

Assignment 1

The Wizard's Lens

An Image Preprocessing Pipeline on the GPU

Released: **2 August 2026** • Due: **23 August 2026, 23:59 IST** • Weight: **10 marks**

1 Prologue: Every Spell Has a Lens

A person points a phone at a leaf and asks, “what plant is this?” Two seconds later the answer comes back, complete with a note about how much water it wants. To them it is magic – a small glass rectangle that simply knows.

You are reading this because you are on the other side of the curtain.

Down here, there is no magic. There is a photograph: two million little triples of numbers, wildly varying in size, brightness and shape, and a neural network that will refuse to look at any of them. The network is a fussy creature. It wants exactly one size. It wants numbers hovering politely around zero, not sprawling from 0 to 255. It wants them arranged just so.

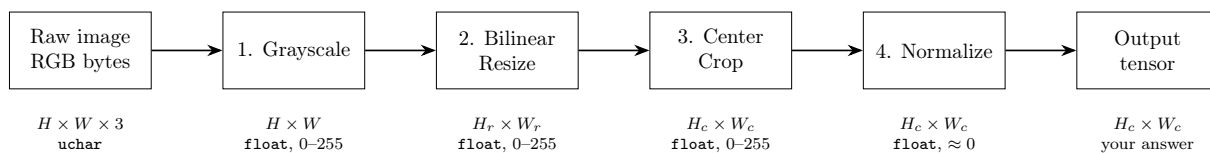
*Between the photograph and the network sits an unglamorous, unloved, absolutely essential piece of machinery: the **preprocessing pipeline**. It is the lens the spell is cast through. Every image that has ever been fed to a vision model, every one of the fourteen million in ImageNet and every frame a self-driving car has ever second-guessed, went through some version of it first. And it must be fast, because a training run that stalls waiting for resized images is a very expensive way to heat a room.*

Here is the thing about that lens: grinding it is embarrassingly parallel. Every output pixel can be computed without asking any other output pixel for permission. A CPU grinds them one at a time, politely, in a queue. A GPU throws ten thousand threads at the problem at once and is done before the CPU has finished clearing its throat.

In this assignment you will grind that lens yourself, in CUDA, from nothing. Four kernels. When you are finished you will have written, by hand, in a few hundred lines, the exact transformation that runs before every image classification you have ever seen. The magic will not survive the experience. Something better will replace it: you will know how it works.

2 The Pipeline You Will Build

You will implement a four-stage pipeline. It takes a raw RGB image and produces the normalised, fixed-size, single-channel tensor that a neural network expects. Each stage is one CUDA kernel. The stages run strictly in order, and the output of one is the input to the next.



This is, almost exactly, what `torchvision.transforms` does for a standard ImageNet model. The only liberty we take is doing the colour conversion first, which makes the arithmetic in the later stages cheaper i.e. one channel instead of three.

An image is a grid of **pixels**, each one a tiny square of colour. Every pixel stores three whole numbers in $[0, 255]$ i.e. how much **R**ed, **G**reen and **B**lue light it holds, so an $H \times W$ photograph is $H \cdot W \cdot 3$ bytes in total. A pixel of $(0, 0, 0)$ is black, $(255, 255, 255)$ is white, and $(250, 225, 80)$ is a warm yellow.

3 The Four Stages

Images are stored **row-major**: the element at row y , column x of an $H \times W$ image lives at flat index $y \cdot W + x$. The raw RGB input is **channel-interleaved**, so the three bytes of pixel (y, x) start at flat index $3(y \cdot W + x)$.

3.1 Grayscale

The ITU-R BT.601 luma weights account for the eye being far more sensitive to green than to blue:

$$\text{gray}(y, x) = 0.299 R(y, x) + 0.587 G(y, x) + 0.114 B(y, x)$$

The weights sum to 1, so the result stays on the 0–255 scale. **Do not round it** - keep it as a float.

- **Input:** `d_rgb`, $H \times W \times 3$ bytes.
- **Output:** `d_gray`, $H \times W$ floats.
- One thread per output pixel; $H \cdot W$ threads. A 1-D grid fits.

3.2 Bilinear Resize

Nearest-neighbour resizing produces blocky results. Bilinear interpolation reads the four source pixels surrounding the point we want and blends them in proportion to how close each one is.

