSeq2 Optimization Overview

Technique 1: Kernel Fusion



3~5μs
=4000 cycles

Kernel Fusion Example

$C = A + B$

$E = C + D$

4 load, 2 stores, 2 matrix add.

needs two kernel executions.

If we write a custom kernel to add three matrices

$E = A + B + C$

3 load, 1 store, 2 matrix add.

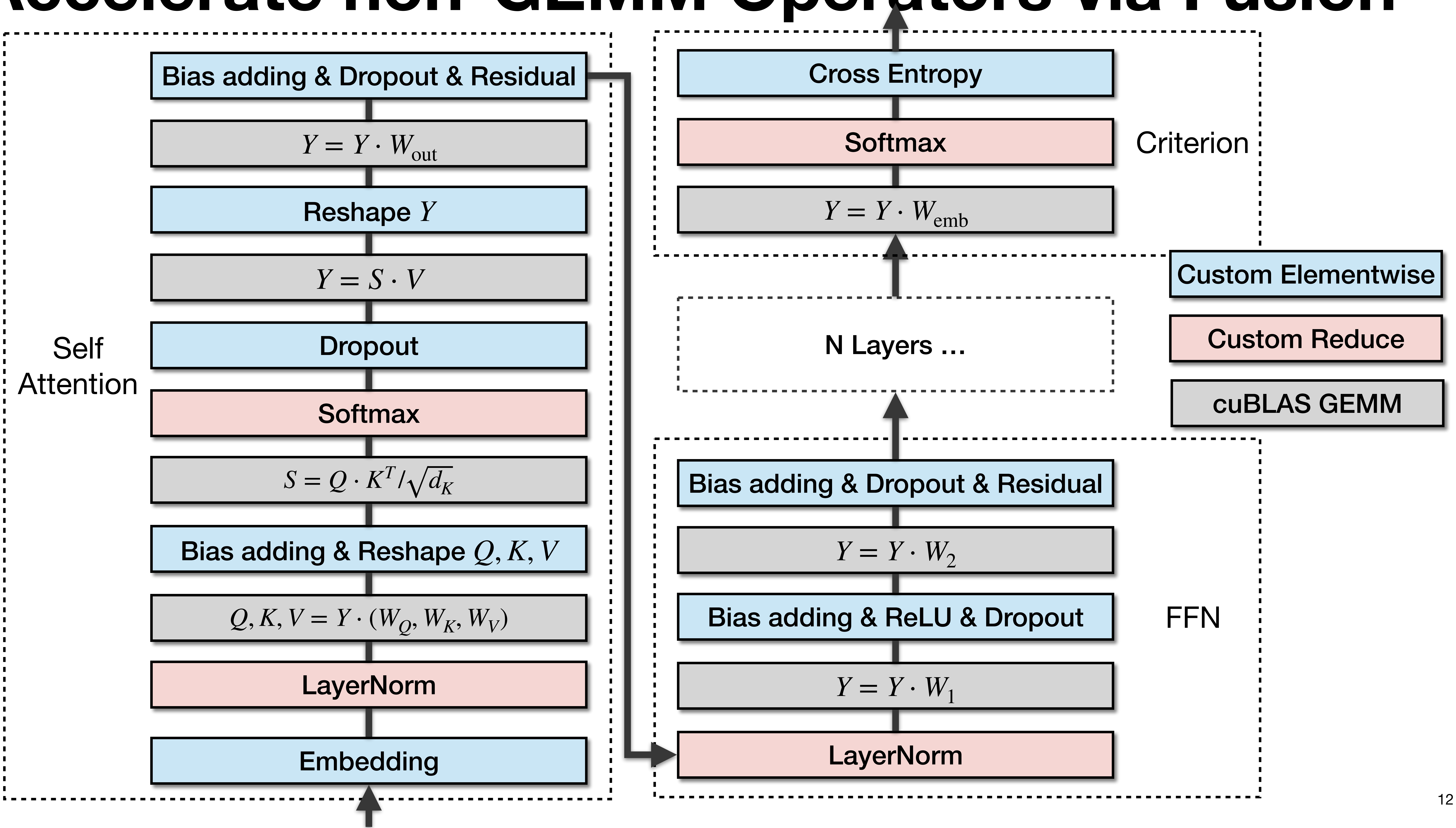
only needs one kernel.

Benefits:

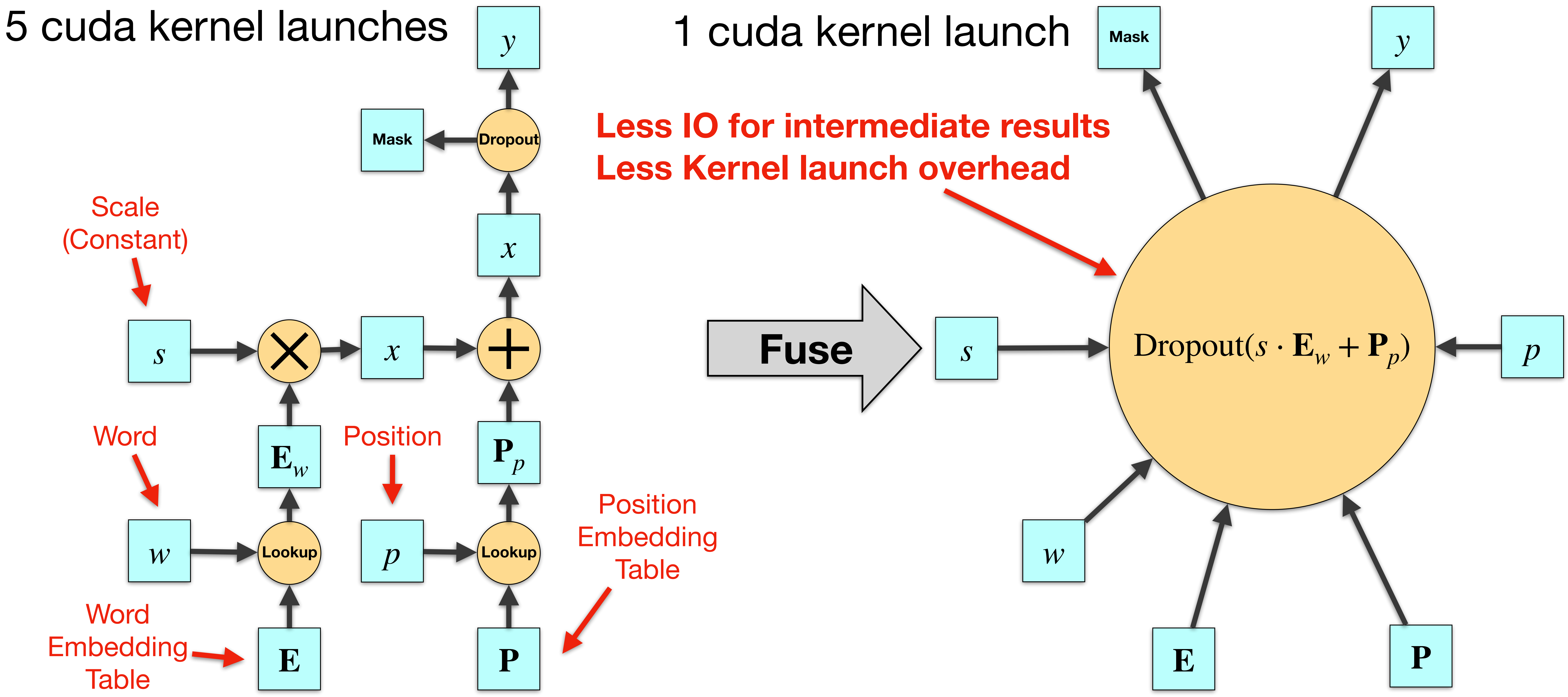
- reduce overhead
- reduce extra memory access

MultiHead Attention And Feed Forward Network

Accelerate non-GEMM Operators via Fusion

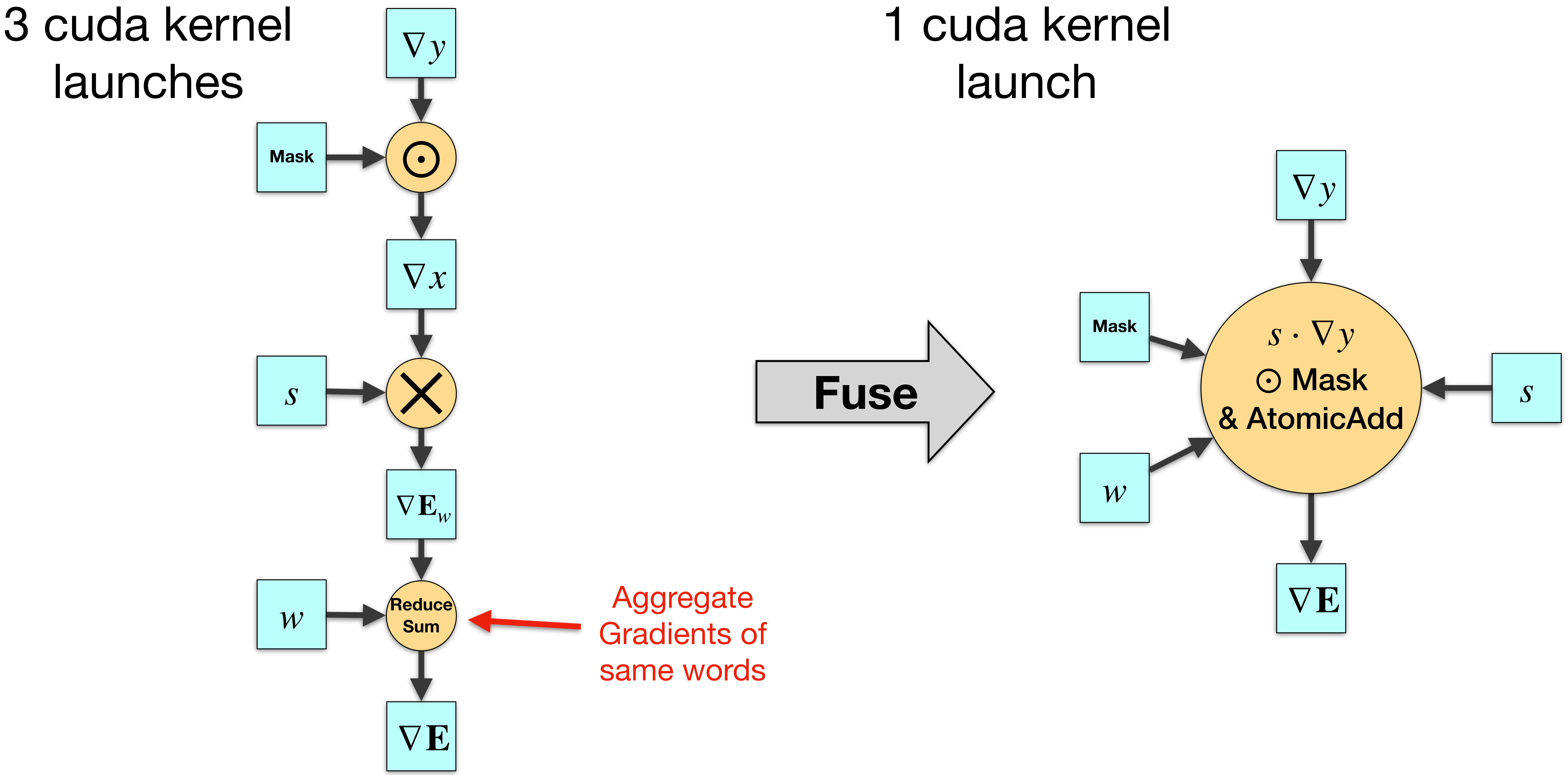


Fused Embedding Forward Operator



$$y = \text{Dropout}(s \cdot \mathbf{E}_w + \mathbf{P}_p)$$

Fused Embedding Backward Operator



$$\nabla \mathbf{E} = \mathrm{ReduceSum}(s \cdot \nabla y \odot \mathrm{Mask})$$

Code Example: Embedding Forward

```
__global__ void lookup_scale_pos_dropout<float>(
    float *output, const int *input, const int *tokens_position,
    const float *embeddings, const float *pos_embeddings, const float *cl
    uint8_t *dropout_mask, int seq_len, int embedding_dim, int padding_id
    float dropout_ratio, float emb_scale, int step, int seed) {
  int batch_id = blockIdx.x;
  int seq_id = blockIdx.y * blockDim.x + threadIdx.x;
  if (seq_id >= seq_len) return;

  int target_pos = batch_id * seq_len + seq_id;
  int start = target_pos * embedding_dim + threadIdx.y;
  int end = (target_pos + 1) * embedding_dim;
  int tid = input[target_pos];

  int token_pos_id = tokens_position[target_pos];

  float4 *output4 = reinterpret_cast<float4 *>(output);
  const float4 *embeddings4 = reinterpret_cast<const float4 *>(embeddings
  const float4 *pos_embeddings4 =
      reinterpret_cast<const float4 *>(pos_embeddings);
  uint32_t *dropout_mask4 = reinterpret_cast<uint32_t *>(dropout_mask);

  // no need to calculate dropout_mask
  if (tid == padding_idx) {
    float4 zero4;
    zero4.x = zero4.y = zero4.z = zero4.w = 0.f;
    for (uint i = start; i < end; i += blockDim.y) {
      output4[i] = zero4;
    }
    return;
  }

  const float dropout_scale = 1.f / (1.f - dropout_ratio);
  float clip_max_val;
  if (clip_max) {
    clip_max_val = clip_max[0];
  }
  curandStatePhilox4_32_10_t state;
```

Word Embedding Lookup

Positional Embedding Lookup

```
  for (uint i = start; i < end; i += blockDim.y) {
    curand_init(seed, i, 0, &state);
    float4 rand4 = curand_uniform4(&state);
    uint8_t m[4];
    // dropout mask
    m[0] = (uint8_t)(rand4.x > dropout_ratio);
    m[1] = (uint8_t)(rand4.y > dropout_ratio);
    m[2] = (uint8_t)(rand4.z > dropout_ratio);
    m[3] = (uint8_t)(rand4.w > dropout_ratio);

    int offset = i - target_pos * embedding_dim;
    // step is non-zero only in inference
    float4 e4 = embeddings4[tid * embedding_dim + offset];
    float4 pe4 =
        pos_embeddings4[(token_pos_id + step) * embedding_dim + offset]
    float4 res4;

    float scale_mask[4];
    scale_mask[0] = dropout_scale * m[0];
    scale_mask[1] = dropout_scale * m[1];
    scale_mask[2] = dropout_scale * m[2];
    scale_mask[3] = dropout_scale * m[3];

    uint8_t clip_mask[4];
    if (clip_max) {
      e4.x = fake_quantize(e4.x, clip_max_val, clip_mask[0], 2);
      e4.y = fake_quantize(e4.y, clip_max_val, clip_mask[1], 2);
      e4.z = fake_quantize(e4.z, clip_max_val, clip_mask[2], 2);
      e4.w = fake_quantize(e4.w, clip_max_val, clip_mask[3], 2);
    }
    res4.x = (emb_scale * e4.x + pe4.x) * scale_mask[0];
    res4.y = (emb_scale * e4.y + pe4.y) * scale_mask[1];
    res4.z = (emb_scale * e4.z + pe4.z) * scale_mask[2];
    res4.w = (emb_scale * e4.w + pe4.w) * scale_mask[3];

    output4[i] = res4;
    uint32_t *m4 = reinterpret_cast<uint32_t *>(m);
    if (clip_max) {
      m4[0] = m4[0] | reinterpret_cast<uint32_t *>(clip_mask)[0];
    }
    dropout_mask4[i] = m4[0];
  }
}
```

Dropout mask

Apply dropout

Scale

https://github.com/bytedance/lightseq/blob/master/lightseq/csrc/kernels/cuda/embedding_kernels.cu

Code Example: Embedding Backward

```
__global__ void d_lookup_scale_pos_dropout<float>(
    float *grad_embeddings, float *grad_clip_max, const float *grad_output,
    const int *input, const uint8_t *dropout_mask, int seq_len,
    int embedding_dim, int padding_idx, float dropout_ratio, float emb_scale) {
  int batch_id = blockIdx.x;
  int seq_id = blockIdx.y * blockDim.x + threadIdx.x;
  if (seq_id >= seq_len) return;

  int target_pos = batch_id * seq_len + seq_id;
  int start = target_pos * embedding_dim + threadIdx.y;
  int end = (target_pos + 1) * embedding_dim;
  int tid = input[target_pos];

  if (tid == padding_idx) {
    return;
  }

  const float scale = 1.f / (1.f - dropout_ratio);
  const float4 *grad_output4 = reinterpret_cast<const float4 *>(grad_output);
  const uint32_t *dropout_mask4 =
      reinterpret_cast<const uint32_t *>(dropout_mask);
  // float block_g_clip_max = 0;
  float thread_cmax_grad = 0;
  float temp_cmax_grad = 0;
```

```
for (uint i = start; i < end; i += blockDim.y) {
  float4 go4 = grad_output4[i];
  uint32_t m4 = dropout_mask4[i];
  uint8_t *m4_ptr = reinterpret_cast<uint8_t *>(&m4);
  float4 res4;
  res4.x = emb_scale * go4.x * (m4_ptr[0] & 1) * scale;
  res4.y = emb_scale * go4.y * (m4_ptr[1] & 1) * scale;
  res4.z = emb_scale * go4.z * (m4_ptr[2] & 1) * scale;
  res4.w = emb_scale * go4.w * (m4_ptr[3] & 1) * scale;
  int offset = i - target_pos * embedding_dim;
  int idx = (tid * (embedding_dim) + offset) << 2;
  clip_bwd(res4.x, temp_cmax_grad, res4.x, m4_ptr[0], 2);
  thread_cmax_grad += temp_cmax_grad;
  clip_bwd(res4.y, temp_cmax_grad, res4.y, m4_ptr[1], 2);
  thread_cmax_grad += temp_cmax_grad;
  clip_bwd(res4.z, temp_cmax_grad, res4.z, m4_ptr[2], 2);
  thread_cmax_grad += temp_cmax_grad;
  clip_bwd(res4.w, temp_cmax_grad, res4.w, m4_ptr[3], 2);
  thread_cmax_grad += temp_cmax_grad;
  atomicAdd(grad_embeddings + idx, res4.x);
  atomicAdd(grad_embeddings + idx + 1, res4.y);
  atomicAdd(grad_embeddings + idx + 2, res4.z);
  atomicAdd(grad_embeddings + idx + 3, res4.w);
}

if (grad_clip_max) {
  __shared__ float block_cmax_grad;
  if (threadIdx.x == 0 && threadIdx.y == 0) {
    block_cmax_grad = 0;
  }
  __syncthreads();
  if (thread_cmax_grad != 0) {
    atomicAdd(&block_cmax_grad, thread_cmax_grad);
  }
  __syncthreads();
  if (threadIdx.x == 0 && threadIdx.y == 0) {
    if (block_cmax_grad != 0) {
      atomicAdd(&grad_clip_max[0], block_cmax_grad);
    }
  }
}
```

Dropout mask
Scale

Reduce sum

Gradient clipping
Gradient accumulation

https://github.com/bytedance/lightseq/blob/master/lightseq/csrc/kernels/cuda/embedding_kernels.cu

Gradient of Criterion Operator

$$\mathscr{L} = -\sum_i p_i \log(q_i)$$

Smoothed one-hot ground truth

$$p = (1-\alpha)y + \frac{\alpha}{V} \cdot 1$$

α: smoothing parameter, 0< α < 1
V: vocabulary size, length of p, q

Softmax output

$$q = \text{Softmax}(h)$$

Gradient of Softmax

$$\frac{\partial \mathbf{q}_i}{\partial \mathbf{h}_j} = \begin{cases} -\mathbf{q}_i\mathbf{q}_j & i \neq j \\ \mathbf{q}_i(1-\mathbf{q}_i) & i = j \end{cases}$$

When i is equal to ground truth token index k:

$$\nabla \mathbf{h}_i = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_i} = -\frac{\alpha}{V}\sum_{j\neq k}\frac{1}{\mathbf{q}_j}\cdot\frac{\partial \mathbf{q}_j}{\partial \mathbf{h}_k} - (1-\alpha+\frac{\alpha}{V})\cdot\frac{1}{\mathbf{q}_k}\cdot\frac{\partial \mathbf{q}_k}{\partial \mathbf{h}_k}$$

$$= \mathbf{q}_k - \frac{\alpha}{V} - 1 + \alpha$$

Otherwise

$$\nabla \mathbf{h}_i = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_i} = -\frac{\alpha}{V}\sum_{j\neq k}\frac{1}{\mathbf{q}_j}\cdot\frac{\partial \mathbf{q}_j}{\partial \mathbf{h}_i} - (1-\alpha+\frac{\alpha}{V})\cdot\frac{1}{\mathbf{q}_k}\cdot\frac{\partial \mathbf{q}_k}{\partial \mathbf{h}_i}$$

$$= -\frac{\alpha}{V}\sum_{j\neq k, j\neq i}\frac{1}{\mathbf{q}_j}\cdot\frac{\partial \mathbf{q}_j}{\partial \mathbf{h}_i} - \frac{\alpha}{V}\cdot\frac{1}{\mathbf{q}_i}\cdot\frac{\partial \mathbf{q}_i}{\partial \mathbf{h}_i}$$

$$- (1-\alpha+\frac{\alpha}{V})\cdot\frac{1}{\mathbf{q}_k}\cdot\frac{\partial \mathbf{q}_k}{\partial \mathbf{h}_i} = \mathbf{q}_i - \frac{\alpha}{V}$$

