Reducing Activation Recomputation in Large Language Models

Ahan Gupta

Korthikanti et al.

Agenda

Tensor Parallelism

The faults in Tensor Parallelism

Sequence + Tensor Parallelism

Activation Checkpointing

Tensor Parallelism

Shoeybi et al.

Motivation

Larger Models yield better quality (provided trained on more data!)

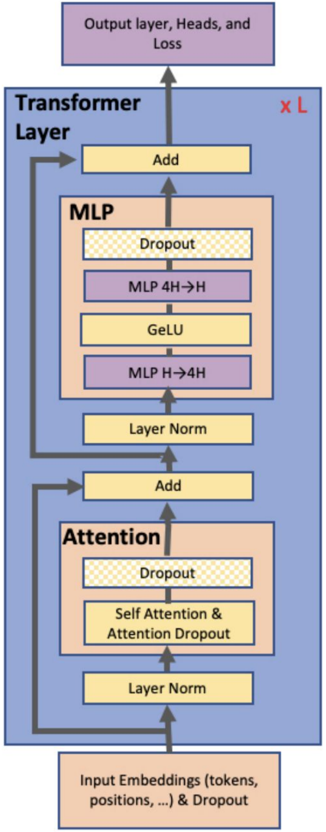
Really large models hit the memory wall

Combination of Data + Model Parallelism is *complicated* and require model re-writing

Solution: Simple intra-layer model parallelism (Tensor parallelism) but this is *not sufficient*

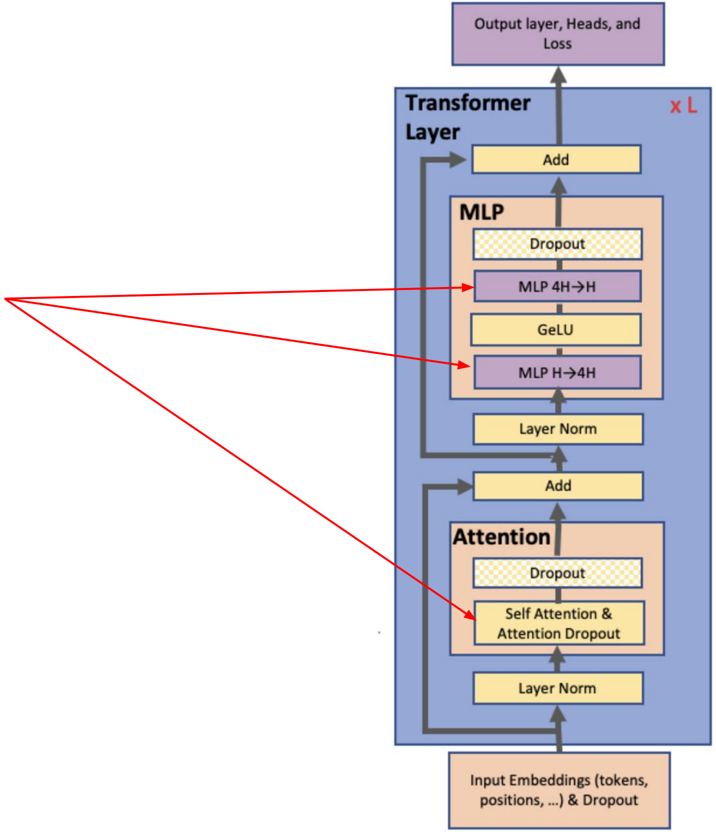
So we bring it one step further.

Tensor Parallelism



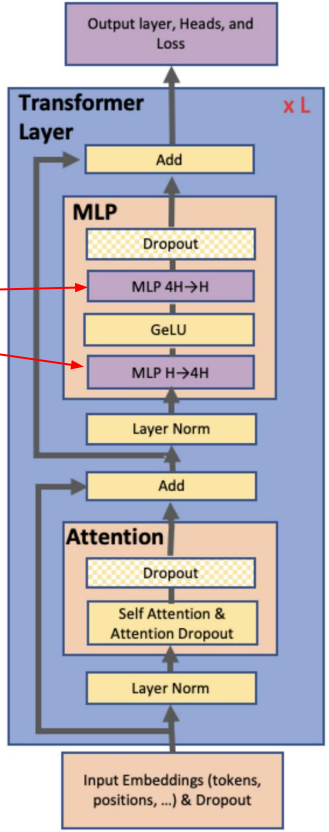
Tensor Parallelism

Apply Tensor Parallelism



Tensor Parallelism

How do we apply Tensor Parallelism to these?



Let's see how we can do this if we want to effectively use 2 GPUs

Tensor Parallelism - FFNs (Better Partitioning)

Intuition:

Split Weights across columns

Replicate Input across GPUs

Tensor Parallelism - FFNs (Better Partitioning)

$$O^1 = XA$$
$$O^2 = Gelu(O^1)$$
$$O^3 = O^2 A^1$$
$$O^4 = Dropout(O^3)$$

Tensor Parallelism - FFNs (Better Partitioning)

$$[A_1|A_2]$$

$$O^1 = XA$$
$$O^2 = Gelu(O^1)$$
$$O^3 = O^2A^1$$
$$O^4 = Dropout(O^3)$$

Step 1: Replicate data, partition weights for local MatMul

GPU 0

Computes:
$$XA_1$$

GPU 1

Computes:
$$XA_2$$

Tensor Parallelism - FFNs (Better Partitioning)

$$[A_1|A_2]$$

$$O^1 = XA$$

$$O^2 = Gelu(O^1)$$

$$O^3 = O^2A^1$$

$$O^4 = Dropout(O^3)$$

Step 2: Compute local GELU's

GPU 0

Computes:

$$Gelu(XA_1)$$

GPU 1

Computes:

$$Gelu(XA_2)$$

Tensor Parallelism - FFNs (Better Partitioning)

$$[A_1 | A_2]$$

$$O^1 = XA$$

$$O^2 = Gelu(O^1)$$

$$O^3 = O^2 A^1$$

$$O^4 = Dropout(O^3)$$

$$\begin{bmatrix} A_1^1 \\ A_2^1 \end{bmatrix}$$

Step 3: Another Partitioning of the weights and local MatMul.

GPU 0

Computes:

$$Gelu(XA_1)A_1^1$$

GPU 1

Computes:

$$Gelu(XA_2)A_2^1$$

Tensor Parallelism - FFNs (Better Partitioning)

$$[A_1 | A_2]$$

$$O^1 = XA$$
$$O^2 = Gelu(O^1)$$
$$O^3 = O^2 A^1$$
$$O^4 = Dropout(O^3)$$

$$\begin{bmatrix} A_1^1 \\ A_2^1 \end{bmatrix}$$

Step 4: All-reduce, synchronize data and add.

GPU 0

Computes:

GPU 1

Computes:

$$Gelu(XA_1)A_1^1 + Gelu(XA_2)A_2^1$$

$$Gelu(XA_1)A_1^1 + Gelu(XA_2)A_2^1$$

Tensor Parallelism - FFNs (Better Partitioning)

$$[A_1|A_2]$$

$$O^1 = XA$$

$$O^2 = Gelu(O^1)$$

$$O^3 = O^2 A^1$$

$$\begin{bmatrix} A_1^1 \\ A_2^1 \end{bmatrix}$$

$$O^4 = Dropout(O^3)$$

Step 4: Dropout.

GPU 0

Computes:

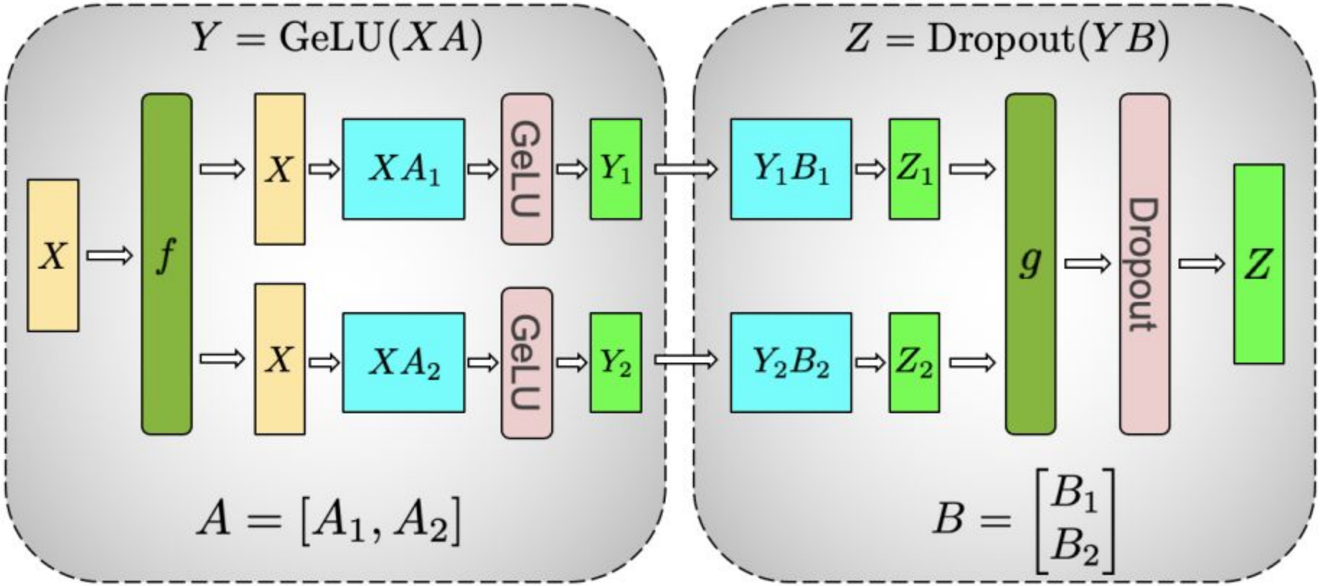
$$Dropout(Gelu(XA_1)A_1^1 + Gelu(XA_2)A_2^1)$$

GPU 1

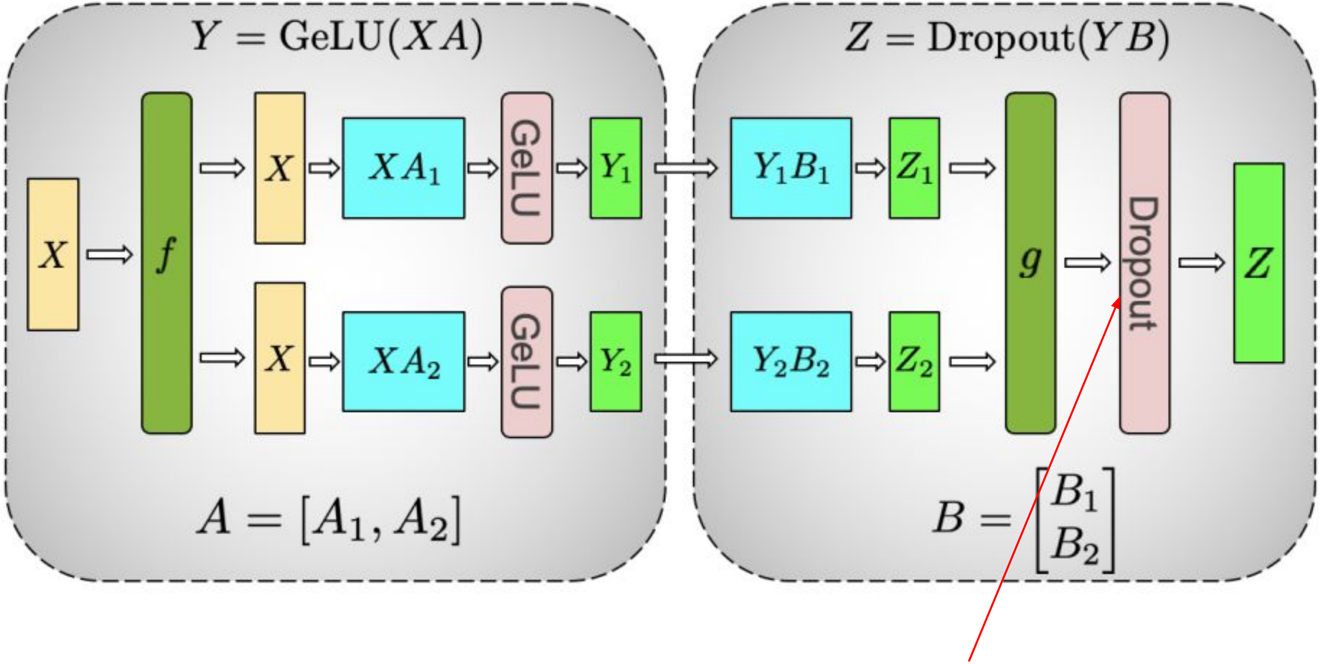
Computes:

$$Dropout(Gelu(XA_1)A_1^1 + Gelu(XA_2)A_2^1)$$

Tensor Parallelism - FFNs (Better Partitioning)



