CSE-234: Data Systems for Machine Learning, Winter 2025

3: Basics: autodiff, ML system architecture overview

Lecturer: Hao Zhang
Scribes: Khushi Patel, Raymond Sun, Matthew Omalley-Nichols, Philip Chi, Sihan Wang, Honghao Lin, Yen-Pu Wang, Forrest Dai, Jintong Luo, Aakash Agrawal

1 Announcements + Multiple Choice Questions

1.1 Pre-lecture:

- CSE will process the enrollments
- Last week we covered: matmul, softmax, computational graphs, programming- imperative vs symbolic, static vs dynamic, and JIT (just in time) and its bottleneck
- The goal for today's lecture is to cover autodiff, and give an MLSys architecture overview

1.2 Multiple Choice Questions

MCQ 1) You are a machine learning engineer at a company that is providing LLM endpoints to users. Your goal is running efficient inference for these LLMs. You are given a framework which has both symbolic and imperative APIs. While designing your system, would you:
A. Use symbolic mode for both testing and deployment of your system.
B. Use imperative mode for development and symbolic mode for deployment.
C. Use symbolic mode for development and imperative mode for deployment.
D. Use imperative mode for both testing and deployment of your system.

The answer is B because you want something that's easier to debug at development and once you figure out your program, you basically deploy it using the most high performance one.

MCQ 2) Which of the following is not true about dataflow graphs?
A. Static dataflow graphs are defined once and executed many times
B. No extra effort is required for batching optimization of static dataflow graphs
C. Dynamic dataflow graphs are easy to debug
D. Define-and-run is a possible way to handle dynamic dataflow graphs

The answer is C. Dataflow graph means its static so you just define once and it will run forever for arbitrary data. Given a static data graph, batching is natural so no extra effort is required. Define-and-run is a possible way to handle dynamic dataflow graphs because you don't care about performance.

2 Forward Mode AD

The classic definition of partial derivative is

$$\frac{\partial f}{\partial \theta} = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon) - f(\theta)}{\epsilon} \approx \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon} + o(\epsilon^2)$$

.

This formula has 2 problems for auto-differentiation: 1. it's slow, requiring two evaluations of f to compute a single gradient, and 2. it's approximal and loses precision from floating point errors in practice.

Instead, autodiff uses symbolic differentiation, where the sum, product, and chain rules for partial differentiation are applied to the computational graph. These rules can be applied either forward or backward.

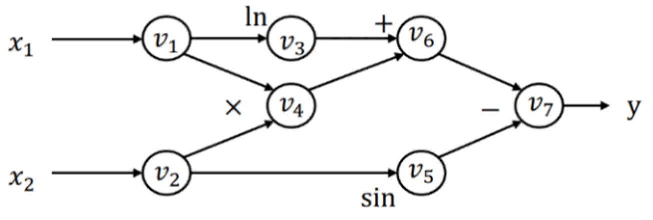
For forward mode autodiff, we start from the input nodes and derive the gradient all the way to the output nodes. To calculate $\partial y/\partial x_1$, define $\dot{v_i} = \partial v_i/\partial x_i$, and solve each $\dot{v_i}$ in the forward order of the graph.

With forward mode autodiff, for a $f : R^n \to R^k$ function, we need n forward passes to get the gradient with respect to each input, but in ML, usually the input dimension n is large, while the output dimension k is 1 or small, so forward mode autodiff is not as effective for ML.

3 Reverse Mode AD

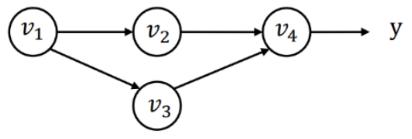
In reverse mode autodiff, we compute the gradients starting from the output y and working back to the inputs x_n. To achieve this, we use the adjoint $\overline{v_i} = \frac{\partial y}{\partial v_i}$, which represents the partial derivative of the output y with respect to the node v_i, where i is the node index. We compute each adjoint $\overline{v}_i$ in the reverse topological order of the computational graph, which must be a directed acyclic graph (DAG) to ensure proper ordering.

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$$



In the example above, the first adjoint, $\overline{v_7}$, is equivalent to the identity function since the output y maps to v_7. From there, we apply the chain rule of calculus to propagate gradients backward through the graph. We calculate the partial derivatives node by node until we reach the original inputs x_n, whose adjoints correspond to $\frac{\partial y}{\partial x_i}$.