Therefore

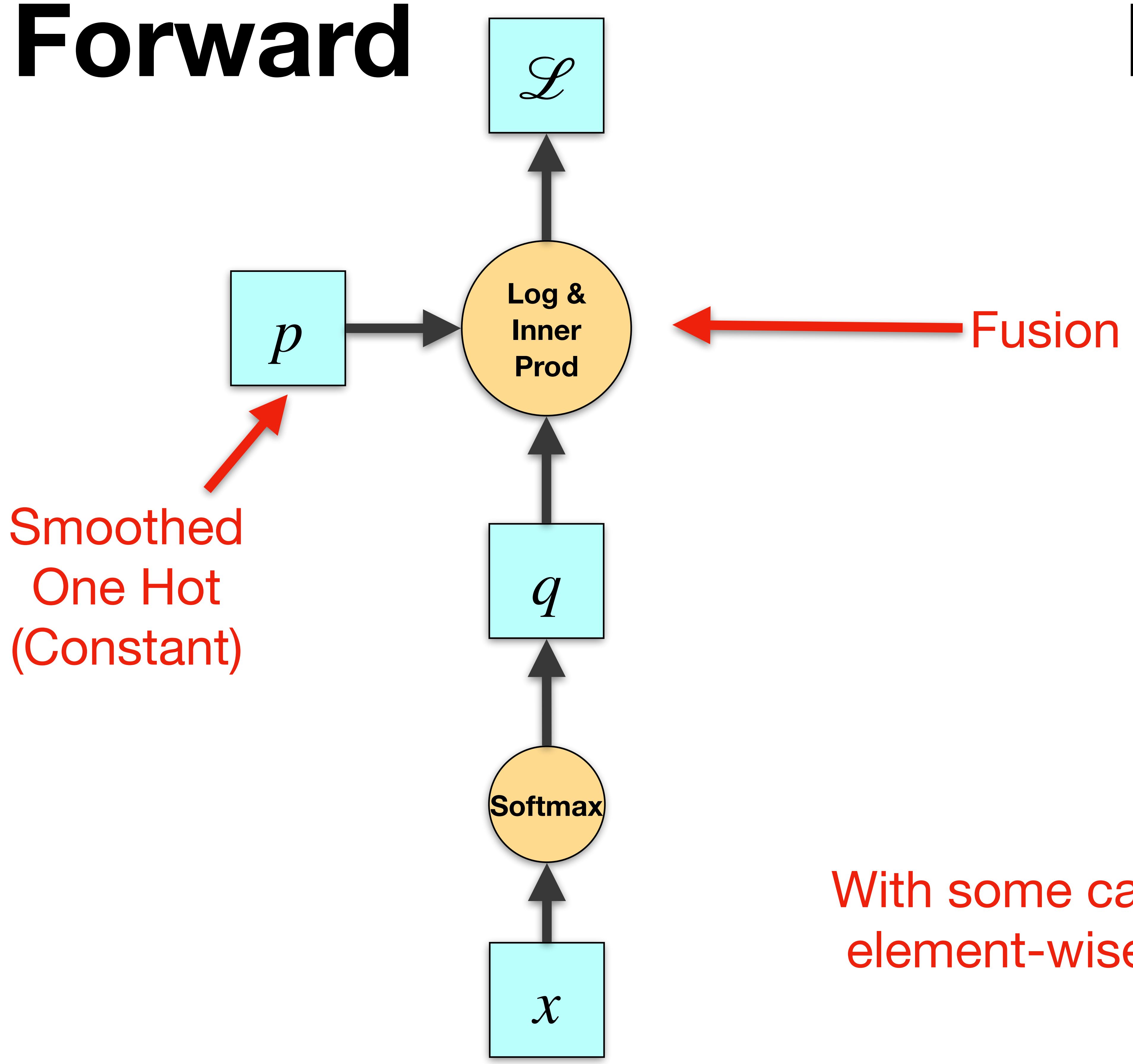
$$\nabla \mathbf{h}_i = \begin{cases} \mathbf{q}_i - \frac{\alpha}{V} - 1 + \alpha & \text{if token } i \text{ is the ground truth} \\ \mathbf{q}_i - \frac{\alpha}{V} & \text{otherwise} \end{cases}$$

=>

$$g = q - p$$

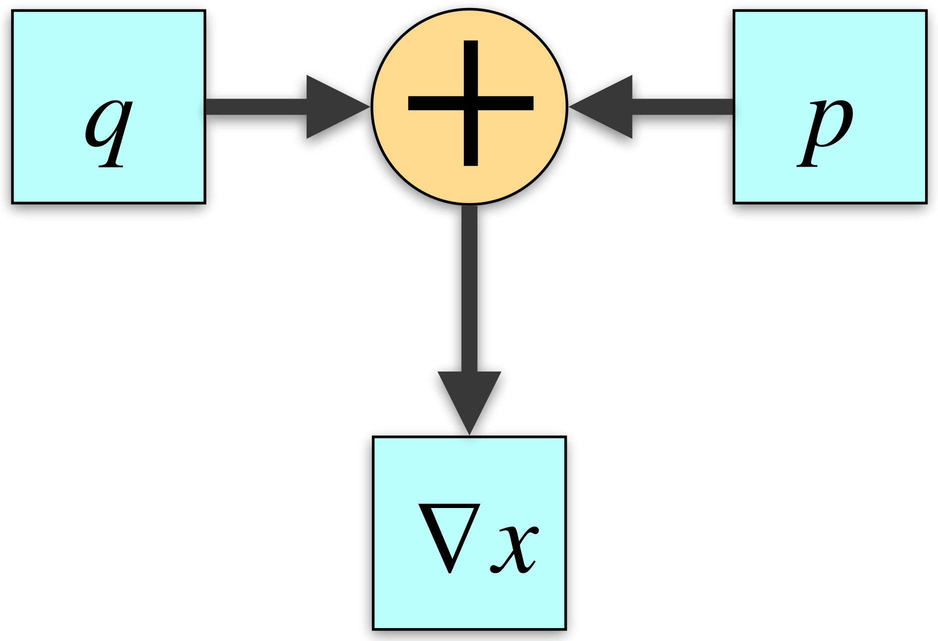
Fused Criterion Operator

Forward



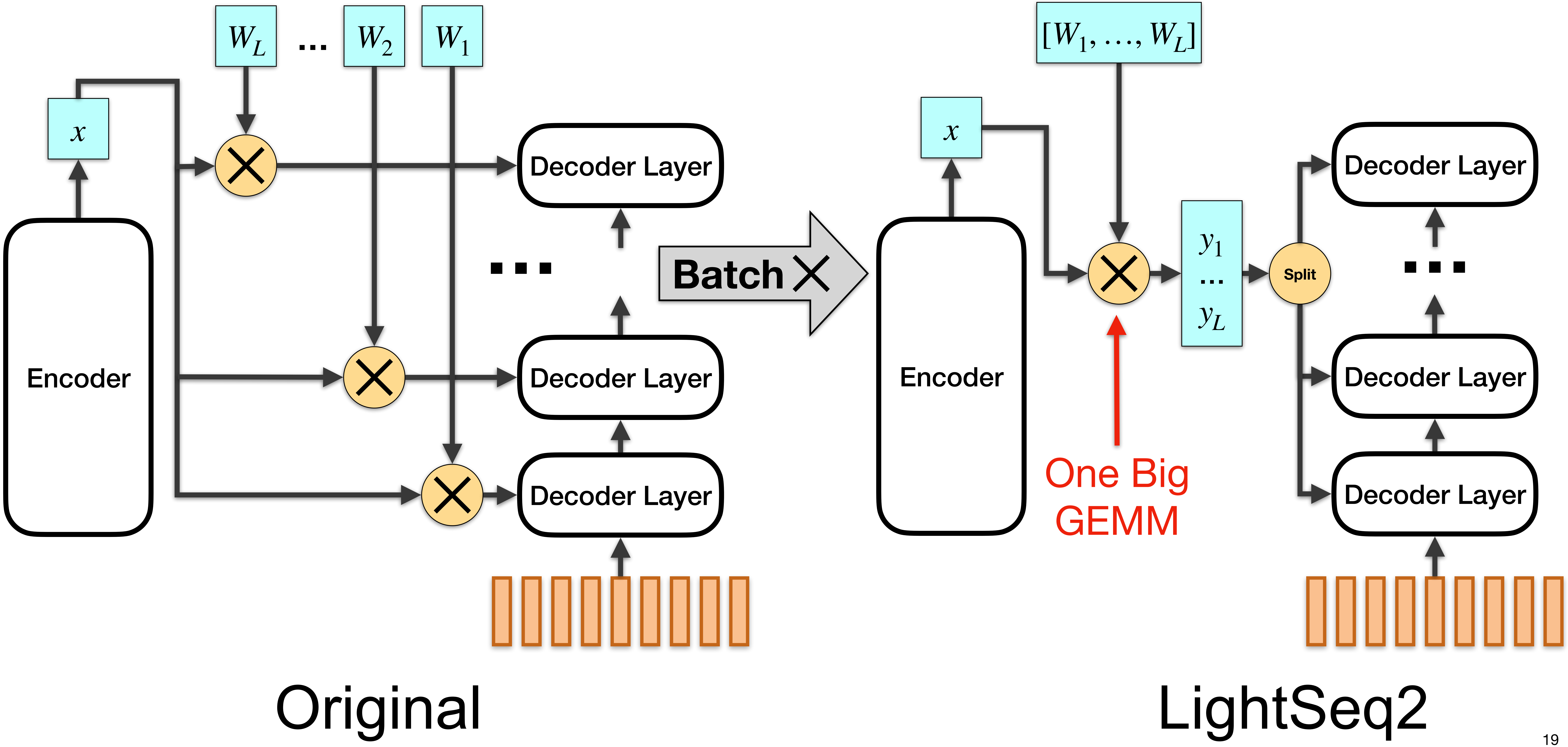
$$\mathscr{L} = -\sum_i p_i \log(q_i)$$

Backward



With some calculations: element-wise operator → $\nabla x = q - p$

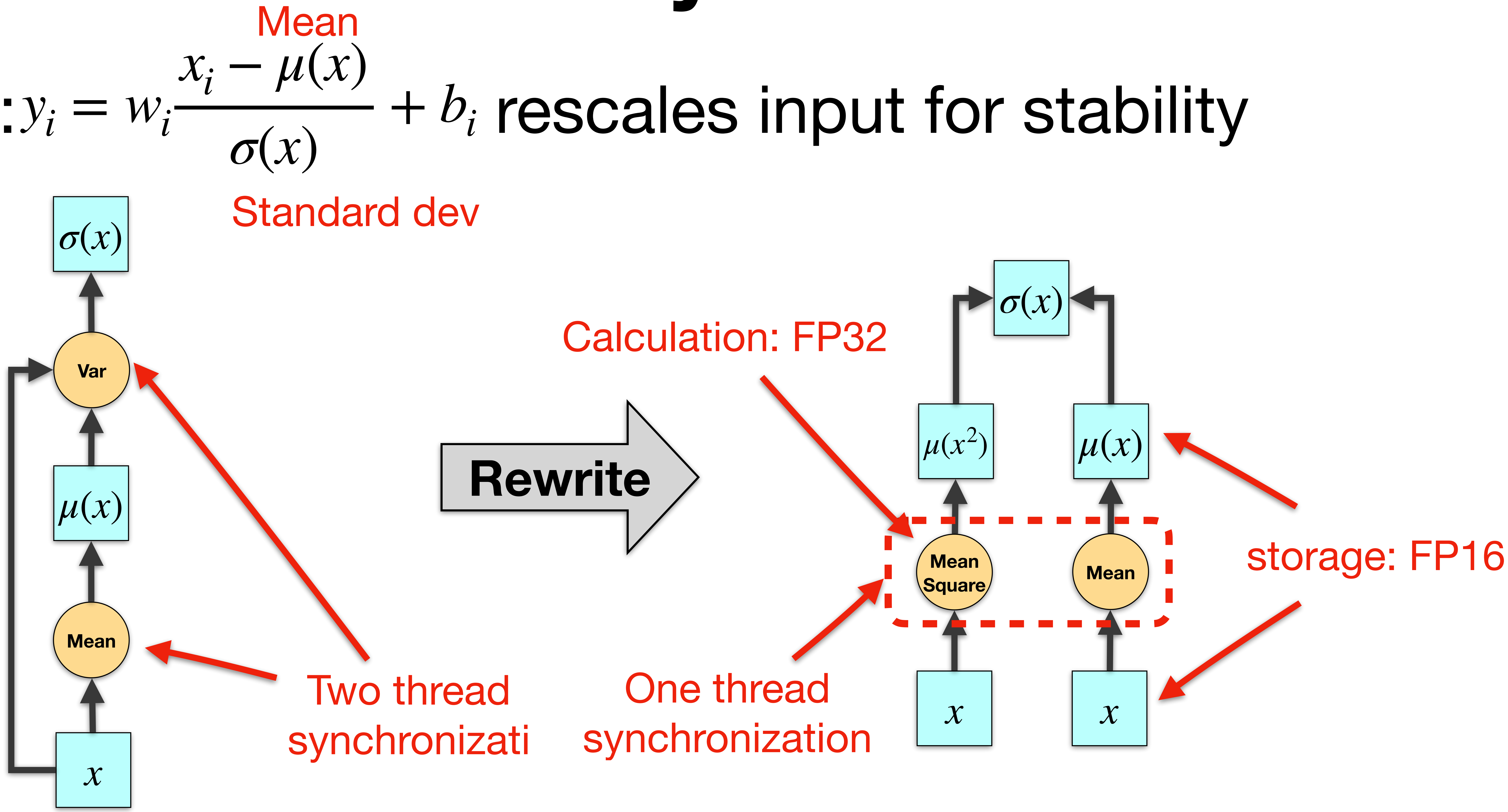
Layer-Batched Cross Attention



Technique 2: Reduce synchronization

Rewrite Reduction: LayerNorm Forward

LayerNorm: $y_i = w_i \frac{x_i - \mu(x)}{\sigma(x)} + b_i$ rescales input for stability



$$\sigma(x) = \sqrt{\frac{1}{N}\sum_i \left(x_i - \mu(x)_i\right)^2}$$

$$\sigma(x) = \sqrt{\mu(x^2) - \mu(x)^2}$$

Rewrite Reduction: LayerNorm Backward

Before:

$$\nabla \mathbf{x}_i = \frac{\mathbf{w}_i \nabla \mathbf{y}_i}{\sigma(\mathbf{x})} - \frac{1}{m\sigma(\mathbf{x})}\left(\sum_j \nabla \mathbf{y}_j \mathbf{w}_j + \hat{\mathbf{x}}_i \sum_j \nabla \mathbf{y}_j \mathbf{w}_j \hat{\mathbf{x}}_j\right)$$

Rearrange:

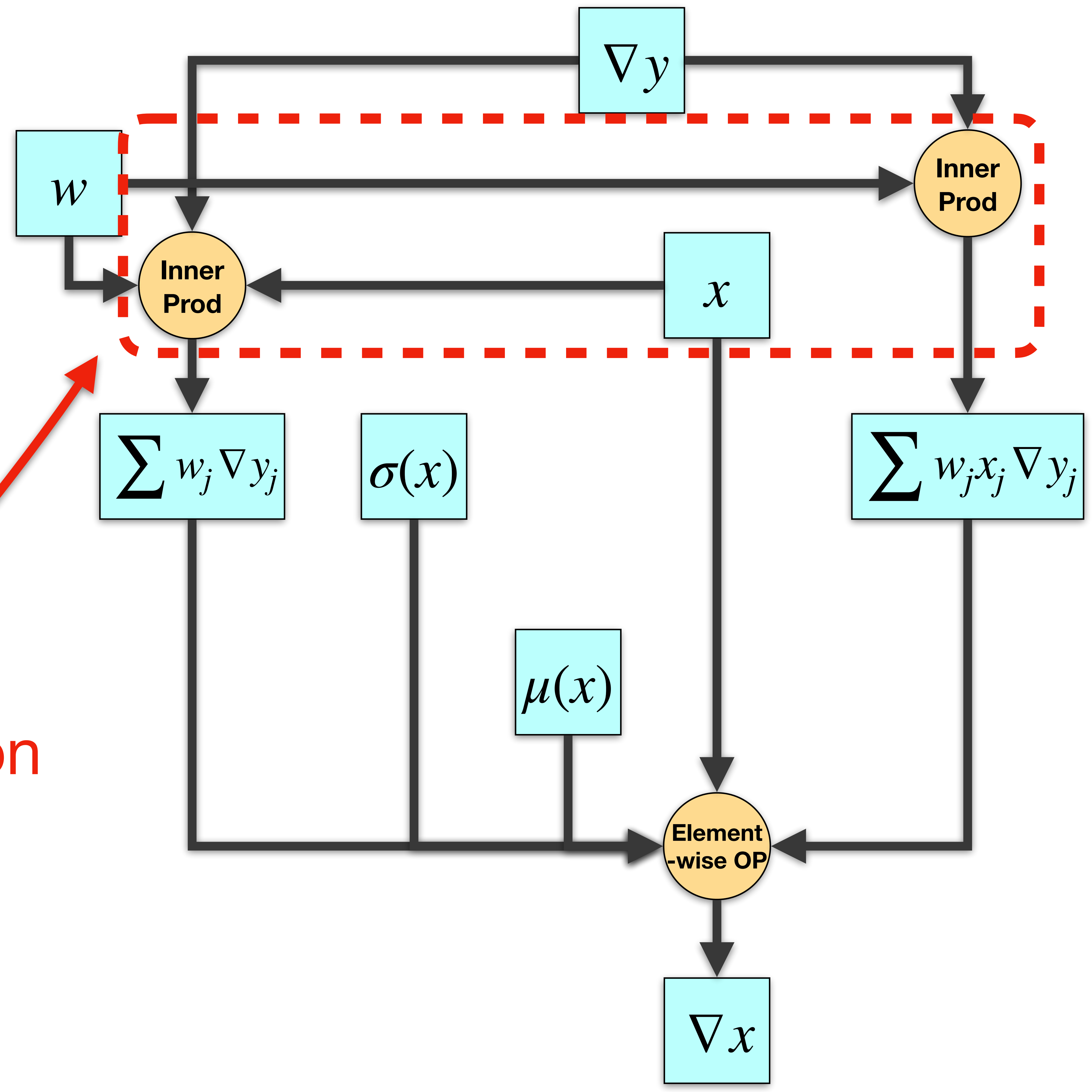
$$\nabla x_i = \frac{w_i \nabla y_i}{\sigma(x)} + \alpha \cdot \sum_j w_j \nabla y_j + \beta \cdot \sum_j w_j \nabla y_j x_j$$

where

$$\alpha = \frac{[x_i - \mu(x)]\mu(x) - \sigma(x)}{m\sigma(x)^3}$$

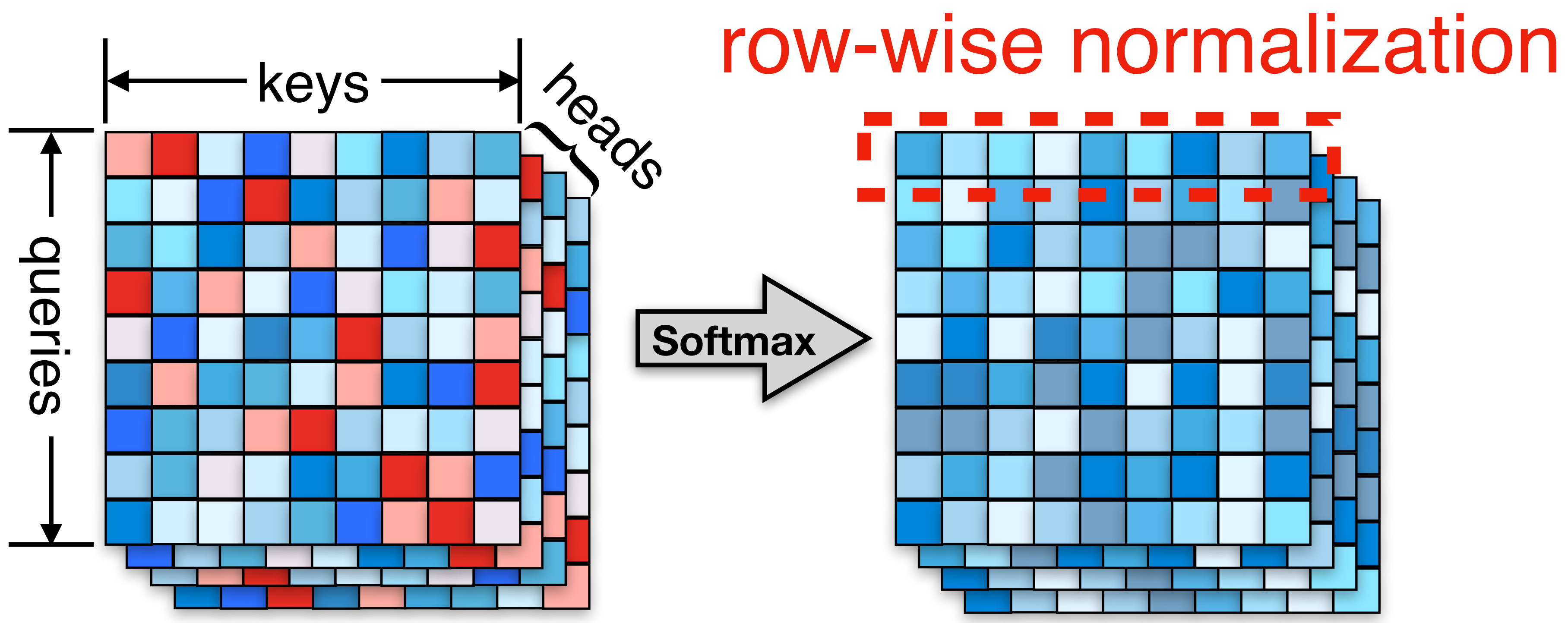
$$\beta = \frac{\mu(x) - x_i}{m\sigma(x)^3}$$

One thread synchronization



You will implement LayerNorm in hw3!

