CENG342 Project-2
Report

Leveraging NVIDIA CUDA for Image Downscaling

Emirhan Akıtürk 19050111065
Semih Gür 19050111017
Emre Özçatal 20050111074
Abdulsamet Haymana 19050111068

Introduction:

To use our Makefile:

make all will compile all 3 programs

make openmp will compile just the openmp program

make run_openmp will run the openmp program

make clean will delete all compiled programs

Similar targets exist for cuda and seq

This projects main goal is to downscale an image using CUDA. CUDA is a parallel computing platform and programming model developed by NVIDIA. It uses GPUs. Using NVIDIA's CUDA technology we want to increase speed and efficiency of downscaling of the image.

The objective of this project is to develop and implement a CUDA downscaling algorithm that can efficiently downscale images. We will try many downscaling techniques and assess their performance on different CUDA GPUs. We will try optimization strategies to maximize the efficiency and minimize memory usage.

This report provides an overview of our project. Explaining the motivation, objectives, methodology, and results achieved. It serves as a valuable resource for understanding the significance of CUDA technology in accelerating image processing tasks, specifically image downscaling. By the challenges encountered, lessons learned, and future possibilities, this report aims to contribute to the growing body of knowledge in GPU-accelerated computing for multimedia applications.

Algorithm Explanation:

The provided code demonstrates an implementation of image downscaling using NVIDIA CUDA technology. It utilizes parallel processing on CUDA-enabled GPUs to accelerate the downscaling process.

1. The `downscaleImage` kernel function is defined. It takes the input image, its width and height, and the output downsampled image as parameters.
2. Inside the kernel function, each thread is responsible for processing one pixel of the downsampled image. The thread's x and y coordinates determine the pixel's position.
3. To downscale the image to half size, the kernel loops through each 2x2 block of pixels in the input image.
4. Within the loop, the red, green, blue, and alpha color components of the four pixels are summed individually.
5. After calculating the sums, the downsampled pixel's index is computed, and the averaged color values are assigned to the corresponding positions in the downsampled image.

6. The `performCUDAImageDownscaling` function is called from the main function. It sets up the necessary device memory and launches the `downscaleImage` kernel.
7. Within `performCUDAImageDownscaling`, device memory is allocated for the input and downsampled images using `cudaMalloc`.
8. The input image data is then copied from the host to the device using `cudaMemcpy`.
9. Grid and block dimensions are defined to determine how the threads are organized for parallel execution.
10. `downscaleImage` kernel is launched with the specified grid and block dimensions.
11. After the kernel finishes execution, the downsampled image is copied back from the device to the host using `cudaMemcpy`.
12. Device memory is freed using `cudaFree`.
13. In the main function, the input image is loaded using the `stbi_load` function from the STB image library.
14. The original width, height, and number of channels (bpp) are obtained.
15. The downsampled width and height are calculated as half the original dimensions.
16. Memory is allocated for the downsampled image.
17. `performCUDAImageDownscaling` is called with the input image, original width and height, and the downsampled image.
18. The resulting downsampled image is written to a file using `stbi_write_jpg` from the STB image library.
19. Memory is freed using `stbi_image_free` and `free`.
20. The execution time is calculated and printed.

Pseudocode:

```
downscaleImage(inputImage, width, height, downsampledImage):
  for each thread (x, y) in the grid:
    if x < width / 2 and y < height / 2:
      redSum = 0, greenSum = 0, blueSum = 0, alphaSum = 0
      for each dy in [0, 1]:
        for each dx in [0, 1]:
          pixelIndex = ((y * 2 + dy) * width + (x * 2 + dx)) * CHANNEL_NUM
          redSum += inputImage[pixelIndex]
          greenSum += inputImage[pixelIndex + 1]
```

```
            blueSum += inputImage[pixelIndex + 2]
            alphaSum += inputImage[pixelIndex + 3]
        downsampledPixelIndex = (y * width / 2 + x) * CHANNEL_NUM
        downsampledImage[downsampledPixelIndex] = redSum / 4
        downsampledImage[downsampledPixelIndex + 1] = greenSum / 4
        downsampledImage[downsampledPixelIndex + 2] = blueSum / 4
        downsampledImage[downsampledPixelIndex + 3] = alphaSum / 4



performCUDAImageDownscaling(inputImage, width, height, downsampledImage):
    allocate device memory for d_inputImage and d_downsampledImage
    copy input image data to device memory
    set grid and block dimensions
    launch downscaleImage kernel with grid and block dimensions
    copy the result back to host memory
    free device memory
main():
    load input image using stbi_load
    obtain original width, height, and bpp
    calculate downsampled width and height
    allocate memory for downsampled image
    call performCUDAImageDownscaling with input image, original width and height, and
downsampled image
    write downsampled image to file using stbi_write_jpg
    free memory
    calculate and print execution time
```

Analysis:

The analysis section of the report focuses on discussing the requirements, challenges, and considerations for implementing image downscaling using CUDA. It includes an examination of the benefits of GPU for this task, limitations and the motivation behind choosing CUDA technology.