Tensor Parallelism - FFNs (Better Partitioning)



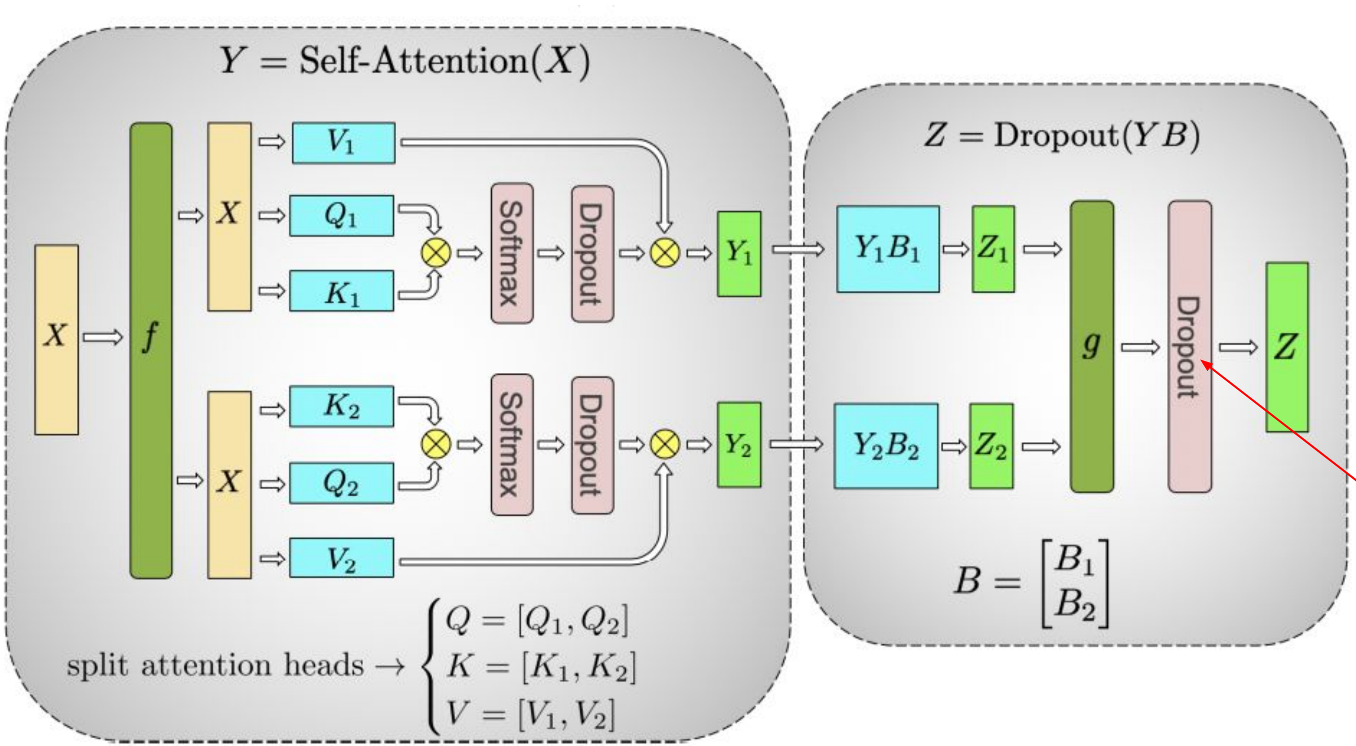
Done on both the GPUs with all the data (redundancies)

Tensor Parallelism - Self-Attention

Concept is the same

Partitioning Scheme is identical

Tensor Parallelism - Self-Attention



Duplication again

Reducing Activation Computation in Large Language Models

Motivation

Layernorm and Dropout in Tensor Parallelism introduces redundant work

Layernorm + Dropout are memory bound but require loads of activations

Duplicating their activations increases Memory usage *drastically*

Intuition

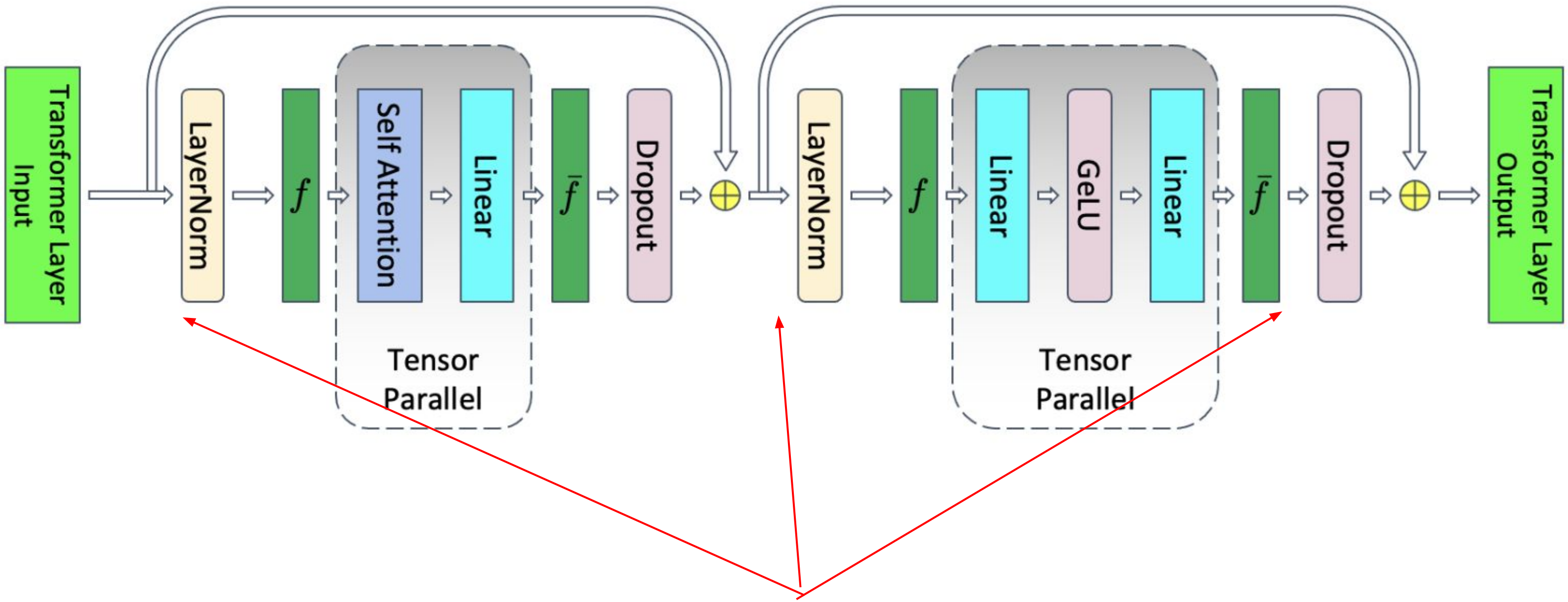
We parallelise both the layernorm and dropout across GPUs, reducing redundant work (save overall memory consumption)

We parallelise across the sequence dimension (Sequence Parallelism)

Put on special activation checkpointing to save memory!

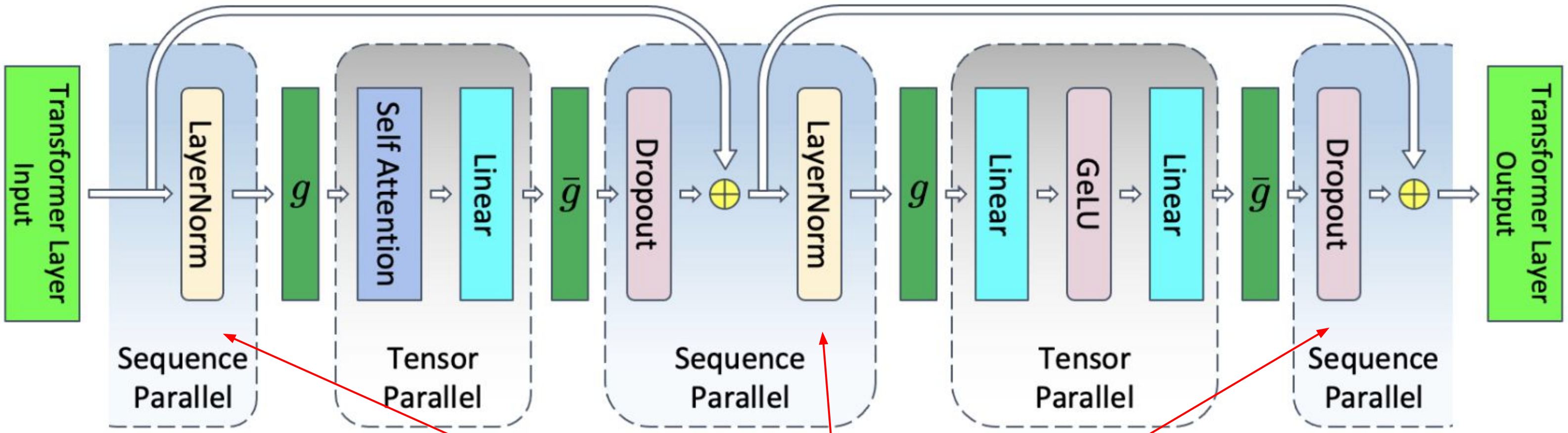
Try not to materialise the full input matrix across *any single GPU*