Rewrite Reduction: Softmax Forward

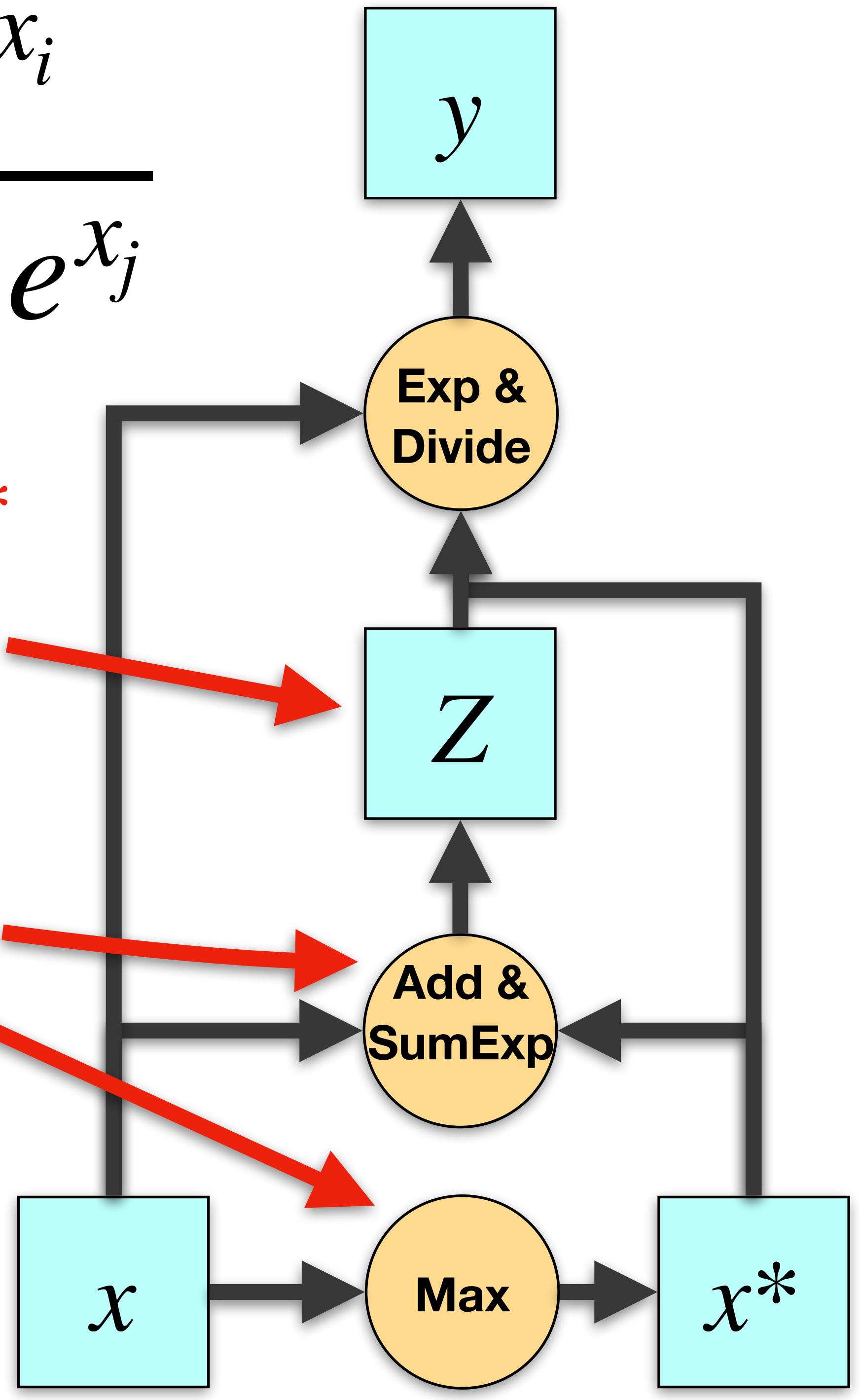


$$y_i = \frac{e^{x_i}}{\sum e^{x_j}}$$

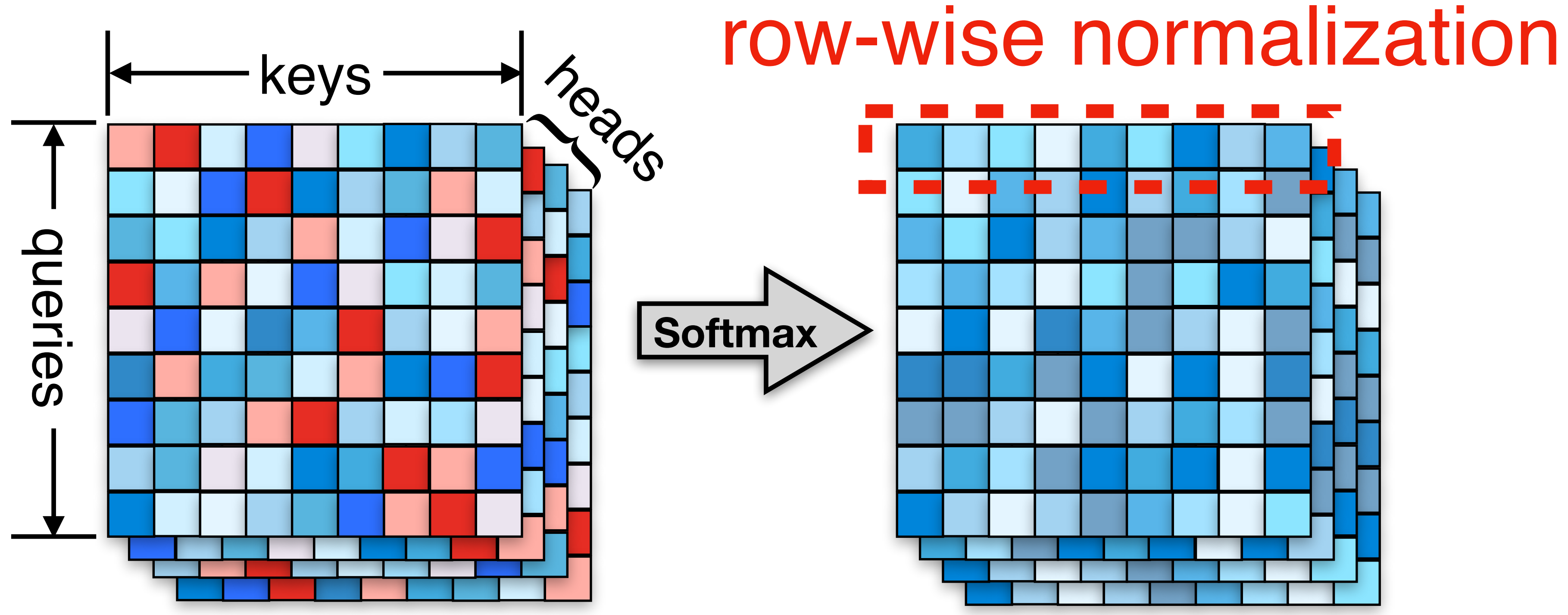
$$Z = \sum_i e^{x_i - x^*}$$

Two thread synchronizations

Two reduce: Costly!

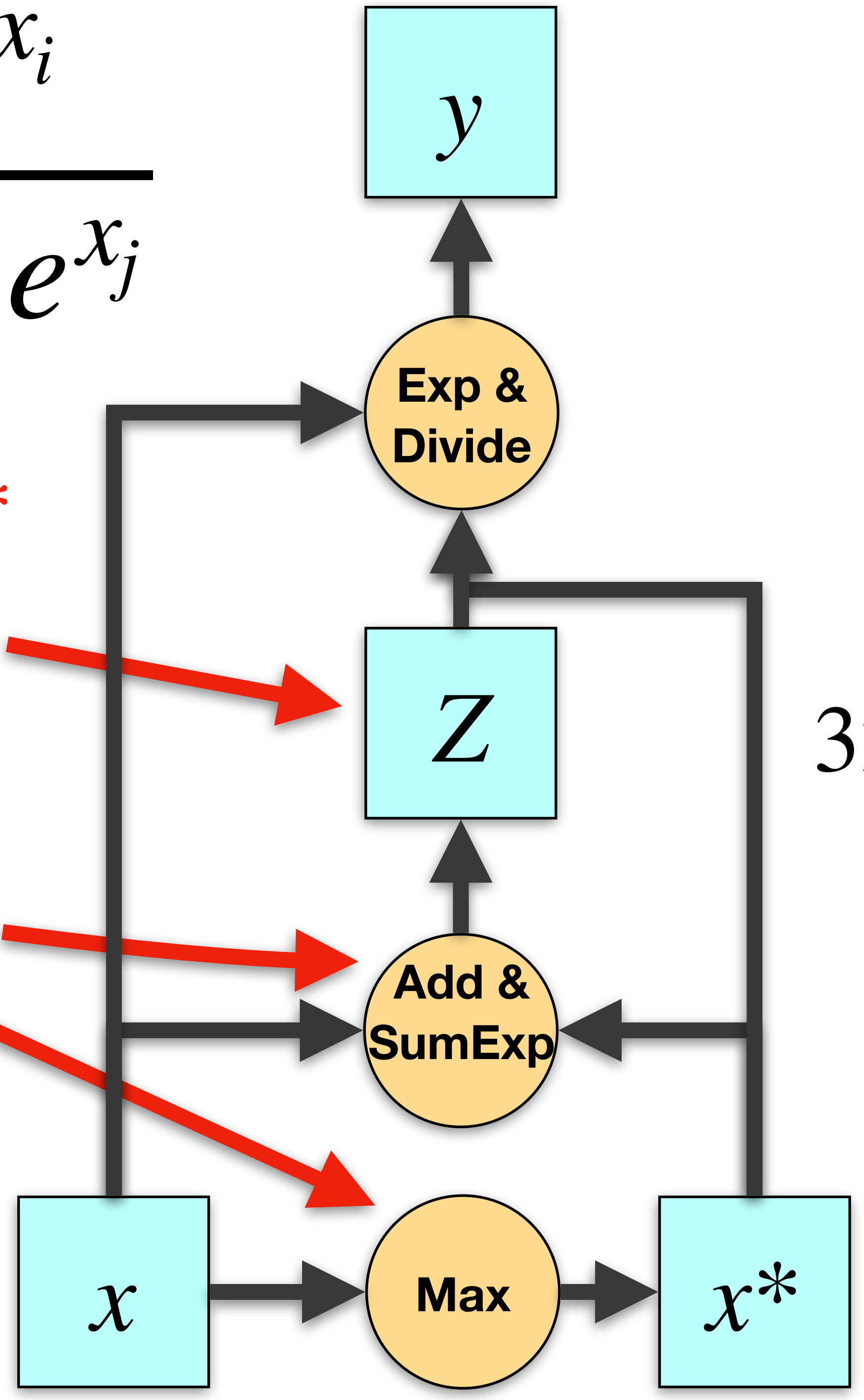


Rewrite Reduction: Softmax Forward



Parameters (e.g. # of blocks, warps per block) are shape dependent for maximal speedup

$$y_i = \frac{e^{x_i}}{\sum e^{x_j}}$$



$$Z = \sum_i e^{x_i - x^*}$$

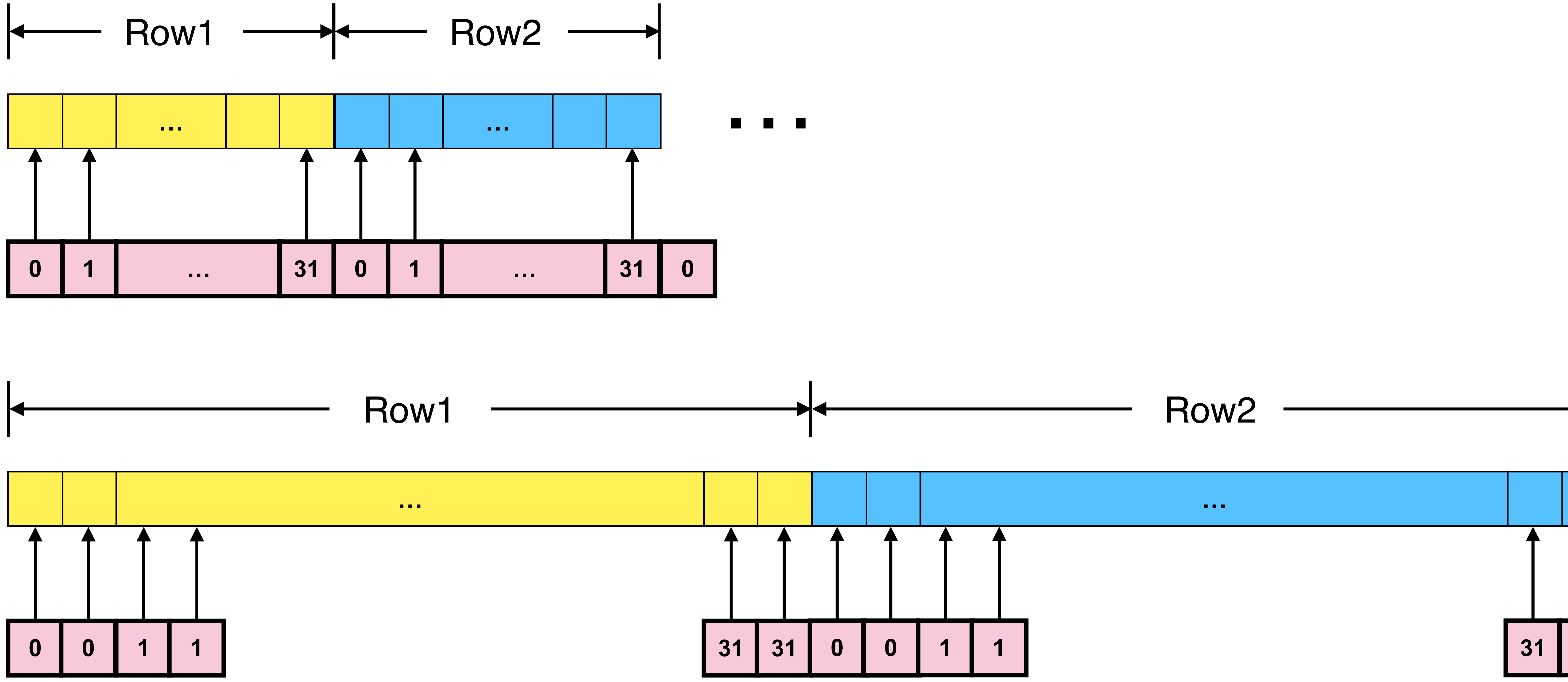
Two thread synchronizations

$\text{column} \leq 32$

Threads

$32 < \text{column} \leq 64$

Threads



And Other Shapes …

You will implement Softmax in HW3!

Code Example: Softmax Forward

Parameters tuning by using templates

```
template <typename T, int block_dim, int ele_per_thread>
__global__ void ker_attn_softmax(T *inp, const T *attn_mask, int from_len,
                                 int to_len, bool mask_future) {
```

```
template <typename T, int block_dim, int ele_per_thread>
__global__ void ker_attn_softmax_lt32(T *inp, const T *attn_mask, int from_len,
                                      int to_len, bool mask_future) {
```


https://github.com/bytedance/lightseq/blob/master/lightseq/csrc/kernels/cuda/softmax_kernels.cu

Code Example: Softmax Forward

Parameters tuning by using templates

```
template <typename T, int block_dim, int ele_per_thread>
__global__ void ker_attn_softmax(T *inp, const T *attn_mask, int from_len,
                                 int to_len, bool mask_future) {

template <typename T, int block_dim, int ele_per_thread>
__global__ void ker_attn_softmax_lt32(T *inp, const T *attn_mask, int from_len,
                                      int to_len, bool mask_future) {
```

Then call with parameters in launch

```
void launch_attn_softmax<float>(float *inp, const float *attn_mask,
                                int batch_size, int nhead, int from_len,
                                int to_len, bool mask_future,
                                cudaStream_t stream) {
  dim3 grid_dim(1, batch_size, nhead);
  if (to_len <= 32) {
    ker_attn_softmax_lt32<float, 32, 1><<<grid_dim, 32, 0, stream>>>(
        inp, attn_mask, from_len, to_len, mask_future);
  } else if (to_len <= 64) {
    ker_attn_softmax_lt32<float, 32, 2><<<grid_dim, 32, 0, stream>>>(
        inp, attn_mask, from_len, to_len, mask_future);
  } else if (to_len <= 128) {
    grid_dim.x = 16;
    ker_attn_softmax<float, 64, 2><<<grid_dim, 64, 0, stream>>>(
        inp, attn_mask, from_len, to_len, mask_future);
  } else if (to_len <= 256) {
    grid_dim.x = 32;
    ker_attn_softmax<float, 128, 2><<<grid_dim, 128, 0, stream>>>(
        inp, attn_mask, from_len, to_len, mask_future);
  } else if (to_len <= 512) {
    grid_dim.x = 64;
    ker_attn_softmax<float, 256, 2><<<grid_dim, 256, 0, stream>>>(
        inp, attn_mask, from_len, to_len, mask_future);
  } else if (to_len <= 1024) {
    grid_dim.x = 128;
    ker_attn_softmax<float, 512, 2><<<grid_dim, 512, 0, stream>>>(
        inp, attn_mask, from_len, to_len, mask_future);
  } else {
    throw std::runtime_error(
        "Sequence length greater than 512 is currently not supported");
  }
}
```

https://github.com/bytedance/lightseq/blob/master/lightseq/csrc/kernels/cuda/softmax_kernels.cu

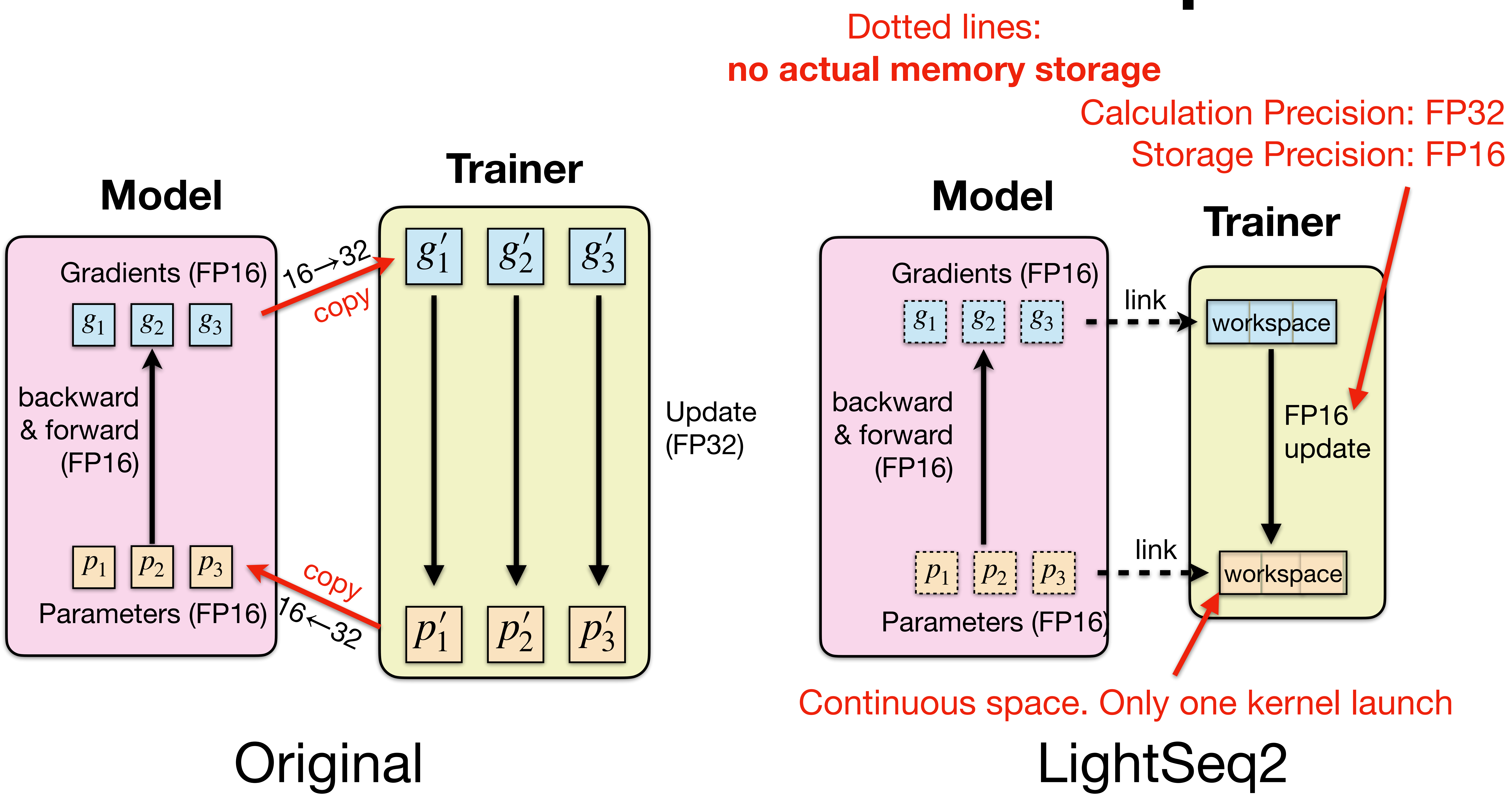
Technique 3: Mixed-precision Calculation

- Modern GPU supports half-precision (FP16) or FP8 (on H100)
- Benefits:
  - lower memory for storing model and data ==> enlarge batch size
  - transfer data at a higher rate (with same bandwith) between GPU main memory and SMs
  - more FLOPs for FP16 (up to 8x more) compared to FP32.

Use low-precision for all data?

