A Presentation by
Leslie TIENTCHEU and Haja RANDRIAMAMONJISOA
Class: TD 40 int

Image Processing Using CUDA

Table of Content

1. Introduction:

This C++ code implements a Gaussian blur operation on a grayscale image using both CPU and GPU (CUDA) implementations. The code reads an input image, applies Gaussian blur using OpenCV's CPU implementation, and compares the performance with a custom CUDA implementation on the GPU.

1.1. Libraries

Libraries in C++ provide pre-written functionality that can be used in a program. In this code, several libraries are included to leverage existing functions and tools for image processing, GPU programming, and timing measurements.

Explanation:

- <iostream>: Input/output stream library for basic I/O operations.
- <chrono>: Library for measuring time durations.
- <opencv2/opencv.hpp>: OpenCV library for computer vision and image processing.
- <cuda_runtime.h>: CUDA runtime library for GPU programming.
- <cmath>: Standard C++ library for mathematical operations.

These libraries are essential for the functionality of the code, providing tools for image loading, processing, GPU programming, and timing measurements. They set the foundation for the subsequent code implementation.

```
1 #include <iostream>
2 #include <chrono>
3 #include <opencv2/opencv.hpp>
4 #include <cuda_runtime.h>
5 #include <cmath>
6
```

1.2. CUDA Kernel for Gaussian Weight Calculation

Now, let's look at the CUDA kernel function for Gaussian weight calculation.

This function calculates the Gaussian weight for a given pair of coordinates (x and y) and a specified standard deviation (sigma). The __device__ keyword ensures that this function runs on the GPU. It's an essential part of the Gaussian blur computation on the CUDA-enabled device.S

Explanation:

- __device__: This keyword is specific to CUDA and indicates that the function will run on the GPU. It's used to define functions that will be executed on the device (GPU) rather than the host (CPU).
- float gaussianWeight(float x, float y, float sigma): This line declares a function named gaussianWeight that takes three float parameters (x, y, and sigma).
- float sigma2 = 2.0f * sigma * sigma;: It calculates sigma2, which is two times the square of sigma. This value is used in the Gaussian weight formula.
- float t = (x * x + y * y) / sigma2;: Calculates the argument for the exponential function in the Gaussian weight formula.
- return exp(-t) / (M_PI * sigma2);: Computes and returns the Gaussian weight using the formula. exp is the exponential function, and M_PI is a constant representing π (pi).

```
6
7 // Kernel to calculate Gaussian weights
8 __device__ float gaussianWeight(float x, float y, float sigma) {
9     float sigma2 = 2.0f * sigma * sigma;
10    float t = (x * x + y * y) / sigma2;
11    return exp(-t) / (M_PI * sigma2);
12 }
13
```

1.3. CUDA kernel for Gaussian blur

This kernel calculates the weighted sum of pixel intensities for each pixel in the image using a 3x3 Gaussian kernel and then normalizes the result before storing it in the output image. The gaussianWeight function is called to compute the weights.

Explanation:

- __global__: Indicates that this function is a CUDA kernel and will be executed on the GPU.
- void gaussianBlurCUDA(...): Declares the CUDA kernel function named gaussianBlurCUDA, which takes input and output image pointers, image dimensions (width and height), and the Gaussian blur parameter sigma.
- Thread Indices Calculation:
    - int x = blockIdx.x * blockDim.x + threadIdx.x;: Calculates the x-coordinate of the current thread in the grid.
    - int y = blockIdx.y * blockDim.y + threadIdx.y;: Calculates the y-coordinate of the current thread in the grid.
- Boundary Check:
    - if (x < width && y < height) { ... }: Ensures that the current thread operates within the valid image dimensions.
- Local Variables Initialization:
    - float sum = 0.0f;: Initializes a variable to accumulate the weighted sum of pixel intensities.
    - float totalWeight = 0.0f;: Initializes a variable to accumulate the total weight of the Gaussian kernel.
- Loop Over the Neighborhood (3x3 Kernel):
    - Nested loops iterate over a 3x3 neighborhood centered around the current pixel.
    - for (int dy = -1; dy <= 1; dy++) { ... }: Loop over the vertical dimension.
    - for (int dx = -1; dx <= 1; dx++) { ... }: Loop over the horizontal dimension.
- Calculate Weighted Sum:

- `int pixelX = x + dx;` and `int pixelY = y + dy;`: Calculate the coordinates of the neighboring pixel.
- `if (pixelX >= 0 && pixelX < width && pixelY >= 0 && pixelY < height) { ... }`: Checks if the neighboring pixel is within the valid image boundaries.
- `float weight = gaussianWeight(dx, dy, sigma);`: Calls the `gaussianWeight` function to get the Gaussian weight.
- `sum += input[pixelY * width + pixelX] * weight;`: Updates the weighted sum.
- `totalWeight += weight;`: Updates the total weight.