Intuition



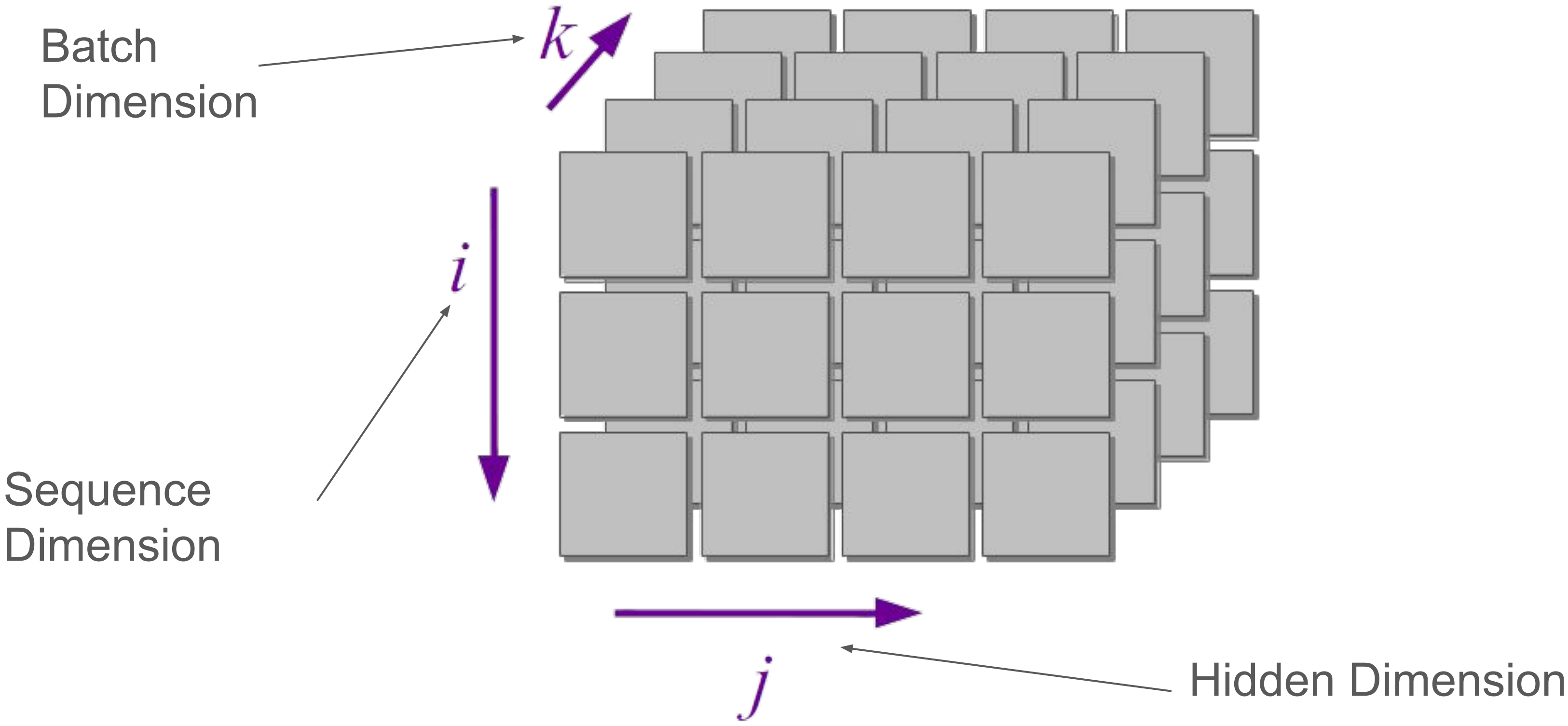
Normally, every GPU will do identical work on these Dropouts and LayerNorms (Duplication)

Intuition

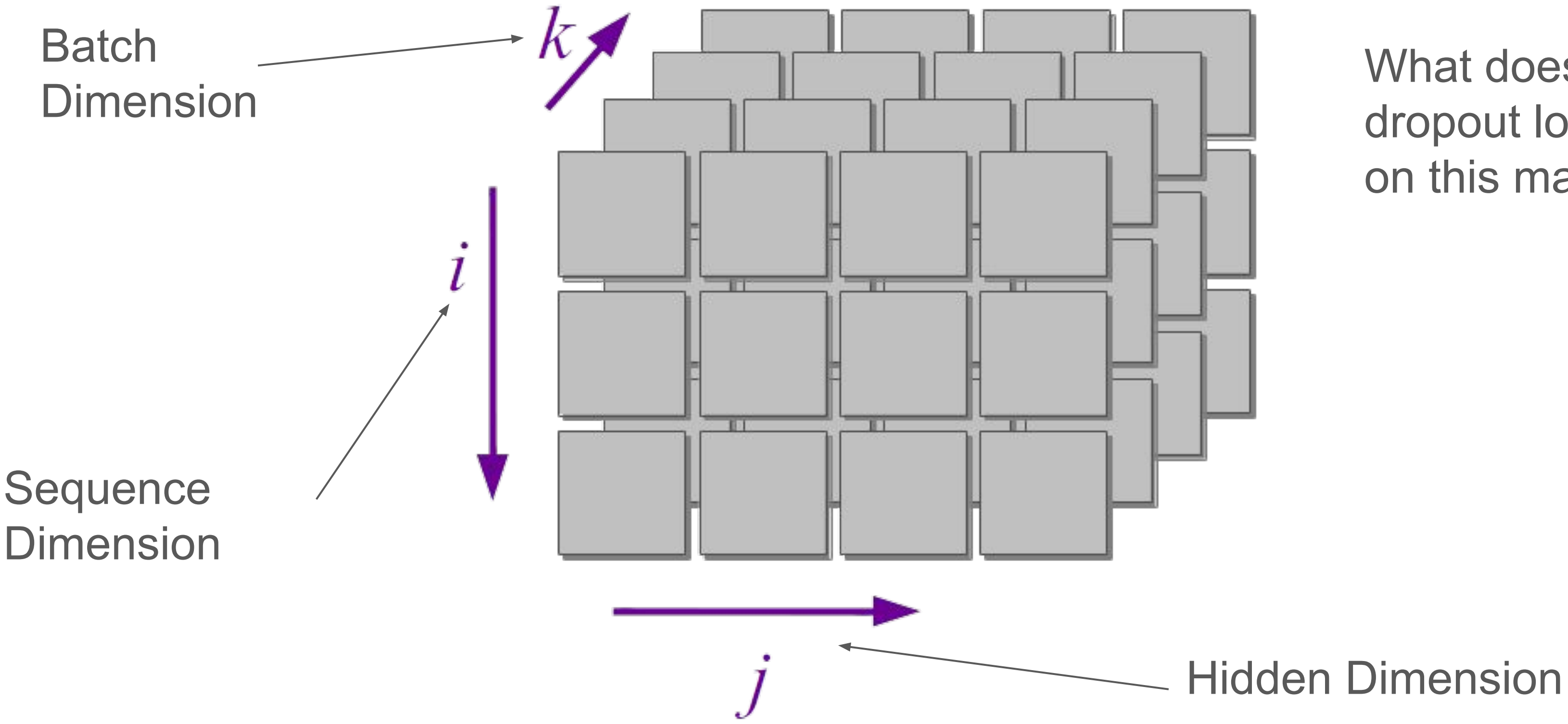


Reducing duplication by parallelising these layers as well

Input Visualisation

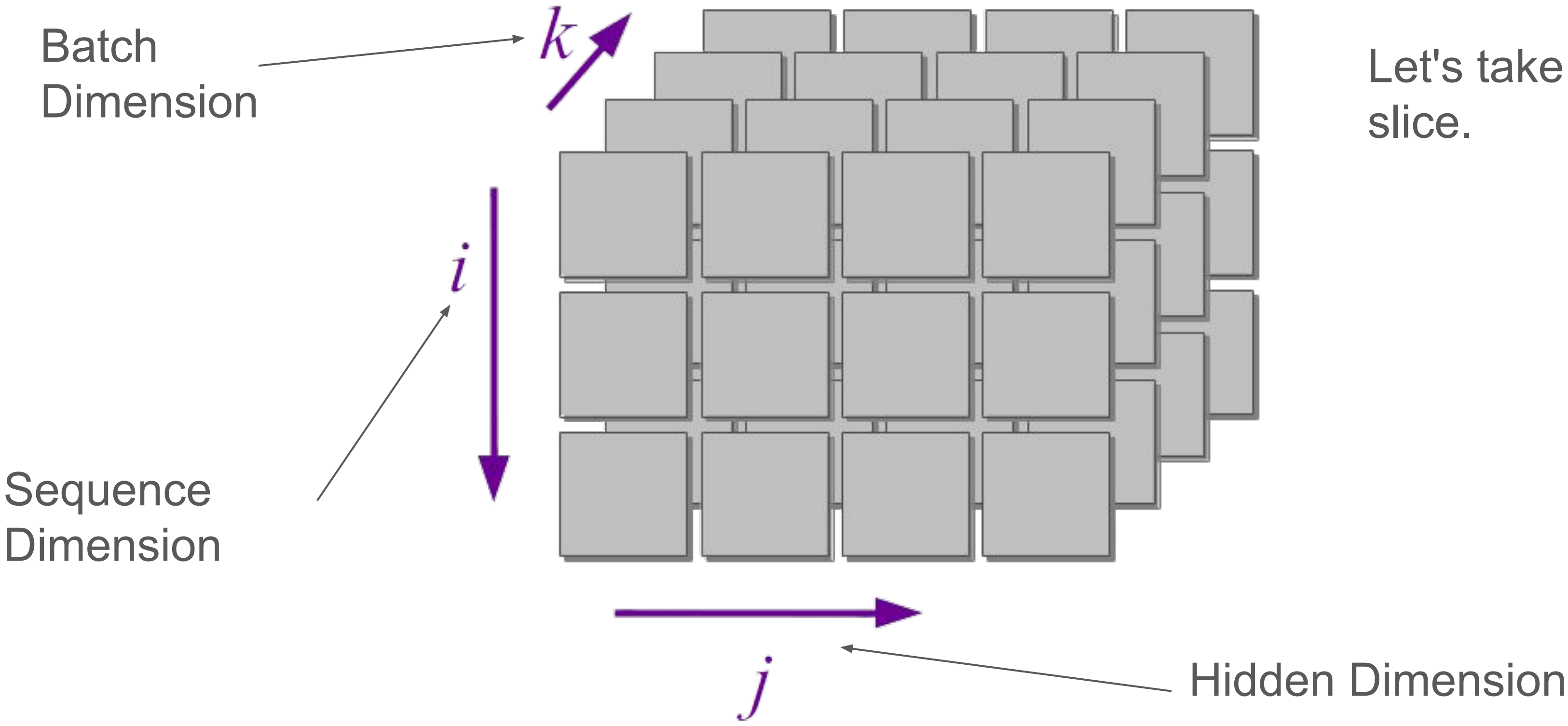


Input Visualisation



What does a dropout look like on this matrix?

Input Visualisation



Let's take one (i,j) slice.

