Task ID	Task	Prompt
1	Sigmoid	Implement a CUDA program for sigmoid activation function: sigmoid$(x) = 1/(1 + \exp(-x))$. Input shape: (batch_size, dim); Output: same shape as input.
2	Matrix Multiplication	Write a program that multiplies two matrices of 32-bit floating point numbers on a GPU. Given matrix A of dimensions $M \times K$ and matrix B of dimensions $K \times N$, compute the product matrix $C = A \times B$, which will have dimensions $M \times N$.
3	Max Pooling 3D	Implement a CUDA program for 3D max pooling function that selects the maximum value within a defined region (a window) of a feature map. Input shape: (batch_size, channels, dim1, dim2, dim3); Output: with 3D max pooling applied.
4	LayerNorm	Implement a GPU program that performs Layer Normalization (LayerNorm) operation, which normalizes across the features for each individual data sample in a layer. Input of shape (batch_size, features, dim1, dim2); Output with Layer Normalization applied, same shape as input.
5	2D Convolution	Write a program that performs a 2D convolution operation on the GPU. Given an input matrix and a kernel (filter), compute the convolved output. The convolution should be performed with a "valid" boundary condition, meaning the kernel is only applied where it fully overlaps with the input. The input consists of: (1) input: A 2D matrix of 32-bit floating-point numbers, represented as a 1D array in row-major order. (2) kernel: A 2D kernel (filter) of 32-bit floating-point numbers, also represented as a 1D array in row-major order. The output should be written to the output matrix (also a 1D array in row-major order). The output matrix will have dimensions: output_rows = input_rows - kernel_rows + 1, output_cols = input_cols - kernel_cols + 1. The convolution operation is defined as: $output[i][j] = \sum_{m=0}^{kernel\_rows-1} \sum_{n=0}^{kernel\_cols-1} input[i+m][j+n] * kernel[m][n]$.
6	Multi-Head Self-Attention	Implement a CUDA program for multi-head self-attention. Given three input matrices Q (queries), K (keys), and V (values) of size $N \times d_{\text{model}}$, compute: MultiHead$(Q, K, V)$ = Concat$(\text{head}_1, \ldots, \text{head}_h)$, where each head computes: $\text{head}_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$ with $d_k = d_{\text{model}}/h$ and Q_i, K_i, V_i being the i-th head's partition of the input matrices.
7	Mean Square Error	Implement a CUDA program to calculate the Mean Squared Error (MSE) between predicted values and target values. Given two arrays of equal length, predictions and targets, compute: MSE $= \frac{1}{N} \sum_{i=1}^{N} (predictions_i - targets_i)^2$ where N is the number of elements in each array. Input: predictions, targets; Output: MSE.
8	Matrix Transpose	Write a program that transposes a matrix of 32-bit floating point numbers on a GPU. The transpose of a matrix switches its rows and columns. Given a matrix A of dimensions rows $\times$ cols, the transpose A^T will have dimensions cols $\times$ rows. All matrices are stored in row-major format.

9	Reverse Array	Implement a program that reverses an array of 32-bit floating point numbers in-place. The program should perform an in-place reversal of input.
10	ReLU Activation Fuction	Implement a program that performs the Rectified Linear Unit (ReLU) activation function on a vector of 32-bit floating point numbers. The ReLU function sets all negative values to zero and leaves positive values unchanged: $\text{ReLU}(x) = \max(0, x)$.
11	Top-K Selection	Implement a GPU program that, given a 1D array input of 32-bit floating point numbers of length N, selects the k largest elements and writes them in descending order to the output array of length k.
12	Sorting	Write a CUDA program that sorts an array of 32-bit floating-point numbers in ascending order using the bubble sort algorithm. Do not use other algorithms.
13	Matrix Copy	Implement a program that copies an $N \times N$ matrix of 32-bit floating point numbers from input array A to output array B on the GPU. The program should perform a direct element-wise copy so that $B_{i,j} = A_{i,j}$ for all valid indices.
