CA Final Project — Image Sharpening

電子所 313510171 郭家均

電子所 313510188 陳柔伊

1. Environment

We're using WSL (Windows subsystem for linux) to run our programs.

With g++ version 13.3.0 and NVCC version 12.0 installed as the following figure shows.

```
ekj@DESKTOP-B76TL5R:~/CA_Final$ g++ --version
g++ (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

ekj@DESKTOP-B76TL5R:~/CA_Final$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Fri_Jan__6_16:45:21_PST_2023
Cuda compilation tools, release 12.0, V12.0.140
Build cuda_12.0.r12.0/compiler.32267302_0
```

2. Program files / attachment

Our program is under the directory of CA_Final, with 5 additional directories containing different files. To use this package, simply use the run script under the CA_Final directory in the terminal. If there's a permission issue, simply use chmod to modify the permission.

1. $chmod +x run
2. $./run

```
CA_Final/
├── CA_Final
├── ReadMe.txt
├── inc
│   ├── BMP.h
│   └── IMAGE.h
├── input
│   ├── Elephant.bmp
│   └── dog.bmp
├── main.cpp
├── makefile
├── obj
│   ├── BMP.d
│   ├── BMP.o
│   ├── IMAGE.o
│   ├── main.d
│   └── main.o
├── output
│   ├── Elephant_CPU_1D.bmp
│   ├── Elephant_CPU_2D.bmp
│   ├── Elephant_GPU.bmp
│   ├── dog_CPU_1D.bmp
│   ├── dog_CPU_2D.bmp
│   └── dog_GPU.bmp
├── run
└── src
    ├── BMP.cpp
    └── IMAGE.cu
```

inc/: contains header files (.h)

src/: contains source files (.cu .cpp)

obj/: contains object files (.o)

input/: contains 2 input images (Small/Large)

output/: contains several output images

main.cpp: main program

run: shell script to run this program package

ReadMe.txt: details of this package

makefile: script to compile this package

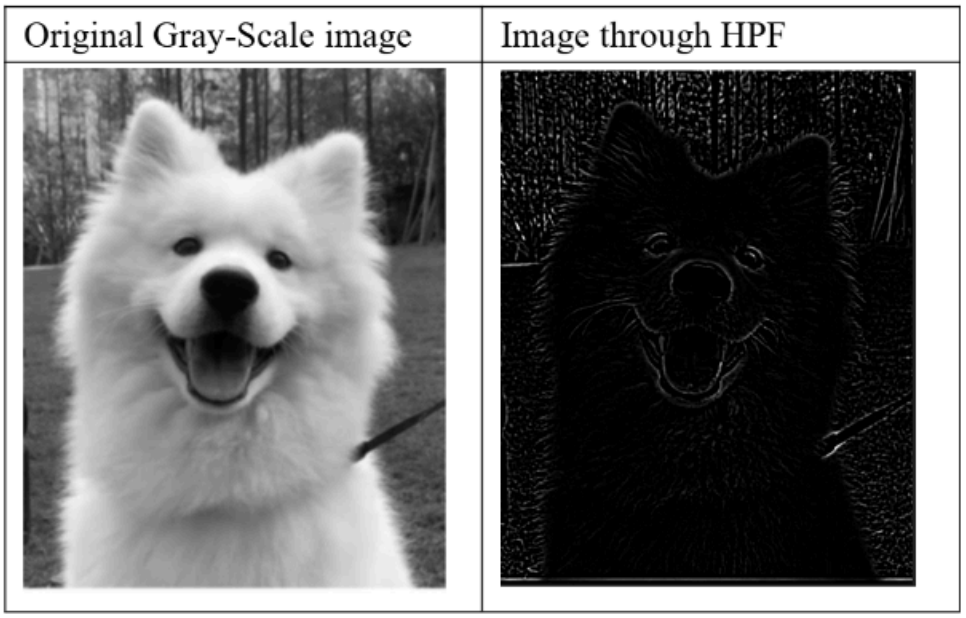
The BMP.h and BMP.cpp contains the basic transformation from bmp files to program data structures, and dumping processed data structures' information into bmp output images.