Intuition - Where is the Parallelism? (Dropout)

From Previous Layer

0.1	0.8	0.9	0.5	-0.2
0.7	0.3	-0.4	0.2	0.1
0.8	-0.1	0.6	0.1	0.6
-0.5	-0.4	0.1	0.3	0.3
-0.6	0.2	-0.6	-0.1	0.2

0	0	1	1	1
1	0	0	1	1
1	0	1	0	1
1	1	0	1	1
1	1	0	0	0

Dropout Mask (D)

To Next Layer

0	0	0.9	0.5	-0.2
0.7	0	0	0.2	0.1
0.8	0	0.6	0	0.6
-0.5	-0.4	0	0.3	0.3
-0.6	0.2	0	0	0

Credits:
https://epynn.net/Dropout.html

Intuition - Where is the Parallelism? (Dropout)

From Previous Layer

0.1	0.8	0.9	0.5	-0.2
0.7	0.3	-0.4	0.2	0.1
0.8	-0.1	0.6	0.1	0.6
-0.5	-0.4	0.1	0.3	0.3
-0.6	0.2	-0.6	-0.1	0.2

Dropout Mask (D)

0	0	1	1	1
1	0	0	1	1
1	0	1	0	1
1	1	0	1	1
1	1	0	0	0

To Next Layer

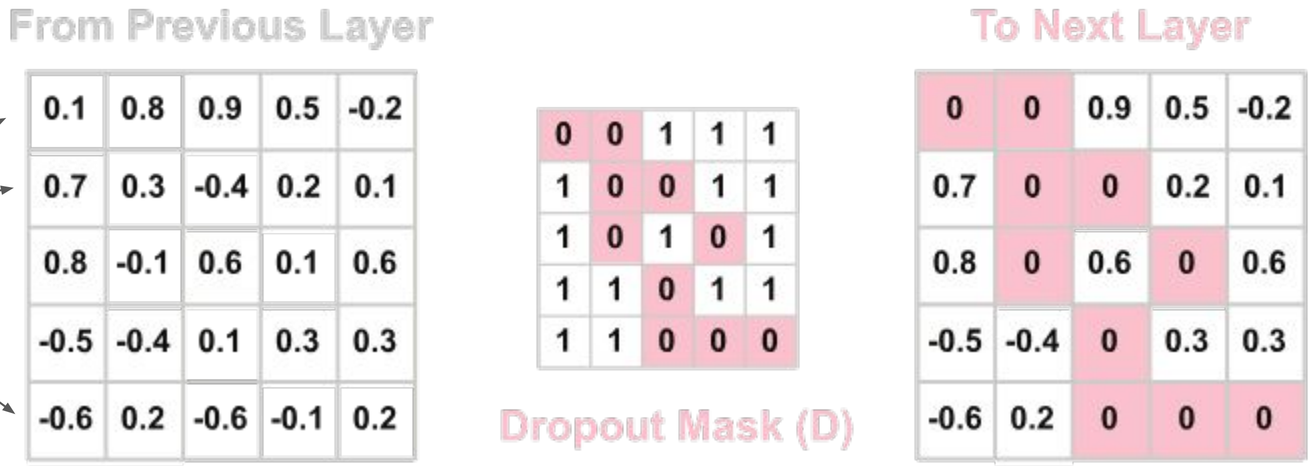
0	0	0.9	0.5	-0.2
0.7	0	0	0.2	0.1
0.8	0	0.6	0	0.6
-0.5	-0.4	0	0.3	0.3
-0.6	0.2	0	0	0

Takes a matrix, and masks out inputs with a particular Probability

Credits: https://epynn.net/Dropout.html

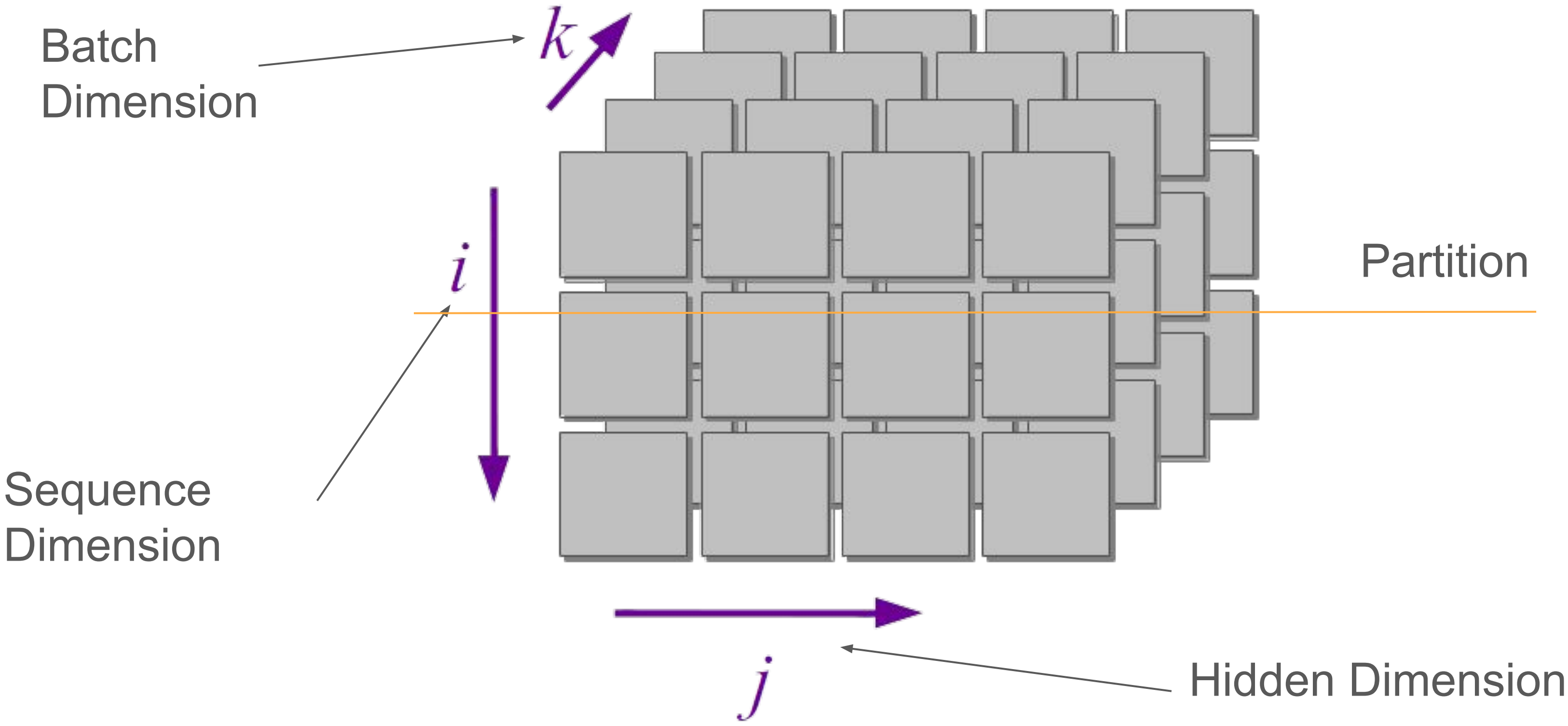
Intuition - Where is the Parallelism? (Dropout)