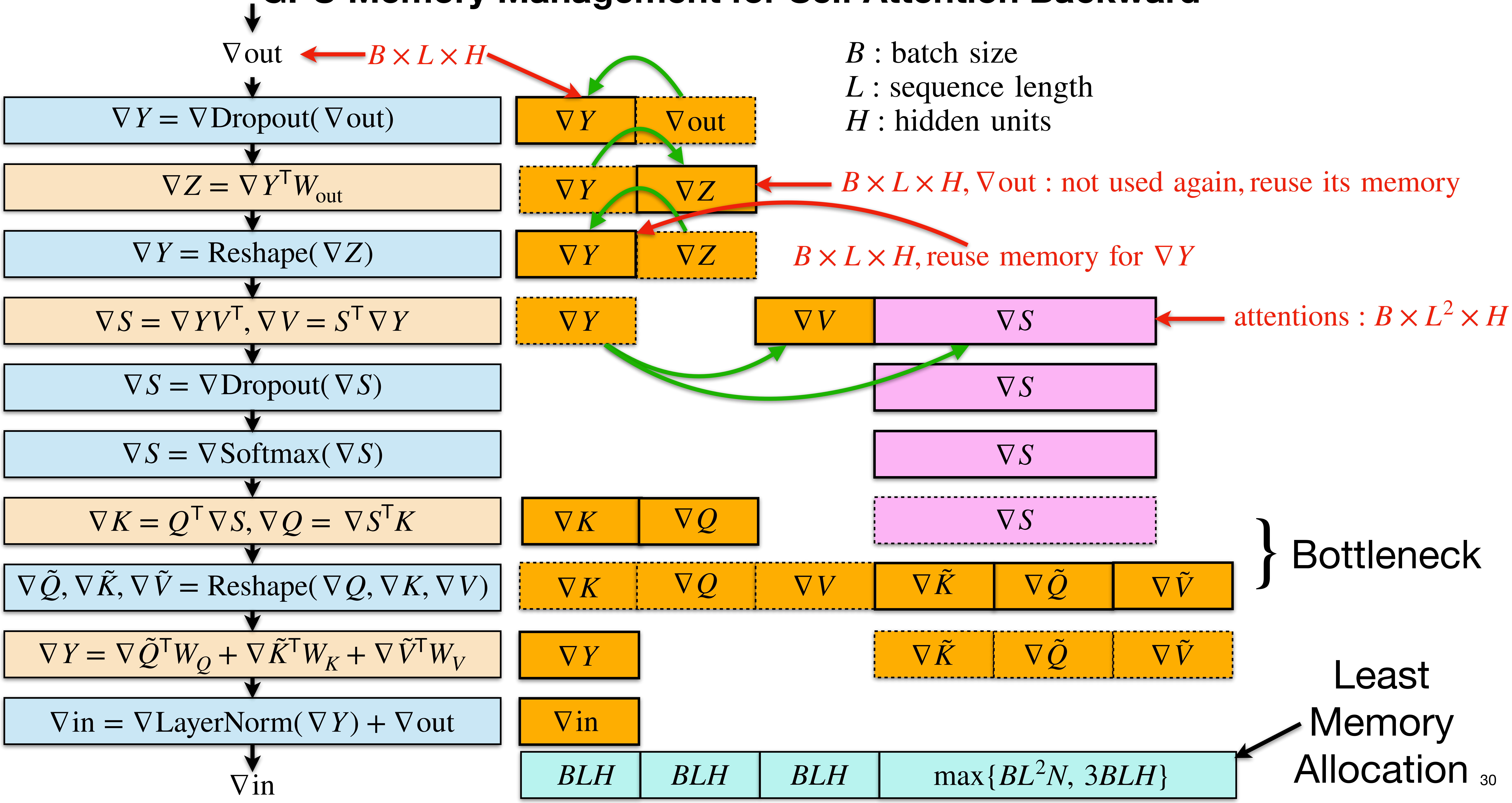
- Forward / Backward could use FP16 or FP8
- Gradient update in Optimizer (or trainer) needs FP32
- Nvidia APEX library provides automatic mixed-precision calculation for many NN layers
    - but still miss fine-grained memory optimization with mixed-precision for LLM.

Accelerated Mixed-Precision Update

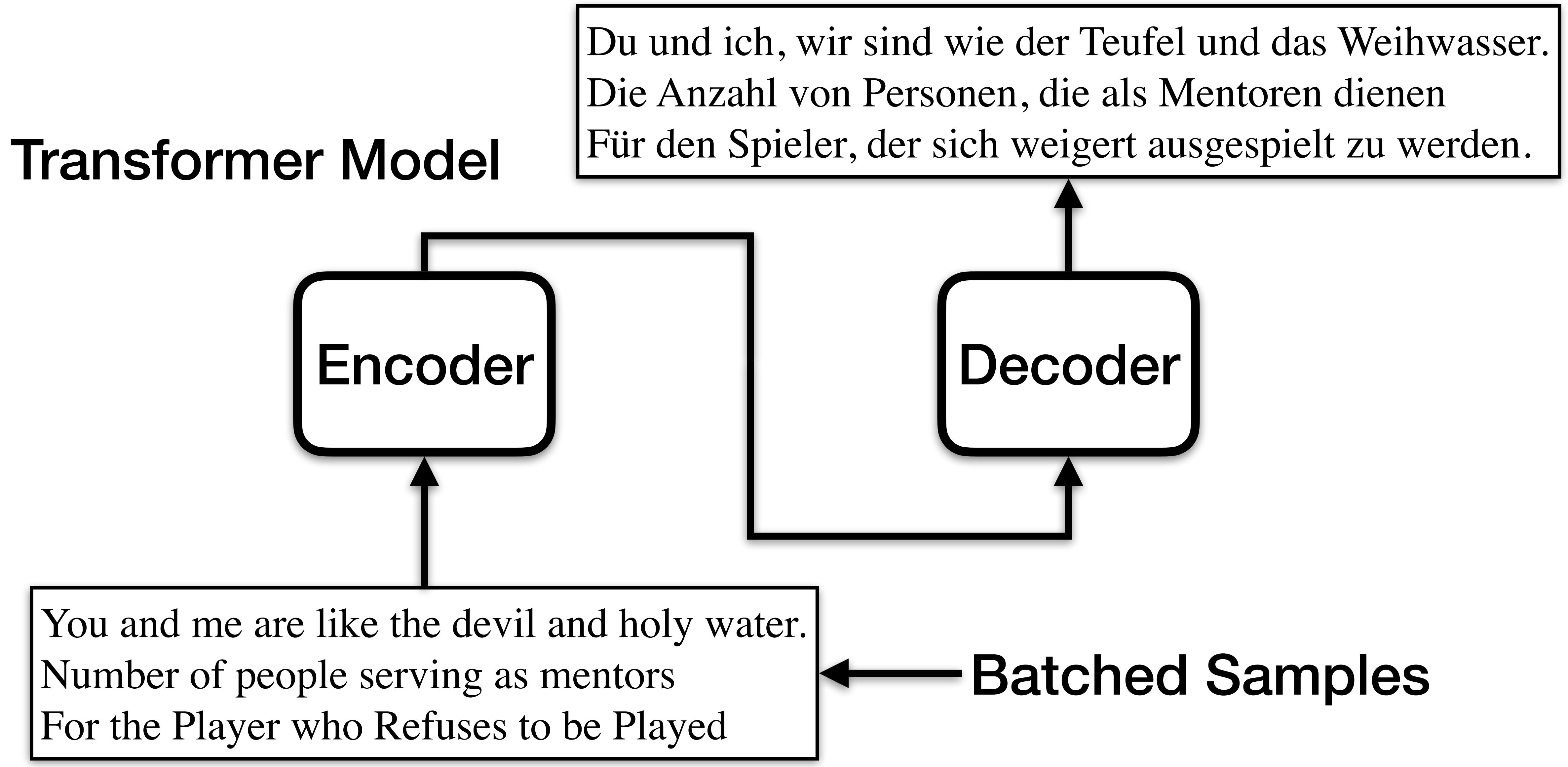


Technique 4: Memory Reuse

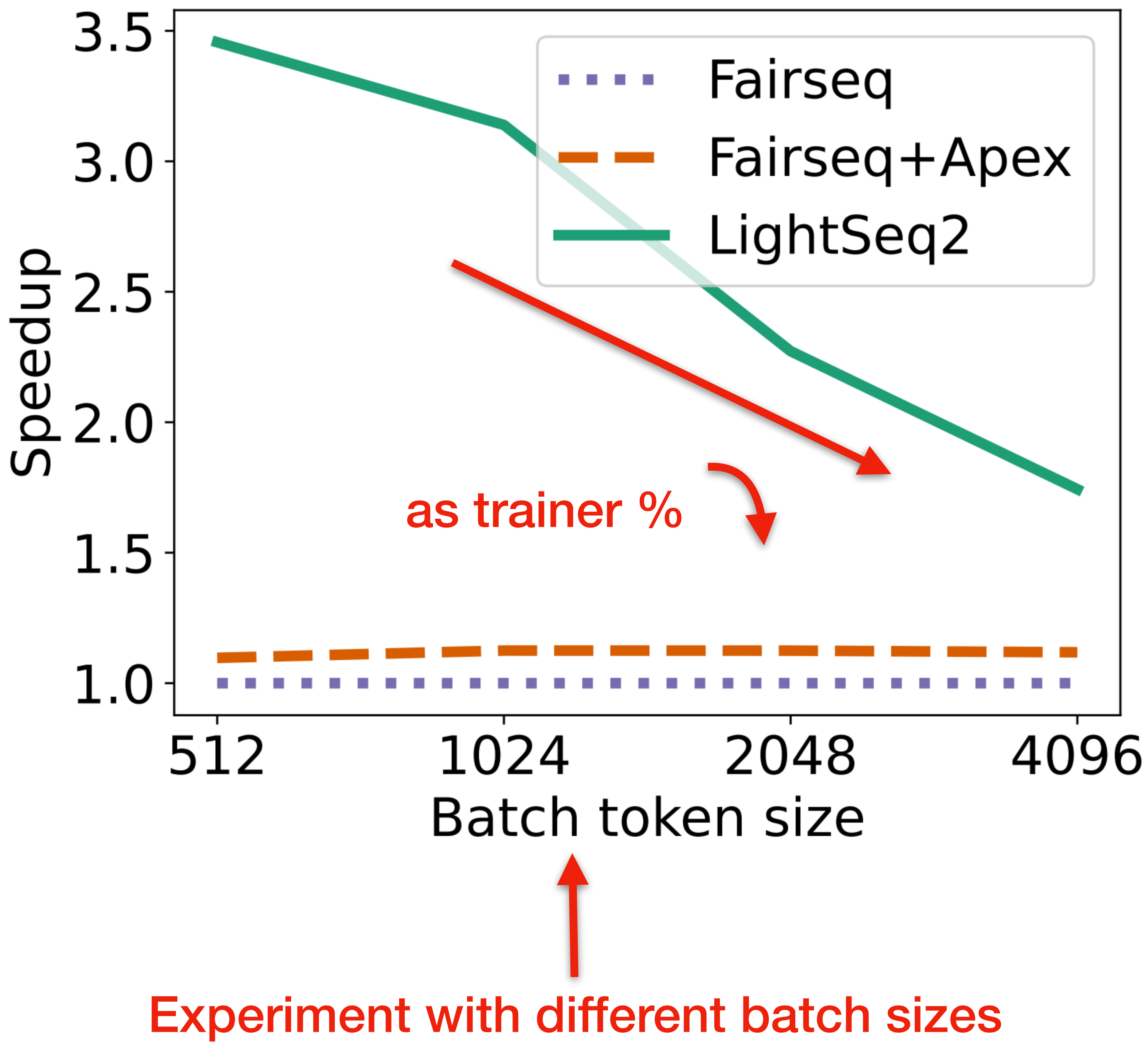
GPU Memory Management for Self Attention Backward



Machine Translation Training: 1.4-3.5x Speedup



DataSet	WMT14 English-German Machine Translation
Model	Transformer: 24 encoder layers + 24 decoder layers
Hardware	1 Worker with 8x A100
Baseline	FairSeq (PyTorch) + Apex (optimized operators)

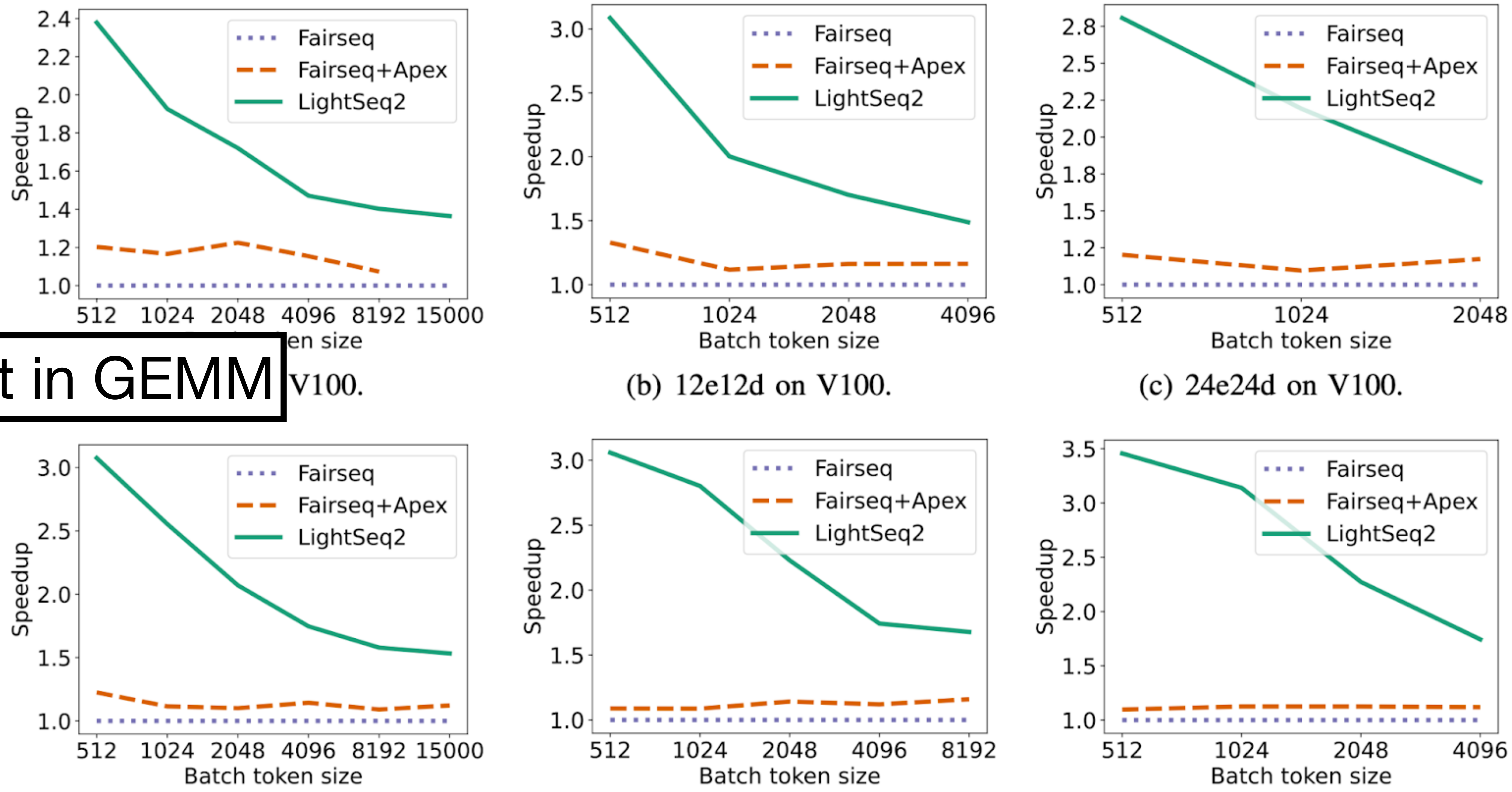


Machine Translation Training: 1.4-3.5x Speedup

V100: 1.4-2.8x

A100: 1.5-3.5x

A100 is more efficient in GEMM



V100.

(b) 12e12d on V100.

(c) 24e24d on V100.

(d) 6e6d on A100.

(e) 12e12d on A100.

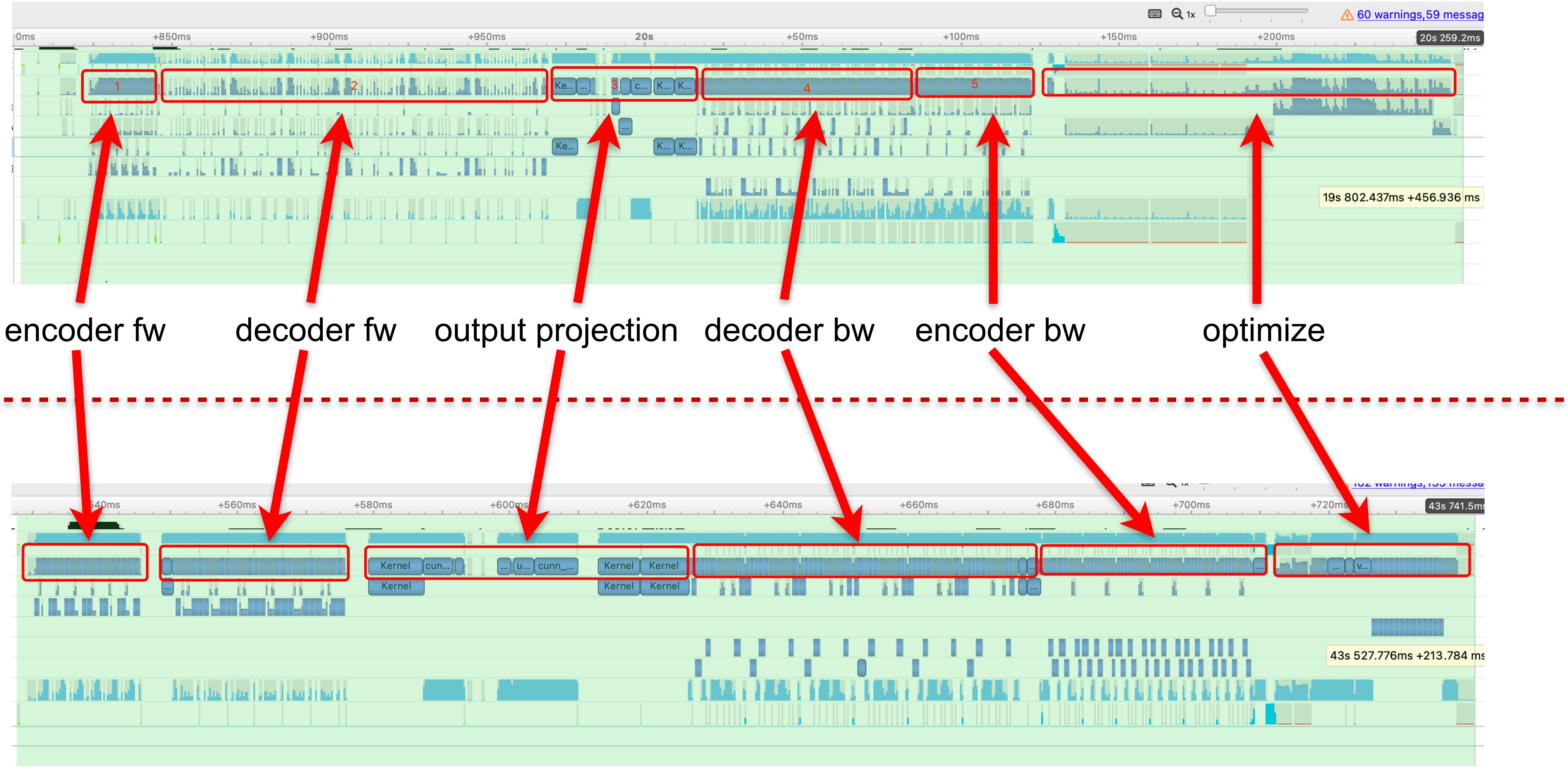
(f) 24e24d on A100.

Model Size

Speedup

Visualization of Training on GPU

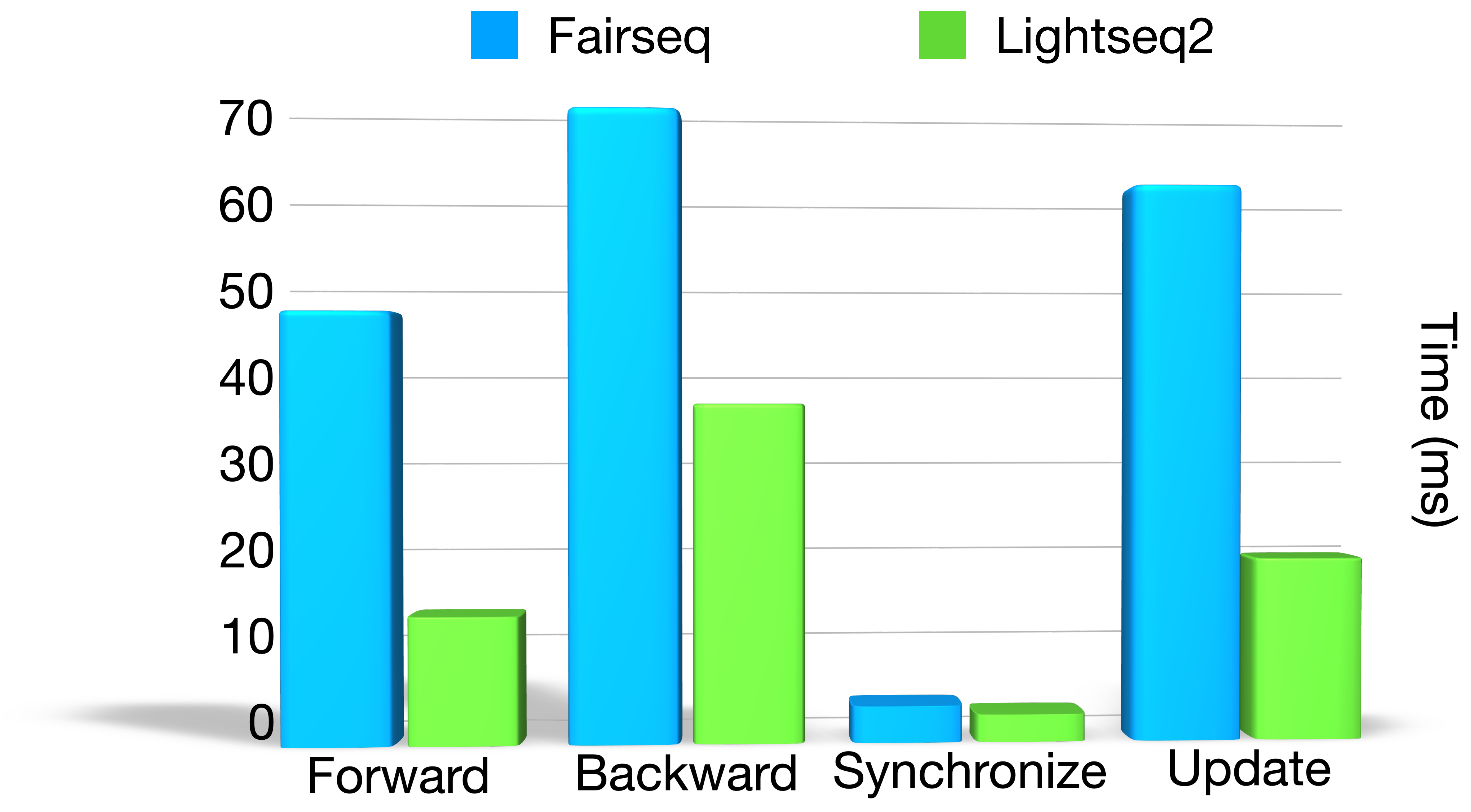
Fairseq: 457ms



LightSeq: 214ms

Training Speedup Breakdown

Task: WMT14 Engish German Machine Translation (same for rest pages)