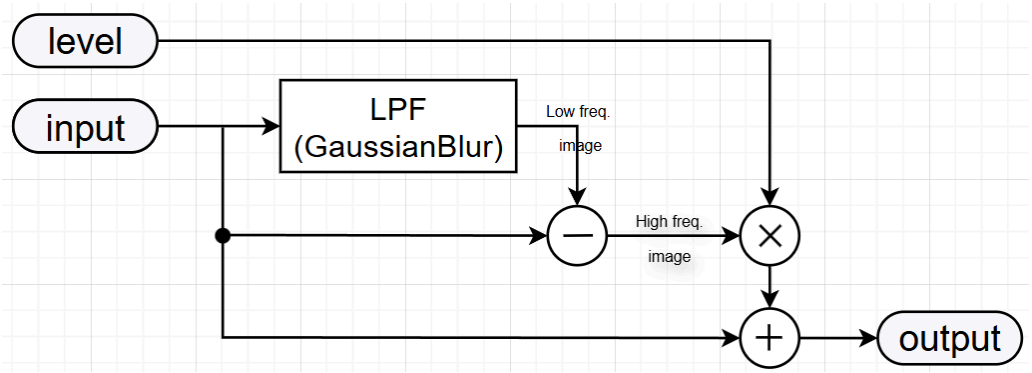
IMAGE.h and IMAGE.cu contain class and functions of image operations. The functions are separated into 2 groups, with one group of functions running on CPU, and the other group running through CUDA acceleration.

2 input images have different sizes, with dog.bmp smaller and Elephant.bmp way larger, so that we can observe the effect of data size.

3. Algorithm: Image sharpening

- We transfer the image from the RGB domain to the YCbCr domain.
- Gaussian blur: We apply gaussian blur on the original image, to obtain the low frequency image, L_image.
- Original_image - L_image = H_image: Through subtraction pixel-by-pixel, we get the high frequency elements of the original image, which is the feature we want to enhance.
- Original image + H_image * Level: Last step is to add the original image with a level scaled high-frequency image, therefore the high-frequency features (like image edges are enhanced).
- We get the sharpened image.



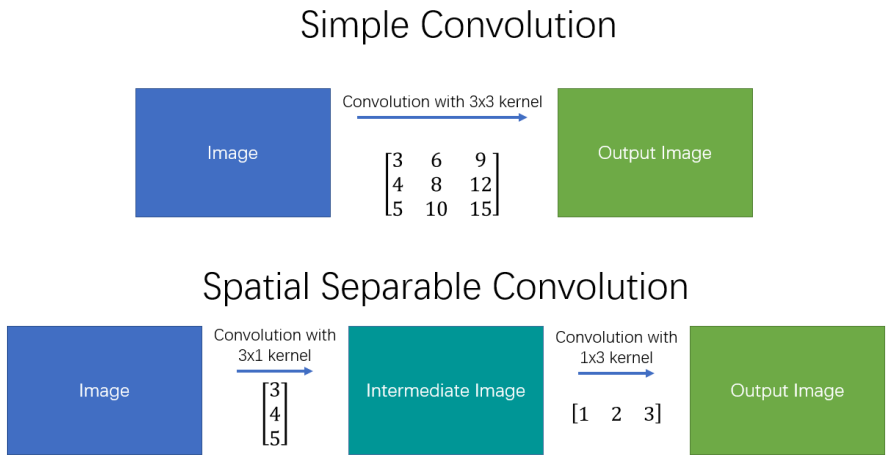


The algorithm flow has 3 versions, each version will produce an output image. Since GaussianBlur requires 2D convolution, where its kernel is linearly separable; we can speedup the algorithm in different ways.

(1) CPU + 2D convolution

This version is purely for comparison, no further speedup optimization utilized. We use normal c++ functions and 2D convolution to produce our desired output.

(2) CPU + 2*1D convolution



In this version, we split the 2D kernel into two 1D kernels, since the kernel is linearly separable. (SW algorithm-wise optimization) However, we still use CPU to run this version.

(3) GPU + 2D convolution

This version utilizes CUDA and NVCC to accelerate our sharpening process. We map the GaussianBlur and some addition/multiplication/subtraction operations on image to GPU's threads.

4. CUDA

This program performs per-pixel operations that are highly parallelizable, making it ideal for execution by many GPU threads simultaneously. Its compute-intensive nature and regular memory access patterns allow CUDA to significantly accelerate processing using shared memory and parallel execution.

(1) Thread-based parallelism : One thread per pixel → massive parallel computation

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

- Each CUDA thread is responsible for computing one pixel at position (x, y).
- Thousands of threads can run simultaneously on the GPU, massively accelerating pixel-wise computations.

(2) Shared memory usage : Fast memory cache for local pixel blocks

```
extern __shared__ float shared[];
```

- Loads the block's region into fast, low-latency shared memory, which is shared by threads in the same block.
- Reduces repeated memory access during the convolution.

(3) Unified Memory : Simplifies memory transfer between CPU/GPU

```
cudaMallocManaged(&Y, H * W * sizeof(float));
cudaMallocManaged(&Y_blur, H * W * sizeof(float));
cudaMallocManaged(&Y_out, H * W * sizeof(float));
cudaMallocManaged(&K, k * k * sizeof(float));
```

- Unified memory allows the CPU and GPU to access the same memory space, reducing data transfer costs.