We can apply this *independently* to each row of the matrix

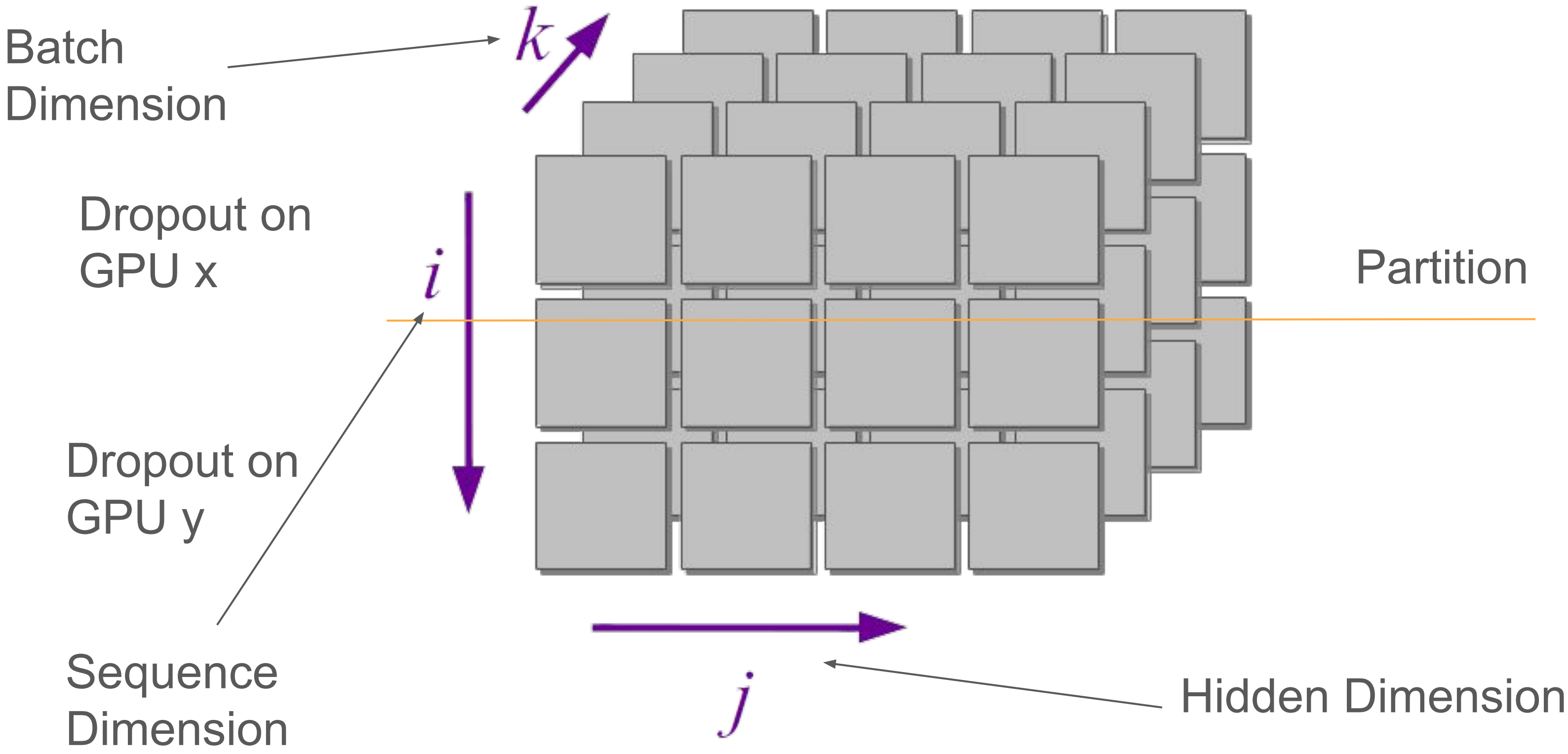


Credits:
https://epynn.net/Dropout.html

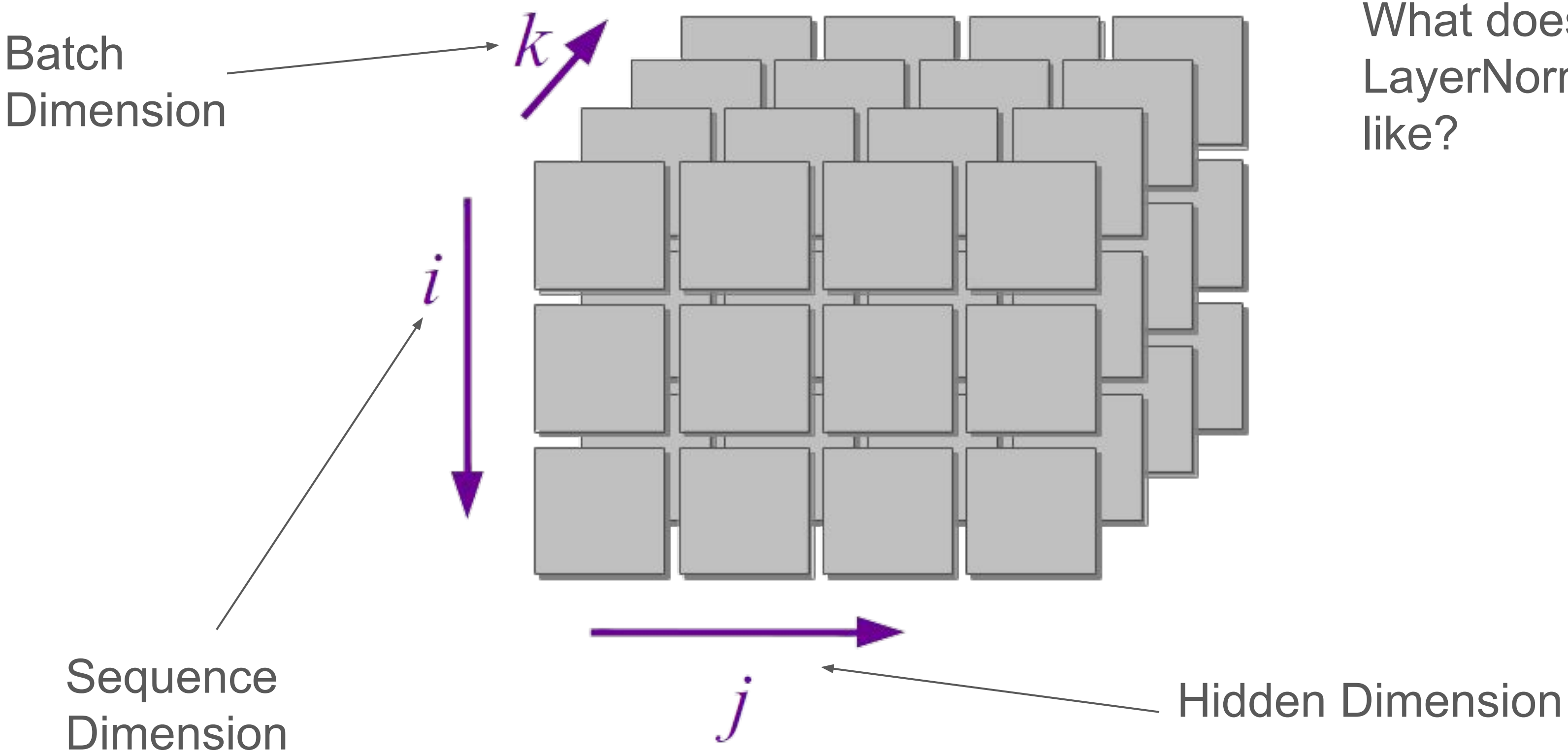
Input Visualisation



Input Visualisation



Input Visualisation

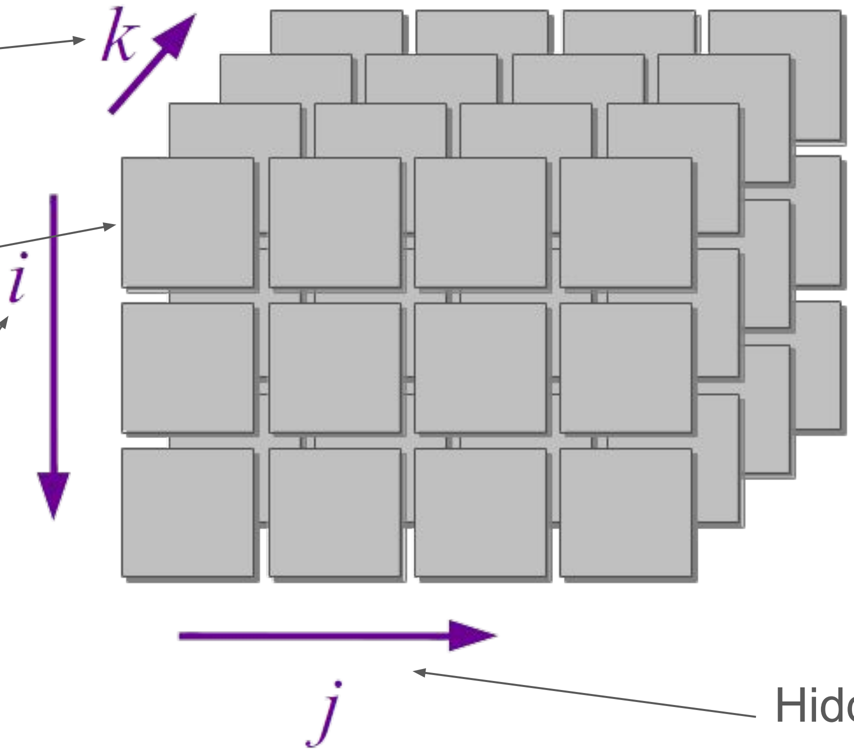


What does LayerNorm look like?

Input Visualisation

Batch Dimension

We take one sequence vector



Sequence Dimension

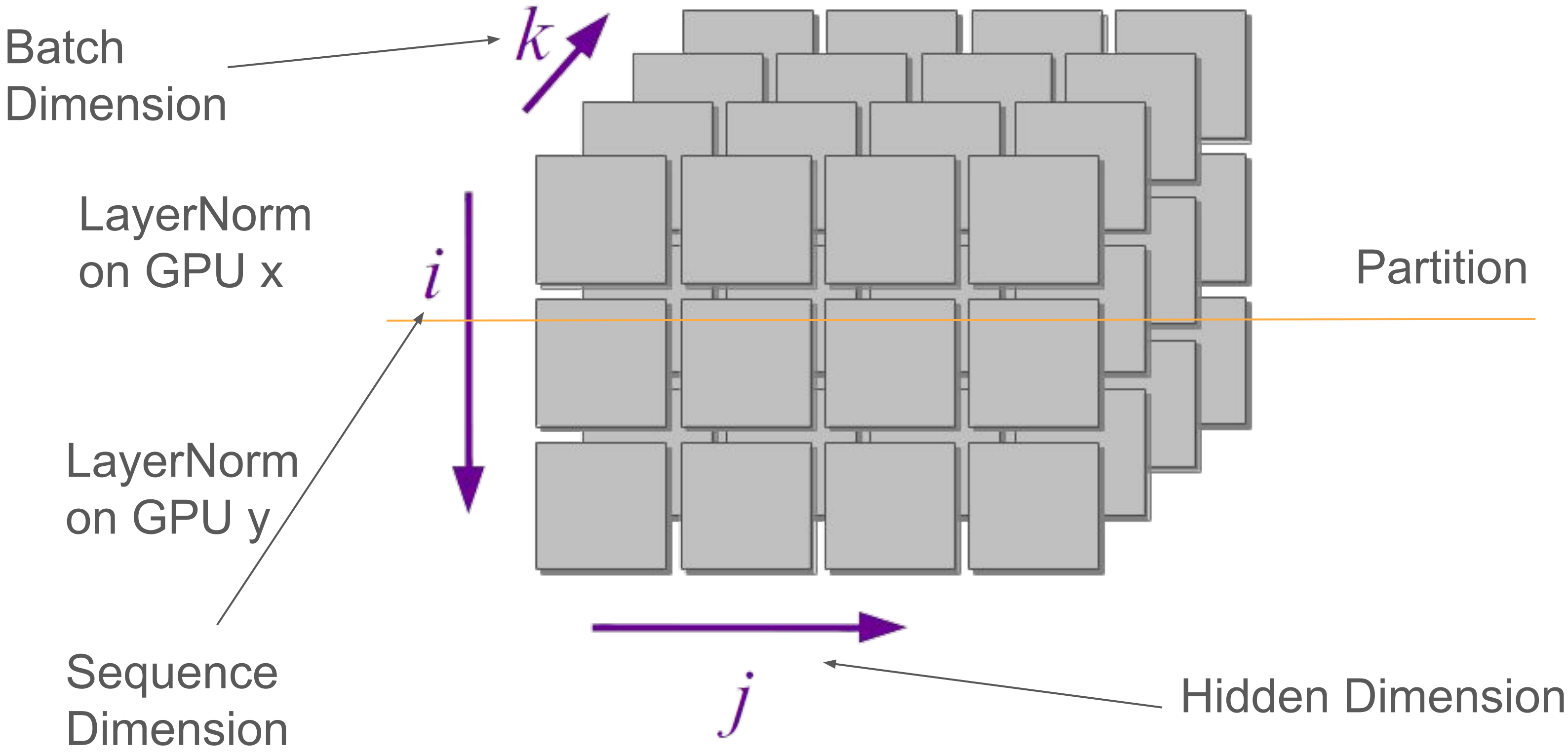
Hidden Dimension

What does LayerNorm look like?

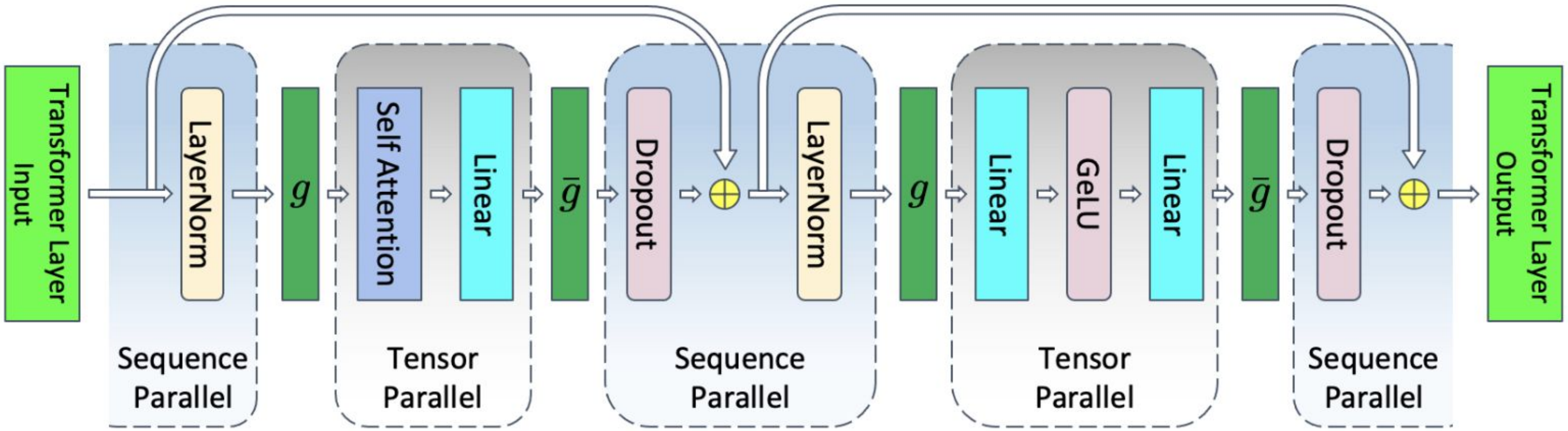
Input Visualisation

$$y = \begin{bmatrix} y_1 \\ \\ y_2 \\ \\ y_3 \end{bmatrix} = normalize(x) = f(x) = \begin{bmatrix} \frac{x_1 - \mu}{\sigma} \\ \\ \frac{x_2 - \mu}{\sigma} \\ \\ \frac{x_3 - \mu}{\sigma} \end{bmatrix}$$