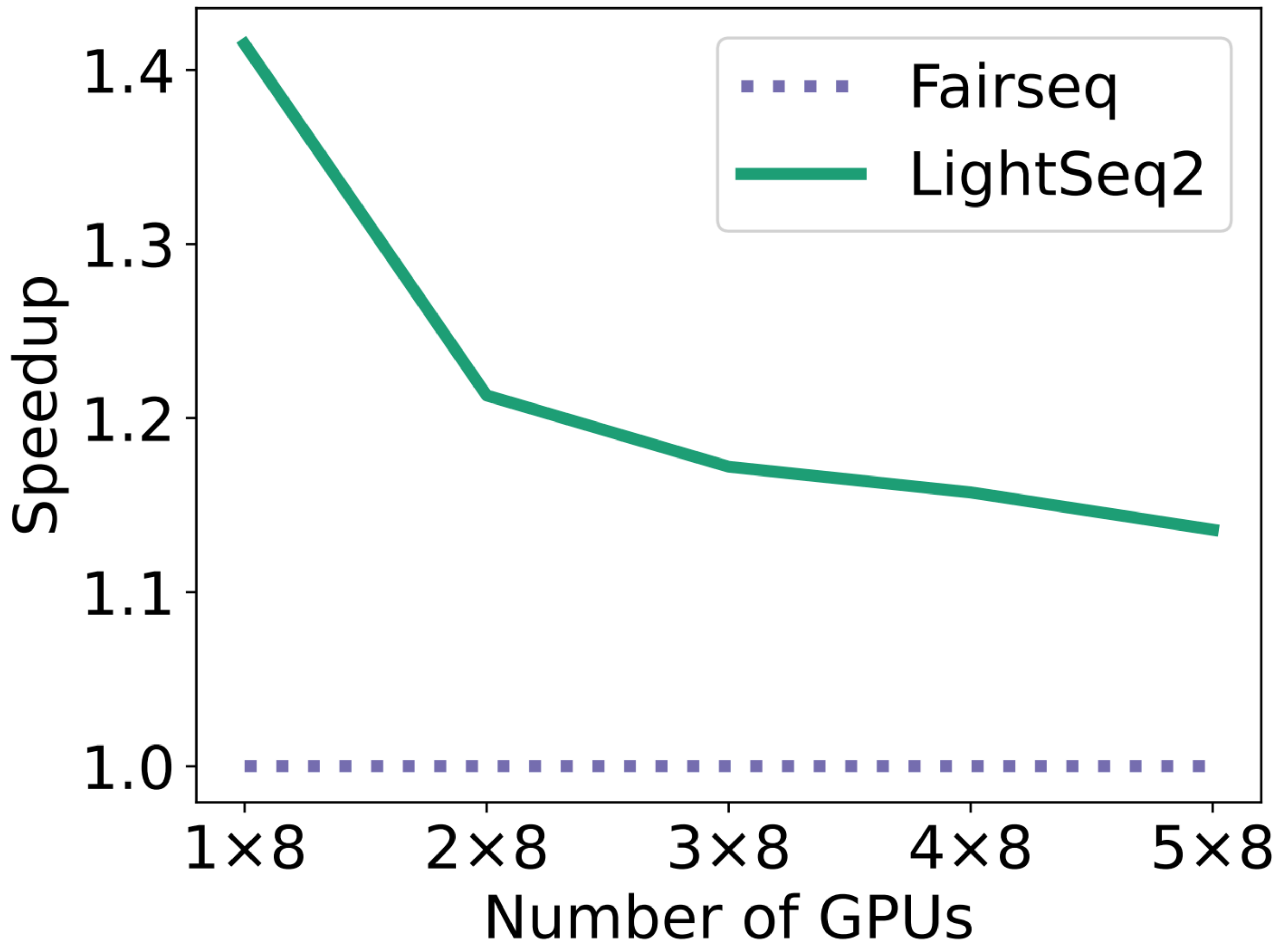


Time cost for each training stages

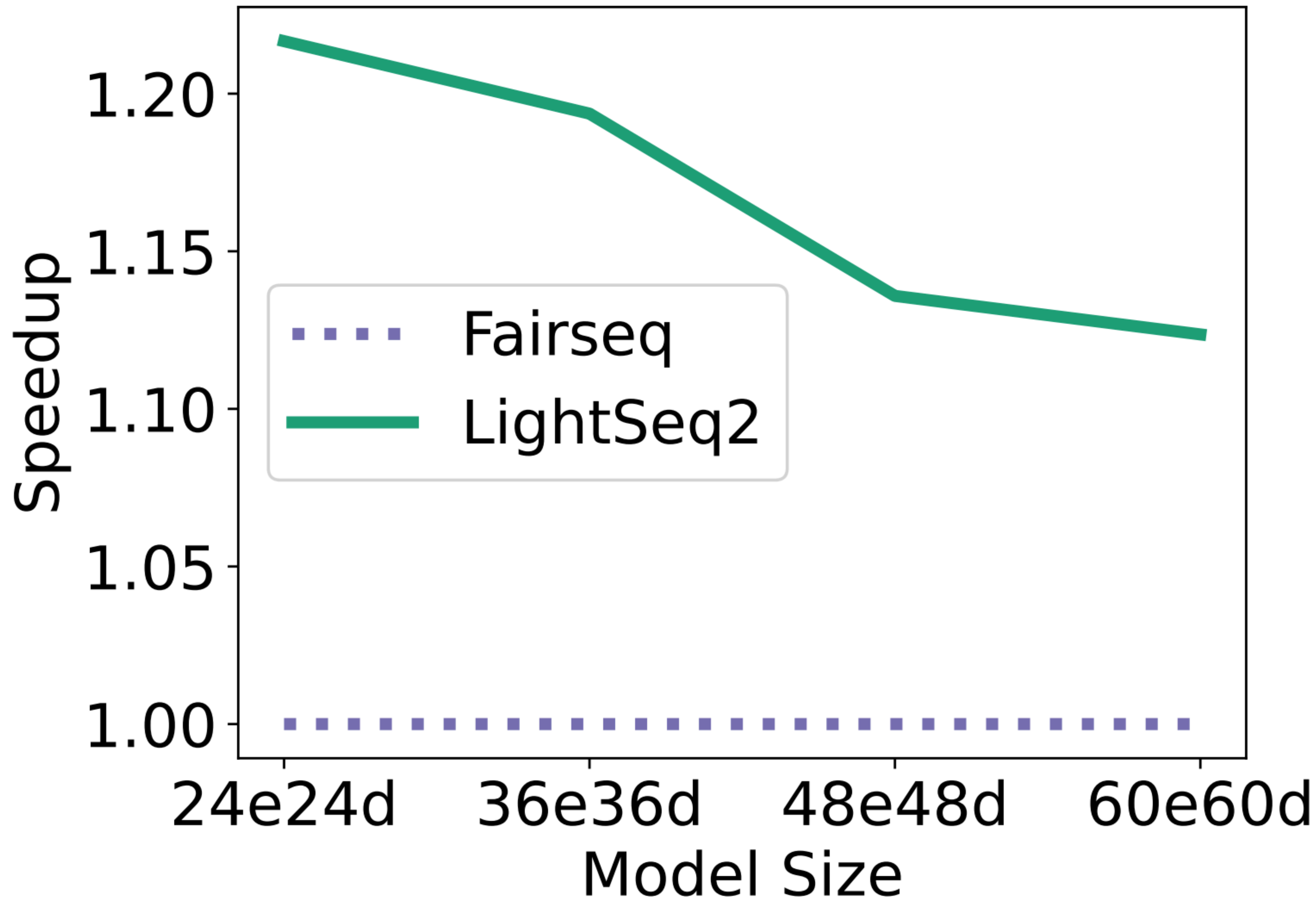
Operator	Speedup
LayerNorm	4x
Softmax	2.5-3.4x
Dropout	1.1-2.5x
Trainer	2.3x

