Compression Concept (Easy Explanation)

This program is all about **compressing an image** by **downsampling** it using a GPU (CUDA).

- **Downsampling** means reducing the image size by averaging pixel values in blocks (like 2x2 or 4x4).
- **Compression** here simply reduces the resolution of the image without significant loss of information, which is useful when we want to make the image smaller or use less memory.

How Does the Compression Work?

- Suppose you start with a large image (like a 1024x1024 pixel image). If you want to compress it with a factor of 2, you will take blocks of 2x2 pixels and replace them with a single pixel that is the **average** of the 4 pixels in that block.
- Similarly, a factor of 4 would use 4x4 blocks to reduce the image even more.

Example:

Imagine you have a 4x4 image:

```
 10 20 30 40
50 60 70 80
90 100 110 120
130 140 150 160
```

-

When you apply a 2x2 downsampling (compression factor = 2), it will be reduced to:

```
 (10+20+50+60) / 4 = 35   (30+40+70+80) / 4 = 55
(90+100+130+140) / 4 = 115   (110+120+150+160) / 4 = 135
```

Result:

```
 35  55
115 135
```

-

The **CUDA kernel** does this downsampling computation on the GPU, making it **faster**.

Line-by-line Explanation of the Code

Header Files

```
#include <opencv2/opencv.hpp>
#include <cuda_runtime.h>
#include <iostream>
```

```
#include <ctime>
```

- These are the libraries you need:
  - `opencv2/opencv.hpp` for image processing functions (like loading and saving images).
  - `cuda_runtime.h` is required to use CUDA (to run GPU code).
  - `iostream` for printing messages.
  - `ctime` for measuring the time it takes to run the compression.

Defining Image Dimensions

```
#define WIDTH 1024
#define HEIGHT 1024
```

- Here, you define the width and height of the input image. In this case, you assume the image will be 1024x1024 pixels.

CUDA Kernel for Downsampling

```
__global__ void downsampleKernel(const uchar* input, uchar* output, int w, int factor) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < w / factor && y < w / factor) {
        int sum = 0;
        for (int i = 0; i < factor; i++)
            for (int j = 0; j < factor; j++)
                sum += input[(y * factor + j) * w + (x * factor + i)];

        output[y * (w / factor) + x] = sum / (factor * factor);
    }
}
```

- This is the **CUDA kernel** that runs on the GPU.
- It takes the `input` image, the `output` image, the width of the image `w`, and the **downsampling factor** (`factor`).
- The `x` and `y` variables calculate the position of each thread on the grid of the image.

- For each thread (each pixel in the downsampled image), it computes the **average of a block** of pixels from the original image (based on the `factor`).
- The result is stored in the corresponding position in the `output` image.

Compression Function

```
void compressImage(const cv::Mat& inputImg, cv::Mat& outputImg, int factor) {
    int newSize = WIDTH / factor;
    outputImg.create(newSize, newSize, CV_8UC1);

    uchar *d_input, *d_output;
    cudaMalloc(&d_input, WIDTH * HEIGHT);
    cudaMalloc(&d_output, newSize * newSize);
    cudaMemcpy(d_input, inputImg.data, WIDTH * HEIGHT, cudaMemcpyHostToDevice);

    // Define grid and block size
    dim3 blockSize(16, 16);
    dim3 gridSize((newSize + 15) / 16, (newSize + 15) / 16);

    // Launch the kernel
    downsampleKernel<<<gridSize, blockSize>>>(d_input, d_output, WIDTH, factor);
    cudaMemcpy(outputImg.data, d_output, newSize * newSize, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_input);
    cudaFree(d_output);
}
```

- **`compressImage`** is the function that handles image compression.
- `newSize` is the new size of the downsampled image based on the factor.
- `outputImg.create(newSize, newSize, CV_8UC1)` creates the new output image (a grayscale image).
- **CUDA Memory Allocation**:
    - `cudaMalloc` allocates memory on the GPU for both the input and output images.
    - `cudaMemcpy` copies the input image from the CPU to the GPU.
- **Grid and Block Setup**:
    - `blockSize(16, 16)` means each block contains 16x16 threads.

    - `gridSize` ensures that enough blocks are created to cover the entire downsampled image.

- The **CUDA kernel** is launched (`downsampleKernel<<<gridSize, blockSize>>>(d_input, d_output, WIDTH, factor)`), and the downsampled image is computed on the GPU.

- Finally, `cudaMemcpy` copies the result from the GPU back to the CPU (to `outputImg`).

- **Memory Cleanup**: `cudaFree` frees up the memory allocated on the GPU.

Main Function

```
int main() {
    // Load input image
    cv::Mat inputImg = cv::imread("Sandesh.jpeg", cv::IMREAD_GRAYSCALE);
    if (inputImg.empty()) {
        std::cerr << "Error loading image!" << std::endl;
        return -1;
    }

    // Resize input image to fixed width and height
    cv::resize(inputImg, inputImg, cv::Size(WIDTH, HEIGHT));

    // Start measuring time
    clock_t start = clock();

    // Compress the image with different factors
    cv::Mat outputImg2, outputImg4;
    compressImage(inputImg, outputImg2, 2);
    compressImage(inputImg, outputImg4, 4);

    // Save the compressed images
    cv::imwrite("compressed_2x_GPU.jpg", outputImg2);
    cv::imwrite("compressed_4x_GPU.jpg", outputImg4);

    // End measuring time
    clock_t end = clock();

    // Calculate and print the elapsed time
    double elapsed_time = double(end - start) / CLOCKS_PER_SEC;
    std::cout << "Compression complete! Time taken: " << elapsed_time << " seconds" << std::endl;

    return 0;
}
```

- The `main` function does the following:

- Loads an image (`cv::imread`), and converts it to grayscale.
- Resizes the image to the defined `WIDTH` and `HEIGHT`.
- Starts measuring time with `clock()`.
- Calls `compressImage` twice, once for each compression factor (2x and 4x).
- Saves the compressed images using `cv::imwrite`.
- Measures the time taken to compress and prints the result.

Summary

- **CUDA** is used to offload the image compression (downsampling) to the **GPU** for faster processing.
- The image is downsampled by averaging pixels in blocks.
- The program takes an image, compresses it using CUDA, and saves the compressed versions.

Let me know if you need further clarification!