For nodes with multiple consumers (i.e., nodes that feed into multiple paths in the graph), the adjoint computation involves unrolling y as a function of intermediary nodes. This scenario is illustrated in the graph below.



In this case, the adjoint $\overline{v_1}$ can be expressed as the partial derivative of y with respect to the index node v_1. Expanding y as a function of intermediary nodes, we have:

$$\overline{v_1} = \frac{\partial y}{\partial v_1} = \frac{\partial f(v_2, v_3)}{\partial v_2}\frac{\partial v_2}{\partial v_1} + \frac{\partial f(v_2, v_3)}{\partial v_3}\frac{\partial v_3}{\partial v_1} = \overline{v_2}\frac{\partial v_2}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1}$$

In general, the adjoint of an index node with multiple consumers is the sum of the adjoints of its consumers, each multiplied by the partial derivative of the consumer with respect to the index node:

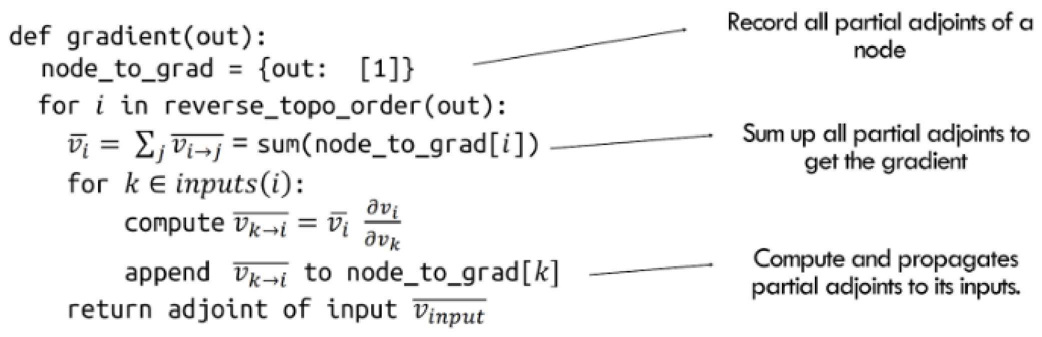
$$\overline{v_i} = \sum_{j \in next(i)} \overline{v_{i\to j}} \text{ , where } \overline{v_{i\to j}} = \overline{v_j}\frac{\partial v_j}{\partial v_i}$$

Notice that this adjoint corresponds to the accumulated gradient of the loss with respect to a certain node in the computational graph. Therefore, the above equation describes backpropagation in a computational graph.

In short, reverse mode autodiff finds the gradient propagation in a neural network starting from the output nodes and traversing back the network in reverse topological order. Reverse mode autodiff computes the gradient of the output y with respect to all inputs x_n in a single backward pass, making it efficient in machine learning where y typically represents a scalar loss function. In scenarios where the dimensions of the output space are larger than the input space, forward mode autodiff should be considered instead.

4 Backward Mode

The goal is to construct a graph that calculates the adjoint values for reverse mode AD. The algorithm is defined as follows:



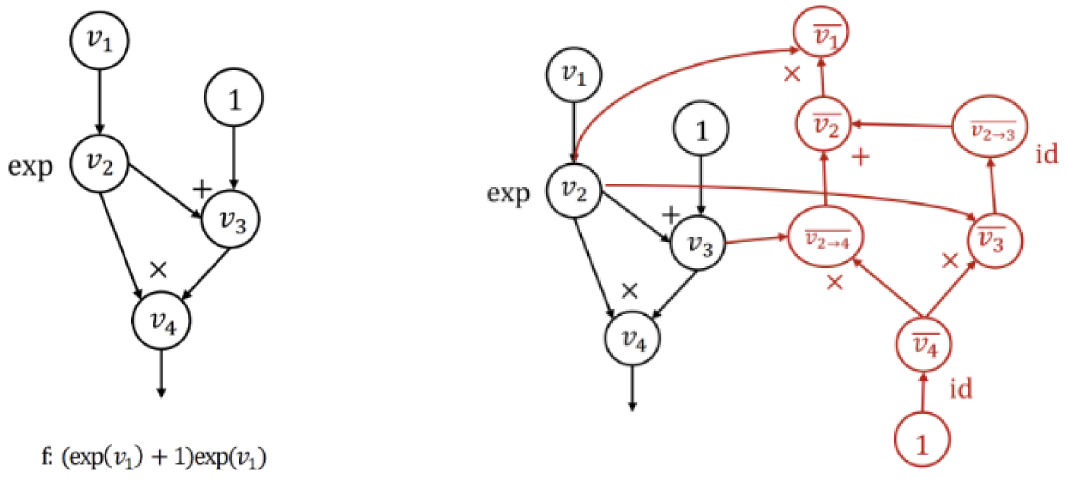
```
def gradient(out):
  node_to_grad = {out:  [1]}
  for $i$ in reverse_topo_order(out):
    $\bar{v_i} = \sum_j \overline{v_{i\to j}}$ = sum(node_to_grad[$i$])
    for $k \in inputs(i)$:
        compute $\overline{v_{k\to i}} = \bar{v_i}\frac{\partial v_i}{\partial v_k}$
        append $\overline{v_{k\to i}}$ to node_to_grad[$k$]
    return adjoint of input $\overline{v_{input}}$
```