14	Reduction	Write a CUDA program that performs parallel reduction on an array of 32-bit floating point numbers to compute their sum. The program should take an input array and produce a single output value containing the sum of all elements.
15	Dot Product	Implement a CUDA program that computes the dot product of two vectors containing 32-bit floating point numbers. The dot product is the sum of the products of the corresponding elements of two vectors. Mathematically, the dot product of two vectors A and B of length n is defined as: $A \cdot B = \sum_{i=0}^{n-1} A_i \cdot B_i$.
16	Prefix Sum	Write a CUDA program that computes the prefix sum (cumulative sum) of an array of 32-bit floating point numbers. For an input array $[a, b, c, d, \ldots]$, the prefix sum is $[a, a+b, a+b+c, a+b+c+d, \ldots]$.
17	Categorical Cross-Entropy Loss	Implement a CUDA program to calculate the categorical cross-entropy loss for a batch of predictions. Given a matrix of predicted logits Z of size $N \times C$ and a vector of true class labels true_labels of size N, compute the average cross-entropy loss over the batch. The loss for a single sample j with logits $z_j = [z_{j1}, \ldots, z_{jC}]$ and true label y_j is calculated using the numerically stable formula: $\text{Loss}_j = \log\left(\sum_{k=1}^{C} e^{z_{jk}}\right) - z_{j,y_j}$. The final output stored in the loss variable should be the average loss over the N samples: $L = \frac{1}{N}\sum_{j=1}^{N} \text{Loss}_j$. Input: logits, true_labels, N (number of samples), and C (number of classes). Output: loss (a pointer to a single float).
18	Monte Carlo Integration	Implement Monte Carlo integration on a GPU. Given a set of function values $y_i = f(x_i)$ sampled at random points uniformly distributed in the interval $[a, b]$, estimate the definite integral: $\int_a^b f(x)dx \approx (b-a) \cdot \frac{1}{n}\sum_{i=1}^{N} y_i$. The Monte Carlo method approximates the integral by computing the average of the function values and multiplying by the interval width.
19	Histogramming	Write a GPU program that computes the histogram of an array of 32-bit integers. The histogram should count the number of occurrences of each integer value in the range [0, num_bins). You are given an input array input of length N and the number of bins num_bins. The result should be an array of integers of length num_bins, where each element represents the count of occurrences of its corresponding index in the input array.
20	Ordinary Least Squares Regression	Solve the Ordinary Least Squares (OLS) regression problem on a GPU. Given a feature matrix X of size $n\_samples \times n\_features$ and a target vector y of size $n\_samples$, compute the coefficient vector β that minimizes the sum of squared residuals: $\min_\beta \|X\beta - y\|^2$. The closed-form solution to OLS is: $\beta = (X^T X)^{-1} X^T y$.

Task ID	Task	Prompt
21	Activation Layer	Implement a CUDA kernel for an Activation Layer that applies a per-element activation function to its input tensor. The output tensor has the same shape as the input. For this implementation, use the ReLU activation function: $y = \max(0, x)$.
22	Assertion Layer	Write a CUDA program that simulates the behavior of an assertion layer in a neural network. This layer takes a boolean tensor (flattened 1D array) as input. If any element in the input is false (0) at runtime, a runtime error should be raised and the network should terminate.
23	Cast Layer	Write a CUDA program that simulates a cast layer in a neural network. This layer takes a tensor of float32 values as input and casts it to int32 values.
24	Concatenation Layer	Write a CUDA program that simulates the behavior of a concatenation layer in a neural network. The layer receives two input tensors of the same shape on all dimensions except for the concatenation axis. It outputs a new tensor with the concatenation performed along the specified axis.
25	Constant Layer	Write a CUDA program that simulates a constant layer in a neural network. This layer produces a fixed constant tensor of a specified shape and value.