- Normalize and Cast to Unsigned Char:
  - `output[y * width + x] = (unsigned char)((sum / totalWeight) + 0.5f);`: Normalizes the sum by dividing by the total weight, adds 0.5 for rounding, and stores the result in the output image.

```
3
4
5 // CUDA kernel for Gaussian blur
6 __global__ void gaussianBlurCUDA(const unsigned char* input, unsigned char* output,
7                                  int width, int height, float sigma) {
8     int x = blockIdx.x * blockDim.x + threadIdx.x;
9     int y = blockIdx.y * blockDim.y + threadIdx.y;
0
1     if (x < width && y < height) {
2         float sum = 0.0f;
3         float totalWeight = 0.0f;
4
5         // Sample neighborhood for simplicity... you can increase the kernel size
6         for (int dy = -1; dy <= 1; dy++) {
7             for (int dx = -1; dx <= 1; dx++) {
8                 int pixelX = x + dx;
9                 int pixelY = y + dy;
0
1                 if (pixelX >= 0 && pixelX < width && pixelY >= 0 && pixelY < height) {
2                     float weight = gaussianWeight(dx, dy, sigma);
3                     sum += input[pixelY * width + pixelX] * weight;
4                     totalWeight += weight;
5                 }
6             }
7         }
8         // Normalize and cast to unsigned char before assigning to output
9         output[y * width + x] = (unsigned char)((sum / totalWeight) + 0.5f);  // Add 0.5 for rounding
0     }
1 }
2
```

1.4. Main Function

The main function directs the entire image processing flow, overseeing tasks from loading the image to executing operations on the GPU, comparing execution times, and saving the resulting images. Let's break down the main function into smaller parts for a clearer understanding:

```
43
44 int main() {
45
46     // Image Path
47     std::string imagePath = "/content/SampleImage.jpg"; // Replace with your uploaded image path
48     cv::Mat image = cv::imread(imagePath, cv::IMREAD_GRAYSCALE);
49     if (image.empty()) {
50         std::cerr << "OpenCV version: " << CV_VERSION << std::endl;
51         std::cerr << "Image load failed!" << std::endl;
52         return -1;
53     }
```

In the beginning, the main function sets the path to the input image and uses OpenCV to read it in grayscale. If the image loading fails, an error message is displayed, and the program exits.

```
54
55     // Create output images for CPU and GPU
56     cv::Mat blurredImageCPU(image.size(), image.type());
57     cv::Mat blurredImageGPU(image.size(), image.type());
58
```

Here, two output images (blurredImageCPU and blurredImageGPU) are created to store the results of the CPU and GPU Gaussian blur operations.

```
58
59     // CPU Gaussian Blur (for timing comparison)
60     auto startCPU = std::chrono::high_resolution_clock::now();
61     cv::GaussianBlur(image, blurredImageGPU, cv::Size(3, 3), 3.0);
62     auto endCPU = std::chrono::high_resolution_clock::now();
63
```

The program then performs a Gaussian blur using the CPU implementation for later comparison with the GPU implementation. The duration of this operation is measured for timing comparison.

```
63
64     // Allocate device memory
65     unsigned char *d_input, *d_output;
66     cudaMalloc(&d_input, image.total());
67     cudaMalloc(&d_output, image.total());
68
```

Device memory is allocated for the input and output images on the GPU using CUDA functions.

```
68
69     // Copy input image to device
70     cudaMemcpy(d_input, image.data, image.total(), cudaMemcpyHostToDevice);
71
72     // Kernel launch configuration
73     dim3 blockSize(16, 16);  // 2D block
74     dim3 gridSize((image.cols + blockSize.x - 1) / blockSize.x,
75                   (image.rows + blockSize.y - 1) / blockSize.y);  // grid
76
77     // CUDA Gaussian Blur
78     auto startGPU = std::chrono::high_resolution_clock::now();
79     gaussianBlurCUDA<<<gridSize, blockSize>>>(d_input, d_output, image.cols, image.rows, 3.0);
80     cudaDeviceSynchronize();
81     auto endGPU = std::chrono::high_resolution_clock::now();
82
```