Design:

The design section gives the overall architecture and design choices made for the CUDA image downscaling project. It includes a description of the chosen downscaling algorithm, its suitability

for parallelization, and the utilization of CUDA's programming model. Additionally, the design section should discuss the allocation of memory on the device, grid and block dimensions for thread organization, and any optimization strategies employed.

Implementation:

The implementation section provides a detailed account of how the code is structured and organized. It covers the various components, functions, and libraries used in the project. It should explain the role of each function, including the main function, the `performCUDAImageDownscaling` function, and the `downscaleImage` kernel. Additionally, this section can highlight any specific considerations or challenges encountered during the implementation process.

Problems We Encountered:

We got various warnings that we couldn't understand the reason at all. So we researched a bit. Afterall what we understood is that its not that much of a problem. It does not have an effect on output or running time of the code.

We also got a wrong output in a system that has non-NVIDIA GPU. But in a system that is NVIDIA. It worked just fine.

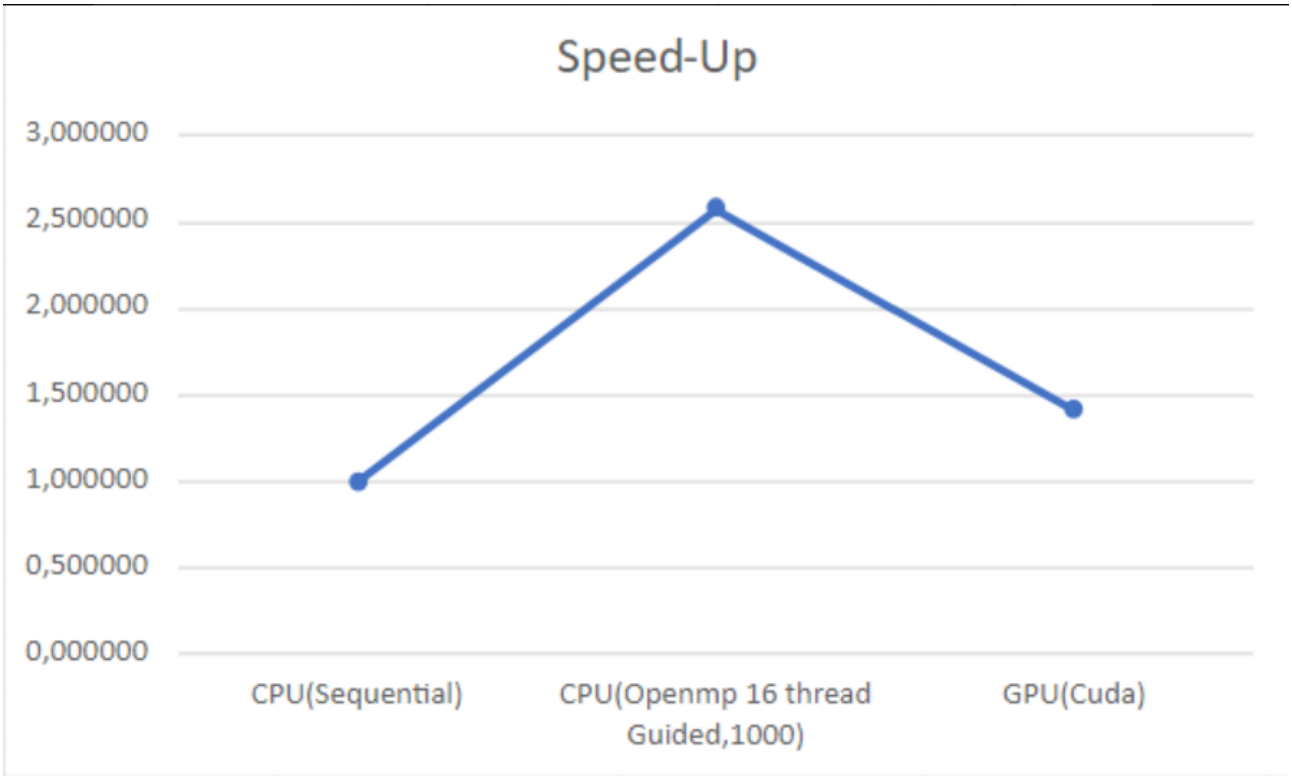
Evaluation:

The evaluation section assesses the performance and effectiveness of the CUDA image downscaling implementation. It includes experimental setup details, such as the hardware and software configurations used. Performance metrics, such as execution time, speedup achieved compared to a CPU approach, and memory utilization, should be measured and reported. The section should also discuss the visual quality of the downsampled images and any trade-offs between speed and quality observed.

Furthermore, the evaluation section can include a comparison of the CUDA implementation with other existing methods or frameworks for image downscaling.

Here is the tests. Our GPU is GTX 1050TI.

Tests \ Cases	CPU(Sequential)	CPU(Openmp 16 thread Guided,1000)	GPU(Cuda)
Test 1	0,120493	0,031936	0,080188
Test 2	0,112584	0,044930	0,076770
Test 3	0,114271	0,041130	0,084612
Test 4	0,114287	0,059357	0,077189
Test 5	0,115879	0,047165	0,091323
Test AVG	0,115503	0,044904	0,082016
Speed-up	1,000000	2,572239	1,408289



So it seems that in our tests CPU openmp 16 thread is the best.