Operator Speedup

Scalability: 1.12-1.41x Speedup

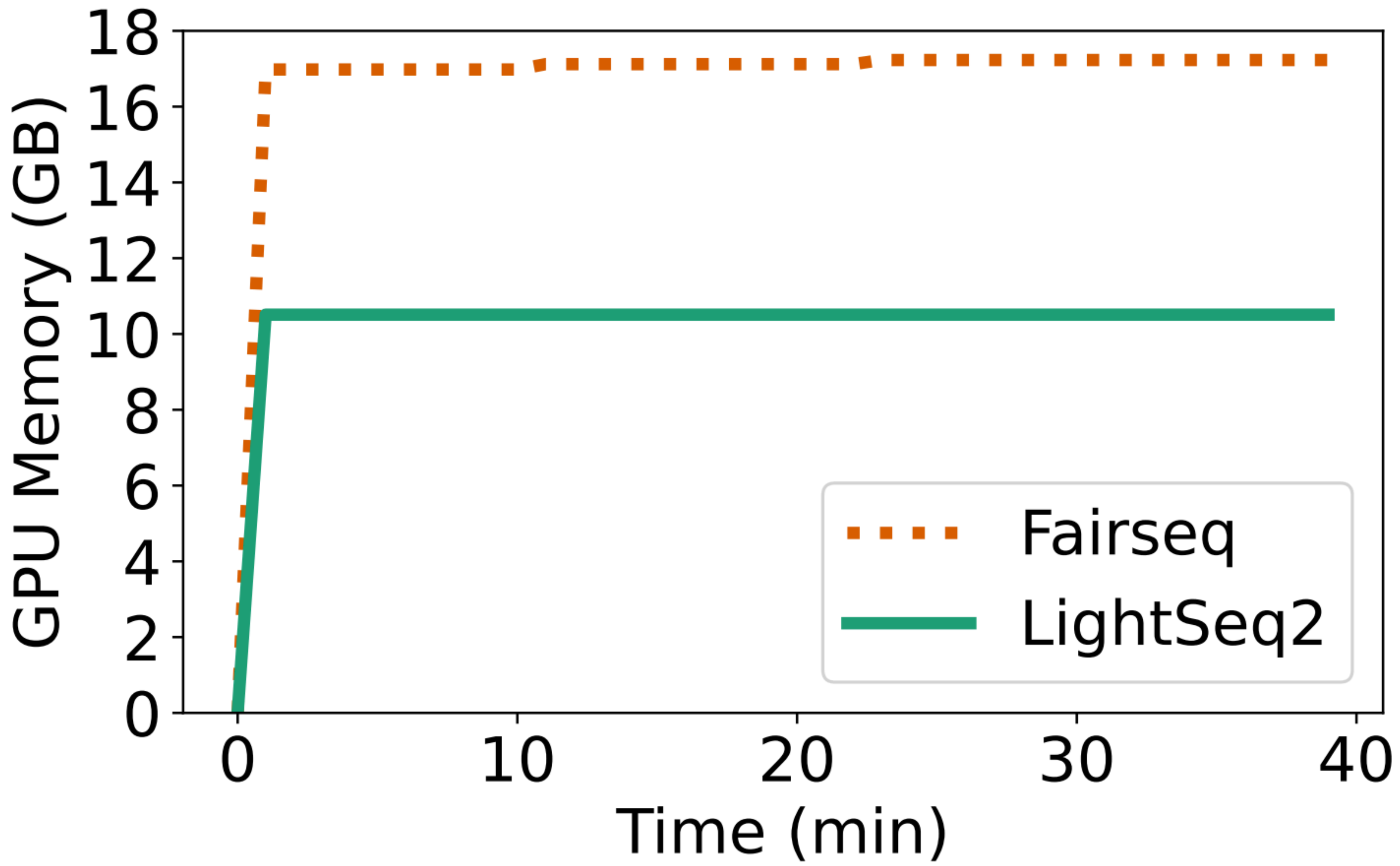


Speedup on 1 to 5 workers,
each has 8x A100

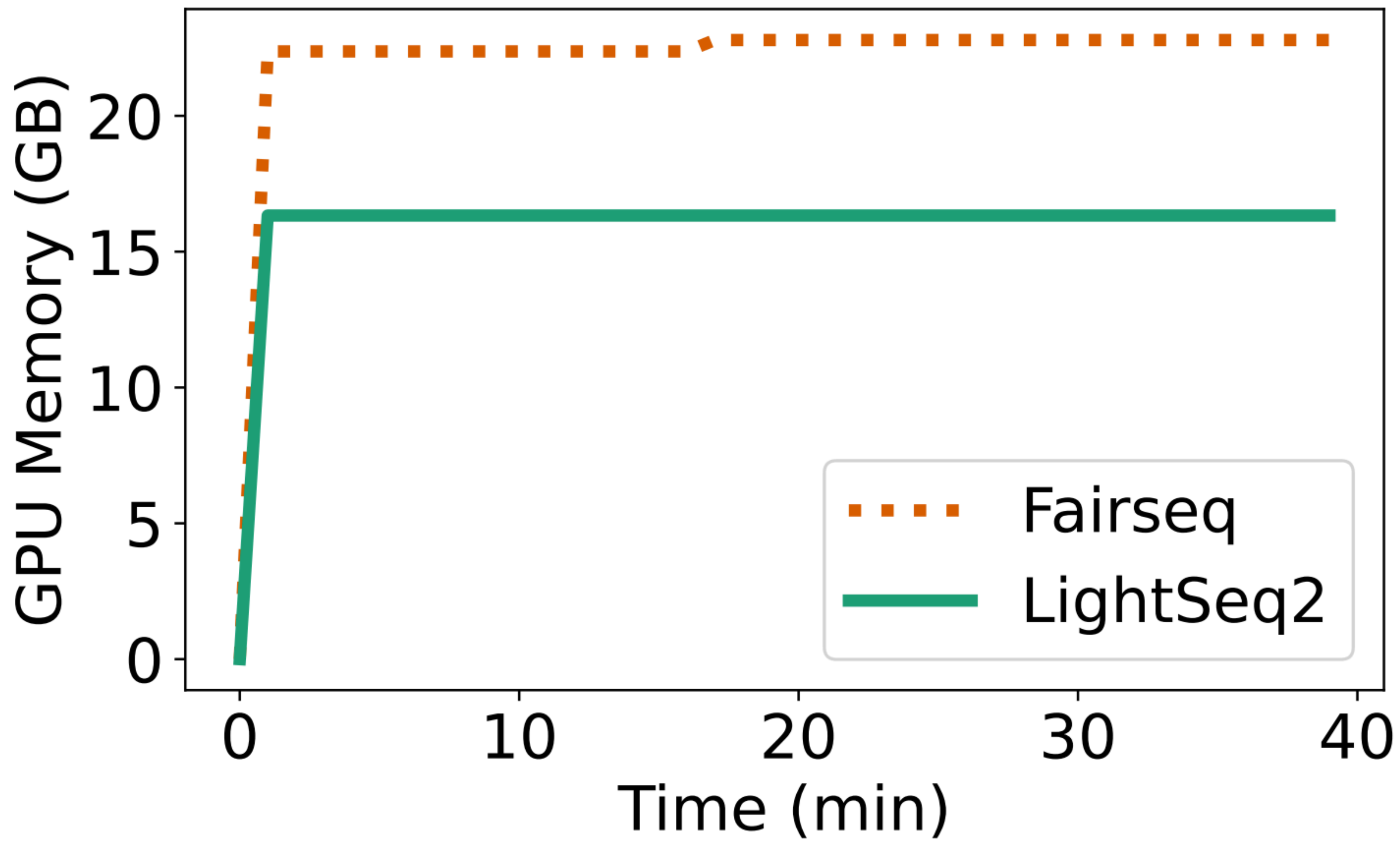


Speedup of Models with
different layers

Training Memory Cost: 6G less

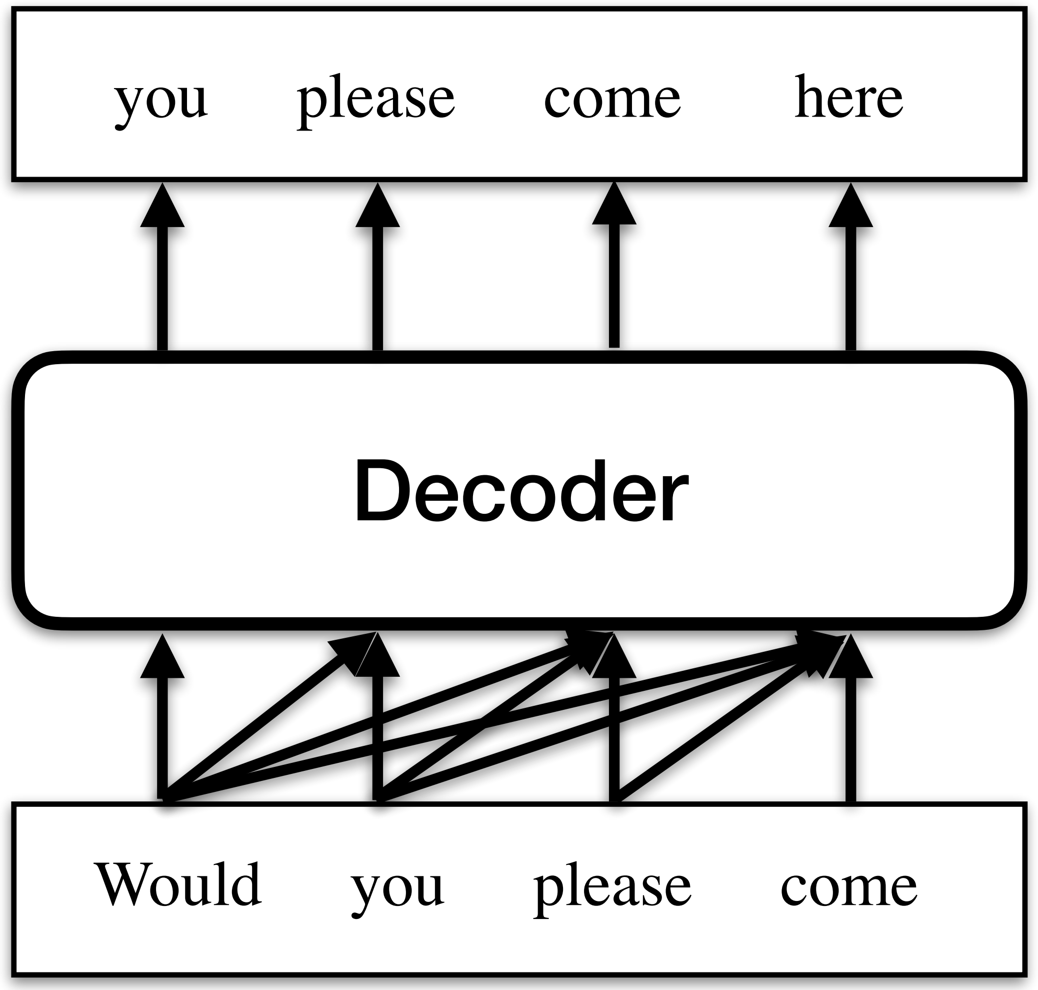


Transformer Base

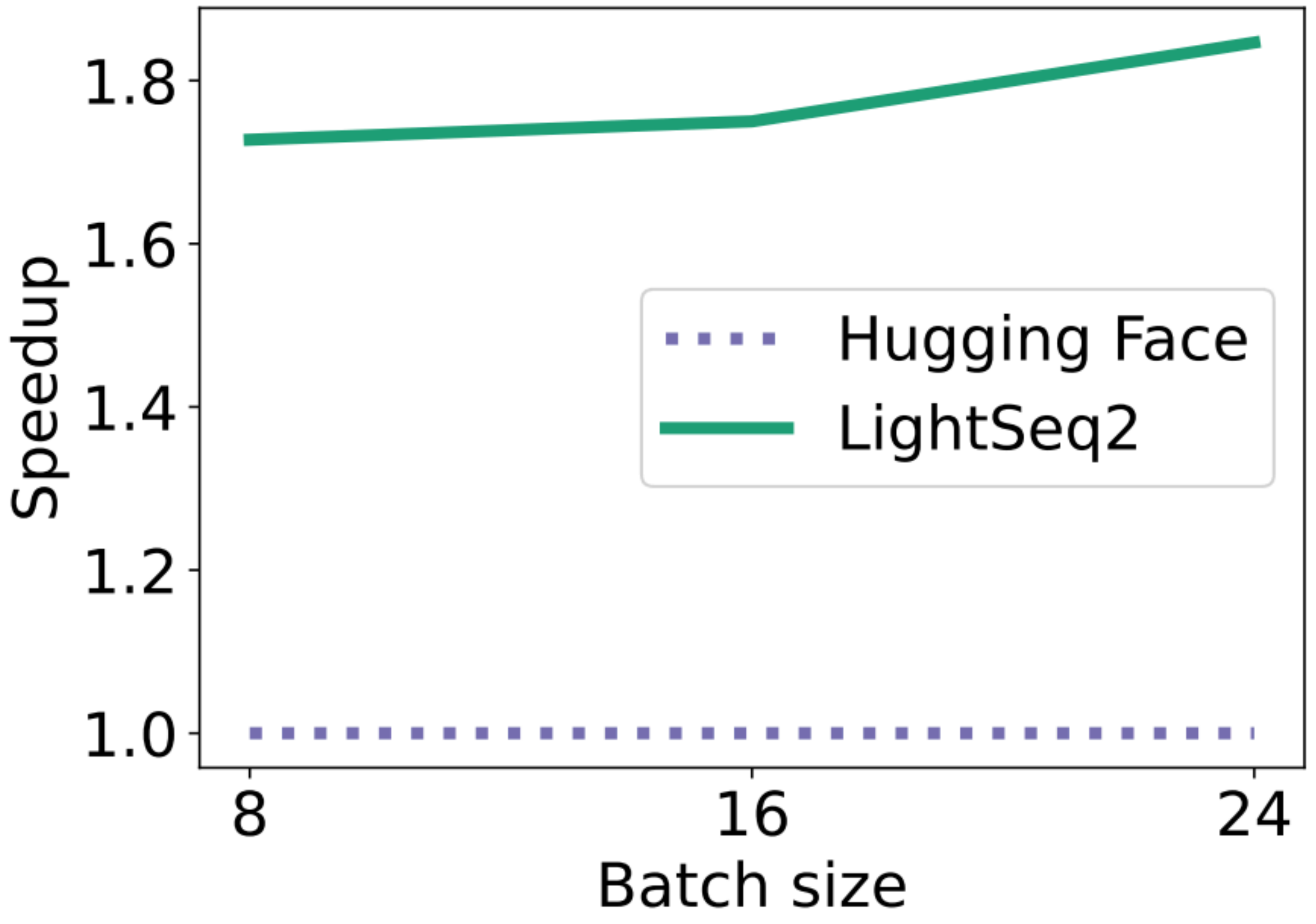


Transformer Large

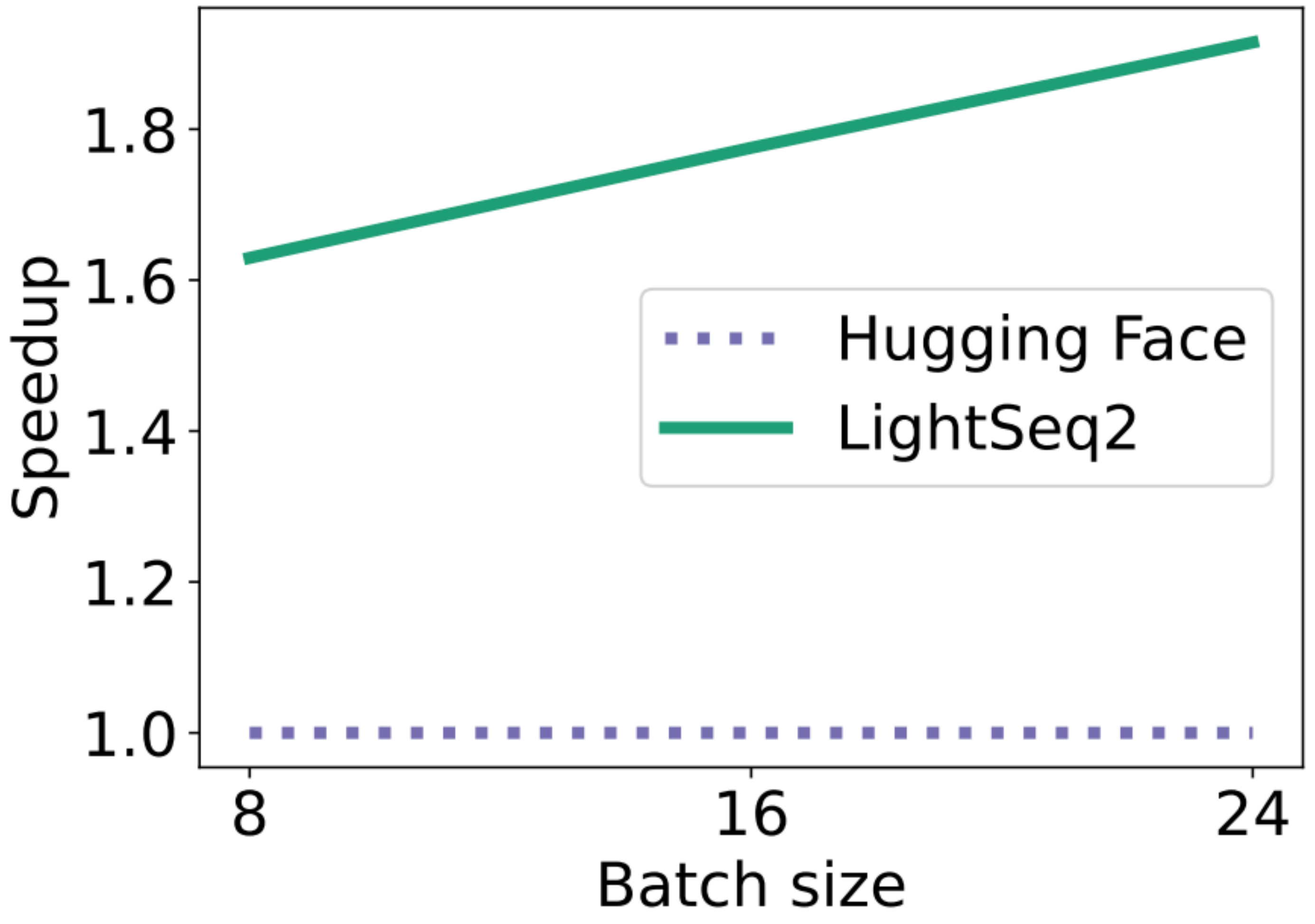
GPT2 Training: 1.6-1.9x Speedup



DataSet	WikiText
Model	GPT2
Hardware	1 Worker with 8x V100/A100
Baseline	Hugging Face (PyTorch)

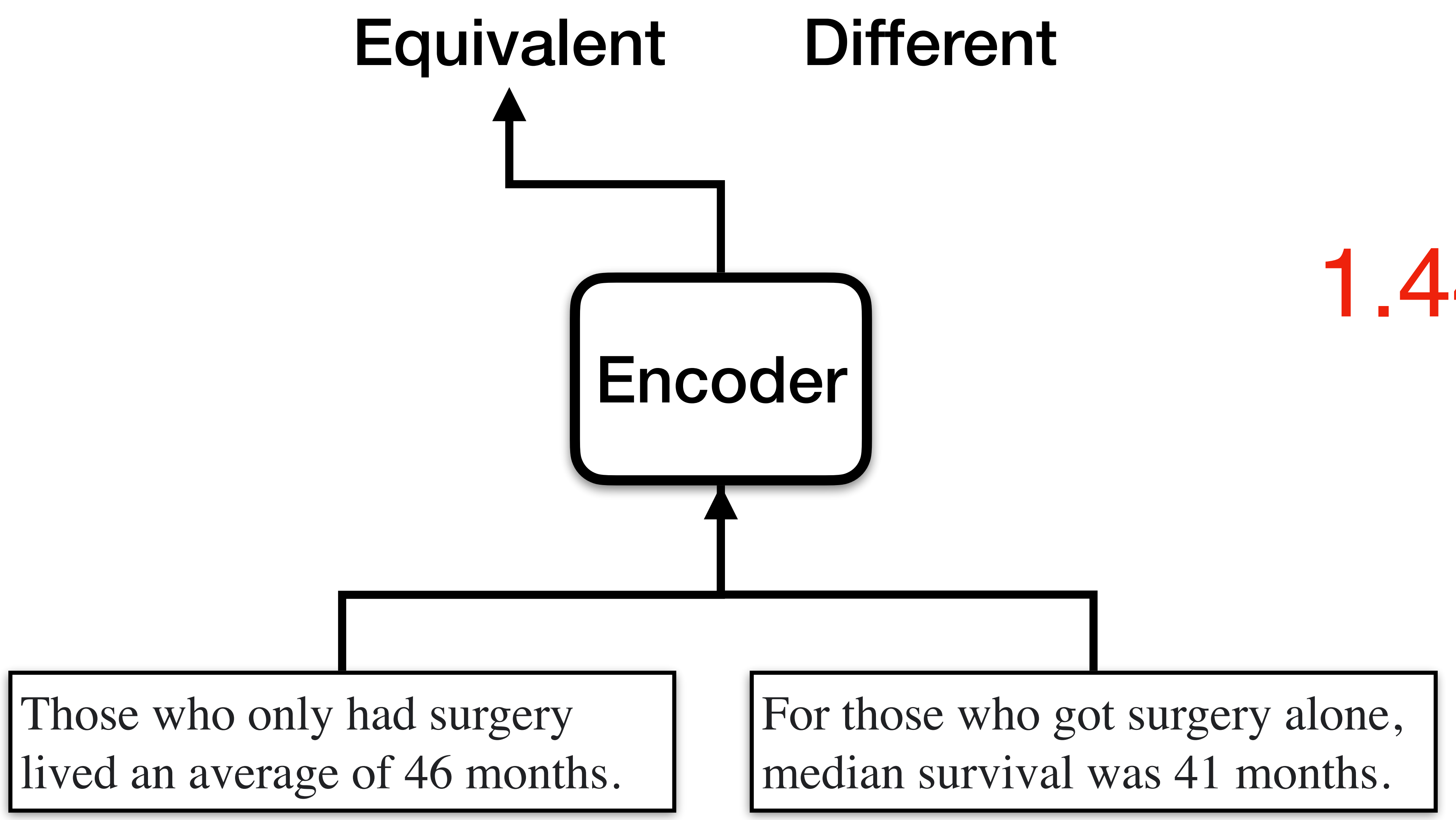


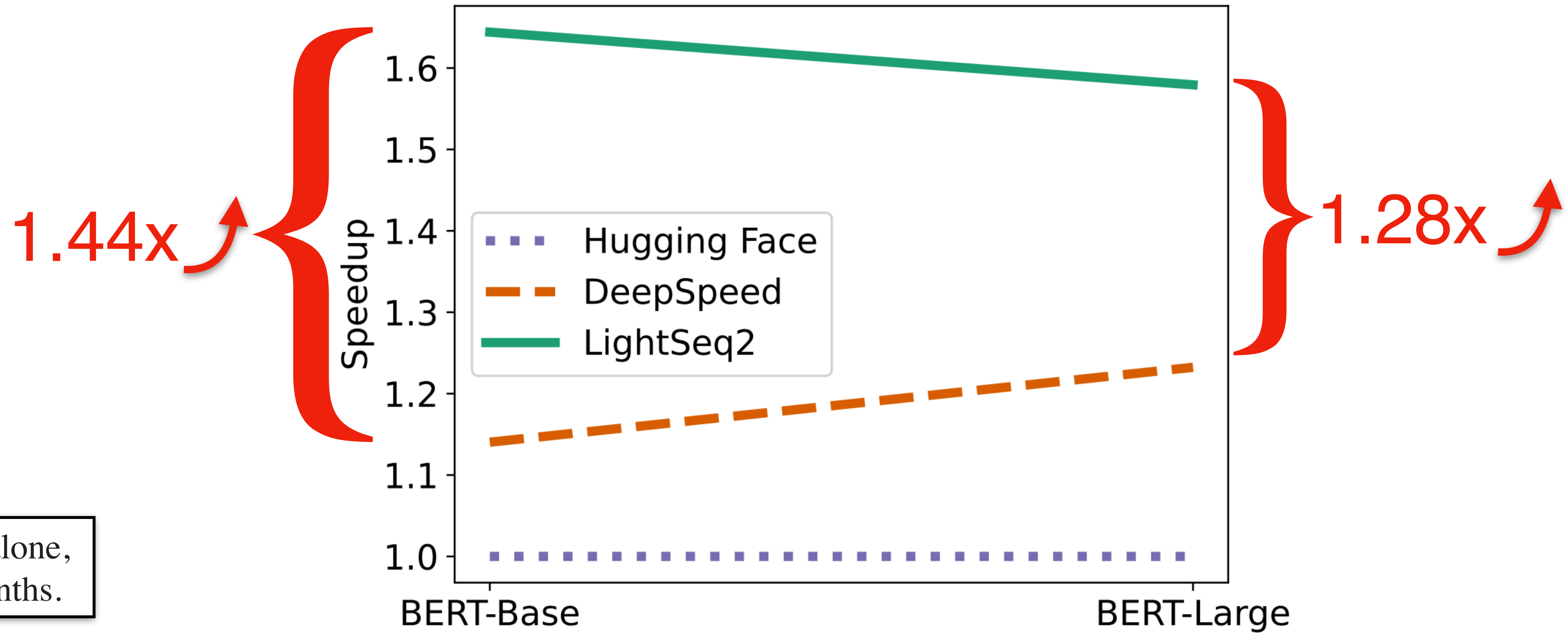
GPT2 Large trained on V100, 1.7-1.8x Speedup



GPT2 Large trained on A100, 1.6-1.9x Speedup

Paraphrase Identification: 1.28-1.44x Speedup





DataSet	Microsoft Research Paraphrase Corpus
Model	BERT
Hardware	1 Worker with 8x V100
Baseline	Hugging Face (PyTorch) DeepSpeed (Kernel Fusion)

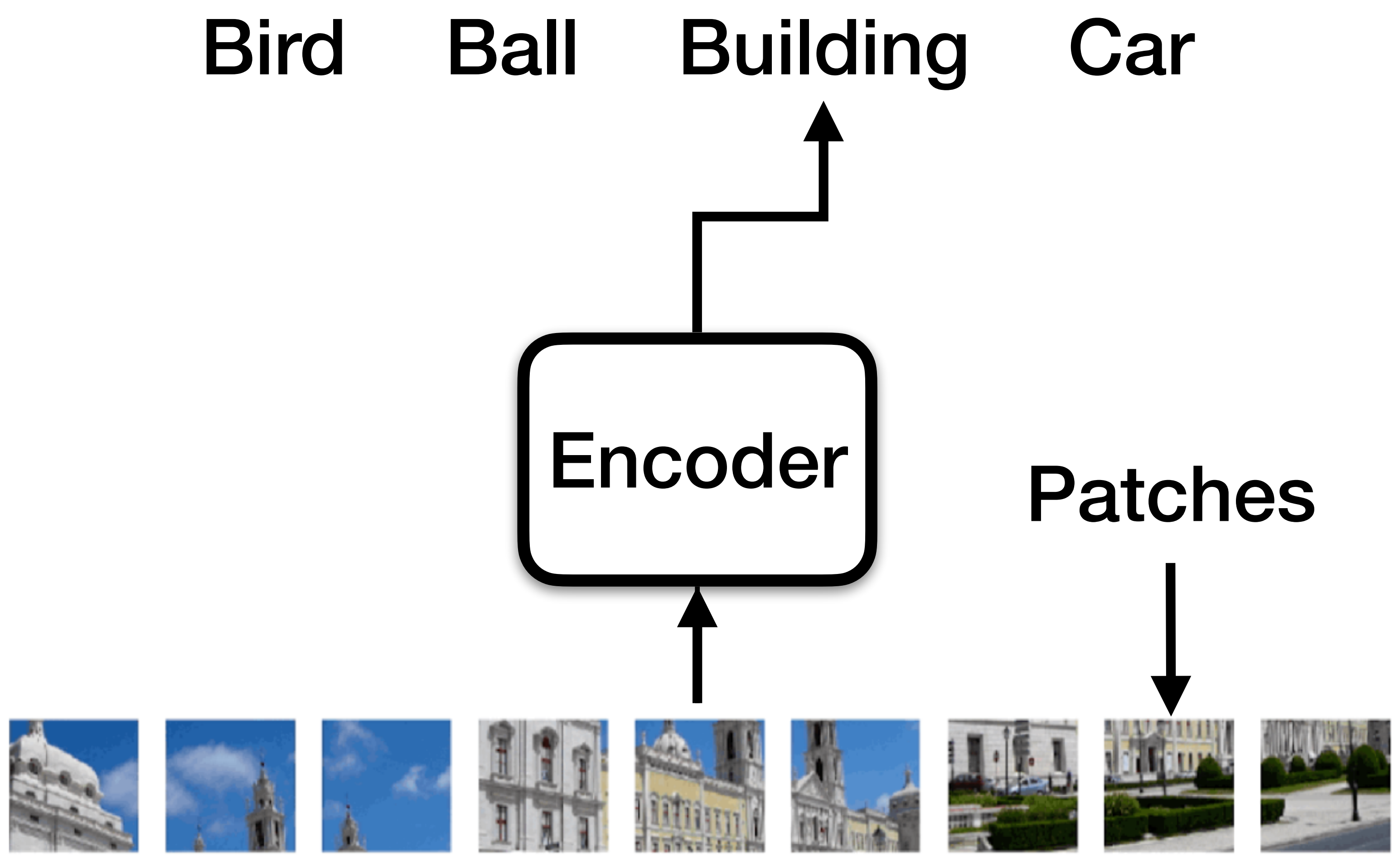
Library	Criterion	Embedding	Trainer
DeepSpeed	✘	✘	✘
LightSeq 2	✔	✔	✔

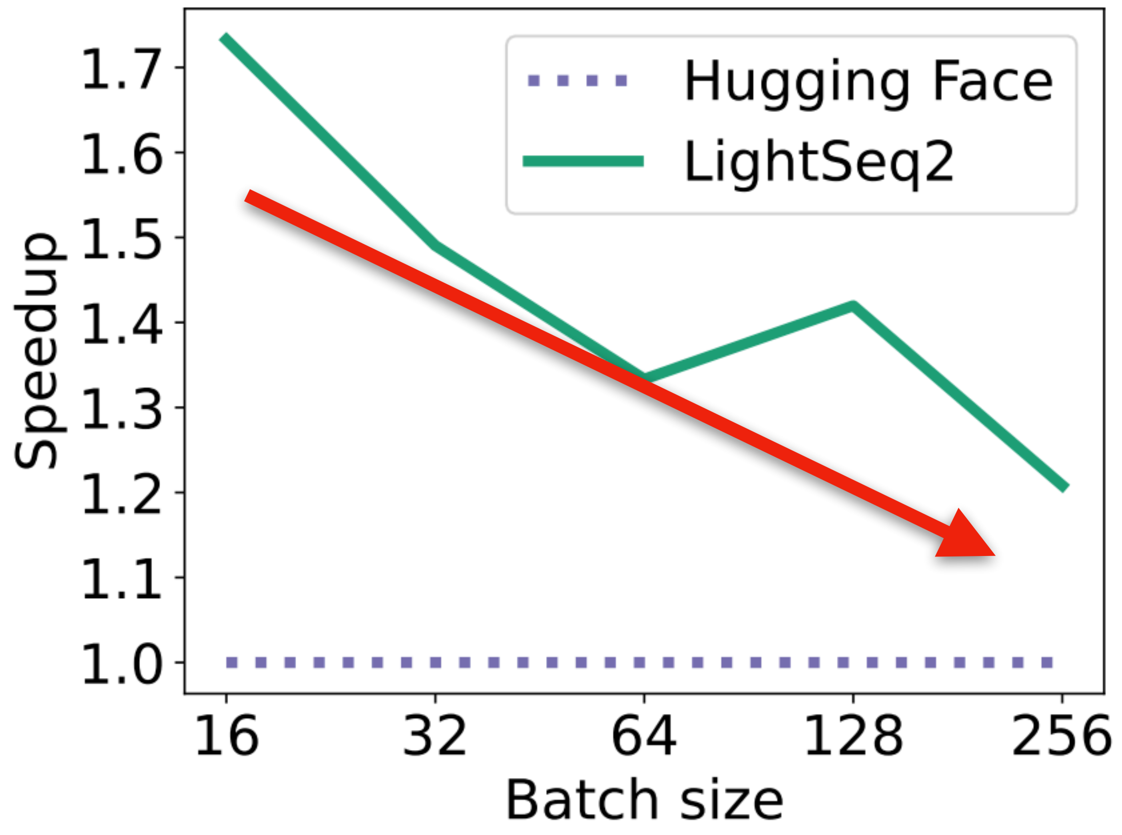
LightSeq2 vs DeepSpeed Major Differences

Image Classification: Vision Transformer (ViT)

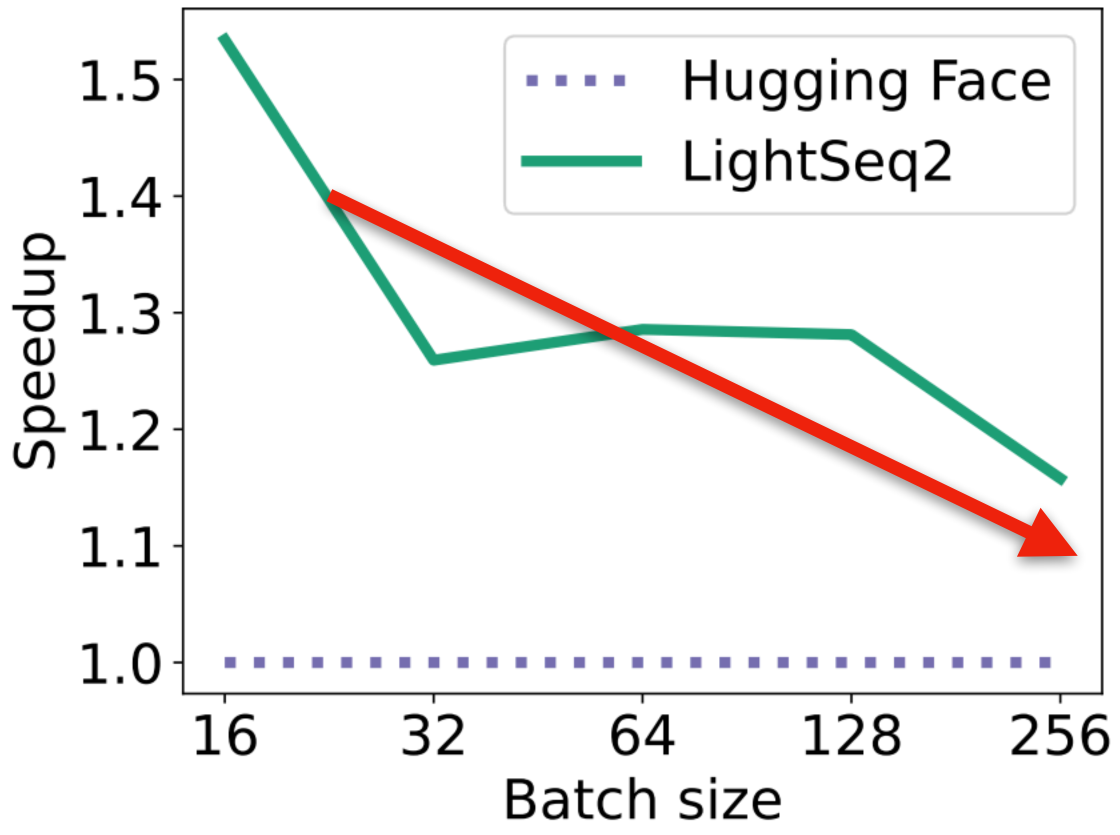
Vision Transformer for Image classification from google AI blog