26	Convolution Layer	Implement a CUDA convolution layer that takes as input: (1) A 4D tensor of shape $[B, C_{\text{in}}, H, W]$; (2) A 4D weight tensor of shape $[C_{\text{out}}, C_{\text{in}}, K, K]$; (3) An optional 1D bias tensor of shape $[C_{\text{out}}]$. The kernel performs standard 2D cross-correlation (not flipped convolution), with unit stride and no padding, producing an output of shape $[B, C_{\text{out}}, H - K + 1, W - K + 1]$.
27	Cumulative Layer	Write a CUDA program that implements a cumulative operation layer in a neural network. The operation is an inclusive forward cumulative sum (i.e., prefix-sum) along axis 1 of a 2D float tensor. The output tensor must have the same shape as the input.
28	Deconvolution Layer	Write a CUDA program that implements a 2D deconvolution (transposed convolution) layer in a neural network. The layer takes an input tensor of shape $[N, C_{\text{in}}, H_{\text{in}}, W_{\text{in}}]$ and a weight tensor of shape $[C_{\text{out}}, C_{\text{in}}, K_h, K_w]$, and produces an output tensor of shape $[N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}}]$ using stride=1 and no padding. The operation is the reverse of 2D convolution.
29	Dequantize Layer	Write a CUDA program that implements a dequantization layer. This layer takes an INT8 input tensor, a corresponding float32 scale tensor, and (optionally) a zero-point tensor (default to 0), and performs the operation: output$[i]$ = (input$[i]$ − zeroPt$[i]$) × scale$[i]$. For simplicity, assume per-tensor quantization with scalar scale and zeroPt values.
30	Dynamic Quantize Layer	Implement a CUDA kernel that simulates a dynamic symmetric quantization layer. Given a 1D floating-point input tensor, the kernel calculates the scale factor as: scale $= \frac{\max(\lvert x \rvert)}{127}$. Each element is then quantized using: $q_i = \text{round}(x_i/\text{scale})$ and stored as INT8. Return both the quantized tensor and the computed scale as outputs.
31	Einsum Layer	Write a CUDA program that implements an Einsum layer. This layer performs a tensor operation based on the Einstein summation convention. For this task, assume the Einsum equation is "$ik, kj \rightarrow ij$" (matrix multiplication). The inputs are two 2D float32 matrices: A with shape $[M, K]$ and B with shape $[K, N]$. The output is a matrix of shape $[M, N]$.
32	Elementwise Layer	Write a CUDA kernel that implements an elementwise operation layer supporting broadcasting, as defined in TensorRT. This layer performs a per-element binary operation (addition in this task) on two tensors. If dimensions mismatch, broadcasting is allowed where one dimension is 1.
33	Fill Layer	Implement a CUDA kernel that simulates the behavior of IFillLayer with FillOperation::kLINSPACE. Given: (1) A 1D shape tensor of the form $[N]$; (2) A scalar start value α (float); (3) A scalar step size β (float). Generate a 1D float tensor of size N, where each output element is computed as: $y_i = \alpha + i \times \beta, \quad i = 0, \ldots, N-1$. Note: β is the step size. This corresponds to IFillLayer in TensorRT when β is provided as a rank-1 vector (not a scalar), representing linspace with constant stride.

34	Gather Layer	Implement a CUDA program that simulates the functionality of a TensorRT Gather layer in GatherMode::kDEFAULT. The CUDA kernel takes as input a 2D data tensor of shape $[N, C]$ and a 1D indices tensor of length N, and outputs a 1D gathered tensor, where each output$[i]$ = data$[i,$ indices$[i]]$.
35	Grid Sample Layer	Implement a CUDA kernel that performs 2D grid sampling with bilinear interpolation and clamping, simulating the GridSample layer (SampleMode::kCLAMP). The input is a 2D float tensor of shape (H, W) (flattened to 1D in row-major order). The grid is a 3D float tensor of shape $(H_{\text{out}}, W_{\text{out}}, 2)$, where each grid point contains normalized coordinates $(x, y) \in [-1, 1]$, also flattened to 1D. The output is a 2D float tensor of shape $(H_{\text{out}}, W_{\text{out}})$, stored as 1D. For each output pixel (h, w), compute the sampled value from the input using bilinear interpolation with clamping behavior (i.e., if the interpolated coordinate is outside the input boundary, use the closest valid pixel value). The interpolation formula is: (1) Convert normalized coordinates to float pixel positions: $f_x = ((x + 1) * W - 1)/2$, $f_y = ((y+1) * H - 1)/2$; (2) Find the surrounding four pixels, apply clamping to (x_0, x_1), (y_0, y_1); (3) Use standard bilinear interpolation to compute the final value.
