Parallel Image Processing with Convolution Filters

Team Members: Hongcheng Tian (ht2657)

Description of the Convolution Filters

The main idea is to modify or enhance images by applying a kernel (also known as a filter) to an image. A kernel is basically a small matrix (like 3x3 or 5x5) that we slide over the image. At each position, we perform a mathematical operation called convolution.

How it works:

1. Kernel Application:
    - For every pixel in the image, we align the center of the kernel with that pixel.
    - We multiply each value in the kernel with the corresponding pixel value in the image.
    - Sum up all these multiplications to get a new pixel value.
    - Replace the original pixel with this new value in the output image.
2. Different Filters for Different Effects:
    - Blurring: Using a kernel where all values are equal and sum up to 1, we can average out the pixel values to create a blur effect.
    - Sharpening: A kernel that emphasizes the center pixel more than its neighbors can enhance edges and details.
    - Edge Detection: Kernels like the Sobel operator can highlight areas with significant intensity changes.
3. Why It's Suitable for Parallelization:
    - Each pixel's new value is calculated independently of others (except for overlapping kernels), which means we can process multiple pixels at the same time.
    - This independence makes it ideal for using parallel computing techniques to speed up the processing.

Broad plan of parallelization

Repa seems to be the best fit for parallelization at the moment. It is specifically optimized for array-based operations, which is exactly what image processing with convolution filters involves. It automatically handles the parallel execution of computations with lower overhead.

1. Prepare the array.
    - Use libraries like JuicyPixels to read the image file.
    - Convert the image into a Repa array for processing.
    - Start with grayscale images and then try color images (RGB). Array U DIM2 Double for grayscale images and Array U DIM3 Double for RGB images.
2. Define the convolution function.
    - Implement convolution by mapping over the array.
    - Store the convolution kernel as a Repa array as well, which makes the element-wise multiplication straightforward.
    - For certain filters like Gaussian blur, implement separable convolution to reduce computational complexity by breaking down a 2D kernel into two 1D kernels.
3. Handle boundary conditions.
    - Decide how to deal with edges (e.g., zero-padding).
    - Create a helper function to handle out-of-bounds indices, or
    - Implement logic within the convolution function to manage indices outside the image bounds.
4. Test and analyze.
    - Run the computation in parallel. Use computeP for parallel computation, which Repa provides out of the box.
    - Analyze the Results. Use benchmarking tools to measure performance or manually compute the speed up by comparing the execution times of the sequential (computeS) and parallel (computeP) versions.
    - Experiment with different ways of splitting the data to achieve better load balancing across threads.

Plan for Getting Input Data

I'll use standard image datasets like the Berkeley Segmentation Dataset or images from sites like Pexels that offer free high-resolution photos. These images will help test how well the filters work on different types of images (landscapes, portraits, etc.).