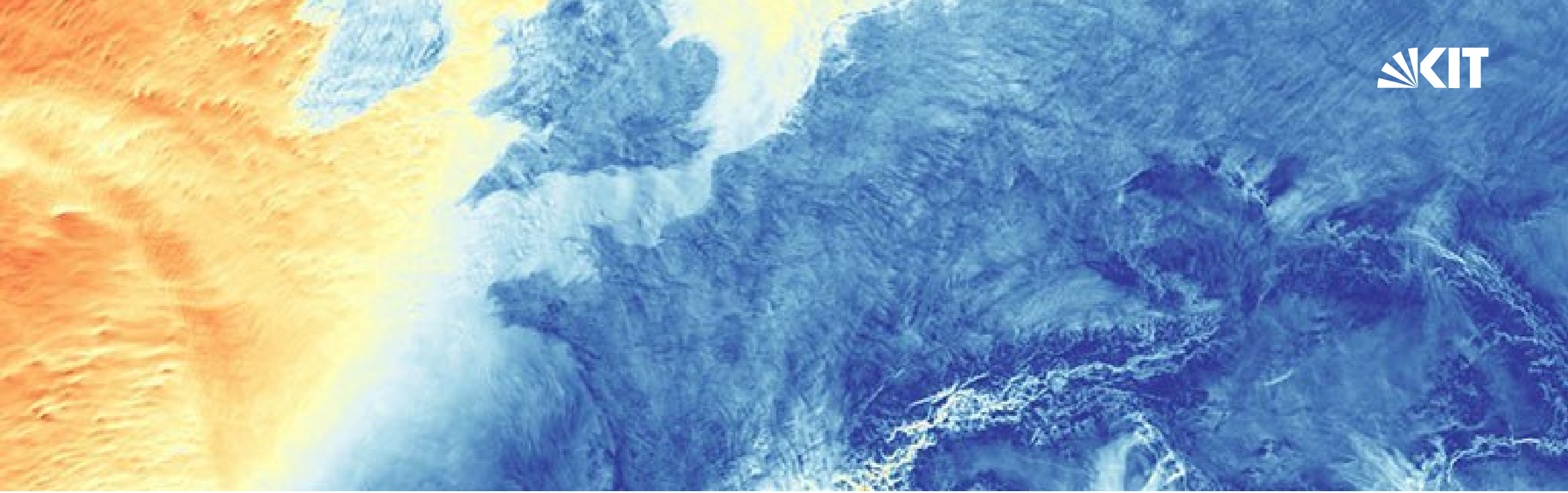


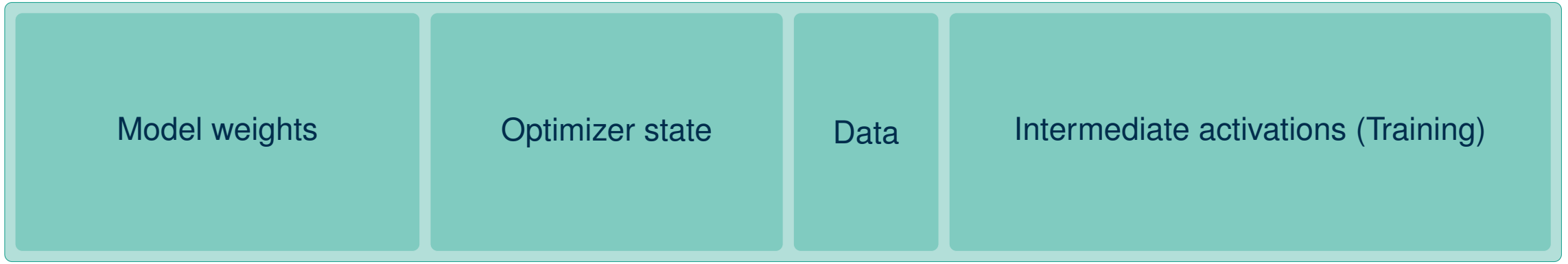
jigsaw: Domain and tensor parallelism for high-resolution input training