Record all partial adjoints of a node
Sum up all partial adjoints to get the gradient
Compute and propagates partial adjoints to its inputs.

'node_to_grad' starts as a key value pair of "out: [1]" where the value is an array with a 1 because the adjoint of the output $\frac{\partial y}{\partial y}$ is always 1.
The loop iterates through all the nodes in the graph in reverse topological order. On each iteration of the loop, we first get $\bar{v}_i$ by summing all $v_{i\bar{\to}j}$. At this point on the graph, we create a new node $\bar{v}_i$ then connect to it from all nodes $v_{i\bar{\to}j}$. Next, for all k in nodes v_k that connect to v_i, we compute their partial adjoint with the expression $v_{k\bar{\to}i} = \bar{v}_i\frac{\partial v_i}{\partial v_k}$. At this step, we can expand the backward graph by creating a node $v_{k\bar{\to}i}$ with edges going from $\bar{v}_i$, and $\frac{\partial v_i}{\partial v_k}$ if the node exists on the graph, to itself. We then append the partial

adjoint to the array ‘node_to_grad[k]‘ for future use.

Following this procedure, the graph on the left would expand to the graph on the right.



5 Backpropagation vs Reverse AD

The summary of Backward AD Rather than working directly with concrete values (numerical computations), the backward graph symbolically represents the operations involved, enabling the automatic computation of derivatives. Once the backward graph is constructed, it can be reused for different sets of input values. This allows efficient computation during training processes in machine learning models, especially neural networks. Popular machine learning libraries like TensorFlow and PyTorch utilize backward automatic differentiation to compute gradients efficiently. This functionality enables optimization algorithms like gradient descent to update model parameters.

5.1 Backpropagation vs. Reverse-mode AD

Reverse mode AD creates a bigger graph, which captures both the forward computations and the reverse gradient propagation in a symbolic form. This graph structure is commonly used by modern frameworks such as TensorFlow and PyTorch because it allows for efficient computation of derivatives with respect to many inputs simultaneously, a necessity for large-scale machine learning models. In contrast, backpropagation (used in frameworks like Caffe and CUDA-convnet) focuses solely on computing the gradient for a specific function or layer. While effective for basic gradient computation, backpropagation lacks the flexibility and reusability provided by reverse mode AD, especially for complex scenarios.

5.2 Missing points

To complete the machine learning training process, the critical missing component is weight update, which ensures the model learns and improves over iterations.

$$\theta^{(t+1)} = f(\theta^{(t)}, \nabla L(\theta^{(t)}, D^{(t)}))$$

The loss function is defined as:

$$L = \text{MSE}(w_2 \cdot \text{ReLU}(w_1 x), y)$$

where the parameters are $\theta = \{w_1, w_2\}$ and the dataset is $D = \{(x, y)\}$.

The training process consists of the following steps: Forward propagation is used to compute the loss $L(\cdot)$. Backward propagation is then performed to calculate the gradient $\nabla L(\cdot)$. Finally, weight update is applied using an update rule such as:

$$f(\theta, \nabla L) = \theta - \eta \nabla L$$

where η is the learning rate.

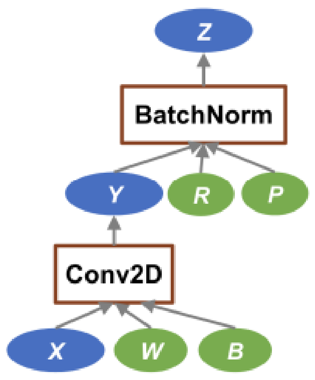
6 MLSys Grand Problem

The grand challenge in ML systems (MLSys) revolves around designing systems that meet ambitious yet essential goals. These include ensuring systems are fast, scalable, and memory-efficient to handle large-scale machine learning workloads. They should be adaptable to diverse hardware platforms, which is crucial for maximizing compatibility and performance across various environments, from cloud GPUs to edge devices. Energy efficiency is another critical priority, addressing growing concerns about the environmental impact of machine learning. Finally, these systems must be user-friendly, making programming, debugging, and deployment intuitive and efficient. Achieving these objectives requires advancements in distributed systems, optimization algorithms, and hardware-software co-design, emphasizing a balanced trade-off between performance and usability in the ever-evolving ML landscape.

7 Graph Optimization

Intuition: The graph written by the user might be inefficient, the system will take the graph defined by the user, try to analyze it and optimize it so that the graph will be more efficient. Goal of the optimization: 1. Rewrite the original graph G to G'. 2. G' runs faster than G.

Below is a portion in the ResNet as a simple example of how graph optimization can be used to make the system faster.



$$Z(n, c, h, w) = Y(n, c, h, w) * R(c) + P(c)$$

$$Y(n, c, h, w) = \left(\sum_{d,u,v} X(n, d, h+u, w+v) * W(c, d, u, v) \right) + B(n, c, h, w)$$

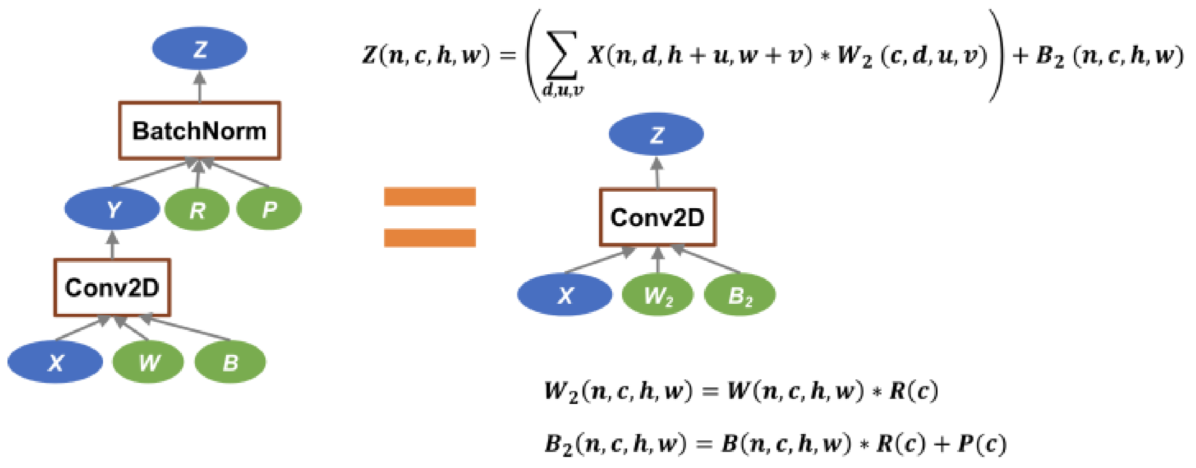
The portion consists of one BatchNorm and one Conv2D operation. To optimize this, we can unroll the computation by plugging $Y(n,c,h,w)$ into the equation of $Z(n,c,h,w)$:

$$Z(n,c,h,w) = [(\Sigma_{d,u,v} X(n,d,h+u,w+v) \cdot W(c,d,u,v)) + B(n,c,h,w)] \cdot R(c) + P(c)$$

$$Z(n,c,h,w) = (\Sigma_{d,u,v} X(n,d,h+u,w+v) \cdot W(c,d,u,v) \cdot R(c)) + B(n,c,h,w) \cdot R(c) + P(c)$$

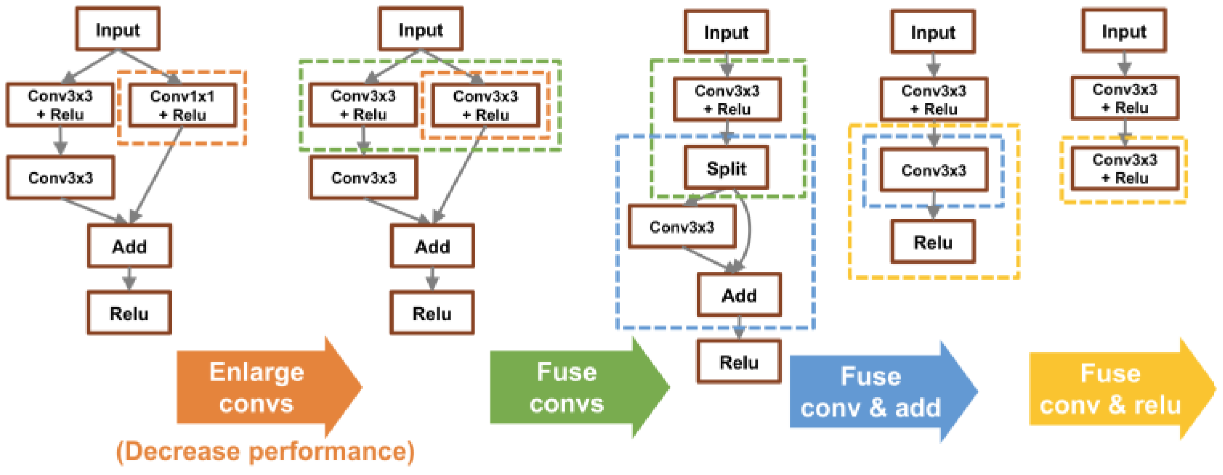
Let $W_2(n,c,h,w) = W(n,c,h,w) \cdot R(n)$ and $B_2(n,c,h,w) = B(n,c,h,w) \cdot R(c) + P(c)$, we can reformat our $Z(n,c,h,w)$ as: Z(n,c,h,w) $=(\Sigma_{d,u,v} X(n,d,h+u,w+v) \cdot W_2(c,d,u,v)) + B_2(n,c,h,w)$

This equation will become a single BatchNorm with weight W_2 and bias B_2, as shown in the diagram below.


Z
BatchNorm
Y
R
P
Conv2D
X
W
B
$Z(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W_2(c,d,u,v)\right) + B_2(n,c,h,w)$
Z
Conv2D
X
W_2
B_2
$W_2(n,c,h,w) = W(n,c,h,w) * R(c)$
$B_2(n,c,h,w) = B(n,c,h,w) * R(c) + P(c)$

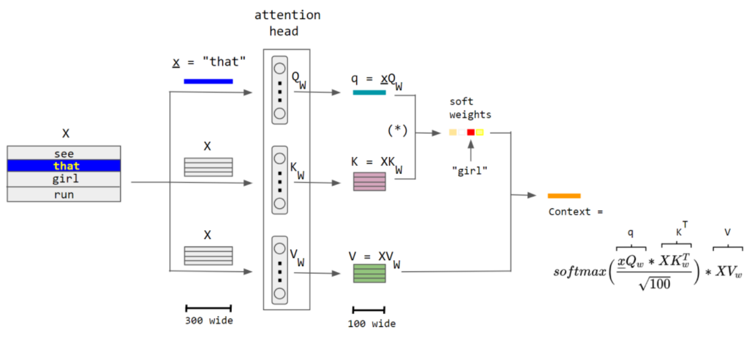
After optimization, it turns out, we have a single Conv2D operation to do the exact same task as the combination of a Conv2D and a BatchNorm can do, and the runtime of the same task is faster as it only uses one operator.

We can check a more aggressive example with graph optimization as shown below.



By looking at the holistic representation of the graph, the system will look for every opportunity to put Conv2D and BatchNorm operations together as those two operations are the key computations in the dataflow graph. As a result after several steps, we are combining all 5 operations into two Conv2D operations. By fusing the operations into smaller numbers of operations, the runtime apparently would be faster as well.

One more example of graph optimization as a motivating example would be the attention mechanism in Transformer.



```
# Original
Q = matmul(W_q, h)
K = matmul(W_k, h)
V = matmul(W_v, h)

# Merged QKV
QKV = matmul(concat(W_q, W_k, W_v), h)
```

In the attention mechanism, each of the three parameters Q, K, and V is calculated by one matmul operation, to optimize this, we can concatenate the weight matrices of Q, K, and V, and perform one single matmul operation to get the merged QKV instead. This would be faster as we reduce the graph size by fusing the operators into smaller numbers of operations.

8 Arithmetic Intensity

Arithmetic Intensity defined as the ratio of total operations over total data movements (AI = #ops / #bytes). The higher the arithmetic intensity, the better / more efficient the algorithm is. Example:

```
void add(int n, float* A, float* B, float* C){
    for (int i = 0; i < n; i++)
        C[i] = A[i] + B[i]
}
```

Algorithm:
Read A[i]
Read B[i]
Add A[i] + B[i]
Stores C[i]

Arithmetic Intensity: 1 compute and 3 read/writes $->$ 1/3

Different programs can perform differently, causing a difference in arithmetic intensity. Program goal: compute E = D((A + B) * C) With

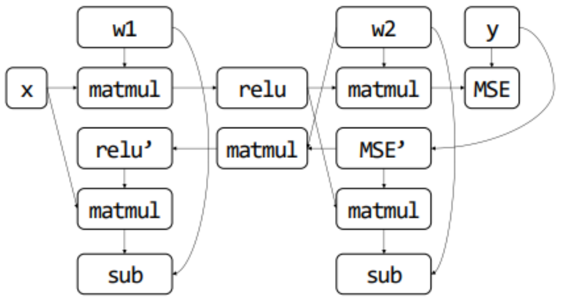
```
void add(int n, float* A, float* B, float* C){
    for (int i = 0; i < n; i++)
        C[i] = A[i] + B[i];
}
```

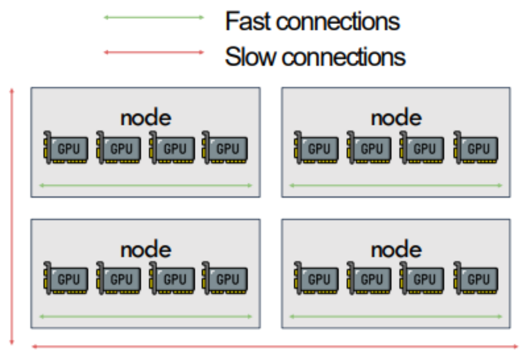
```
void mul(int n, float* A, float* B, float* C){
    for (int i = 0; i < n; i++)
        C[i] = A[i] * B[i];
}

float *A, *B, *C, *D, *E, *tmp1, *tmp2;


Algorithm 1:
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);

Algorithm 2:
void fused(int n, float* A, float* B, float* C, float* D, float* E){
    for (int i=0; i<n; i++){
        E[i] = D[i] + (A[i] + B[i]) * C[i];
  }
}

fused(n, A, B, C, D, E)
```

Arithmetic Intensity for Algorithm 1: 3 compute and 9 read/writes $- >$ 1/3 Arithmetic Intensity for Algorithm 2: 3 compute and 5 read/writes $- >$ 3/5

9 Parallelization

Goal: parallelize the graph compute over multiple devices Device cluster: Over a device cluster, there may be multiple nodes, each with several GPUs within the node connected through NVLink. Connections between GPUs are faster in the same node and slower in different nodes(typically 10 - 100x slower). Therefore, it is critical to cut the graph into components and put them on appropriate GPUs within the cluster, **minimizing connections across nodes and maximizing connections within nodes.**





10 Runtime and Scheduling, Operator Implementation

Parallelization Problems: how to partition, how to communicate, how to schedule, how about consistency, how to auto-parallelize.

Runtime and Scheduling Goal: schedule compute, communicate and memory so that they are as fast as possible, overlap communicate with compute, and everything should be subject to memory constraints.

Operator Implementation Goal: get the fastest possible implementation of operations like matmul, conv2d, etc. for different hardwares (V100, A100, H100, phone, TPU, etc.), precisions (fp32, fp16, fp8, fp4) and shapes (conv2d_3x3, conv2d_5x5, matmul2D, matmul3D, attention) From a high-level perspective, we have already discussed mathematical primitives (primarily matrix multiplication) and representations that express computations using these primitives. The next step is to explore how to execute these computations efficiently on clusters comprising diverse hardware types. Beyond graph optimization, parallelization, and runtime scheduling, our current focus is on optimizing operators. The primary objective is to maximize the arithmetic intensity, defined as #ops / #bytes, or the ratio of the number of operations to the number of bytes transferred.

11 Optimize Arithmetic Intensity

11.1 Key Focus Areas in Optimization

Vectorization: This technique uses hardware instructions to process multiple data points simultaneously, reducing execution time. CPUs and GPUs can handle several elements in one operation instead of one at a time, boosting efficiency.
Data Layout Optimization: The way data is stored in memory (row-major vs. column-major) affects access speed. Aligning data with the memory hierarchy improves cache use and accelerates read/write operations.
Parallelization: Splitting tasks across multiple cores or threads speeds up processing. GPUs excel here, using thousands of cores to handle large computations quickly.

11.2 Vectorization

Vectorization speeds up operations by using hardware-optimized batch instructions. Consider the following unvectorized code, which loops 256 times to add two arrays and store the values in a third array.

```
float A[256], B[256], C[256];
for (int i = 0; i < 256; ++i) {
    C[i] = A[i] + B[i];.
}
```

Instead of adding array elements one by one, vectorized code uses functions like load_float4, add_float4, and store_float4 to load, compute, and store four elements at a time.

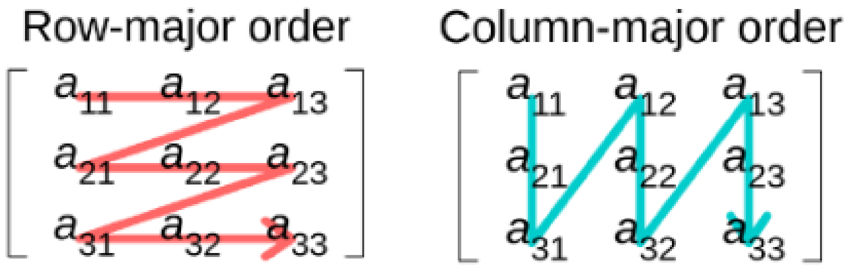
```
for (int i = 0; i < 64; ++i) {
    float4 a = load_float4(A + i*4);
    float4 b = load_float4(B + i*4);
    float4 c = add_float4(a, b);
    store_float4(C + i*4, c);
}
```

Now, instead of looping 256 times to perform the operation, we only need to loop 64 times since we are processing 4 floats at a time. This reduces loop iterations and memory access, significantly improving performance.

11.3 Data Layout

Efficient memory access is critical for high-performance computing. Since memory is inherently linear, multi-dimensional data structures like tensors must be stored sequentially. Two common data layouts are row-major and column-major:

- **Row-major** order stores data row by row, meaning consecutive memory addresses hold elements of the same row.
- **Column-major** order stores data column by column, meaning consecutive memory addresses hold elements of the same column.


Row-major order
Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Consider the following code:

```
for $j = 0$ to $N - 1$ do
    for $i = 0$ to $M - 1$ do $sum$ += $a[i][j]$
```

Assuming the array a is stored in row-major order, this loop structure results in inefficient memory traversal, as it involves iterating through the column dimension first, resulting in hops while accessing the elements from memory. A simple improvement involves swapping the loops, which ensures that memory is accessed sequentially, enhancing cache efficiency and reducing memory latency.

A modern approach that allows for dynamic memory access patterns is the strides format. This format acts as a generalization of the row and col major accesses. Consider an N-dimensional tensor A and let A_internal be its underlying storage (can be row-major or column-major storage). Accessing the element A[i0][i1][i2]... using the strides format is done as follows:

A[i0][i1][i2]... = A_internal[
stride_offset
+ i0 * A.strides[0]