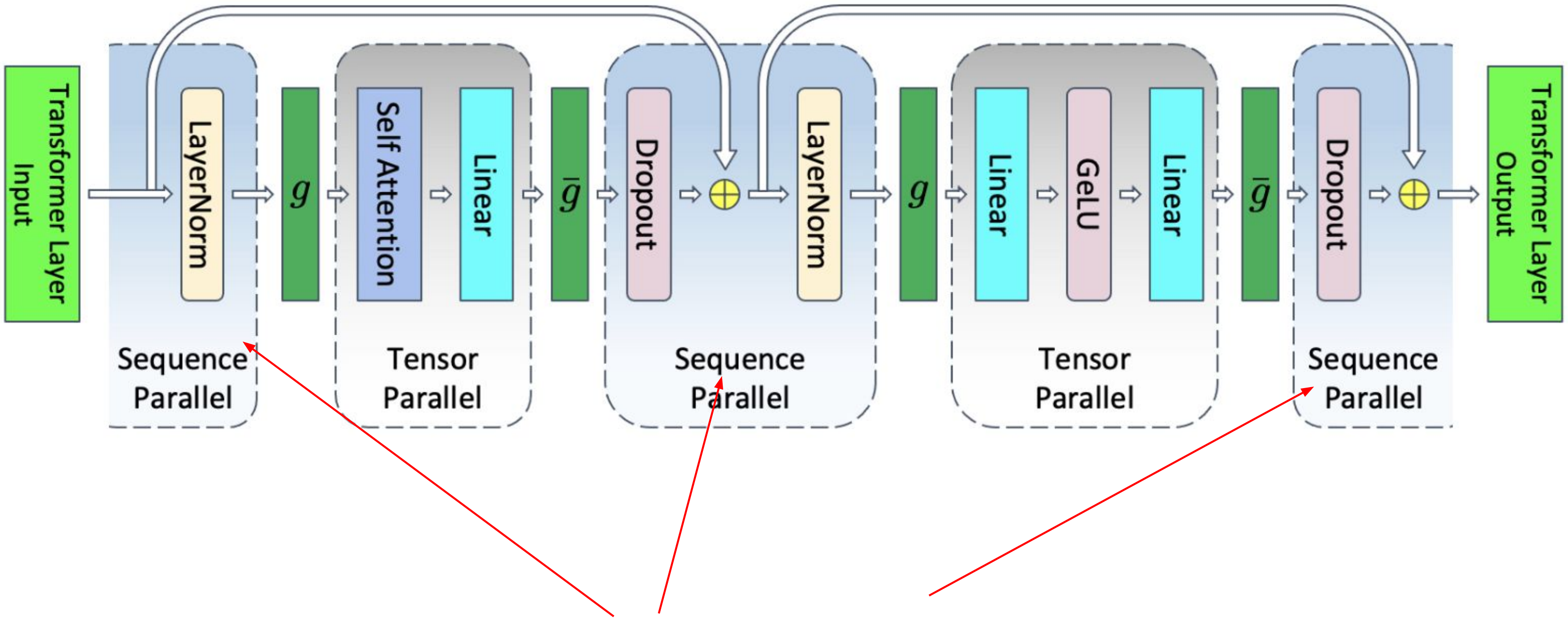
Input Visualisation



Full Flow - Sequence & Tensor Parallelism

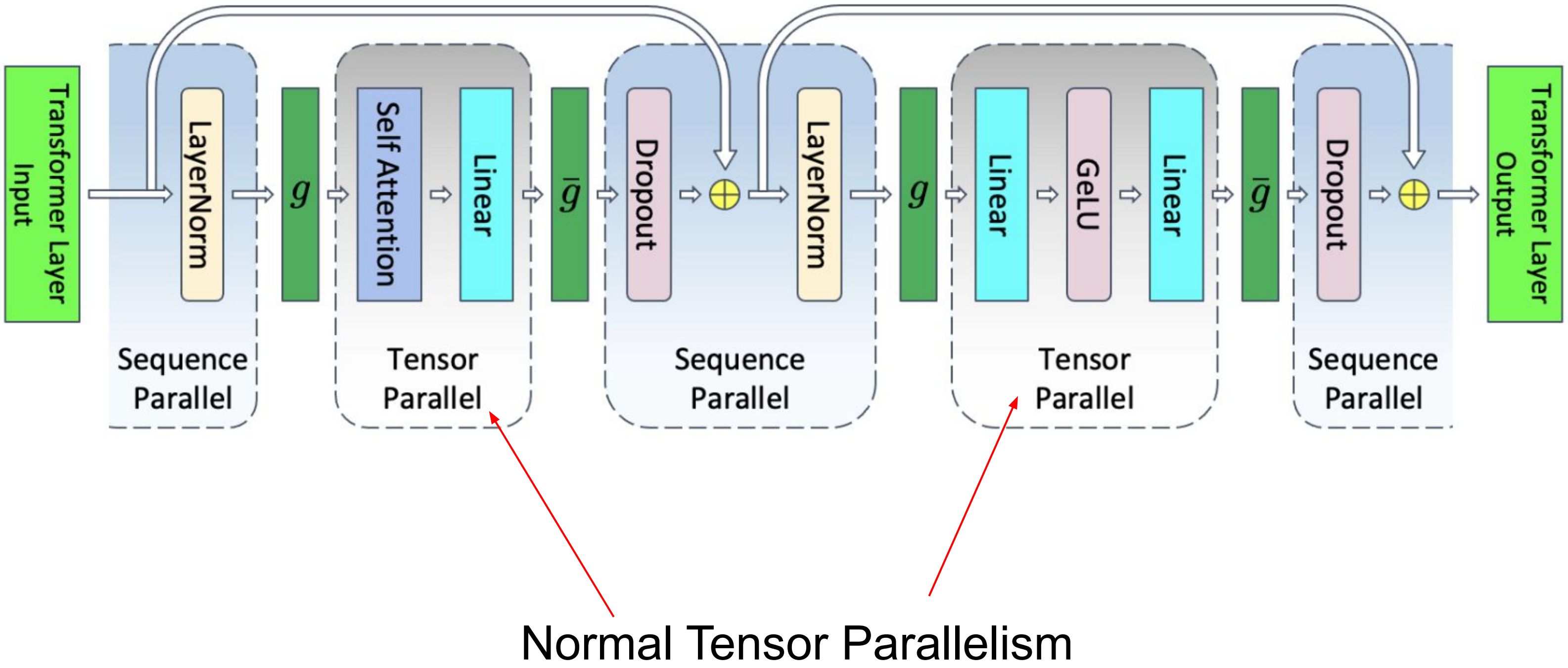


Full Flow - Sequence & Tensor Parallelism

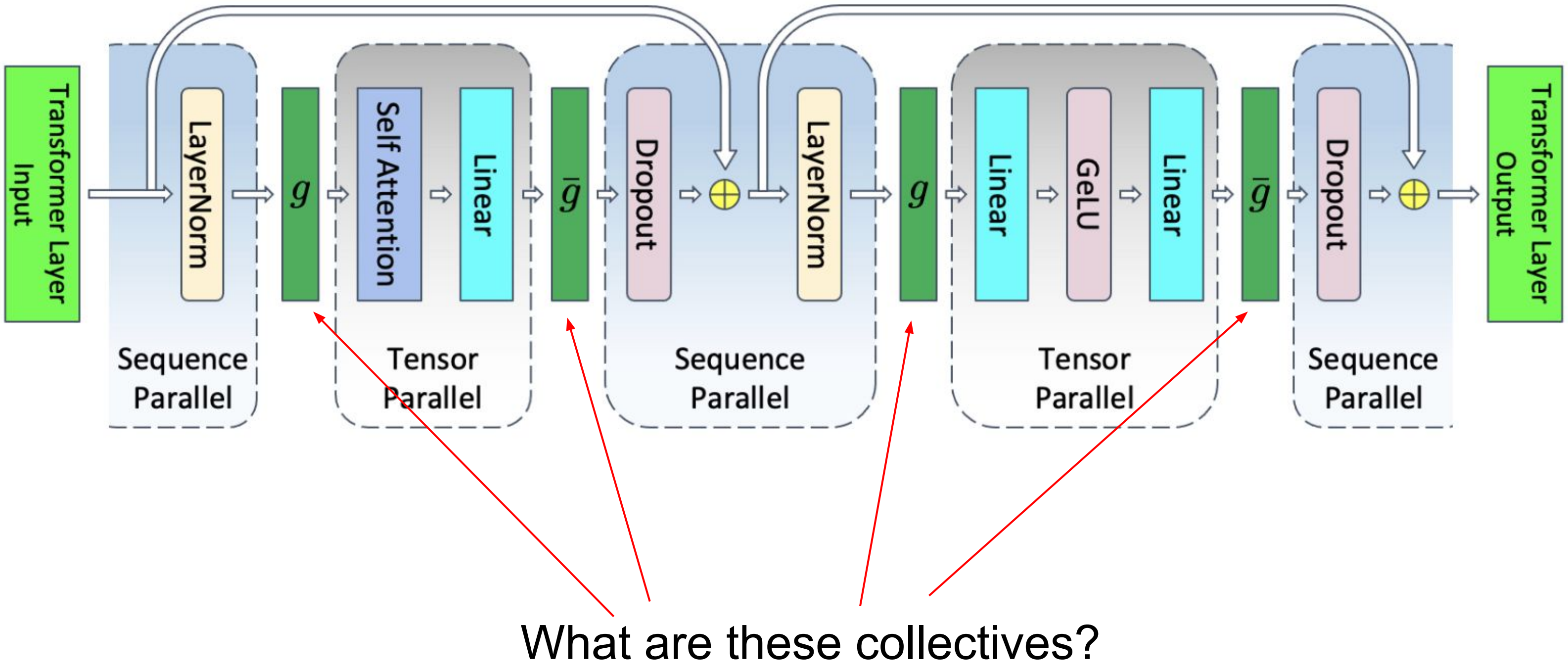


Parallelise across sequence dimension

Full Flow - Sequence & Tensor Parallelism



Full Flow - Sequence & Tensor Parallelism



Full Flow - Sequence & Tensor Parallelism

Let's walk through how to do this on 2 GPUs

$$O^1 = LayerNorm(X)$$
$$O^2 = O^1 A^1$$
$$O^3 = Gelu(O^2)$$
$$O^4 = O^3 A^2$$
$$O^5 = Dropout(O^4)$$

Full Flow - Sequence & Tensor Parallelism

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

$$O^1 = LayerNorm(X)$$
$$O^2 = O^1 A^1$$
$$O^3 = Gelu(O^2)$$
$$O^4 = O^3 A^2$$
$$O^5 = Dropout(O^4)$$

Step 1: Replicate data across sequence dimension. Compute LayerNorm

GPU 0	GPU 1
Computes: $LayerNorm(X_1)$	Computes: $LayerNorm(X_2)$

Full Flow - Sequence & Tensor Parallelism

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

$$O^1 = LayerNorm(X)$$
$$O^2 = O^1 A^1$$
$$O^3 = Gelu(O^2)$$
$$O^4 = O^3 A^2$$
$$O^5 = Dropout(O^4)$$

Step 2: All-gather, and apply tensor Parallelism

GPU 0

Computes:
$$LayerNorm(X_1)$$

GPU 1

Computes:
$$LayerNorm(X_2)$$

Full Flow - Sequence & Tensor Parallelism

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

$$O^1 = LayerNorm(X)$$
$$O^2 = O^1 A^1$$
$$O^3 = Gelu(O^2)$$
$$O^4 = O^3 A^2$$
$$O^5 = Dropout(O^4)$$

$$[A_1^1 | A_2^1]$$

$$\begin{bmatrix} A_1^2 \\ A_2^2 \end{bmatrix}$$

Step 2: State After Tensor Parallelism has been applied.