Image Classification: 1.2-1.7x Speedup





Vit Base

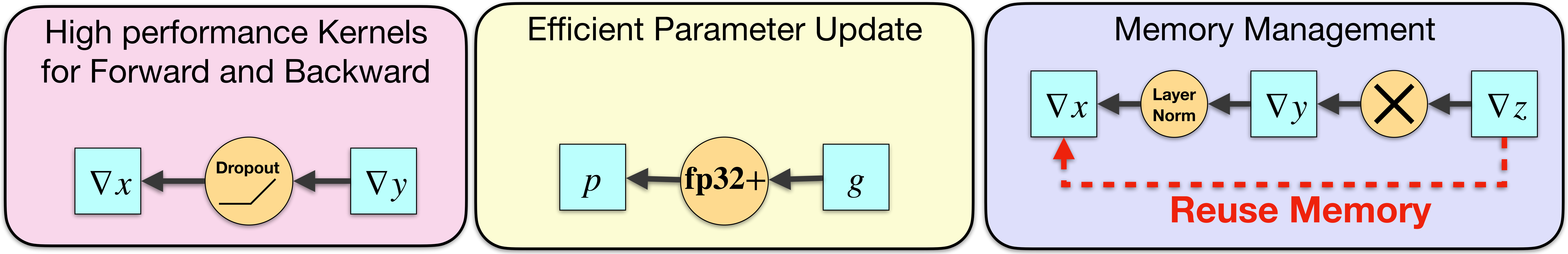


Vit Large

DataSet	CIFAR-10
Model	Vision Transformer (ViT)
Hardware	1 Worker with 8x V100
Baseline	Hugging Face (PyTorch)

Batch Size (#Patches) ⇨ Speedup

Summary for Accelerating Transformer Training



key techniques

- Kernel Fusion: merge kernels other than matmul
- Algebraic Transformation: reduce sync
- Mix-precision calculation (use half precision whenever possible)
- Memory reuse (dependent on architecture)

Fast Inference for Transformer

LightSeq: A High Performance Inference Library for Transformers

Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, Lei Li

NAACL 2021

TurboTransformers: An Efficient GPU Serving System For Transformer Models

Jiarui Fang, Yang Yu, Chengduo Zhao, Jie Zhou

PPoPP 2021

Inference: Beam Search

argmax_y P(Y|X)

1. start with empty S
2. at each step, keep k best partial sequences
3. expand them with one more forward generation
4. collect new partial results and keep top-k

Code Example

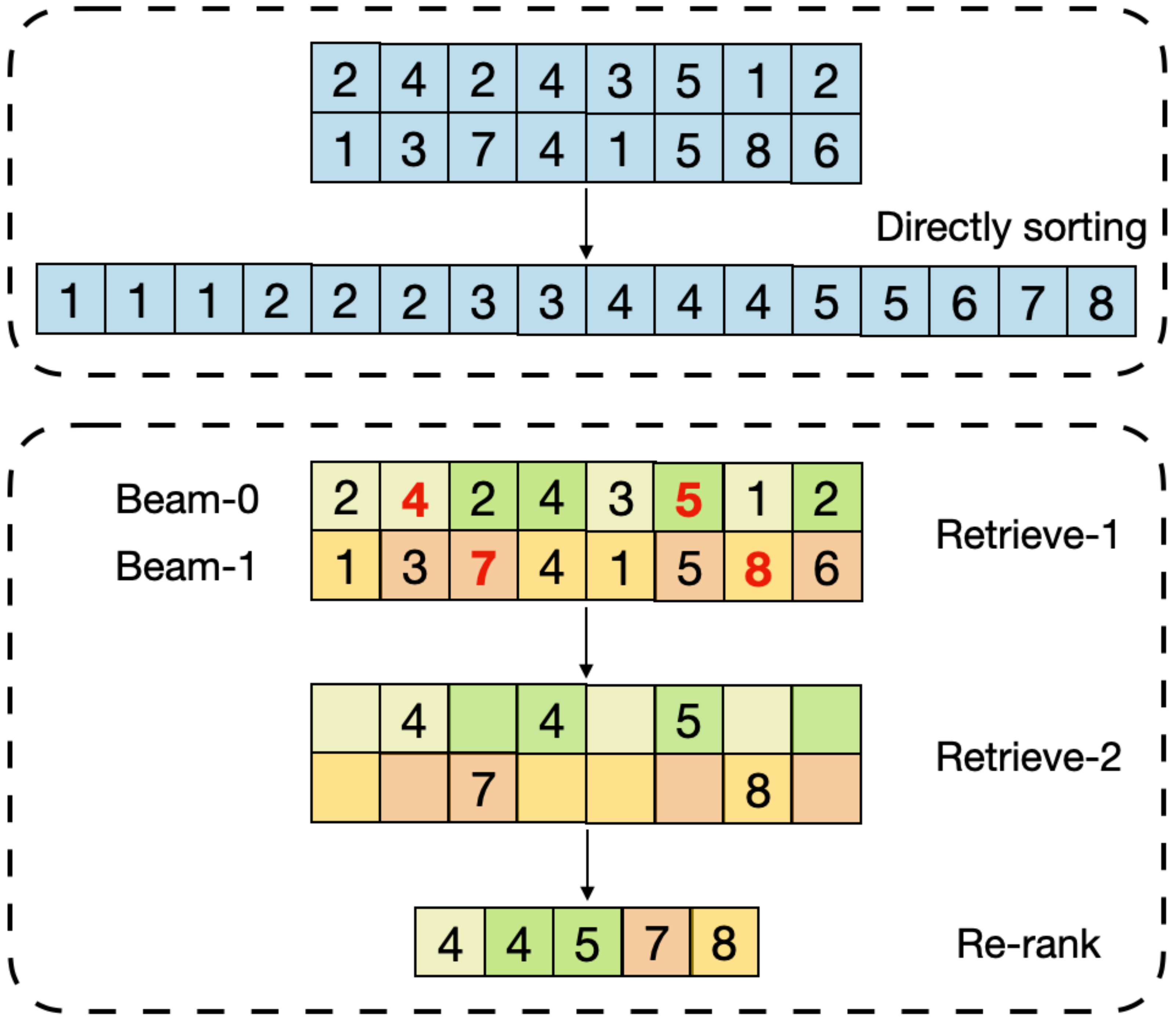
```
# 1.compute next token log probability
log_token_prob = tf.nn.log_softmax(logit) # [batch_size, beam_size,
vocab_size]
log_seq_prob += log_token_prob # [batch_size, beam_size,
vocab_size]
log_seq_prob = tf.reshape(log_seq_prob, [-1, beam_size *
vocab_size])
# 2. compute the top k sequence probability for each batch
sequence
topk_log_probs, topk_indices = tf.nn.top_k(log_seq_prob, k=K)
# 3. refresh the cache (decoder key and values) based on beam id
refresh_cache(cache, topk_indices)
```

Hierarchical Auto Regressive Search (HARS) for decoding

- Two calculations are needed in one step of beam search:
  - Compute the conditional probability of each token in vocab using Softmax
  - Select the top-k beams by sequential probability.
    - need sorting k*V elements!
- Key Idea for acceleration: HARS for decoding
  - retrieve and re-rank to reduce complexity

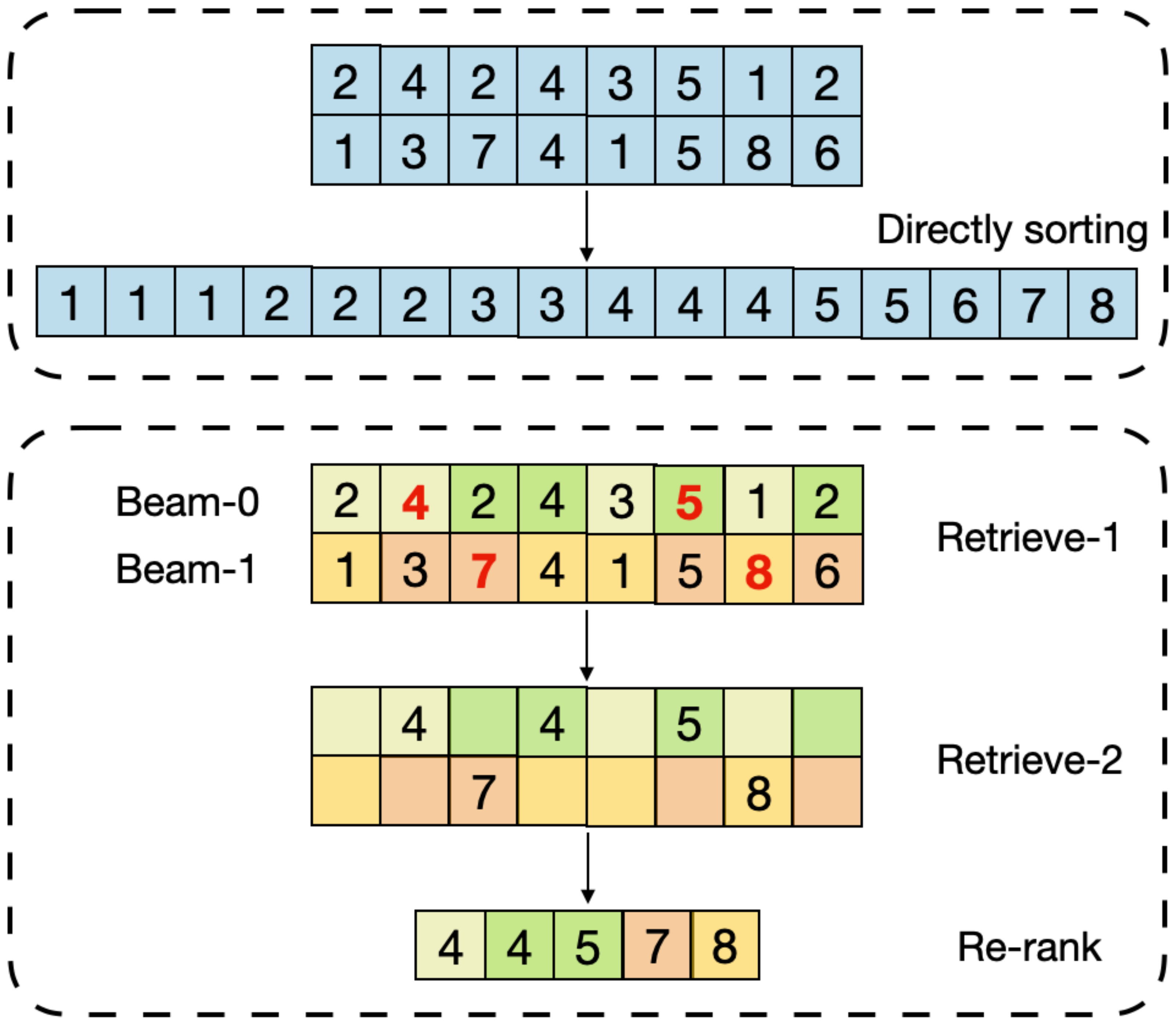
HARS - Retrieve step

- Divide logits into k groups.
- Calculate the maximum of group i, denoted as m_i, marked red
- Calculate the minimum of m_i in each beam, denoted as rough top-kth logit $\mathscr{R}$.
- Select logits larger than $\mathscr{R}$ and write them into GPU memory.



HARS - Re-rank step

- Re-rank (sorting) on candidate logits



HARS Decoding Example

- Original logits, with Beam size = 2 and Vocab size = 8.

2	4	2	4	3	5	1	2
1	3	7	4	1	5	8	6

HARS Decoding Example

- For each beam, divide the eight logits into two groups.

2	4	2	4	3	5	1	2
1	3	7	4	1	5	8	6

HARS Decoding Example

- Calculate the *maximum* of each group.

2	**4**	2	4	3	**5**	1	2
1	3	**7**	4	1	5	**8**	6

HARS Decoding Example

- For each beam, calculate the *minimum* of each group's maximum.

2	**4**	2	4	3	5	1	2
1	3	**7**	4	1	5	8	6

HARS Decoding Example

- For each beam, select logits larger than the minimum in previous step.

2	**4**	2	**4**	3	**5**	1	2
1	3	**7**	4	1	5	**8**	6

HARS Decoding Example

- For each beam, select logits larger than the minimum in previous step.

	4		4		5		
		7				8	

HARS Decoding Example

- Re-rank only on five logits

4	4	5	7	8

Details in GPU Inference Implementation

- share tensor memory across layers
- mixed precision computation, mostly using FP16 for computation
- Using `float4 and half2 to increase bandwidth`
- No need to keep intermediate results and gradients during infernece, similar to `with torch.no_grad()`

LightSeq Software Architecture

API Example: HuggingFace BERT

```
from lightseq.training import LSTransformerEncoderLayer

config = LSTransformerEncoderLayer.get_config(
            model="bert-base",
            max_batch_tokens=4096,
            max_seq_len=512,
            fp16=True,
            local_rank=0)

ls_layer = LSTransformerEncoderLayer(config)

# replace the 1st Hugging Face layer with LightSeq2
bert_model.layer[0] = ls_layer
```

Step 1: import LightSeq

Step 2: Config and define your model/layer

Step 3: Replace HuggingFace Layer

LightSeq + Fairseq Integration

```
lightseq-train DATA_SET \
    --task translation \
    --arch ls_transformer_wmt_en_de_big_t2t \
    --optimizer ls_adam \
    --criterion ls_label_smoothed_cross_entropy \
    --OTHER_PARAMS
```

- LightSeq can be seamlessly used with Fairseq
  - Training: `lightseq-train`, using prefix `ls_`

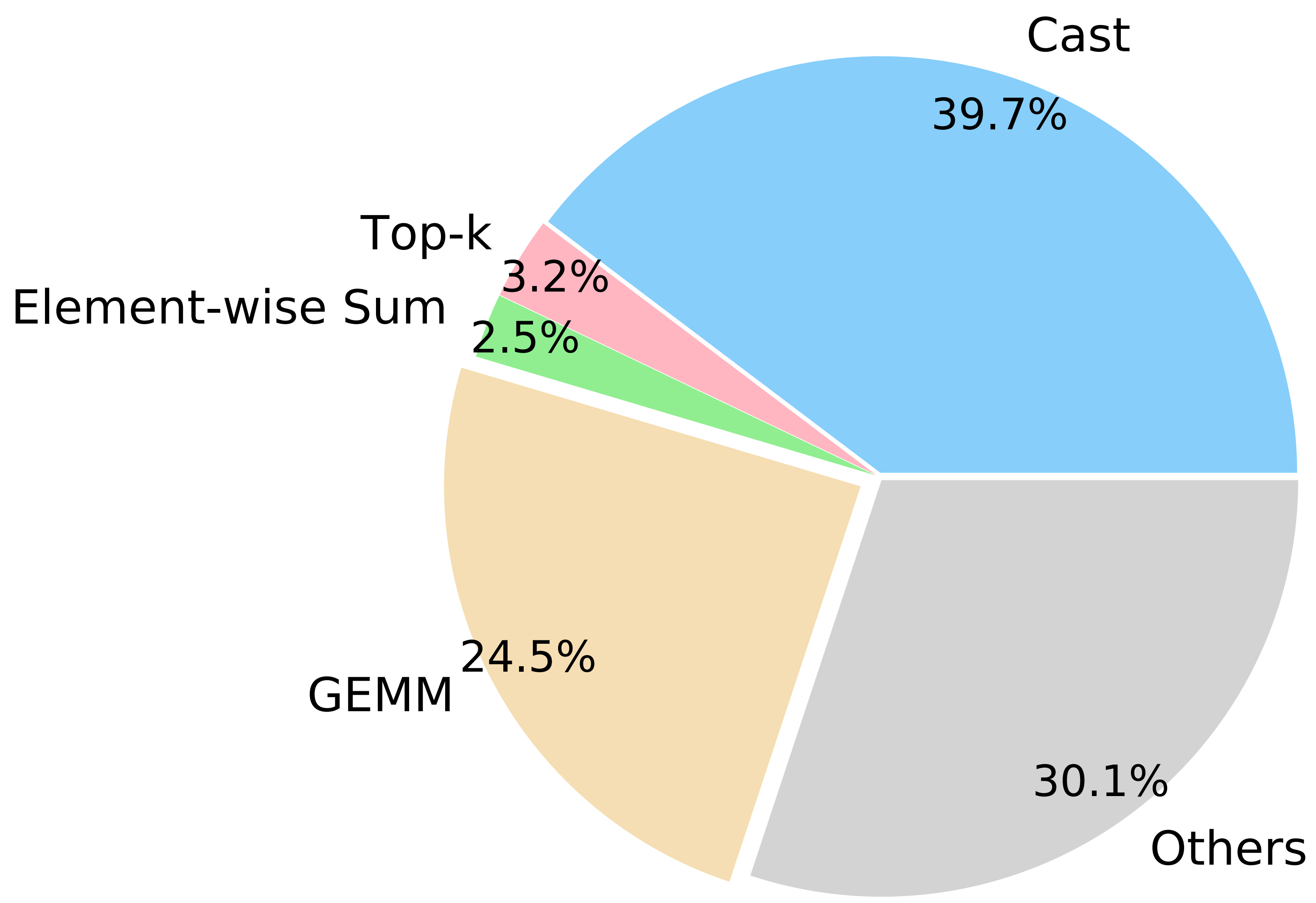
LightSeq + Fairseq Integration

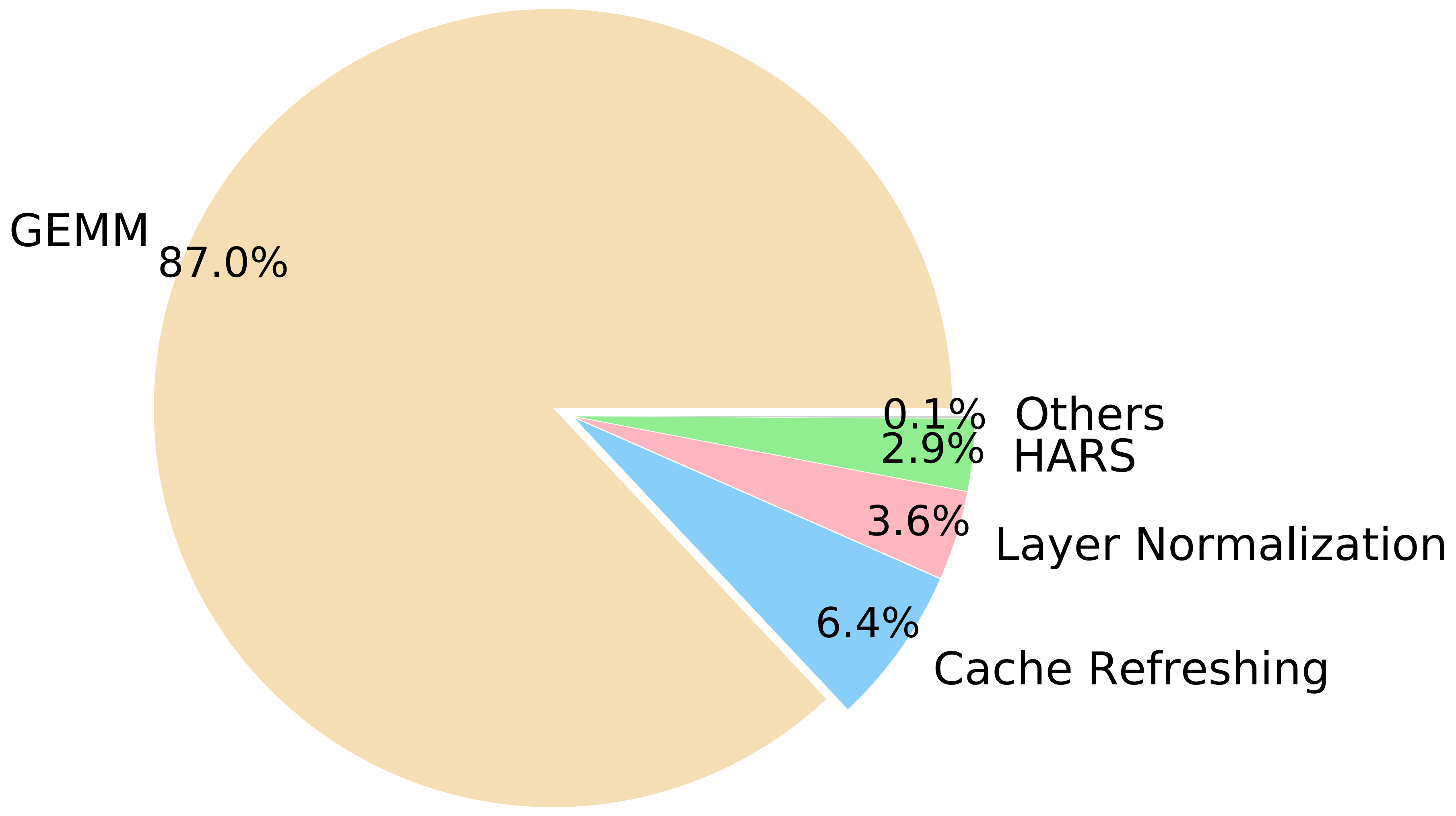
- LightSeq accelerated Transformer embedding / encoder / decoder、Adam and cross entropy for Fairseq
- LightSeq is compatible with Fairseq cache and reorder
- LightSeq is compatible with Apex and DeepSpeed together with Fairseq。

```
deepspeed ds_fairseq.py DATA_SET \
    --user-dir fs_modules \
    --deepspeed_config deepspeed_config.json \
    --task translation \
    --arch ls_transformer_wmt_en_de_big_t2t \
    --optimizer ls_adam \
    --criterion ls_label_smoothed_cross_entropy \
    --OTHER_PARAMS
```

GPU Occupation

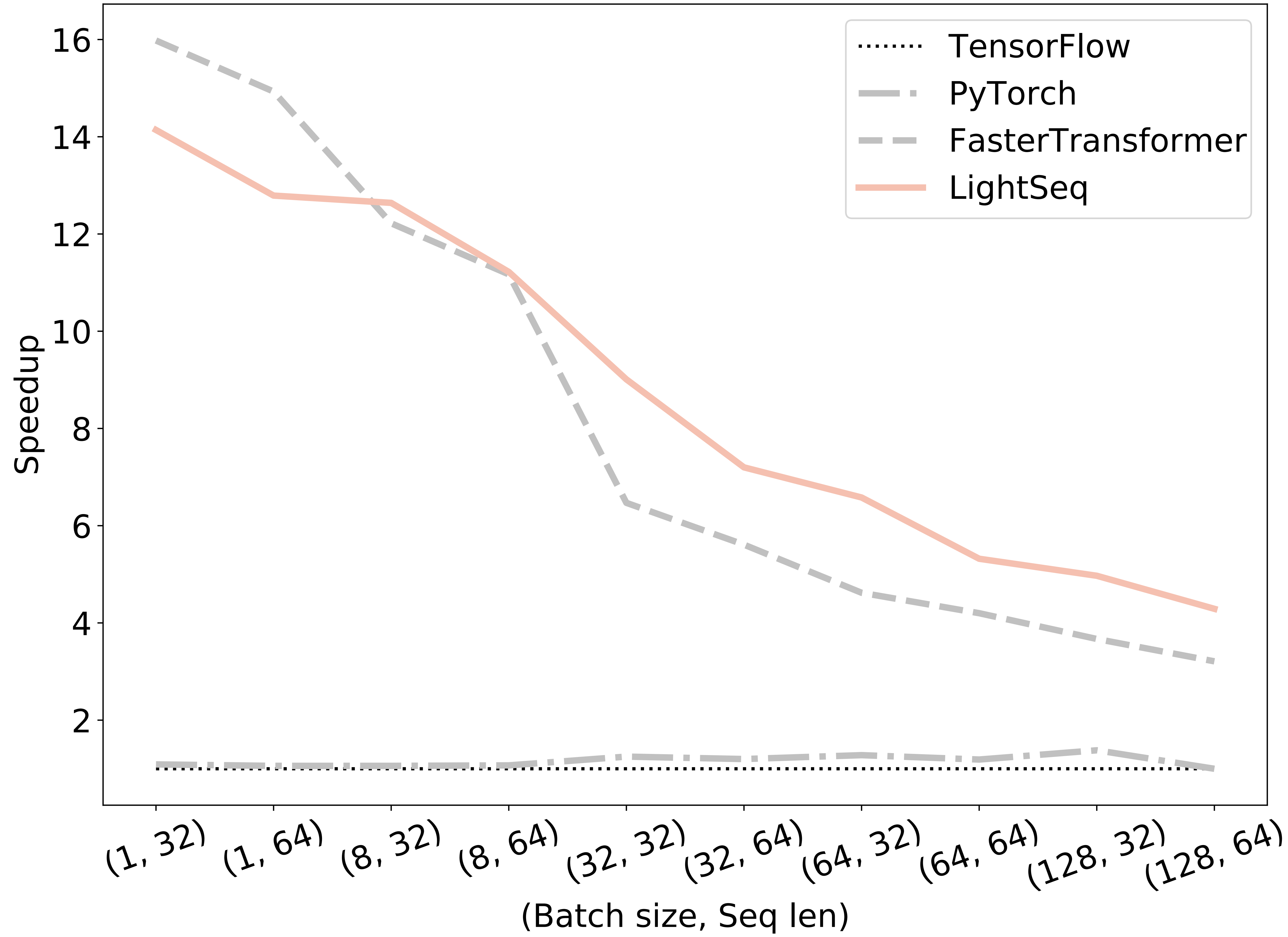
- LightSeq greatly reduces the proportion of kernels other than GEMM.