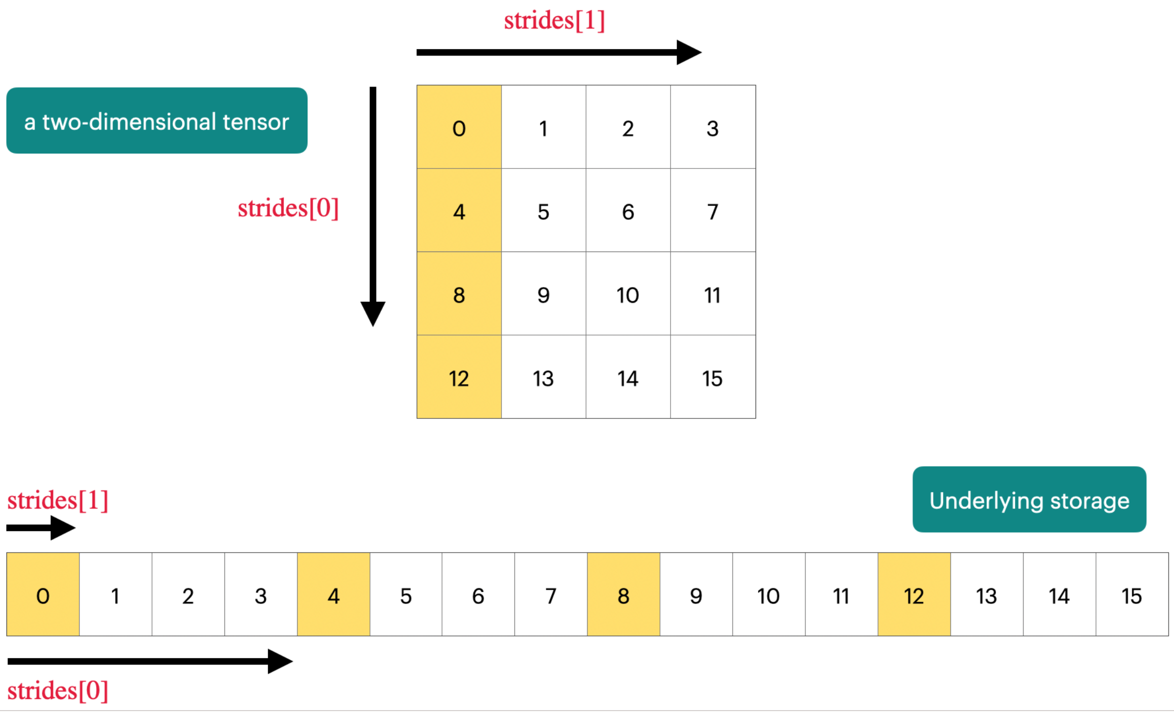
+ i1 * A.strides[1]
+ i2 * A.strides[2]
+ ...
+ in-1 * A.strides[n-1]
]

This format consists of two parameters: offset and strides.

- **Offset** indicates the offset of the tensor relative to the underlying base storage
- **Strides** is an array of the same shape as the total number of dimensions of the tensor (i.e., len(strides) = A.shape). Strides[i] indicate how many elements need to be skipped in memory in order to move "one-unit" in the ith dimension of the tensor.

This approach can be better understood by taking an example of a 2D tensor, where the access is given by: A[i, j] = A.data[offset $+ i \times A$.strides[0] $+ j \times A$.strides[1]]

For the given 2D array example of shape (4 x 4), the underlying storage follows a row-major layout. Here, we would skip 4 elements in memory in order to access the next row element in the array. Hence, Strides[0] = 4. Since we only need to skip 1 element to the right to access the next array element in the column dimension, Strides[1] = 1.



The strides format offers great flexibility and acts as a generalization for (i, j) indexing for both row-major and col-major layouts. For a 2D array, the strides array is given by:

- If the underlying storage of an array A is row-major, strides = [A.shape[1], 1]
- If the underlying storage of an array A is col-major, strides = [1, A.shape[0]]

11.4 Benefits of using the Strides format

Strides separate the underlying storage and the view of the tensor. We can perform various tensor operations by just changing how the data is accessed (via stride values) rather than moving or duplicating the data in a new memory location, this is called Zero-Copy. The underlying storage in memory remains unaffected. These tensor operations include: slicing, transpose, broadcasting, etc.

- **Permute/Transpose:** Strides format helps achieve permute operations by modifying the strides array values.
- **Slice:** This operation is achieved by manipulating both the offset values and the strides array values.
- **Broadcast:** The only thing that changes to achieve broadcasting is the strides array.

11.5 Issues with the Strides format

In operations like slicing, the elements accessed by indexing are not contiguous in the original memory. However, many vectorized operations require the elements to be stored sequentially in the memory.

A way to mitigate this issue is to use tensor.contiguous module provided by prominent libraries like Pytorch. This will make the underlying storage of results (post operations like slicing) contiguous. However, this involves copying elements from one location in the memory to another.

12 Contributions

- Khushi Patel: Section 1, Assigning Tasks, Converting google doc notes from others to latex
- Raymond Sun: Section 2
- Matthew Omalley-Nichols: Section 3
- Philip Chi: Section 4
- Sihan Wang: Section 5, 6
- Honghao Lin: Section 7
- Yen-Pu Wang: Section 8, 9
- Forrest Dai: Section 10
- Jintong Luo: Section 11.1, 11.2
- Aakash Agrawal: Section 11.3, 11.4, 11.5