GPU 0

Computes:

$$Gelu(O^1 A_1^1) A_1^2$$

GPU 1

Computes:

$$Gelu(O^1 A_2^1) A_2^2$$

Now we've finished Tensor Parallelism (Step Prior to All-Gather)

Full Flow - Sequence & Tensor Parallelism

$$O^1 = LayerNorm(X) \leftarrow \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

$$O^2 = O^1 A^1 \leftarrow \left[A_1^1 | A_2^1\right]$$

$$O^3 = Gelu(O^2)$$

$$O^4 = O^3 A^2 \leftarrow \begin{bmatrix} A_1^2 \\ A_2^2 \end{bmatrix}$$

$$O^5 = Dropout(O^4)$$

Step 2: State After Tensor Parallelism has been applied.

GPU 0

Computes:

$$Gelu(O^1 A_1^1) A_1^2$$

GPU 1

Computes:

$$Gelu(O^1 A_2^1) A_2^2$$

What we need to do is:

1. Add results
2. Split across rows

Full Flow - Sequence & Tensor Parallelism

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

$$O^1 = LayerNorm(X)$$

$$O^2 = O^1 A^1 \quad \leftarrow \quad [A_1^1 | A_2^1]$$

$$O^3 = Gelu(O^2)$$

$$O^4 = O^3 A^2 \quad \leftarrow \quad \begin{bmatrix} A_1^2 \\ A_2^2 \end{bmatrix}$$

$$O^5 = Dropout(O^4)$$

Step 2: State After Tensor Parallelism has been applied.

GPU 0

Computes:

$$Gelu(O^1 A_1^1) A_1^2$$

GPU 1

Computes:

$$Gelu(O^1 A_2^1) A_2^2$$

We'll do a reduce scatter!

Full Flow - Sequence & Tensor Parallelism

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

$$O^1 = LayerNorm(X)$$
$$O^2 = O^1 A^1$$
$$O^3 = Gelu(O^2)$$
$$O^4 = O^3 A^2$$
$$O^5 = Dropout(O^4)$$

$$\left[A_1^1 | A_2^1\right]$$

$$\begin{bmatrix} A_1^2 \\ A_2^2 \end{bmatrix}$$

$$\begin{bmatrix} O_1^4 \\ O_2^4 \end{bmatrix}$$

Step 3: Reduce-Scatter

GPU 0

Has:

O_1^4

GPU 1

Has:

O_2^4

Full Flow - Sequence & Tensor Parallelism

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

$$O^1 = LayerNorm(X)$$
$$O^2 = O^1 A^1$$
$$O^3 = Gelu(O^2)$$
$$O^4 = O^3 A^2$$
$$O^5 = Dropout(O^4)$$

$$\left[A_1^1 | A_2^1\right]$$

$$\begin{bmatrix} A_1^2 \\ A_2^2 \end{bmatrix}$$

$$\begin{bmatrix} O_1^4 \\ O_2^4 \end{bmatrix}$$

Step 4: Apply Dropout

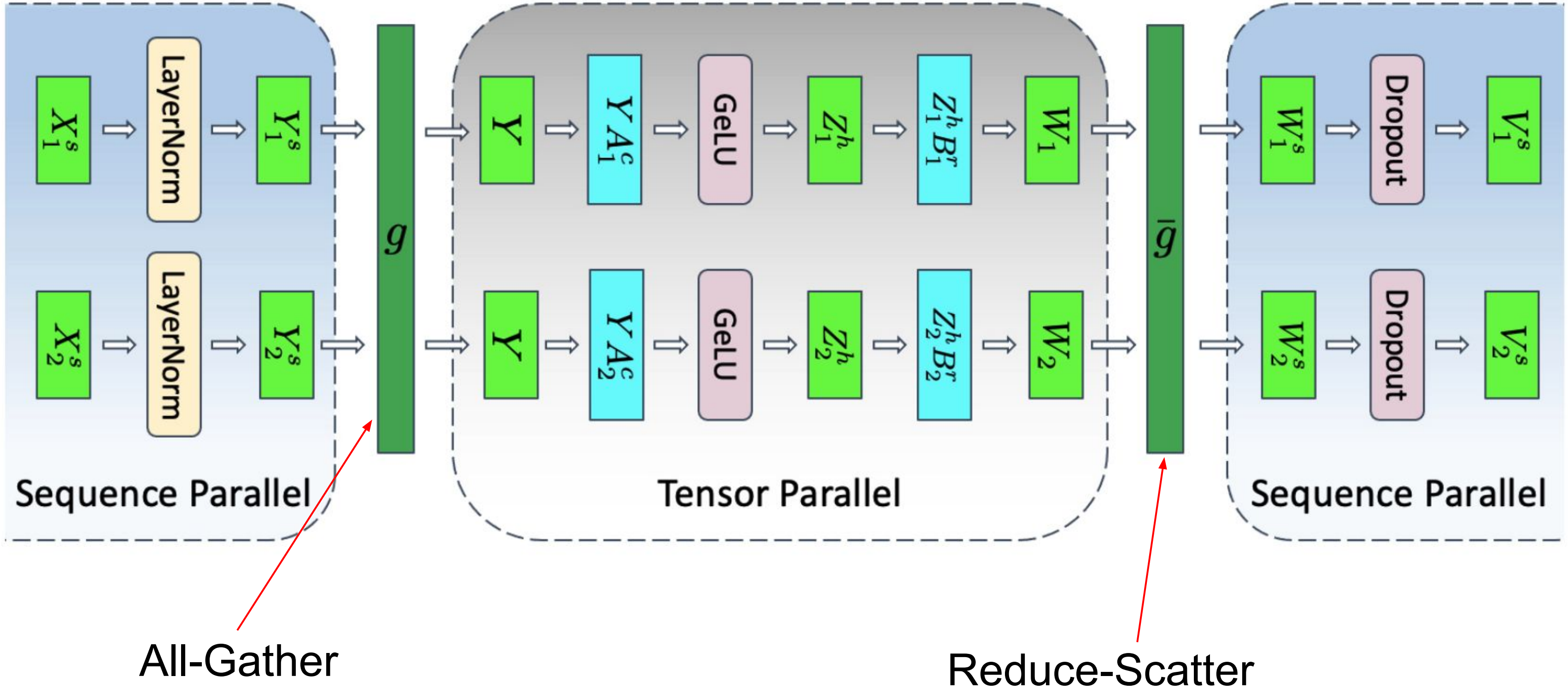
GPU 0

Computes:
$$Dropout(O_1^4)$$

GPU 1

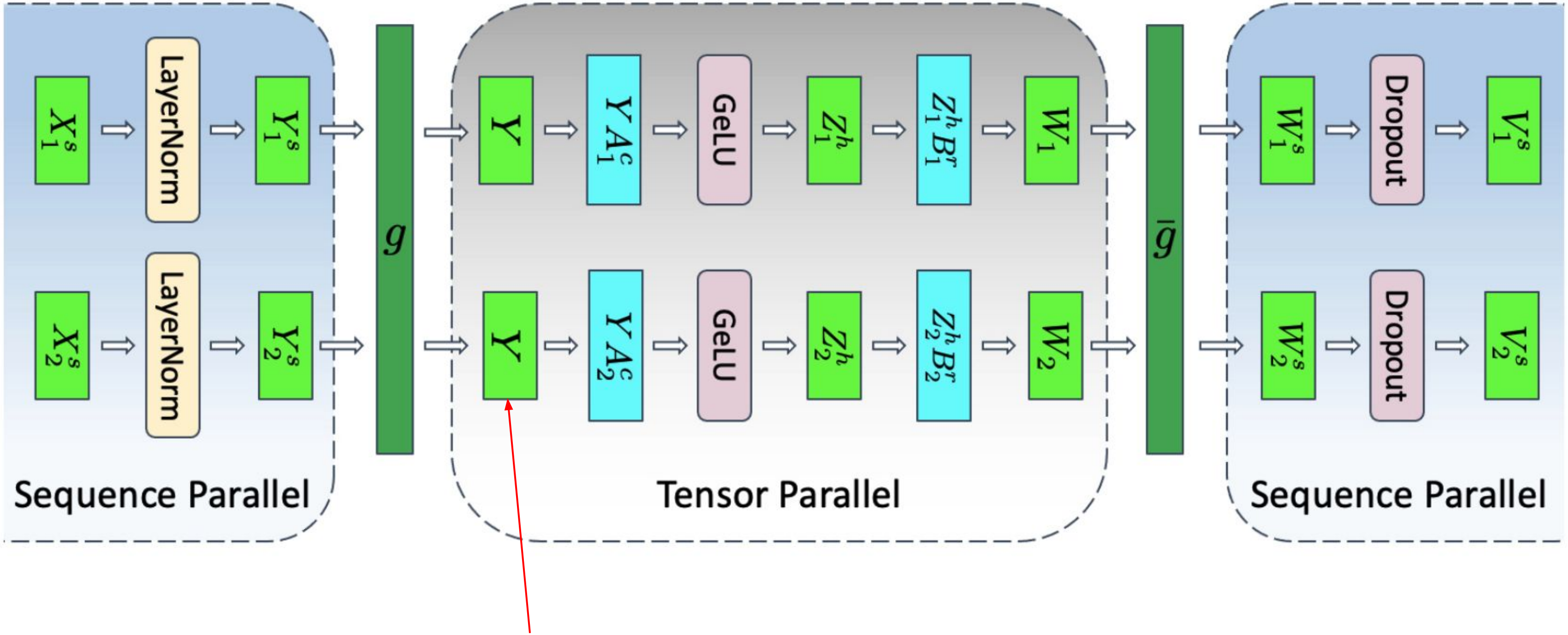
Computes:
$$Dropout(O_2^4)$$

Full Flow



X_1^s
LayerNorm
Y_1^s
X_2^s
LayerNorm
Y_2^s
Sequence Parallel
g
Y
YA_1^c
GeLU
Z_1^h
$Z_1^h B_1^r$
W_1
Y
YA_2^c
GeLU
Z_2^h
$Z_2^h B_2^r$
W_2
Tensor Parallel
$\bar{g}$
W_1^s
Dropout
V_1^s
W_2^s
Dropout
V_2^s
Sequence Parallel
All-Gather
Reduce-Scatter

Full Flow

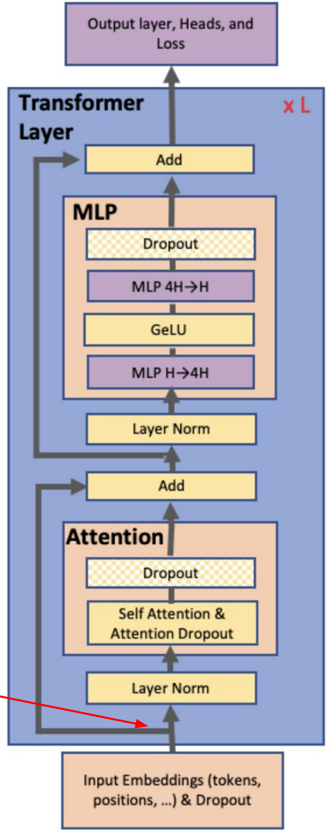


We materialise all the activations here.