We use the *align-corners* convention, which pins the four corner pixels of the input exactly onto the four corner pixels of the output:

$$s_y = \begin{cases} \frac{H-1}{H_r-1} & \text{if } H_r > 1 \\ 0 & \text{if } H_r = 1 \end{cases} \quad s_x = \begin{cases} \frac{W-1}{W_r-1} & \text{if } W_r > 1 \\ 0 & \text{if } W_r = 1 \end{cases}$$

For each output pixel (o_y, o_x) :

$$\begin{aligned}
 f_y &= o_y \cdot s_y & f_x &= o_x \cdot s_x \\
 y_0 &= \lfloor f_y \rfloor & x_0 &= \lfloor f_x \rfloor \\
 y_1 &= \min(y_0 + 1, H - 1) & x_1 &= \min(x_0 + 1, W - 1) \\
 w_y &= f_y - y_0 & w_x &= f_x - x_0
 \end{aligned}$$

Then blend horizontally along the two rows, and vertically between the results:

$$\begin{aligned}
 \text{top} &= \text{in}(y_0, x_0) (1 - w_x) + \text{in}(y_0, x_1) w_x \\
 \text{bottom} &= \text{in}(y_1, x_0) (1 - w_x) + \text{in}(y_1, x_1) w_x \\
 \text{out}(o_y, o_x) &= \text{top} (1 - w_y) + \text{bottom} w_y
 \end{aligned}$$

The min in y_1 and x_1 clamps you inside the image at the bottom and right edges, where y_0 can already be the last row.

- **Input:** `d_gray`, $H \times W$ floats.
- **Output:** `d_resized`, $H_r \times W_r$ floats.
- Two-dimensional: use a 2-D grid of 2-D blocks.

Which index maps to which dimension?

Map `threadIdx.x` to the **column** (x) and `threadIdx.y` to the **row** (y). Threads within a warp have consecutive `threadIdx.x`, so this makes them access consecutive memory addresses, which the hardware merges into a few wide transactions. Swapping them is still correct, just several times slower.

3.3 Center Crop

$$\text{off}_y = \left\lfloor \frac{H_r - H_c}{2} \right\rfloor \quad \text{off}_x = \left\lfloor \frac{W_r - W_c}{2} \right\rfloor \quad \text{out}(y, x) = \text{in}(y + \text{off}_y, x + \text{off}_x)$$

Those are **integer** divisions, rounding down. When $H_r - H_c$ is odd the crop sits one pixel off-centre towards the top-left; that is intended.

- **Input:** `d_resized`, $H_r \times W_r$ floats.
- **Output:** `d_cropped`, $H_c \times W_c$ floats.
- Note that the input row stride is W_r but the output row stride is W_c .

3.4 Normalize

$$\text{out}(y, x) = \frac{\text{in}(y, x) - \mu}{\sigma}$$

μ and σ are given in the input file.

- **Input:** `d_cropped`, $H_c \times W_c$ floats.
- **Output:** `d_out`, $H_c \times W_c$ floats.
- Element-wise; a 1-D grid again.

4 Input and Output Format

Input

Your program reads one test case from **standard input**. It also accepts a filename as `argv[1]`, which is convenient while debugging. The parser is already written for you.

Line	Contents
1	H W — height and width of the input image
2	H_r W_r — target height and width after resizing
3	H_c W_c — target height and width after cropping
4	<code>mean std</code> — the normalisation constants μ and σ
5 onwards	$H \cdot W \cdot 3$ integers in $[0, 255]$: the pixel data, row-major, channel-interleaved as <i>R G B R G B ...</i>

Output

Write to **standard output**, and nothing else:

- Line 1: H_c W_c , separated by a single space.
- Then H_c lines, each with W_c values printed with exactly six decimal places (`%.6f`), separated by single spaces.

The printing code is supplied. Debugging messages belong on `stderr`, which the grader ignores.

How your output is compared.

The grader does not do a byte-for-byte `diff`. It parses your numbers and accepts any value satisfying $|a - e| \leq 10^{-3} + 10^{-4}|e|$, where e is the expected value. A different-but-equivalent order of arithmetic operations will still pass; a wrong formula will not.

5 Worked Example

This is public test case 01. A 4×4 image is resized to 3×3 , cropped to 2×2 , and normalised with $\mu = \sigma = 0.5$.