Deifilia Kieckhefen | 07.04.2026

Complex tasks demand large models

- State-of-the-art models for, i.e., LLMs, have billions to trillions of parameters
- Modern hardware accelerators have limited memory $\approx$ 80 GB per GPU

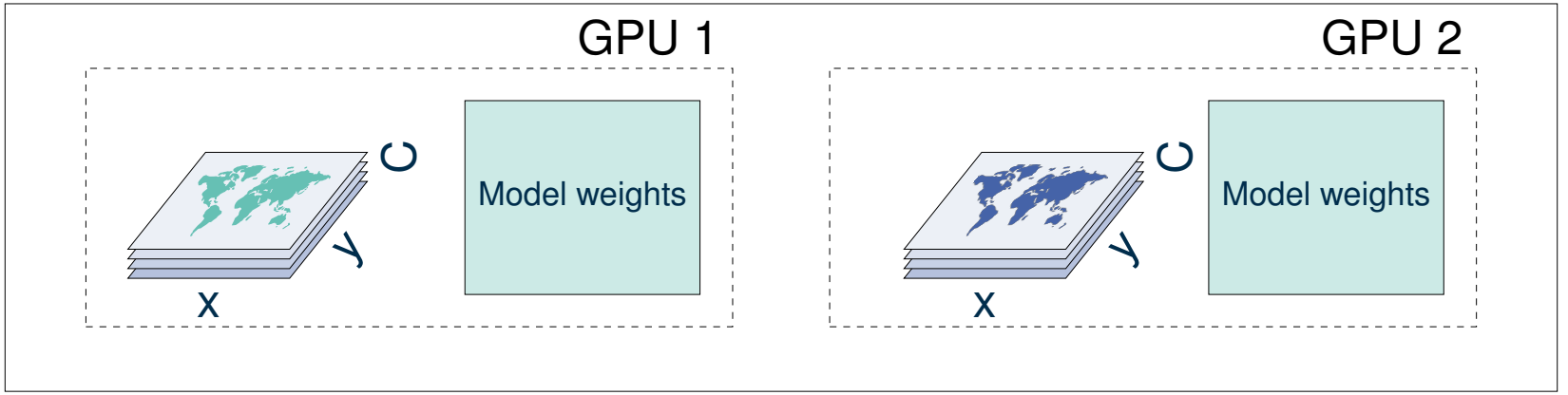
GPU memory in training



- Approaches such as FSDP, ZeRO, or pipeline parallelism to train larger models on more GPUs

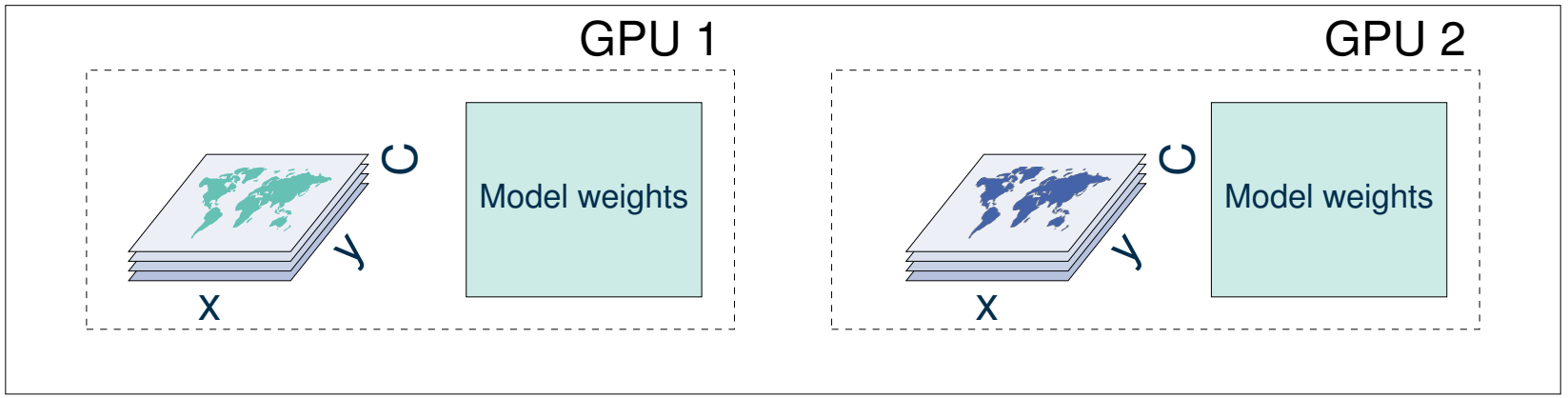
Parallelism to circumvent memory limitations

Data parallel

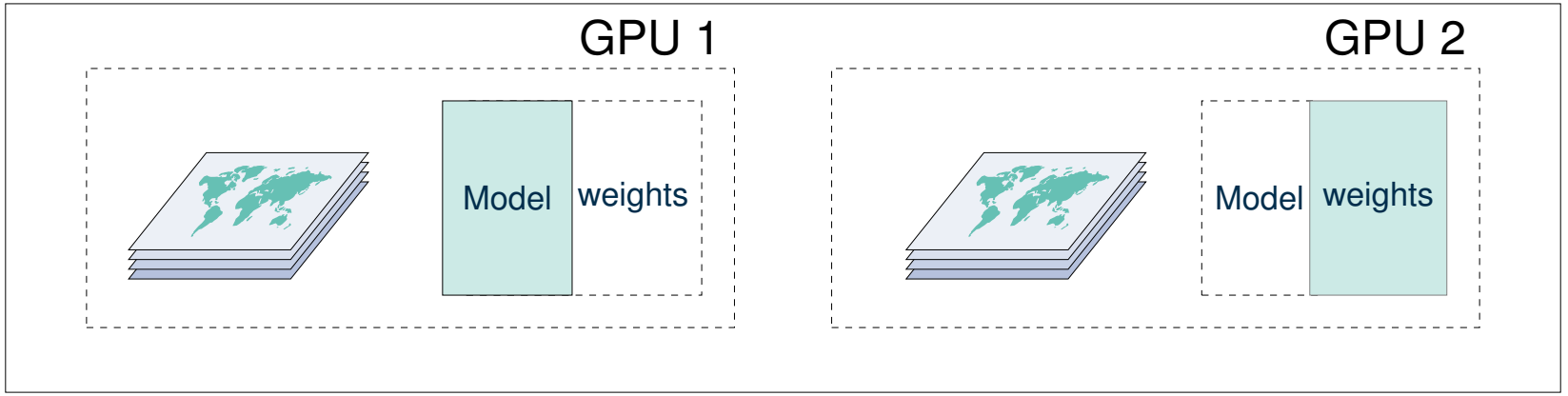


Parallelism to circumvent memory limitations

Data parallel



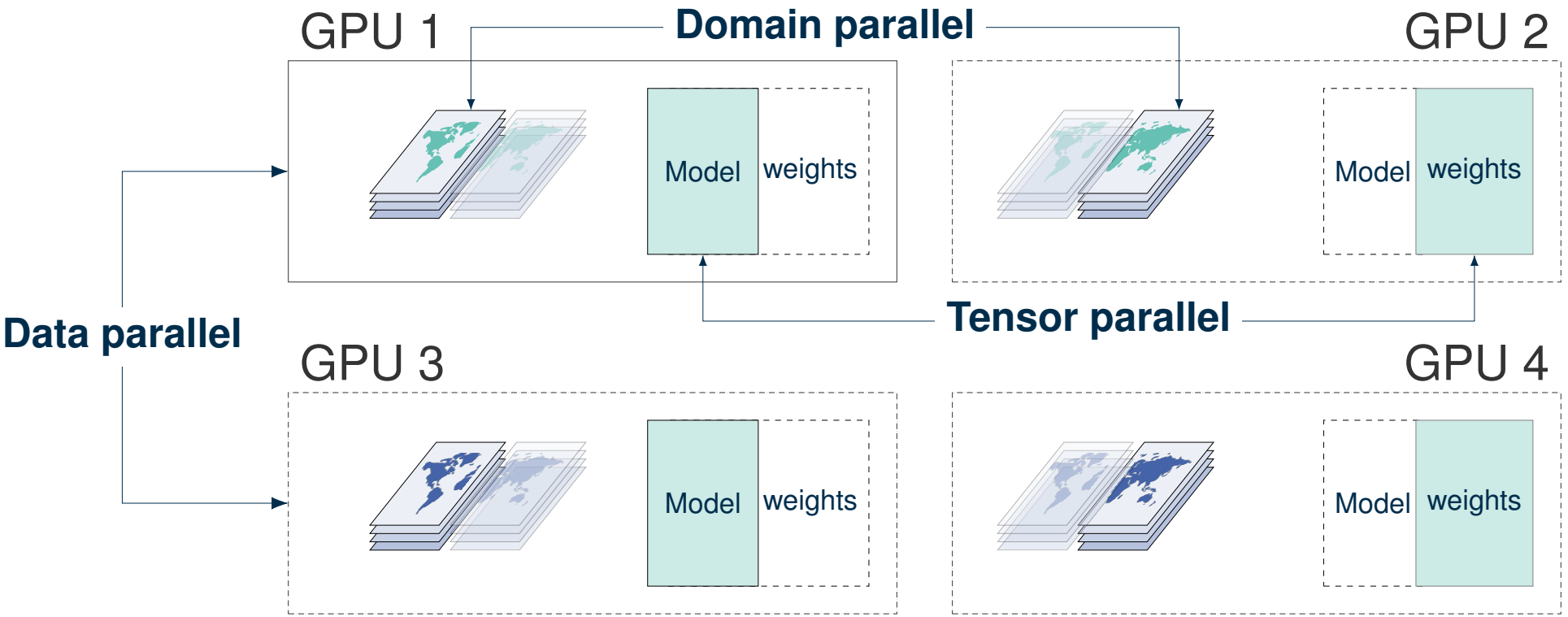
Tensor parallel



What if the data itself is large?

- Image-based data can be orders of magnitude larger than text