Activation Checkpointing

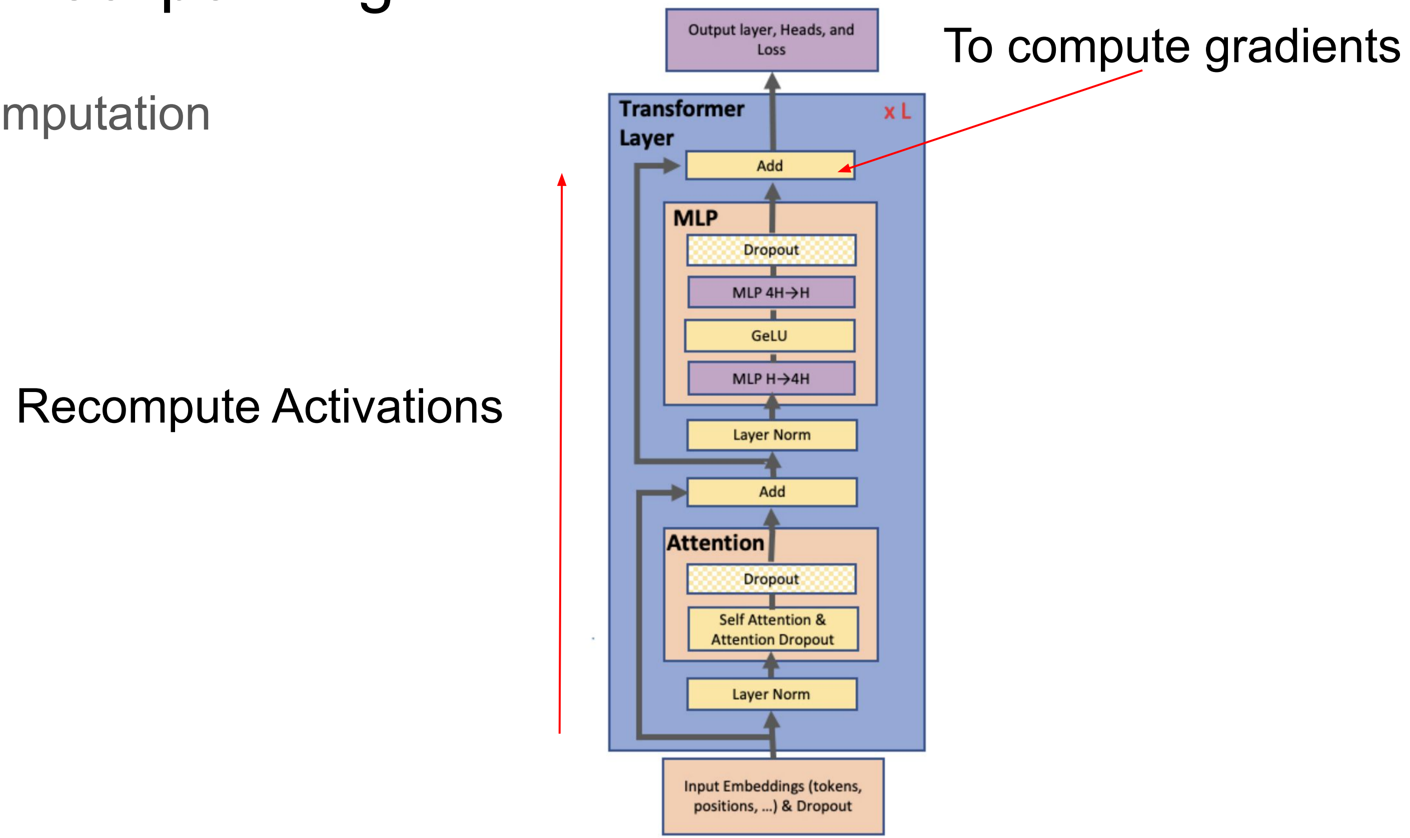
Naive Full-Recomputation

Store these checkpoints



Activation Checkpointing

Naive Full-Recomputation



Selective Recomputation

Activation Checkpointing is effective in reducing memory consumption

Which layers' activations to *not* checkpoint?

Selective Recomputation

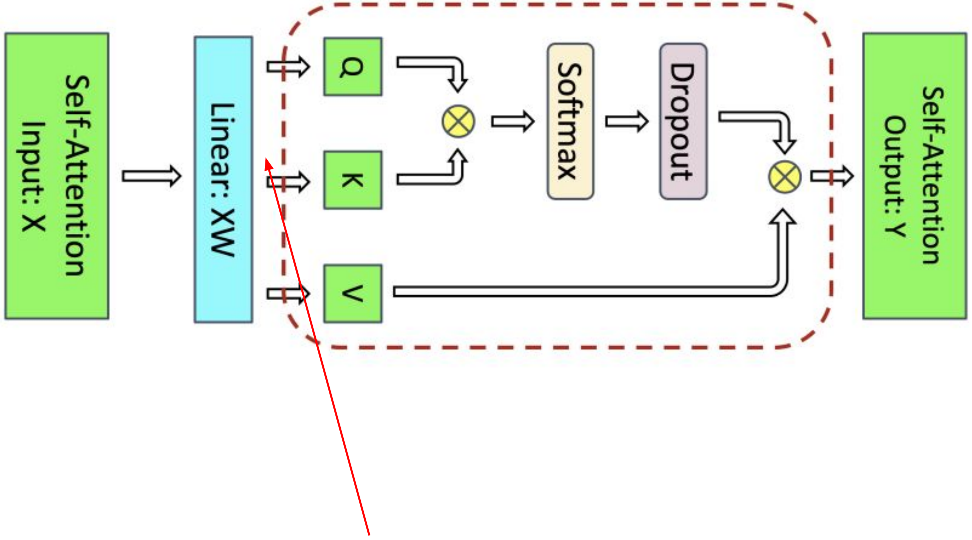
Activation Checkpointing is effective in reducing memory consumption

Which layers' activations to *not* checkpoint?

Layers with low FLOPs, but high number of activations (softmax, dropout).

Selective Recomputation

Checkpoint activations
post linear transformation



Checkpoint activations

Evaluation

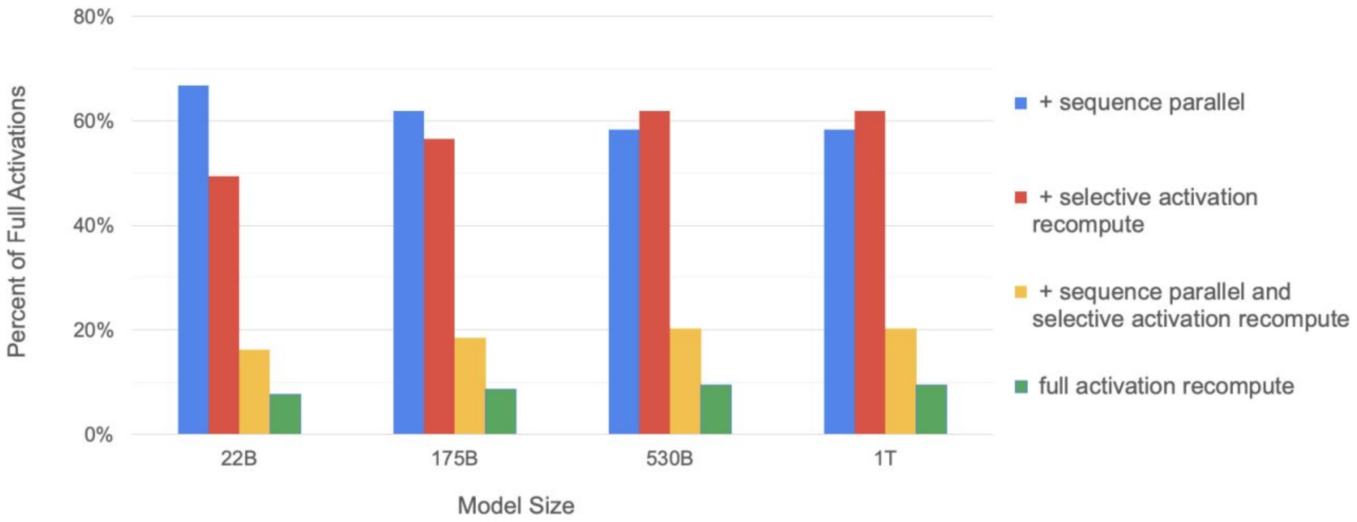


Figure 7: Percentage of required memory compared to the tensor-level parallel baseline. As the model size increases, both sequence parallelism and selective activation recomputation have similar memory savings and together they reduce the memory required by $\sim 5\times$.

Evaluation

Experiment	Forward (ms)	Backward (ms)	Combined (ms)	Overhead (%)
Baseline no recompute	7.7	11.9	19.6	–
Sequence Parallelism	7.2	11.8	19.0	−3%
Baseline with recompute	7.7	19.5	27.2	39%
Selective Recompute	7.7	13.2	20.9	7%
Selective + Sequence	7.2	13.1	20.3	4%

Table 4: Time to complete the forward and backward pass of a single transformer layer of the 22B model.

Evaluation

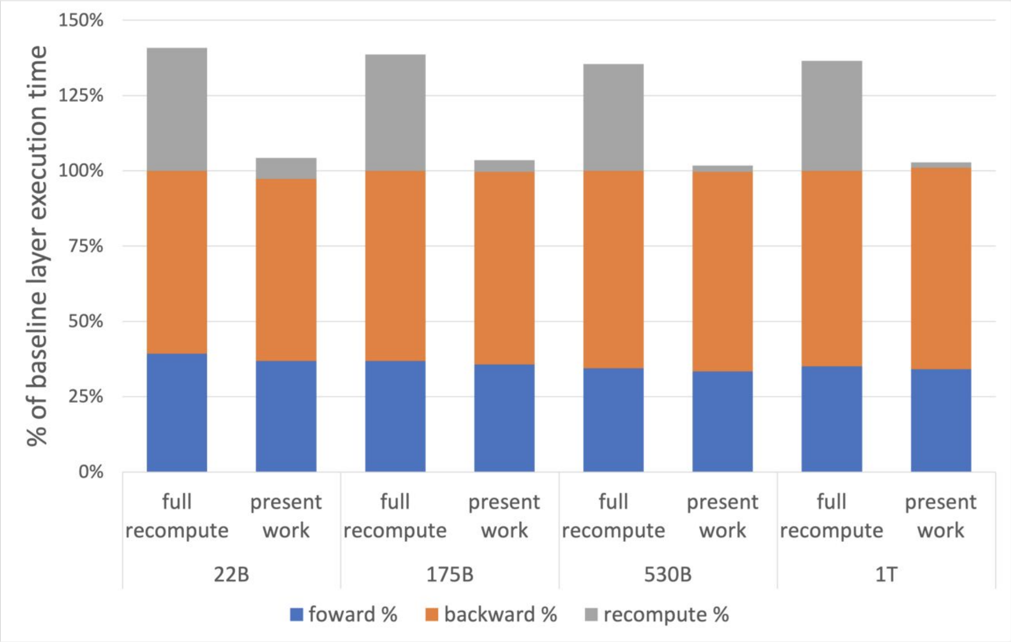


Figure 8: Per layer breakdown of forward, backward, and recompute times. Baseline is the case with no recomputation and no sequence parallelism. Present work includes both sequence parallelism and selective activation recomputation.

Opinion

Doesn't seem to accelerate inference

Main speedup is for training.

Discussion

HPC Community has been working on distributed Matmul for a while. Can some of their methods be adapted?

Is there a way to systematically explore the space of communication operations + partitioning strategies?

Can we leverage offload strategies as learnt earlier in the class?