Input (`testcases/public/input_01.txt`)

```
4 4
3 3
2 2
0.500000 0.500000
0 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44 45 46 47
```

The pixel values are a ramp: pixel (y, x) has $R = 12y + 3x$, $G = R + 1$, $B = R + 2$.

Stage 1 — Grayscale (4×4)

Because the weights sum to 1, the luma of pixel (y, x) is $0.299R + 0.587(R+1) + 0.114(R+2) = R + 0.815$:

0.8150	3.8150	6.8150	9.8150
12.8150	15.8150	18.8150	21.8150
24.8150	27.8150	30.8150	33.8150
36.8150	39.8150	42.8150	45.8150

Stage 2 — Bilinear resize to 3×3

Here $s_y = s_x = \frac{4-1}{3-1} = 1.5$, so output row 1 samples source row 1.5, exactly halfway between rows 1 and 2, with $w_y = 0.5$. This image is a linear ramp, so the interpolation reproduces the ramp exactly — convenient for checking by hand.

0.8150	5.3150	9.8150
18.8150	23.3150	27.8150
36.8150	41.3150	45.8150

Stage 3 — Center crop to 2×2

$\text{off}_y = \lfloor (3-2)/2 \rfloor = 0$ and likewise $\text{off}_x = 0$, so we keep the top-left 2×2 block:

0.8150	5.3150
18.8150	23.3150

Stage 4 — Normalize with $\mu = \sigma = 0.5$

$\text{out} = (v/255 - 0.5)/0.5$. For the first element: $(0.815/255 - 0.5)/0.5 = (0.003196 - 0.5)/0.5 = -0.993608$.

Expected output (testcases/public/expected_01.txt)

2 2
-0.993608 -0.958314
-0.852431 -0.817137

6 Constraints

$1 \leq H, W \leq 1024$	input image dimensions
$1 \leq H_r \leq 1024, 1 \leq W_r \leq 1024$	resize target
$1 \leq H_c \leq H_r, 1 \leq W_c \leq W_r$	crop target
$0 \leq R, G, B \leq 255$	pixel values, integers
$0.0 \leq \mu \leq 1.0$	normalisation mean
$0.01 \leq \sigma \leq 2.0$	normalisation standard deviation

Upscaling is allowed: H_r may be larger than H .

7 Instructions

- Everything lives in a single file, `pipeline.cu`. It compiles and runs before you have written a line; it just produces zeros.
- **Do not modify anything outside the TODO markers.** The parser, the printer and the kernel signatures are checked automatically during grading.
- Passing the public tests is necessary, not sufficient.
- Refer `README.md` for building and testing.
- Sequential codes will not fetch marks.
- Your submission is graded on the five public test cases supplied with the assignment and on a set of hidden test cases you will not see until after the deadline.
- Your submission will be evaluated automatically on a standalone server. Therefore, if the code contains non-compilable text such as `%%writefile`, the corresponding submission will receive 0 marks.
- If you have any general questions, please post on Moodle.
- Write your own code. You are welcome to discuss the ideas with your classmates, but the code you submit must be yours, written by you.

8 Submission Guidelines

1. Rename your file to your roll number: `<RollNumber>.cu` - for example `CS24S009.cu`. Capitalisation matters.
2. Verify it one last time, exactly as the grader will:

```
$ ./run_tests.sh -s CS24S009.cu
```

3. Create a folder named with your roll number (e.g., `CS24S009`), place the `<RollNumber>.cu` file inside it, and compress the folder into `<RollNumber>.zip`.
4. Submit **only the `<RollNumber>.zip` file**. Do not include any additional files, reports, test outputs, or nested archives.
5. Upload to the course portal before **23 August 2026, 23:59 IST**.

9 Practice Problems (no marks, just curiosity)

1. Your pipeline makes four round trips to global memory. Could all four stages be one kernel, where each thread computes its own source coordinates and does everything in registers? How much faster would you expect that to be, and why?
2. Time your kernels with `cudaEvent`. Now time the `fscanf` loop that reads the input file. Which one dominates? What does that suggest about where real pipelines spend their effort?
3. Real preprocessing runs on batches of thousands of images. What would you change to process 1000 images per launch instead of one?
4. Try to implement other image processing routines like Gaussian blur, alpha blend two images, flip the image horizontal/vertical/transpose, etc.