36	Identity Layer	Implement a CUDA kernel for an Identity layer that simply copies input tensor values to the output tensor. This identity operation preserves values but may optionally allow layout or data type conversions (which can be ignored in this basic test).
37	Conditional Input Layer	Implement a CUDA kernel simulating a simplified conditional-sum operation in Conditional Input Layer. The kernel takes a 1D input tensor x of length N and a boolean condition tensor cond of the same length. For each element $x[i]$, if cond$[i]$ is true, then $x[i]$ is routed to the "then" branch; otherwise, to the "else" branch. The kernel should compute the sum of all values routed to the then branch and the else branch separately. The final outputs are two scalar values: then_sum and else_sum.
38	LRN Layer	Implement a CUDA kernel for the Local Response Normalization (LRN) layer used in deep learning. The layer takes a 2D input tensor of shape (N, C), where C is the channel dimension. For each element, the output is computed using the formula: $y_i = \frac{x_i}{\left(1+\frac{\alpha}{n}\sum_{j=\max(0,i-n/2)}^{\min(C-1,i+n/2)} x_j^2\right)^{\beta}}$. This should be implemented with the normalization applied across channels, using the default parameters: $\alpha = 1e-4$, $\beta = 0.75$, and $n = 5$.
39	Conditional Output Layer	Implement a CUDA kernel for the ConditionalOutput layer (IIfConditionalOutputLayer). This layer takes two floating-point input tensors: true_output and false_output, and a boolean flag array of the same shape. It outputs true_output$[i]$ if flag$[i]$ is true, otherwise false_output$[i]$. Inputs: (1) flag: $[N]$ boolean mask (0 or 1); (2) true_output: $[N]$ float array; (3) false_output: $[N]$ float array. Output: output: $[N]$ float array chosen based on the flag.
40	Matrix Multiply Layer	Write a CUDA program that implements a matrix multiplication layer. The layer accepts two float32 input tensors A and B, which may be 2D matrices or vectors. The operation computes the matrix product $A@B$ following broadcasting rules: if one operand has a singleton batch dimension, it is broadcast to match the other.
41	NMS Layer	Implement a CUDA kernel for Non-Maximum Suppression (NMS) that operates per batch and per class. The input includes bounding boxes of shape [batchSize, numBoxes, 4] and confidence scores of shape [batchSize, numBoxes]. For each batch, select up to MaxOutputBoxesPerClass boxes with confidence scores above ScoreThreshold and mutually IoU $\leq$ IoUThreshold. The selected indices should be stored as (batchIndex, boxIndex) pairs, sorted by batch index and descending score. You can follow these five structured steps in the CUDA kernel: (1) For each batch independently, iterate through numBoxes and collect boxes where score $\geq$ ScoreThreshold; (2) Sort the valid box indices by their corresponding scores in descending order; (3) Apply NMS suppression; (4) Select top-k boxes; (5) Pad unused output with -1.
42	NonZero Layer	Implement a CUDA program for a NonZero layer. This layer takes a 2D input tensor and returns the indices of all non-zero elements, following ONNX NonZero semantics. The output is a $2 \times N$ matrix of int32 values where N is the number of non-zero elements. Each column represents a (row, column) coordinate, and the columns must be lexicographically ordered. Input data types may include float32 or int32.

43	Normalization Layer	Implement a CUDA kernel for a normalization layer that applies the following transformation to the input tensor: $Y = \frac{X - \text{Mean}(X, \text{axes})}{\sqrt{\text{Variance}(X) + \epsilon}} \cdot S + B$, where Mean and Variance are computed over specified axes. This implementation assumes normalization is done along the last dimension of a 2D input (i.e., row-wise normalization). The scale S and bias B tensors are broadcasted along the rows.
44	OneHot Layer	Write a CUDA kernel and testing framework to implement the behavior of a OneHot layer, as defined in TensorRT. This kernel should support the case where axis = -1, i.e., the one-hot dimension is appended as the last axis. The kernel takes three inputs: (1) Indices – an int32 tensor of arbitrary shape; (2) Values – a 1D tensor of shape [off_value, on_value], type float32; (3) Depth – a scalar int32, indicating the number of classes. The output is a tensor of shape Indices.shape + [Depth]. Each output element is set to on_value at the index specified in Indices, and off_value elsewhere.
45	Padding Layer	Implement a CUDA kernel for a Padding layer that applies zero-padding only to the last two dimensions of a 4D input tensor. The padding values can be positive (for padding) or negative (for cropping). The input tensor is in NCHW layout. The output tensor shape is adjusted according to the specified padding amounts for the height and width dimensions.
46	Parametric ReLU Layer	Implement a CUDA kernel for a Parametric ReLU (PReLU) layer. The input is a 2D tensor of shape $[N, C]$, and each channel has a corresponding learned slope. For positive inputs, the output equals the input; for negative inputs, the output is the input multiplied by the corresponding slope. The slope tensor is a build-time constant with shape $[C]$.
47	PluginV2 Layer	Write a CUDA program to implement a custom plugin layer for TensorRT, which computes the element-wise square of a 2D float tensor. The CUDA kernel should take a $[N, C]$ float input tensor and produce a $[N, C]$ float output tensor, where each element is computed as $y = x * x$.
48	PluginV3 Layer	Write a CUDA program simulating a V3 plugin layer behavior. The plugin performs an element-wise cube ($y = x^3$) operation on a 2D float input tensor of shape $[N, C]$, and produces a tensor of the same shape as output. Implement the CUDA kernel, memory management, and correctness testing.
49	Pooling Layer	Implement a CUDA kernel that simulates a basic Max Pooling layer in a neural network. The kernel should apply a max reduction operation over a fixed-size window along the last dimension of a 2D input tensor. For each test case, input is a flattened 2D tensor of shape $[B, W]$ and output is the pooled tensor of shape $[B, W' = W//\text{pool\_size}]$, where pool_size is fixed. The kernel should work with float32 data.
50	Quantize Layer	Implement a CUDA kernel that simulates the output of a TensorRT Quantize layer followed by implicit dequantization. The kernel should quantize a floating-point input tensor using the formula: output $= \text{clamp}(\text{round}(\text{input}/\text{scale})) \times \text{scale}$, where scale is a positive scalar (per-tensor quantization). The rounding method should be round-to-nearest ties-to-even. Clamp the intermediate quantized value to the range $[-128, 127]$ to simulate int8 symmetric quantization. The kernel should support 2D float input and produce float32 output.
51	Ragged Softmax Layer	Implement a CUDA kernel for a RaggedSoftmax layer. The input is a 2D float tensor of shape $Z \times S$, and a 1D bounds tensor of shape $Z \times 1$ specifying the valid length of each of the Z sequences (i.e., number of elements to apply softmax on per row). For each row i, apply softmax only on the first bounds$[i]$ values of that row. The remaining entries in the row can be left as zero. This simulates ragged or variable-length softmax computation across sequences. The output tensor should have the same shape as input.
52	Reduce Layer	This task is about implementing a reduction layer in a neural network. The reduction layer takes a tensor of arbitrary shape and performs a specified reduction operation (e.g., sum, mean, max) along the specified axis or across the entire tensor. The output tensor will have the same shape as the input tensor, except for the reduced dimensions. In this particular implementation, the operation is to sum all elements of the input tensor along the second dimension (i.e., summing each row), and the results are stored in the output tensor. Input Tensor Shape: $[Z, M]$ where Z is the number of samples (batch size) and M is the number of features (columns); Output Tensor Shape: $[Z]$, as we perform reduction along the second dimension (i.e., sum the elements in each row).

53	Resize Layer	The Resize Layer is responsible for resizing multi-dimensional tensors, supporting kNEAREST interpolation mode. It resizes the input tensor's last m dimensions, where $m \leq \min(8, N)$ and $N > 0$. Key Functionalities: Nearest Neighbor Resizing: The kNEAREST mode resizes the tensor by mapping the output coordinates to the nearest input tensor coordinates. The kernel performs resizing with proper coordinate mapping and interpolation. The task is to implement the resizing operation in CUDA, perform the resizing with nearest neighbor interpolation, and compare the result with a CPU-generated reference.
54	Reverse Sequence Layer	Implement a CUDA kernel for the ReverseSequence layer. Given a 2D input tensor of shape [batch_size, sequence_length] and a 1D tensor sequenceLens of size [batch_size], reverse the first sequenceLens[i] elements along the sequence dimension for each batch i, and leave the remaining elements unchanged. The output tensor must match the shape of the input. This implementation assumes batchAxis = 0 and sequenceAxis = 1.
55	Scale Layer	The task is to implement a CUDA kernel for the Scale layer in a neural network definition. This layer performs a per-element computation: output $=$ (input $\times$ scale $+$ shift)$^{\text{power}}$. The coefficients for scale, shift, and power can be applied on a per-tensor, per-channel, or per-element basis. If no weights are provided, the default values are used for shift (0), power (1), and scale (1). The output tensor has the same shape as the input tensor. This layer can be used for operations such as INT8 quantization when combined with specific data types (FP32 input and INT8 output).
56	Scatter Layer	Write a CUDA program to implement a Scatter layer in ScatterMode::kELEMENT. Each input tensor is 4D in NCHW format, and scatter is performed along axis 2 (the H dimension) only. The kernel takes three inputs: (1) data: a float32 tensor $[N, C, H, W]$; (2) indices: an int32 tensor $[N, C, H, W]$ specifying where to scatter; (3) updates: a float32 tensor $[N, C, H, W]$. Each value updates $[n, c, h, w]$ is written to output $[n, c, \text{indices}[n, c, h, w], w]$. All indices values are guaranteed to be in $[0, H)$. The output tensor is initialized as a copy of data, and then modified by the scatter operation.
57	Select Layer	Implement a CUDA kernel that performs element-wise selection from two input tensors x and y, based on a condition tensor cond. For each index i, the output is: output[i] = cond[i] != 0 ? $x[i]$: $y[i]$. All three tensors (cond, x, y) have the same 1D shape.
58	Shape Layer	Write a CUDA kernel that implements the functionality of a "Shape Layer". The kernel should accept an input tensor of arbitrary dimensions (minimum rank 1) and output a 1D tensor containing the dimensions of the input tensor. For example, if the input shape is $[2, 3, 5, 7]$, the output tensor should be $[2, 3, 5, 7]$ of type int64. This mimics the behavior of TensorRT's Shape Layer.
59	Shuffle Layer	Implement a CUDA kernel for a Shuffle Layer that applies a fixed sequence of operations to a 4D input tensor X of shape $[N, C, H, W]$. The operations are: (1) First transpose: permute axes to $[C, N, H, W]$; (2) Reshape: flatten into $[C, N * H * W]$; (3) Second transpose: permute to $[N * H * W, C]$. The input is stored in row-major format. The final output is also row-major.
60	Slice Layer	Implement a CUDA kernel that performs static slicing on a 2D input tensor using the specified 1D arrays start, size, and stride. The slicing is applied independently across each axis without specifying axes (i.e., operates on the full rank of the input tensor). The input is a matrix of shape $[H, W]$, and the output is obtained by applying slicing rules: output$[i][j]$ = input[start[0] + $i \cdot$ stride[0]][start[1] + $j \cdot$ stride[1]]. Validate correctness by comparing output to a precomputed reference.