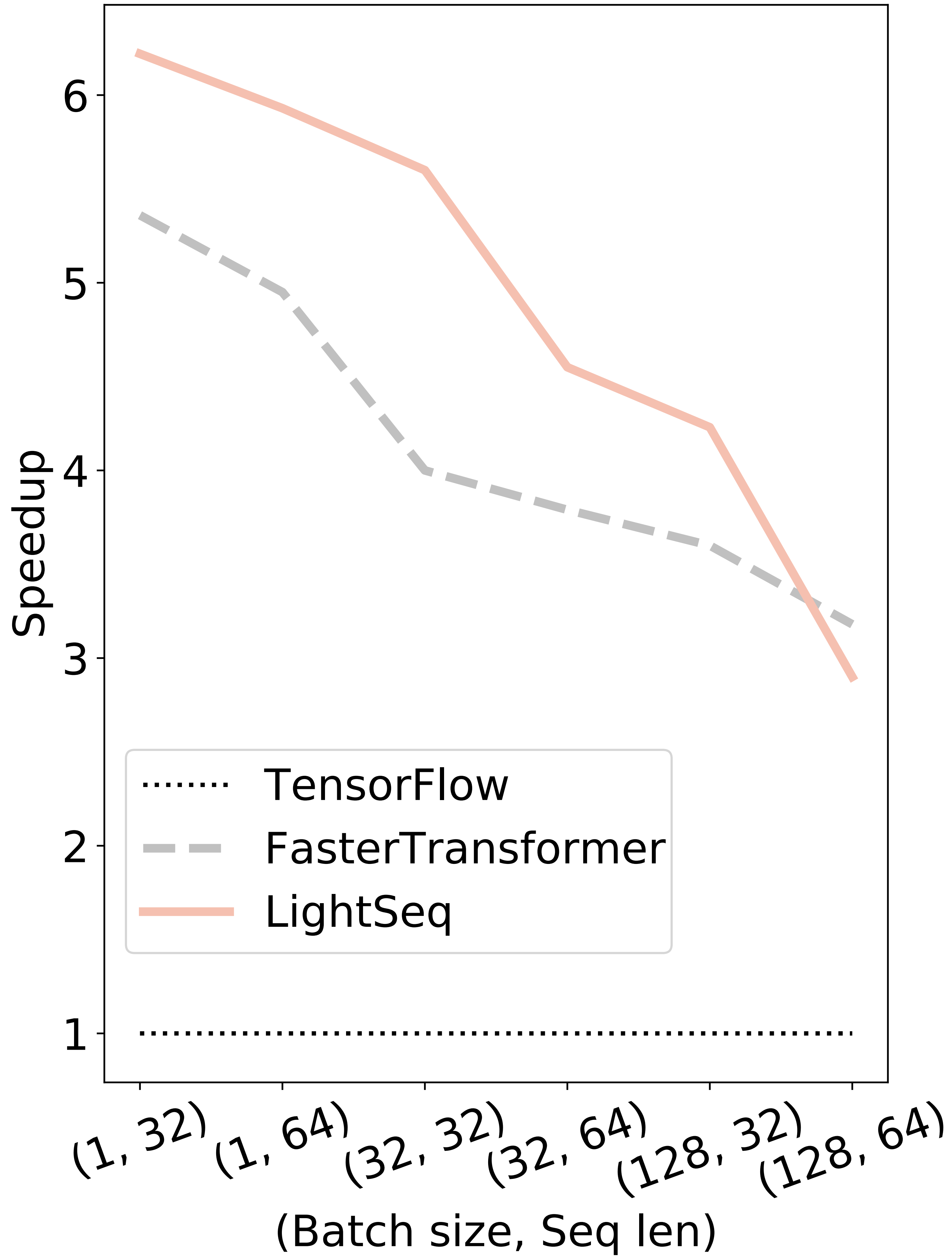
Machine Translation Inference: 14x speedup

- LightSeq outperforms others in most cases, especially in large batch size.



GPT2 Inference: 6x speedup

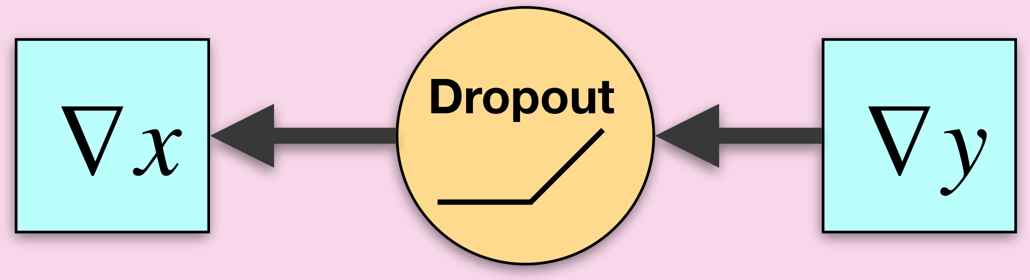
- LightSeq outperforms others in most cases



Summary

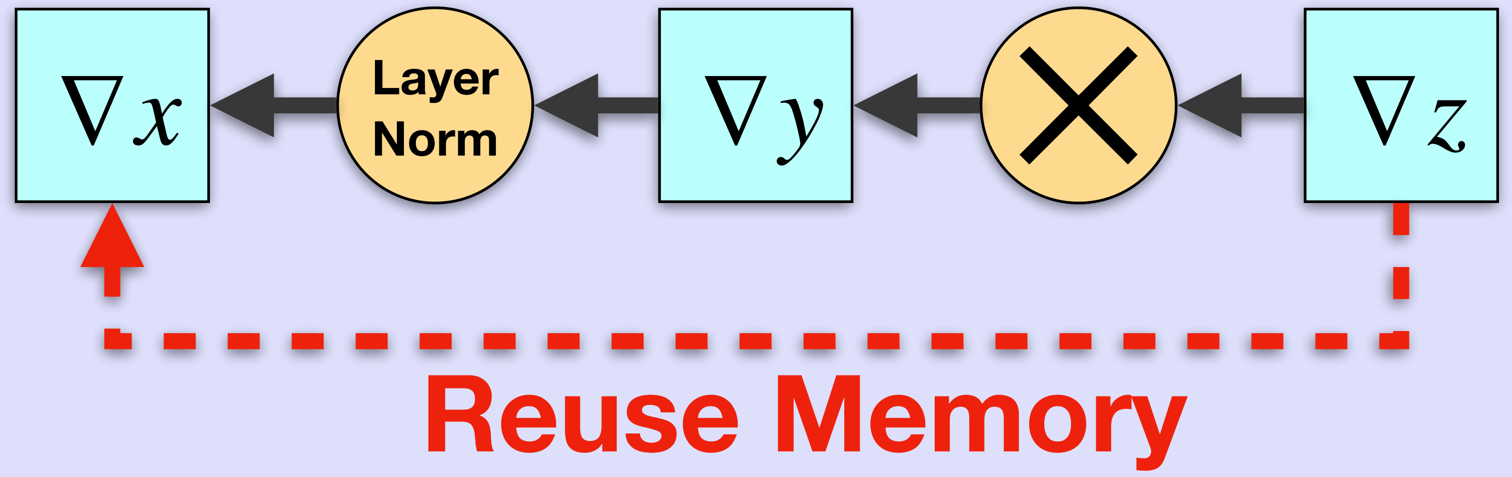
We optimize the training process from 3 aspects

High performance Kernels for Forward and Backward via Fusion & Algebra trick



Efficient Parameter Update via mixed precision

Memory Management

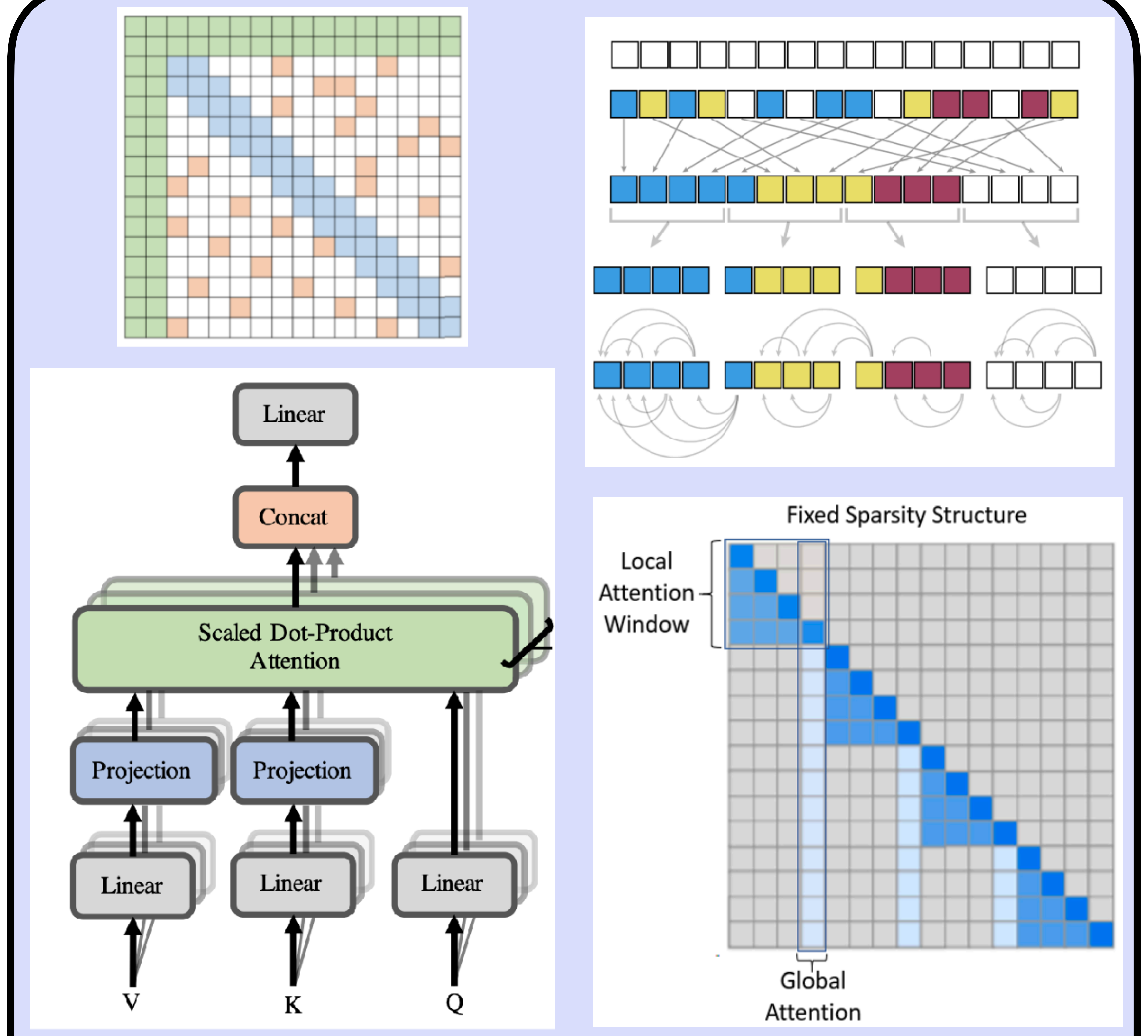


Operators that can be reused in Other Networks:
Dropout, LayerNorm, Softmax, Cross Entropy

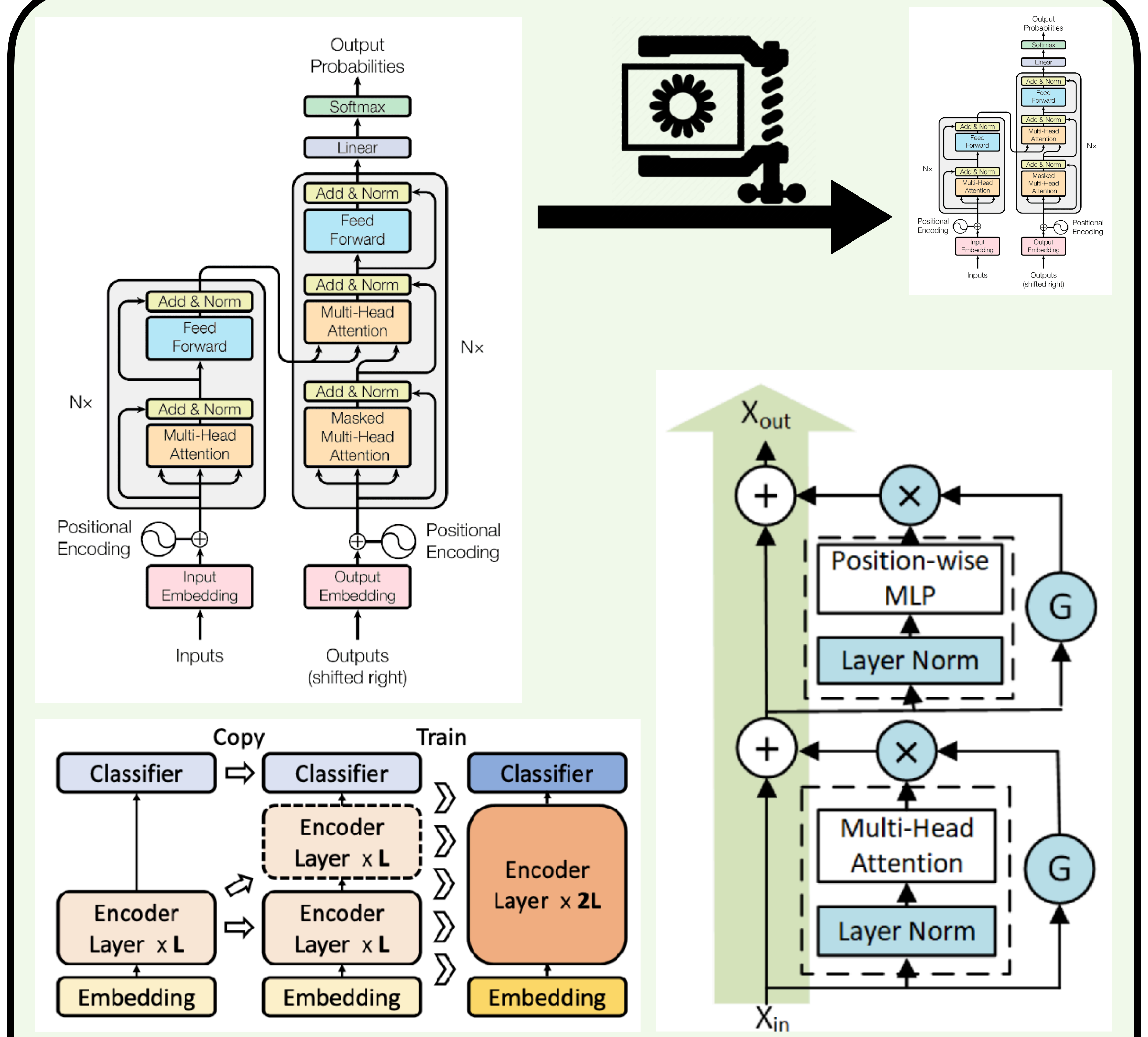
Accelerating decoding

hierachical auto-regressive search for large vocabulary converting sorting to parallel operations (max, filter, re-rank)

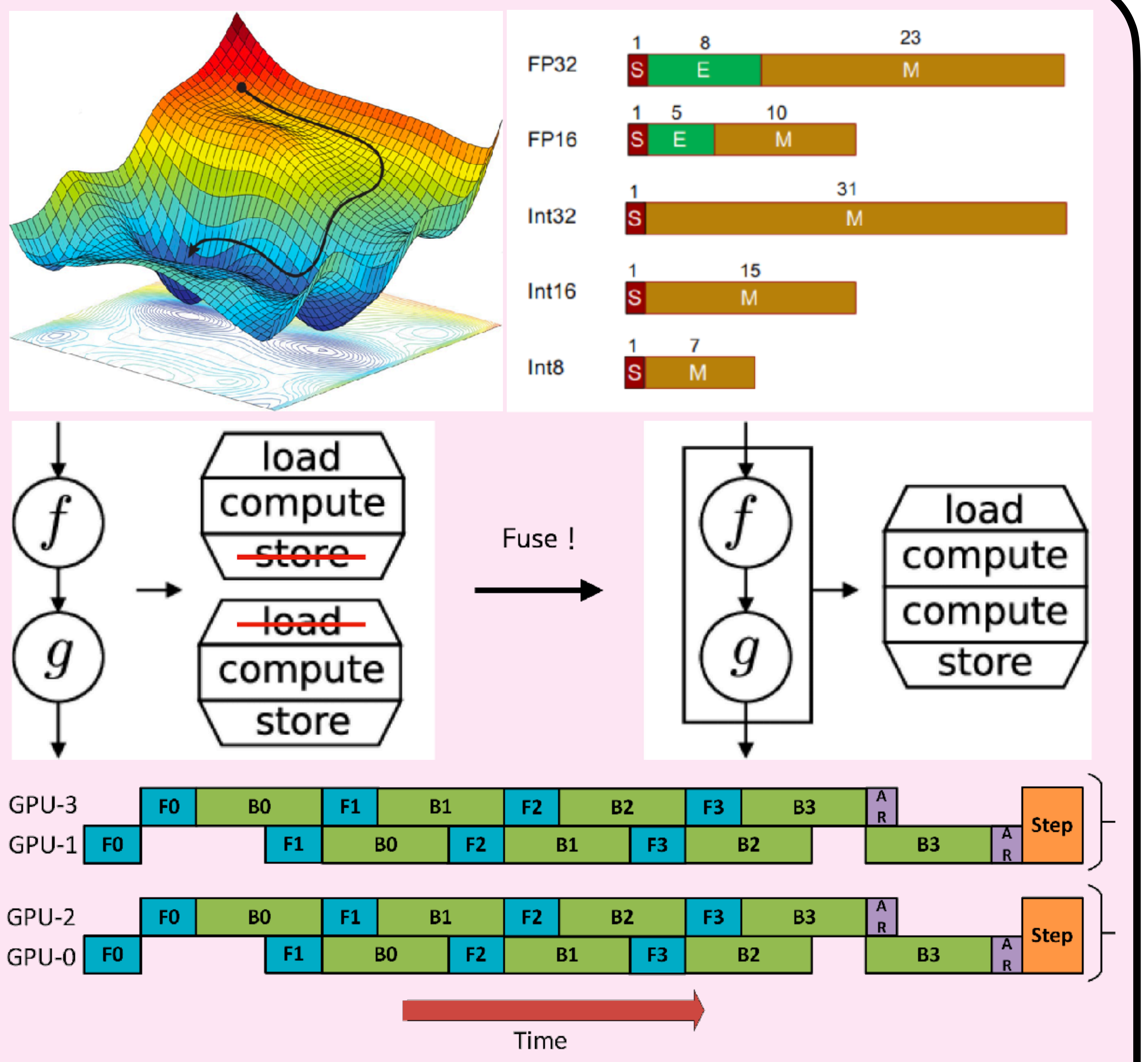
Other Approaches for Acceleration



Alternative Model Structures: Linformer, Reformer



Training Strategy: Shallow to Deep, Layer Dropout



Efficient Computation: LAMB, Quantization, Hardware Optimization code is available at

https://github.com/bytedance/lightseq

Stay tuned for Deepseek's FlashMLA

- High-performance decoding kernel optimized for Multi-head Latent Attention (MLA) on Hopper GPUs
- https://github.com/deepseek-ai/FlashMLA

Reading for Next

- PyTorch Distributed: Experiences on Accelerating Data Parallel Training
- PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel