Paper: *Ring Attention with Blockwise Transformers for Near-Infinite Context*

BUAN 6382.SW1 - 24S

Group 6

Ring Attention with Blockwise Transformers for Near-Infinite Context

Authors:
Hao Liu,
Matei Zaharia,
Pieter Abbeel

UC Berkeley

October 2023

https://arxiv.org/pdf/2310.01889.pdf

Problem Statement

- Transformers demonstrate outstanding performance across numerous state-of-the-art AI models.
- However, their capacity to effectively process lengthy sequences of tokens is constrained.
- As the popularity of AI models and applications exponentially grow, so does the need for LLM architectures to have memory capabilities to process extremely large amounts of data efficiently.

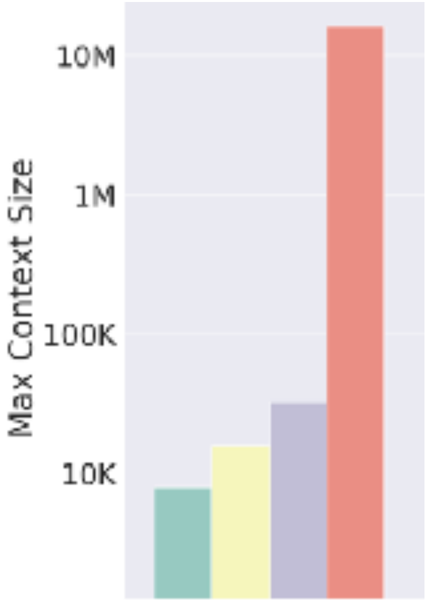
Why is this Important?

LLMs are being applied to larger and larger bodies of data

→ including books, high-resolution images, and long videos.

Reduce memory usage and increase speed of algorithms

→ vital to processing vast amounts of data

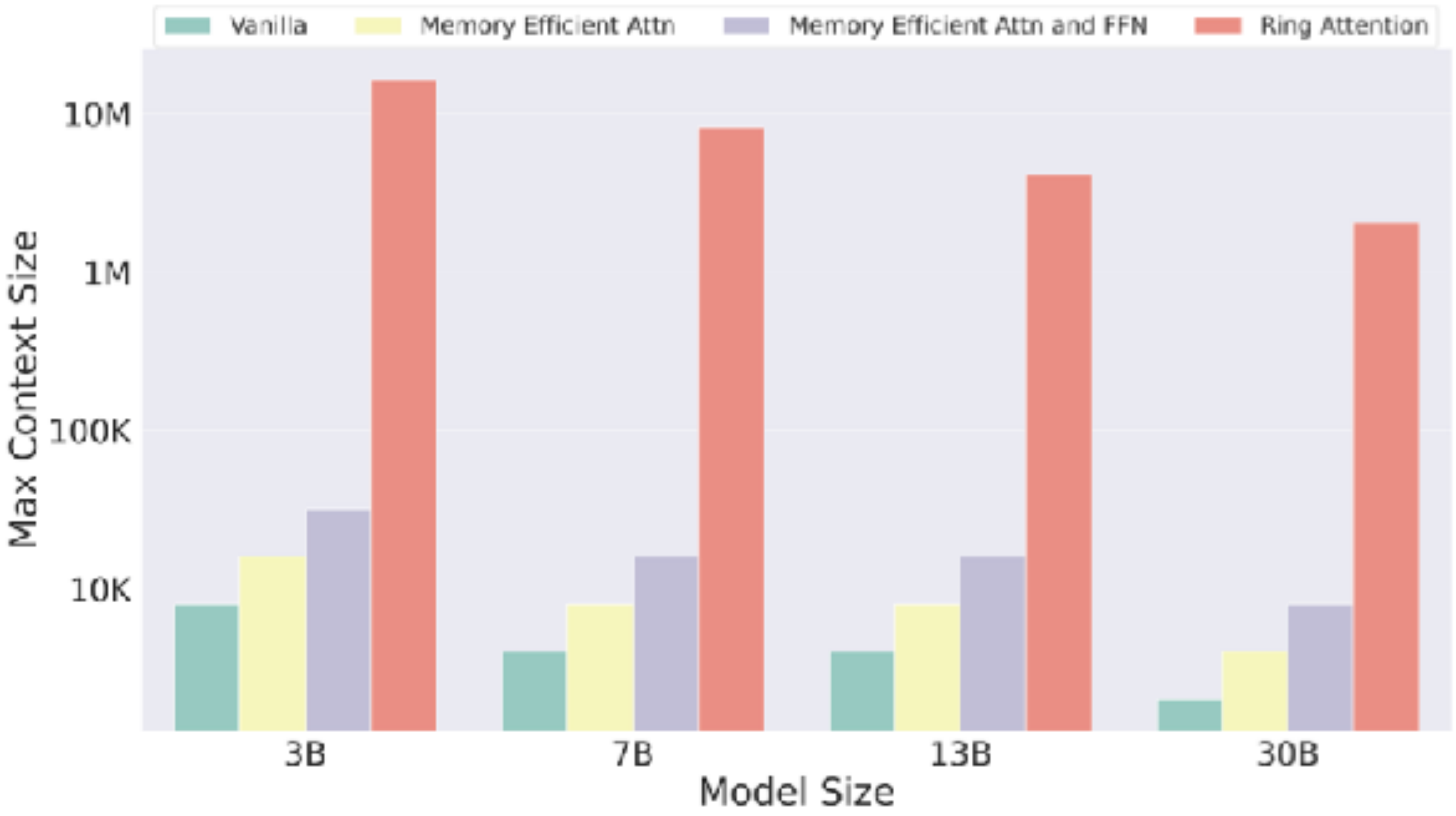


Proposed Solution:
Ring Attention with Blockwise Transformers

Key Advantage:
Reduces memory requirements

Enables training of:
- 500x longer token sequence than other transformers
- sequences exceeding100 million tokens in length without making approximations to attention

Ability to achieve near-infinite context size.

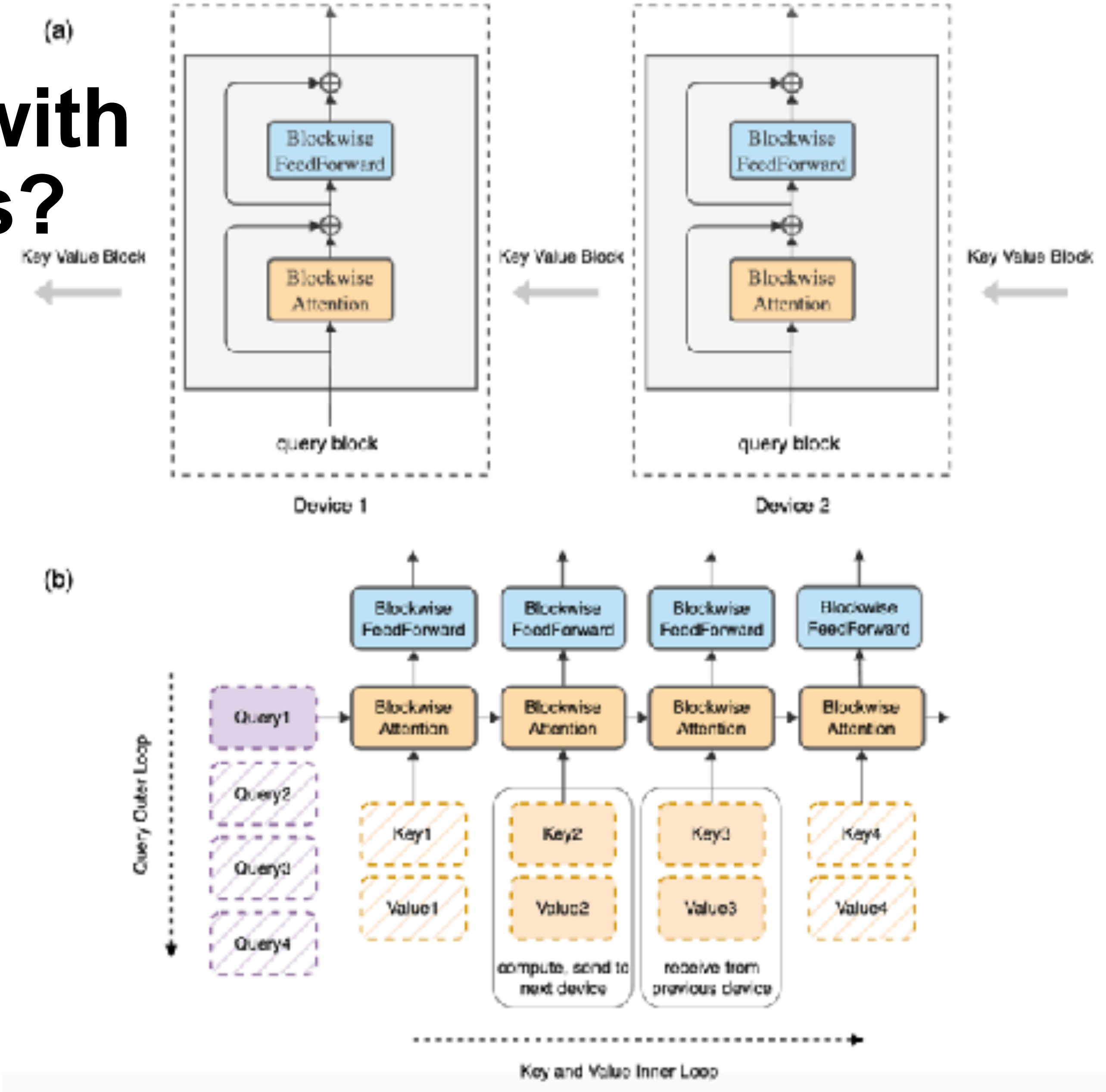


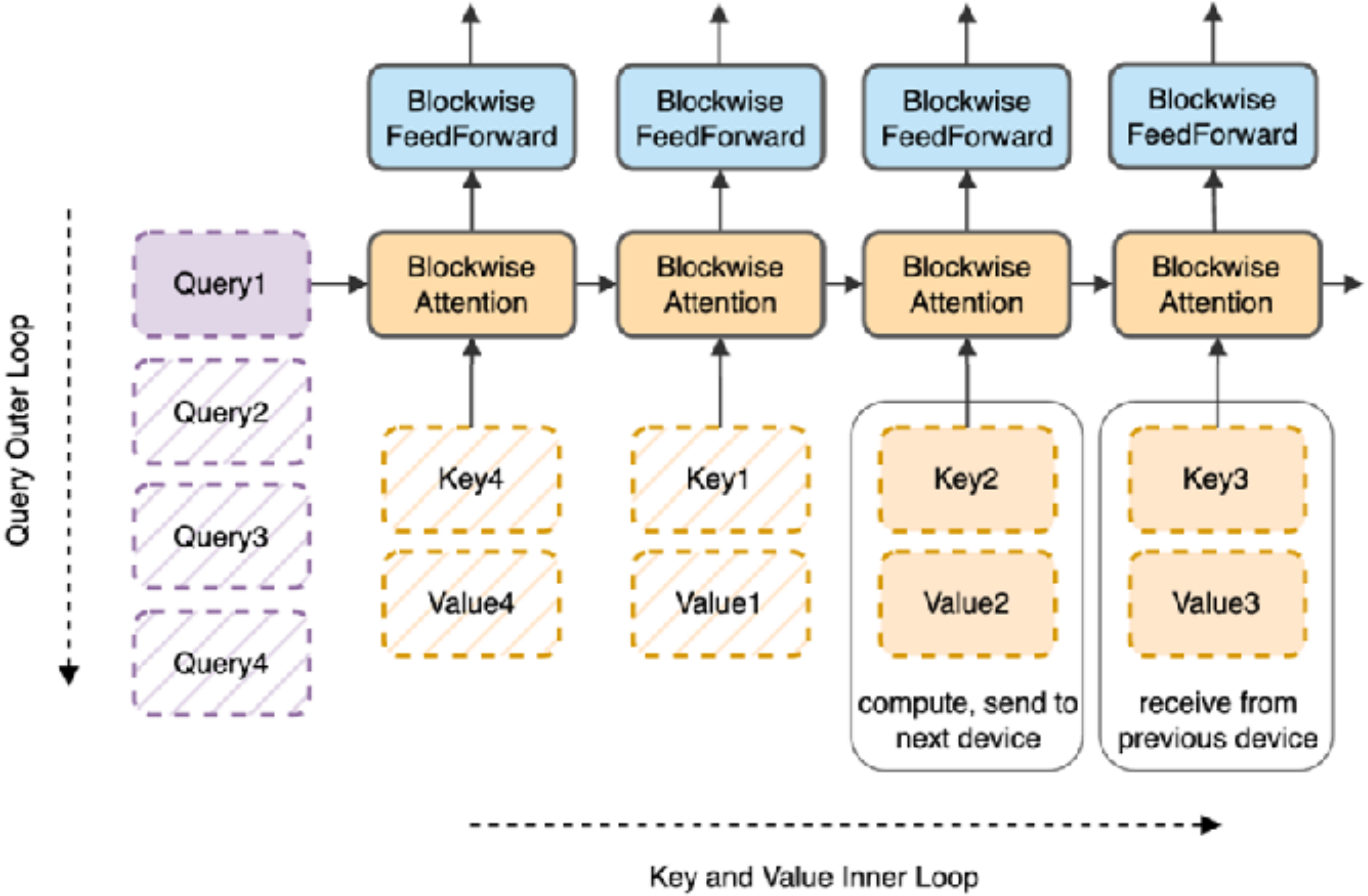
What is Ring Attention with Blockwise Transformers?

Combines: Ring Attention + Blockwise Processing

Ring Attention:
- Restricts attention to a fixed radius of distance from each token.
- Limits computational complexity
- More efficient process for long sequences

Blockwise Processing:
- Chunks input sequence into smaller blocks, each containing a fixed number of tokens
- Processes sequences incrementally
- Reduces memory requirements




Blockwise FeedForward
Blockwise FeedForward
Blockwise FeedForward
Blockwise FeedForward
Query Outer Loop
Query1
Blockwise Attention
Blockwise Attention
Blockwise Attention
Blockwise Attention
Query2
Query3
Query4
Key4
Value4
Key1
Value1
Key2
Value2
compute, send to next device
Key3
Value3
receive from previous device
Key and Value Inner Loop

Code and Demo

Ring Attention

```
class RingAttention(nn.Module):
    def __init__(self, input_size, num_heads=8, block_size=16):
        super(RingAttention, self).__init__()
        self.input_size = input_size
        self.num_heads = num_heads
        self.block_size = block_size

        self.query_projection = nn.Linear(input_size, input_size)
        self.key_projection = nn.Linear(input_size, input_size)
        self.value_projection = nn.Linear(input_size, input_size)
        self.output_projection = nn.Linear(input_size, input_size)
```

Attention Computation on Q,K,V

```python
def forward(self, x):
    batch_size, seq_len, input_size = x.size()
    assert input_size == self.input_size

    # Reshape input into blocks
    x = x.view(batch_size, -1, self.block_size, input_size)

    # Apply projections
    queries = self.query_projection(x)  # [batch_size, num_blocks, block_size, input_size]
    keys = self.key_projection(x)  # [batch_size, num_blocks, block_size, input_size]
    values = self.value_projection(x)  # [batch_size, num_blocks, block_size, input_size]

    # Compute attention scores
    attention_scores = torch.einsum('bijk,bilk->bijl', queries, keys) / (input_size ** 0.5)  # [batch_size, num_blocks, block_size, block_size]
    attention_weights = F.softmax(attention_scores, dim=-1)  # [batch_size, num_blocks, block_size, block_size]

    # Apply attention to values
    attended_values = torch.einsum('bijl,bilk->bijk', attention_weights, values)  # [batch_size, num_blocks, block_size, input_size]

    # Reshape back to original shape
    attended_values = attended_values.view(batch_size, seq_len, input_size)

    # Apply output projection
    outputs = self.output_projection(attended_values)

    return outputs
```

Blockwise Transformer

```
class BlockwiseTransformer(nn.Module):
    def __init__(self, input_size, num_layers=4, num_heads=8, block_size=16):
        super(BlockwiseTransformer, self).__init__()
        self.input_size = input_size
        self.num_layers = num_layers
        self.num_heads = num_heads
        self.block_size = block_size
```

Initialization of Ring Attention Layers in Blockwise Transformer (ie Devices)

```python
# Define Ring Attention layers
RA_Layers = []
for _ in range(num_layers):
  RA_Layers.append(RingAttention(input_size, num_heads, block_size))
self.attention_layers = nn.ModuleList(RA_Layers)
```

Ring Attention Layer Execution and FF Initialization

```
    # Feedforward layer
    self.feedforward = nn.Sequential(
        nn.Linear(input_size, 4 * input_size),
        nn.ReLU(),
        nn.Linear(4 * input_size, input_size)
    )

def forward(self, x):
    for layer in self.attention_layers:
        x = x + layer(x)  # Residual connection
        x = F.layer_norm(x, normalized_shape=x.size()[1:])  # Layer normalization
        x = self.feedforward(x)  # Feedforward layer
    return x
```

Synthetic Dataset

```
# Define SyntheticDataset class
class SyntheticDataset(Dataset):
    def __init__(self, num_samples, seq_len, input_size):
        self.num_samples = num_samples
        self.seq_len = seq_len
        self.input_size = input_size

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        # Generate random sequence tensor
        sequence = torch.randn(self.seq_len, self.input_size)

        # Determine label based on some criteria (e.g., sum of the sequence)
        label = torch.tensor(1 if sequence.sum() > 0 else 0, dtype=torch.long)

        return sequence, label

# Instantiate the dataset and dataloader
num_samples = 1000  # Number of samples in the dataset
seq_len = 1024      # Length of each sequence
input_size = 512    # Dimensionality of each element in the sequence
batch_size = 32     # Batch size for training
dataset = SyntheticDataset(num_samples, seq_len, input_size)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Instantiate the model
model = BlockwiseTransformer(input_size=input_size, num_layers=4, num_heads=8, block_size=16)

# Define device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Training loop & Results: Synthetic Dataset

```python
# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    total_loss = 0.0
    for batch_idx, (inputs, targets) in enumerate(dataloader):
        inputs, targets = inputs.to(device), targets.to(device)

        # Forward pass
        optimizer.zero_grad()
        outputs = model(inputs)

        # Compute loss
        loss = criterion(outputs.mean(dim=1), targets.squeeze())

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        if (batch_idx + 1) % 10 == 0:
            print(f"Epoch [{epoch+1}/{num_epochs}], Batch [{batch_idx+1}/{len(dataloader)}], Loss: {loss.item():.4f}")

    print(f"Epoch [{epoch+1}/{num_epochs}], Average Loss: {total_loss / len(dataloader):.4f}")
```

```
Epoch [1/10], Batch [10/32], Loss: 0.6545
Epoch [1/10], Batch [20/32], Loss: 1.1160
Epoch [1/10], Batch [30/32], Loss: 0.6245
Epoch [1/10], Average Loss: 1.1188
Epoch [2/10], Batch [10/32], Loss: 0.7041
Epoch [2/10], Batch [20/32], Loss: 0.7914
Epoch [2/10], Batch [30/32], Loss: 0.7351
Epoch [2/10], Average Loss: 0.7848
Epoch [3/10], Batch [10/32], Loss: 0.7019
Epoch [3/10], Batch [20/32], Loss: 0.6059
Epoch [3/10], Batch [30/32], Loss: 0.7423
Epoch [3/10], Average Loss: 0.7463
Epoch [4/10], Batch [10/32], Loss: 0.7776
Epoch [4/10], Batch [20/32], Loss: 0.7830
Epoch [4/10], Batch [30/32], Loss: 0.6873
Epoch [4/10], Average Loss: 0.7228
Epoch [5/10], Batch [10/32], Loss: 0.7369
Epoch [5/10], Batch [20/32], Loss: 0.6925
Epoch [5/10], Batch [30/32], Loss: 0.7179
Epoch [5/10], Average Loss: 0.7131
Epoch [6/10], Batch [10/32], Loss: 0.7483
Epoch [6/10], Batch [20/32], Loss: 0.7060
Epoch [6/10], Batch [30/32], Loss: 0.6869
Epoch [6/10], Average Loss: 0.7188
Epoch [7/10], Batch [10/32], Loss: 0.7529
Epoch [7/10], Batch [20/32], Loss: 0.7376
Epoch [7/10], Batch [30/32], Loss: 0.7207
Epoch [7/10], Average Loss: 0.7027
Epoch [8/10], Batch [10/32], Loss: 0.6874
Epoch [8/10], Batch [20/32], Loss: 0.7057
Epoch [8/10], Batch [30/32], Loss: 0.6671
Epoch [8/10], Average Loss: 0.6996
Epoch [9/10], Batch [10/32], Loss: 0.6970
Epoch [9/10], Batch [20/32], Loss: 0.7019
Epoch [9/10], Batch [30/32], Loss: 0.7659
Epoch [9/10], Average Loss: 0.7189
Epoch [10/10], Batch [10/32], Loss: 0.7399
Epoch [10/10], Batch [20/32], Loss: 0.7024
Epoch [10/10], Batch [30/32], Loss: 0.7105
Epoch [10/10], Average Loss: 0.7149
```

Ring Attention Implementation on IMDb Dataset

```
# Load IMDB dataset
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)

# Preprocess the data
max_len = 200  # Limiting sequence length to 200 for padding
train_data = pad_sequences(train_data, maxlen=max_len, padding='post')
test_data = pad_sequences(test_data, maxlen=max_len, padding='post')
```

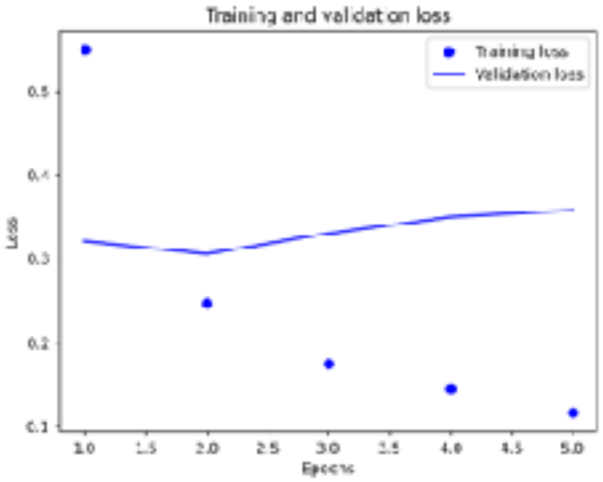
RA on IMDb – Model Build

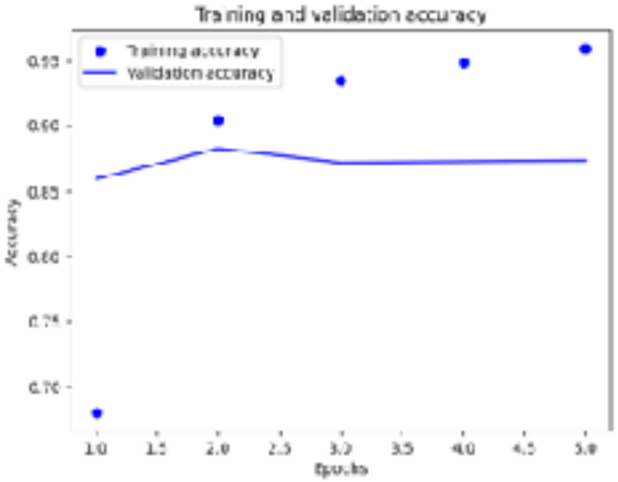
```
def build_model():
    inputs = Input(shape=(max_len,))
    x = Embedding(input_dim=10000, output_dim=128)(inputs)  # Embedding layer
    x = BlockwiseTransformer(num_blocks=6, embed_dim=128, num_heads=8, mlp_dim=128, dropout=0.1)(x)
    x = GlobalAveragePooling1D()(x)
    outputs = Dense(1, activation='sigmoid')(x)
    model = Model(inputs=inputs, outputs=outputs)
    return model

# Create and compile the model
model = build_model()
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Basic training & Results: IMDb Dataset

```python
# Train the model
history = model.fit(train_data, train_labels, epochs=5, batch_size=64, validation_split=0.1)

# Evaluate the model on test data
loss, accuracy = model.evaluate(test_data, test_labels)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Epoch 1/5
352/352 [==============================] - 44s 81ms/step - loss: 0.5507 - accuracy: 0.6803 - val_loss: 0.3206 - val_accuracy: 0.8592
Epoch 2/5
352/352 [==============================] - 17s 47ms/step - loss: 0.2409 - accuracy: 0.9045 - val_loss: 0.3057 - val_accuracy: 0.8824
Epoch 3/5
352/352 [==============================] - 14s 39ms/step - loss: 0.1743 - accuracy: 0.9351 - val_loss: 0.3298 - val_accuracy: 0.8720
Epoch 4/5
352/352 [==============================] - 12s 35ms/step - loss: 0.1444 - accuracy: 0.9484 - val_loss: 0.3493 - val_accuracy: 0.8728
Epoch 5/5
352/352 [==============================] - 12s 34ms/step - loss: 0.1159 - accuracy: 0.9597 - val_loss: 0.3571 - val_accuracy: 0.8736
782/782 [==============================] - 8s 11ms/step - loss: 0.4116 - accuracy: 0.8555
Test Accuracy: 0.8555
```





Basic training & Results: IMDb Dataset

fcNN (model from class lecture)

```
model.predict(x_test)
```

```
782/782 [==============================]
Out[19]:
array([[0.21844056],
       [0.9994658 ],
       [0.8811928 ],
       ...,
       [0.10073161],
       [0.05596441],
       [0.5775681 ]], dtype=float32)
```

Ring Attention model

```
model.predict(test_data)
```

```
782/782 [==============================]
array([[0.01934421],
       [0.99492955],
       [0.9266383 ],
       ...,
       [0.08135927],
       [0.18948923],
       [0.9576044 ]], dtype=float32)
```

Ring Attention Implementation on Reuters Dataset

```
# Load IMDB dataset
(train_data, train_labels), (test_data, test_labels) = reuters.load_data(num_words=10000)

# Preprocess the data
max_len = 2400  # Limiting sequence length to 200 for padding
train_data = pad_sequences(train_data, maxlen=max_len, padding='post')
test_data = pad_sequences(test_data, maxlen=max_len, padding='post')

from tensorflow.keras.utils import to_categorical

# Convert labels to one-hot encoding
train_labels = to_categorical(train_labels, num_classes=46)
test_labels = to_categorical(test_labels, num_classes=46)
```

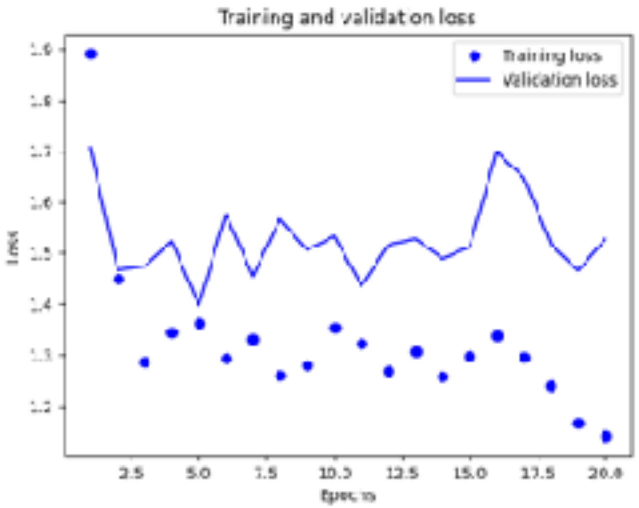
RA on Reuters – Model Build

```
def build_model():
    inputs = Input(shape=(max_len,))
    x = Embedding(input_dim=10000, output_dim=128)(inputs)  # Embedding layer
    x = BlockwiseTransformer(num_blocks=6, embed_dim=128, num_heads=8, mlp_dim=128, dropout=0.1)(x)
    x = GlobalAveragePooling1D()(x)
    outputs = Dense(46, activation='softmax')(x)
    model = Model(inputs=inputs, outputs=outputs)
    return model

# Create and compile the model
model = build_model()
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Basic training & Results: Reuters Dataset

```python
# Train the model
history = model.fit(train_data, train_labels, epochs=20, batch_size=16, validation_split=0.1)

# Evaluate the model on test data
loss, accuracy = model.evaluate(test_data, test_labels)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Epoch 1/20
506/506 [==============================] - 160s 305ms/step - loss: 1.8775 - accuracy: 0.5378 - val_loss: 1.5441 - val_accuracy: 0.6396
Epoch 2/20
506/506 [==============================] - 153s 302ms/step - loss: 1.4365 - accuracy: 0.6432 - val_loss: 1.4434 - val_accuracy: 0.6596
Epoch 3/20
506/506 [==============================] - 153s 302ms/step - loss: 1.3060 - accuracy: 0.6772 - val_loss: 1.4204 - val_accuracy: 0.6630
Epoch 4/20
506/506 [==============================] - 153s 302ms/step - loss: 1.1933 - accuracy: 0.7011 - val_loss: 1.4411 - val_accuracy: 0.6574
Epoch 5/20
506/506 [==============================] - 153s 302ms/step - loss: 1.1523 - accuracy: 0.7053 - val_loss: 2.0182 - val_accuracy: 0.5573
Epoch 6/20
506/506 [==============================] - 153s 302ms/step - loss: 1.1552 - accuracy: 0.7058 - val_loss: 1.4671 - val_accuracy: 0.6440
Epoch 7/20
506/506 [==============================] - 153s 302ms/step - loss: 1.3330 - accuracy: 0.6568 - val_loss: 1.4623 - val_accuracy: 0.6429
Epoch 8/20
506/506 [==============================] - 153s 302ms/step - loss: 1.4294 - accuracy: 0.6328 - val_loss: 1.4116 - val_accuracy: 0.6463
Epoch 9/20
506/506 [==============================] - 153s 302ms/step - loss: 1.2793 - accuracy: 0.6687 - val_loss: 1.3501 - val_accuracy: 0.6529
Epoch 10/20
506/506 [==============================] - 153s 302ms/step - loss: 1.1689 - accuracy: 0.6921 - val_loss: 1.3545 - val_accuracy: 0.6630
Epoch 11/20
506/506 [==============================] - 153s 302ms/step - loss: 1.2007 - accuracy: 0.6881 - val_loss: 1.4764 - val_accuracy: 0.6185
Epoch 12/20
506/506 [==============================] - 153s 302ms/step - loss: 1.2205 - accuracy: 0.6771 - val_loss: 1.4237 - val_accuracy: 0.6374
Epoch 13/20
506/506 [==============================] - 153s 302ms/step - loss: 1.1527 - accuracy: 0.6958 - val_loss: 1.4012 - val_accuracy: 0.6541
Epoch 14/20
506/506 [==============================] - 153s 302ms/step - loss: 1.0658 - accuracy: 0.7183 - val_loss: 1.4762 - val_accuracy: 0.6385
Epoch 15/20
506/506 [==============================] - 153s 302ms/step - loss: 1.1000 - accuracy: 0.7085 - val_loss: 1.3708 - val_accuracy: 0.6529
Epoch 16/20
506/506 [==============================] - 153s 302ms/step - loss: 1.0718 - accuracy: 0.7177 - val_loss: 1.3033 - val_accuracy: 0.6707
Epoch 17/20
506/506 [==============================] - 153s 303ms/step - loss: 1.0768 - accuracy: 0.7169 - val_loss: 1.3744 - val_accuracy: 0.6563
Epoch 18/20
506/506 [==============================] - 153s 303ms/step - loss: 1.1007 - accuracy: 0.7105 - val_loss: 1.3485 - val_accuracy: 0.6641
Epoch 19/20
506/506 [==============================] - 153s 303ms/step - loss: 1.1072 - accuracy: 0.7010 - val_loss: 1.4090 - val_accuracy: 0.6429
Epoch 20/20
506/506 [==============================] - 154s 303ms/step - loss: 1.0849 - accuracy: 0.7130 - val_loss: 1.4300 - val_accuracy: 0.6506
71/71 [==============================] - 14s 192ms/step - loss: 1.4331 - accuracy: 0.6327
Test Accuracy: 0.6327
```



Compare with Alternative Solutions

Simple attention

- Keys, Values, Query vectors -> attention scores

Cross attention

- Learns relationships across separate queries. Good for Q&A.

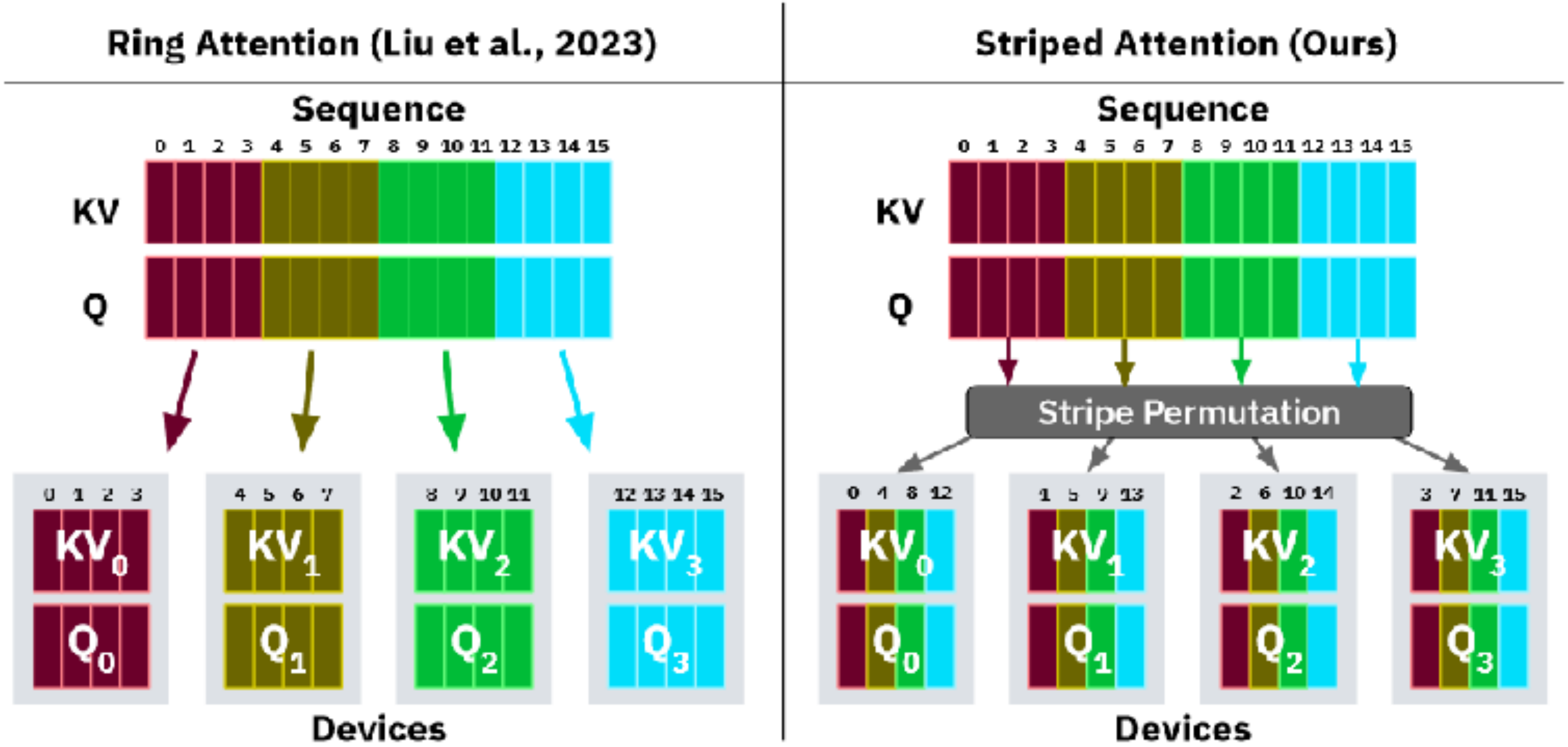
Vanilla attention mechanisms

- Do not scale well with increasing sequence lengths due to quadratic complexity.
- Ring attention scales linearly with the number of devices, allowing it to handle near-infinite context sizes.

Blockwise Attention (*Ring attention builds upon this idea*)

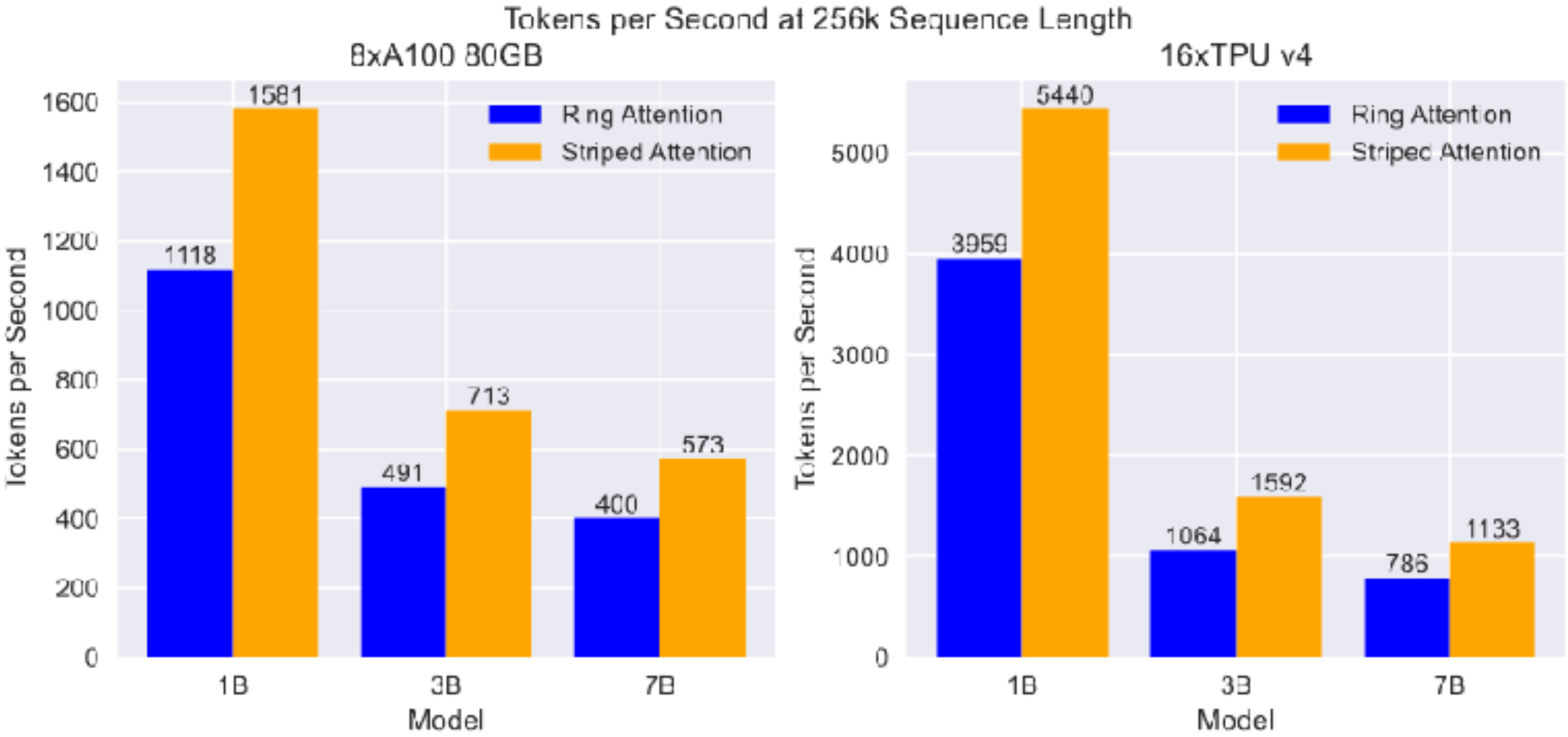
- Chunks sequence into blocks (groups of tokens) for stepwise processing.

Especially Noteworthy Alternate Solution: Striped Attention



https://arxiv.org/pdf/2311.09431.pdf

Especially Noteworthy Alternate Solution: Striped Attention



https://arxiv.org/pdf/2311.09431.pdf

Conclusions

- Ring attention is built using a blockwise implementation.
- Ring attention provides significant improvement in memory utilization over vanilla transformers.
- Newer approaches, such as striped attention, are taking this idea even further by achieving faster application speeds.
- Evolution of algorithms in this field is occurring rapidly.
- Large World Model (LWM) uses ring attention to handle larger contexts
  - eg: understands up to 1-hour videos

References

Ring Attention with Blockwise Transformers for Near-Infinite Context

- https://arxiv.org/pdf/2310.01889.pdf

Blockwise Self-Attention for Long-Document Understanding

- https://arxiv.org/abs/1911.02972

Striped Attention: Faster Ring Attention for Causal Transformers

- https://arxiv.org/pdf/2311.09431.pdf

Large World Model (LWM)

- https://github.com/LargeWorldModel/LWM/blob/main/README.md

Ring Attention in Large World Model (LWM)

- https://jrodthoughts.medium.com/inside-large-world-model-uc-berkeley-multimodal-model-that-can-understand-1-hour-long-videos-d1a97c5c7fa0#:~:text=LWM%20provides%20a%20strong%20foundation,to%20interact%20with%20long%20videos.

Questions?

Thank you!