61	SoftMax Layer	Implement a CUDA kernel that performs a per-channel Softmax operation over a 2D input tensor with shape $[N, C]$, where C is the softmax axis (channel). The output tensor must retain the same shape.
62	Squeeze Layer	Implement a CUDA kernel simulating a squeeze operation along a known unit axis. The input is a 2D tensor of shape $[N, 1]$, and the squeeze removes the singleton dimension to produce a 1D tensor of shape $[N]$.
63	TopK Layer	Implement a CUDA program that performs TopK reduction on a 2D input tensor. For each row, the kernel finds the top-K maximum values and writes them to the output tensor. This implementation assumes static K, which is fixed and passed as a macro definition. The output should contain the top-K values for each row in descending order.

64	UnaryOp Operation Layer	Implement a CUDA program to simulate a UnaryOp layer in a neural network, specifically for the EXP operation. The EXP operation computes the exponential function element-wise, defined as: $y_i = \exp(x_i)$, where x_i is the i-th element of the input tensor, and y_i is the corresponding output. The layer applies the exponential function to each element of a 1D input tensor and returns the result as the output tensor. The CUDA kernel should handle large input sizes efficiently.
65	Unsqueeze Layer	Implement a CUDA kernel simulating the behavior of an Unsqueeze operation. The operation inserts a unit dimension at a specified axis of the input tensor's shape. In this implementation, we assume the input is a 2D tensor of shape $[N, D]$, and we insert a new axis at position 1, producing an output tensor of shape $[N, 1, D]$. The actual memory layout of the data remains the same, but we simulate this transformation by copying input to output buffer shaped accordingly.
66	Loop Trip Limit Layer	Implement a CUDA kernel simulating the behavior of a LoopTripLimit layer with TripLimit::kCOUNT mode. Each thread performs a fixed number of iterations as specified by a scalar INT32 input. In each iteration, the thread increments a value by 1. The final output is an array where each element equals the trip limit (i.e., the total number of iterations).
67	Loop Recurrence Layer	Implement a CUDA kernel to simulate the behavior of a Loop Recurrence Layer as in TensorRT. The layer has two inputs: an initial value tensor and an update (step) value tensor. The kernel computes the result of a loop-like recurrence defined as: $\text{out}[i] = \text{init}[i] + \text{trip\_count} \times \text{delta}[i]$. That is, starting from an initial value init, the output after a fixed number of iterations trip_count is computed by adding delta in each iteration. The goal is to validate this recurrence over a batch of N samples.
68	Loop Iterator Layer	Implement a CUDA kernel that simulates the behavior of a Loop Iterator Layer which iterates over axis 0 of a 2D input tensor. Each iteration produces a slice (i.e., a row of the matrix). In this implementation, we assume the input is a matrix of shape (N, D), and we extract all rows in order into a contiguous output buffer of shape (N, D). The kernel copies each row individually into the output, simulating a loop's behavior with iterator over axis 0.
69	Loop Output Layer	Implement a CUDA kernel simulating the LoopOutput layer in kCONCATENATE mode along axis 0. Each iteration outputs a 1D tensor of fixed size D, and the number of iterations is K. The final result is a 2D tensor of shape $[K, D]$, where each row corresponds to the output of one iteration. The output tensor is laid out in row-major format.
70	Plugin Layer	Implement a CUDA kernel for a custom Plugin Layer. The plugin takes a 2D float32 input tensor of shape $[N, C]$, and for each row, it computes the L2 norm (i.e., the square root of the sum of squares across the C features).
71	Condition Layer	Implement a CUDA kernel that simulates the behavior of a TensorRT IIfConditional layer. This layer takes a scalar boolean predicate and two candidate value vectors: the "then" and "else" branches. If the scalar predicate is true (non-zero), then the output is equal to the then_branch vector; otherwise, the output is equal to the else_branch vector. The kernel must perform this conditional selection efficiently using a single branch evaluation across the entire array.