- Applications in: image processing, scientific analysis, satellite image classification, weather forecasting

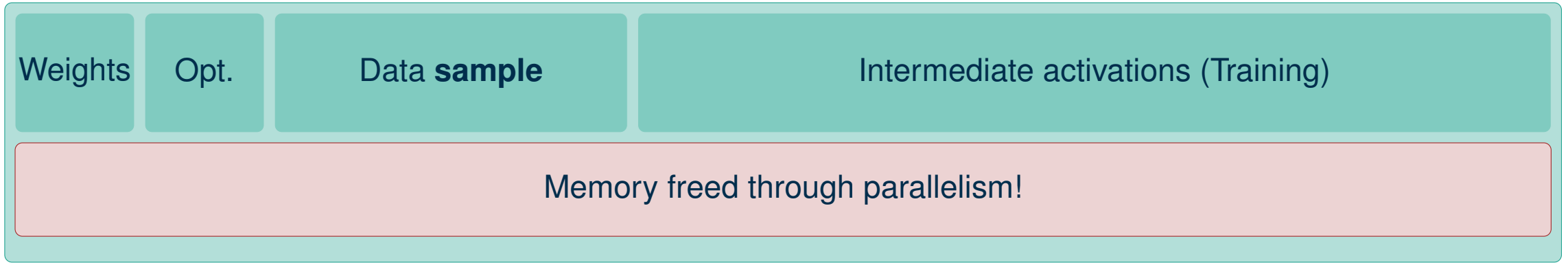
Parallelism to circumvent memory limitations



jigsaw framework

- A user-friendly PyTorch framework to speed up training with **domain and tensor parallelism**