The input image data is then copied from the host (CPU) to the device (GPU).The configuration for launching the CUDA kernel is set, specifying the block and grid dimensions based on the image size.The GPU Gaussian blur operation is performed using the custom CUDA kernel (gaussianBlurCUDA). The duration of this operation is measured for timing comparison.

```
82
83      // Copy result back to host for CPU that is GPU
84      cv::Mat blurredImageHost(image.size(), image.type());
85      cudaMemcpy(blurredImageHost.data, d_output, image.total(), cudaMemcpyDeviceToHost);
86
87      // Calculate execution times
88      auto cpuDuration = std::chrono::duration<double, std::milli>(endCPU - startCPU).count();
89      auto gpuDuration = std::chrono::duration<double, std::milli>(endGPU - startGPU).count();
90
91      std::cout << "GPU Time: " << cpuDuration << " ms" << std::endl;
92      std::cout << "CPU Time: " << gpuDuration << " ms" << std::endl;
93
94      // Save ONLY blurred images
95      cv::imwrite("/content/cpu_blurred_image.jpg", blurredImageHost);
96      cv::imwrite("/content/gpu_blurred_image.jpg", blurredImageGPU);
97
```

The resulting image from the GPU is copied back to the host for later comparison with the CPU result.Execution times for both CPU and GPU operations are calculated and displayed.The resulting images from both CPU and GPU operations are saved to files.

```
97
98      // Display confirmation message
99      std::cout << "Blurred images saved as: \n"
100               << " - cpu_blurred_image.jpg\n"
101               << " - gpu_blurred_image.jpg\n";
102
103     // Release memory
104     cudaFree(d_input);
105     cudaFree(d_output);
106
107     return 0;
108 }
```

A confirmation message is displayed, indicating the names of the saved images, and Finally, memory allocated on the GPU is released, and the main function returns 0, indicating successful execution. The main function essentially manages the flow of the entire image processing task, from input to output, involving both CPU and GPU operations.

2. CMakeList.txt

The CMakeLists.txt file is an essential configuration file for the CMake build system. It is used to specify how the project should be built, including source files, dependencies, and compiler options. In this particular CMakeLists.txt file for a CUDA-enabled OpenCV project, several key elements are present.

The cmake_minimum_required command sets the minimum required version of CMake to 3.5. The project command defines the project name as "CudaOpenCVProject" and specifies the programming languages used in the project (C++ and CUDA).

The find_package command is utilized to locate the OpenCV library, which is a crucial dependency for the project. The set(CMAKE_CUDA_FLAGS ...) line sets the CUDA architecture flag to 'sm_75', reflecting the compute capability of the GPU. This should be adjusted according to the specific GPU architecture being used.

The include_directories command specifies the directories where CMake should look for header files, and in this case, it includes the directories required for OpenCV.

The add_executable command creates an executable named "CudaImage" from the source file "CudaImage.cu". Notably, the source file has a ".cu" extension, indicating that it contains CUDA code.

The set_target_properties command is used to set properties for the target, specifically the CUDA architecture (compute capability). In this case, the target property CUDA_ARCHITECTURES is set to 75, which corresponds to the specified GPU architecture.

Finally, the target_link_libraries command links the "CudaImage" executable with the OpenCV libraries, ensuring that the necessary OpenCV functions are available during compilation and linking.

Overall, the CMakeLists.txt file plays a crucial role in configuring the build process for a CUDA-enabled OpenCV project, specifying dependencies, compiler flags, and target properties.

```
cmake_minimum_required(VERSION 3.5)
project(CudaOpenCVProject LANGUAGES CXX CUDA)

# Find OpenCV
find_package(OpenCV REQUIRED)

# Set CUDA architecture (change according to your GPU architecture)
set(CMAKE_CUDA_FLAGS ${CMAKE_CUDA_FLAGS} -arch=sm_75)

# Specify include directories
include_directories(${OpenCV_INCLUDE_DIRS})

# Add CUDA executable with explicitly specifying source file
add_executable(CudaImage CudaImage.cu)

# Set CUDA architectures property for the target (replace with your GPU's compute capability)
set_target_properties(CudaImage PROPERTIES CUDA_ARCHITECTURES 75)

# Link OpenCV libraries
target_link_libraries(CudaImage ${OpenCV_LIBS})
```