5. Simulation result

[dog.bmp] (Small Case 1893 KB)

Original Image	Sharpened Image
	

[Elephant.bmp] (Large Case 24540 KB)

Original Image	Sharpened Image
	

As the above result shows, our sharpening algorithm works pretty well. The large case is originally a high resolution image, then blurred using online image edit tools. The sharpened image has finer details and more defined edges. However, due to high frequency information being amplified, the surface may appear to be grain-like.

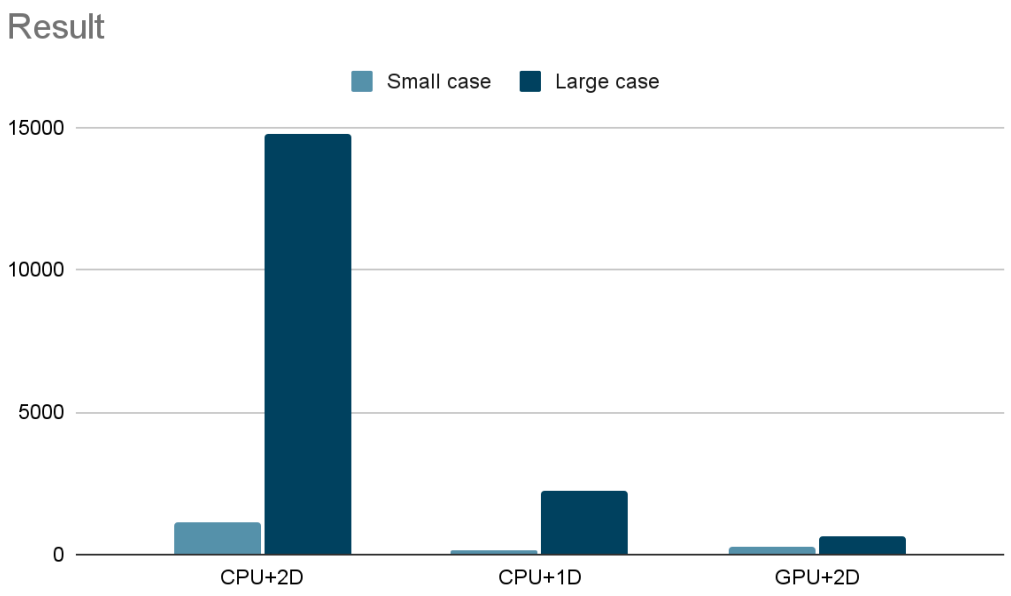
Here is the comparison of 3 versions:

```
ekj@DESKTOP-B76TL5R:~/CA_Final$ ./run
nvcc -std=c++11 -Iinc -c src/IMAGE.cu -o obj/IMAGE.o
nvcc -std=c++11 -Iinc obj/BMP.o obj/main.o obj/IMAGE.o -o CA_Final
Compile Done
Generate Sharper Image
-- Load Image input/dog.bmp
-- improve sharpness with CPU 2D input/dog.bmp
[CPU_2D] 1136.924 ms

-- Load Image input/dog.bmp
-- improve sharpness input/dog.bmp
[CPU_1D] 181.218 ms

-- Load Image input/dog.bmp
-- improve sharpness with CUDA input/dog.bmp
[GPU] 281.123 ms

-- Load Image input/Elephant.bmp
-- improve sharpness with CPU 2D input/Elephant.bmp
[CPU_2D] 14793.950 ms

-- Load Image input/Elephant.bmp
-- improve sharpness input/Elephant.bmp
[CPU_1D] 2269.179 ms

-- Load Image input/Elephant.bmp
-- improve sharpness with CUDA input/Elephant.bmp
[GPU] 670.742 ms
```

	[CPU + 2D]	[CPU + 1D]	[GPU + 2D]
Small case	1137 ms	181 ms	281 ms
Large case	14794 ms	2269 ms	671 ms

For the small case, we found out that [CPU + 1D] runs the fastest, while [GPU + 2D] runs slightly slower. That is because the data transfer overhead is larger on GPUs. The speedup process of the GPU does not compensate for it.

For the large case, we can see that [GPU + 2D] runs almost 3.4 times faster than the [CPU + 1D] approach, and 22 times faster than [CPU + 2D] version.



6. Conclusion

For smaller images, the benefit of GPU is weakened due to the large data transfer overhead. On the other hand, the runtime is significantly reduced when processing large images.

7. Project Distribution

郭家均	Basic C++ structure / CUDA optimization / Report
陳柔伊	Generate test case / CUDA optimization / Report