GPU memory in training



The linear layer is the heart of the neural network

```
# Perform distributed matrix multiplication across n=2 processes
def AB_distributed(A, B, model_parallelism=n): ...
```

- Implement distributed matrix multiplication in kernels
- Implementation via PyTorch Distributed

$$
\begin{aligned}
XW &= \underbrace{\begin{bmatrix} X_1 & X_2 \end{bmatrix}}_{1\times 2} \underbrace{\begin{bmatrix} W_1 & W_2 \end{bmatrix}}_{1\times 2} = \underbrace{\begin{bmatrix} Y_1 & Y_2 \end{bmatrix}}_{1\times 2} \\
&= \underbrace{\begin{bmatrix} X_{1,A} & X_{2,A} \\ X_{1,B} & X_{2,B} \end{bmatrix}}_{2\times 2} \underbrace{\begin{bmatrix} W_{1,A} & W_{2,A} \\ W_{1,B} & W_{2,B} \end{bmatrix}}_{2\times 2} \\
&= \begin{bmatrix} X_{1,A}W_{1,A} + X_{2,A}W_{1,B} & X_{1,A}W_{2,A} + X_{2,A}W_{2,B} \\ \underbrace{X_{1,B}W_{1,A} + X_{2,B}W_{1,B}}_{\texttt{proc. 1}} & \underbrace{X_{1,B}W_{2,A} + X_{2,B}W_{2,B}}_{\texttt{proc. 2}} \end{bmatrix} \\
&= \begin{bmatrix} Y_1 & Y_2 \end{bmatrix}
\end{aligned}
$$

Building an MLP

```python
class DistributedMLP(torch.nn.Module):
    def __init__(self, in_dim, hidden_dim, out_dim):
        self.linear1 = DistributedLinear(in_dim, hidden_dim)
        self.linear2 = DistributedLinear(hidden_dim, out_dim)

    def forward(self, x):
        x = self.linear1(x)
        x = self.activation(x)
        x = self.drop(x)
        x = self.linear2(x)
        x = self.drop2(x)
        return x
```

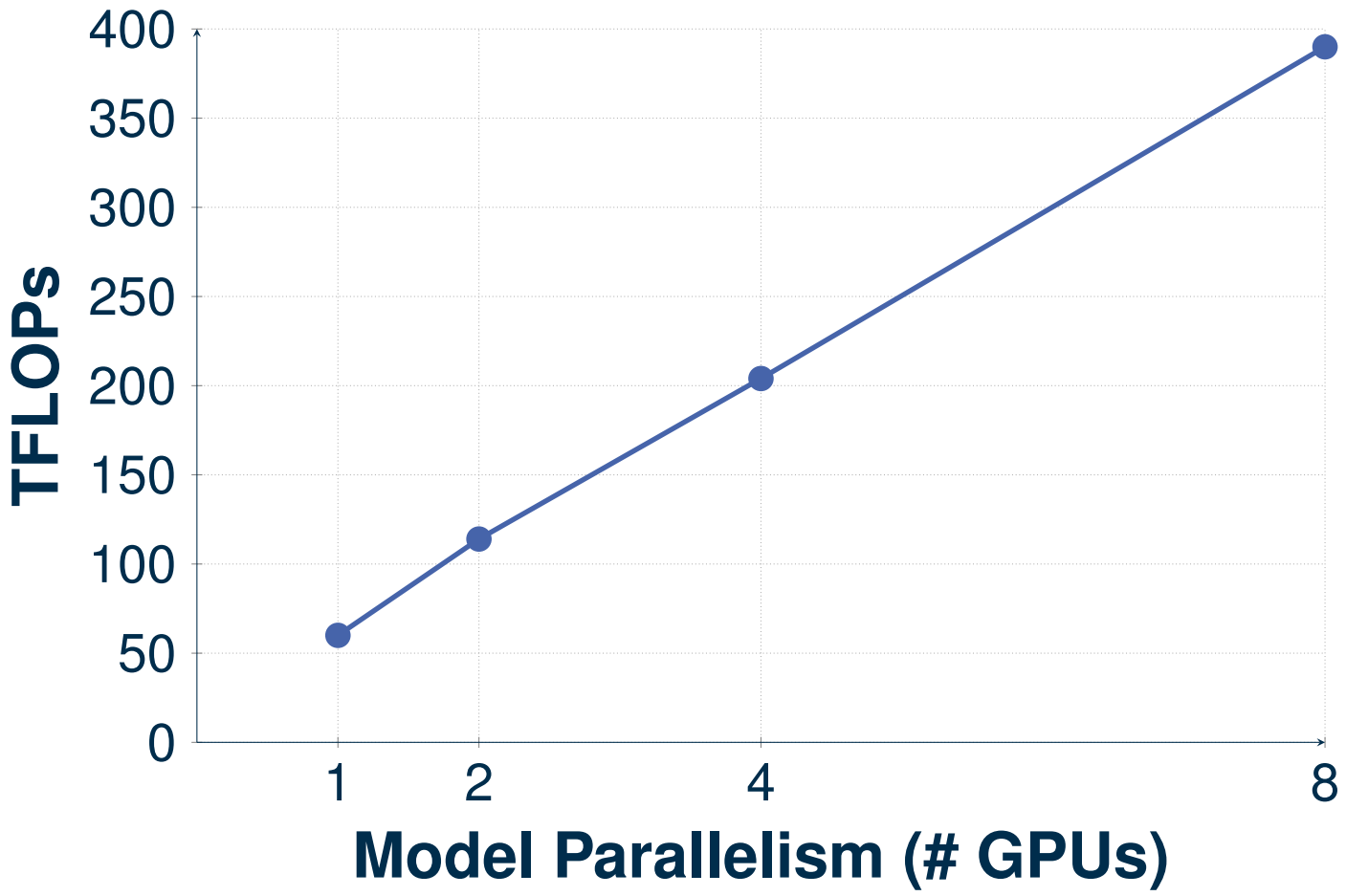
Transformers: Mat-muls are all you need

$$\text{attn} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

```
class DistributedAttention(torch.nn.Module):
    def __init__(self, x_features, embed_dim, output_dim):
        self.qkv = DistributedLinear(x_features, embed_dim * 3)
        self.softmax = DistributedSoftMax()

    def forward(self, x):
        q, k, v = self.qkv(x_patches_local)
        attention = distributed_x_wt(q, k)
        attention = self.softmax(attention)
        values = distributed_x_w(attention, v)
        return values
```

jigsaw allows for faster model training



- Input data: 400 MB per sample
- 1-billion parameters on 1 GPU $\rightarrow$ 4-billion parameters on 8 GPUs
- Distributed across up to 8 NVIDIA A100-40GB GPUs

jigsaw in summary

- Framework for distributed training and inference, while using PyTorch's
  - Auto-differentiation
  - CUDA backend for efficient computations
- Highly efficient memory usage
- Near-identical interface to PyTorch
- Seamless integration with DDP and other forms of parallelism
- Work-in-progress! Feedback welcome

Contact information:

deifilia.kieckhefen@kit.edu

DistributedLinear by overwriting autograd

```
class DistributedLinear(torch.nn.Module):
def __init__(self):
   self.local_weights = nn.Parameter(...)
   self.XWT = XWT().apply
def forward(self, x):
    x = self.XWT()
```

```
class XWT(torch.autograd.Function):
  @staticmethod
  def forward(x, w):
    xwt = xwt_distributed(x, w)
    return xwt

  @staticmethod
  def backward(ctx, grad_output)
    grad_input = xw_dist(grad_output, w)
    grad_weight = xtw_dist(grad_output, x)
    return grad_input, grad_weight
```