HPC_Based_ImageManipulation

Sangeethkumar I, Parthasarathy R, Ankitha M, Om Prakash, Animesh Kumar
sangeethkumar@iisc.ac.in, rparthasarat@iisc.ac.in, ankitham@iisc.ac.in, prakashom@iisc.ac.in, animeshkumar@iisc.ac.in

***Abstract* – Image processing incurs a lot of overhead when operating over the entire image. High Performance Computing helps address this increasing demand for processing speed and analyzing/processing huge amount of data. One can try to parallelize different steps in image processing by leveraging HPC based implementation and bring down the runtime. This mini-project focusses on comparison between a normal implementation v/s HPC based implementation for image augmentation.**

***Keywords*: HPC, OpenMP, CI, CUDA, High Resolution, Flip, Rotate**

SECTION I – METHODOLOGY

It is quite common to perform image processing/image augmentation using C/C++. For applications like medical imaging, image augmentation is done on very large images, and using just one CPU to process the image will be quite time consuming. Ensuring that image augmentation process does not take too much time and pose as a bottleneck, becomes pertinent.

With technology advancing at such a fast pace over the last few years, it is common to find personal computers/laptops to have multiple physical cores, and also with ability to operate over multiple logical cores. This is where High Performance Computing comes into picture. High performance computing (HPC) is the ability to process data and perform complex calculations at high speeds.

OpenMP is a library for parallel programming in shared-memory processors (i.e.) all threads share memory and data. An OpenMP program has sections that are sequential and sections that are parallel. The sequential section sets up the environment, initializes variables, etc and it runs on the master thread. Parallel sections of the program (under omp pragma) start execution on multiple threads. (slave threads)

CUDA is an extension of C/C++ programming that uses the Graphical Processing Unit (GPU). It is a parallel computing platform that allows computations to be performed in parallel while providing well-formed speed. Using CUDA, one can utilize the power of the Nvidia GPU to perform common computing tasks, such as processing matrices and other linear algebra operations.

It is also imperative to maintain versions for all the files being worked on. For this, Github Actions are used.

GitHubActions/CI: For seamless project file handling, Continuous Integration was utilized on the GitHub repository[5]. Single workflow with one job is used, job: "build" involves multiple steps to install – it runs build for CUDA and OpenMP. These actions are clubbed along with git push/pull. This maintains data version control history.

In this paper, different methodologies, and techniques tried to implement image augmentation for flip/rotate by leveraging shared memory parallel computing will be discussed. Image augmentation was tried with OpenMP, as well as CUDA, and the performance/speedup was compared with a CPU based implementation.

SECTION II –EXPERIMENTATION

A - OpenMP

OpenMP implementation was done to flip the images vertically and horizontally which had a huge impact in the speedup compared to the sequential implementation for the same. The *" # pragma omp parallel for "* was used to parallelize the flipping of image where the pixels have been manipulated and stored in the new array. The number of threads that needs to be used in the OpenMP implementation was taken as an argument.

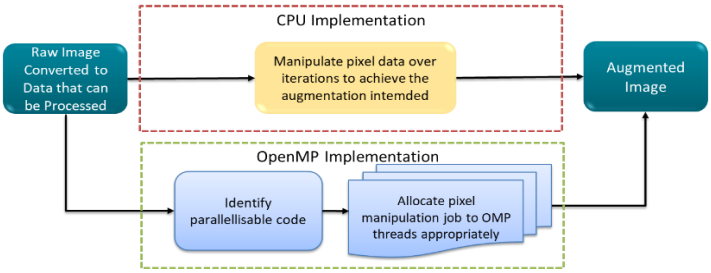

Figure 1 - OpenMP Image Augmentation Flow

B - CUDA

General CUDA flow includes Computation on the Host, Computation on the Device, Memory transfer from Host to Device and Device to Host, and CUDA Kernel calls. Code/Logic that can be parallelized is identified and allocated to different threads to generate the final augmented image.

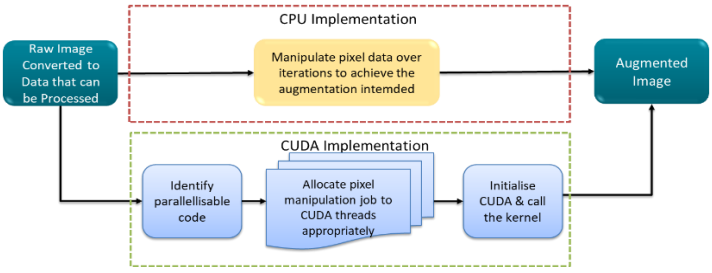

Figure 2 - CUDA Image Augmentation Flow

Different experiments were tried for CUDA based implementation.

CV::Mat

OpenCv C++ was another method that was used to load and save the images. This required a lot of effort since the loaded image was stored as a Mat datatype and processing the Mat datatype was quite tedious. Also, copying the image into the GPU was not straight forward, and the use of GPU Mat was required which did not have proper documentation. So, this idea was dropped off to load and save the images.

LodePNG

LodePNG is a header that helps one load PNG extension images of any size onto C/C++. It offers different in-built functions that helps one to encode an image into a vector of characters that contain the RBGAlpha information, as well as decode a character array/vector of RBGAlpha information back into a PNG image.
The only drawback with this header is that it is limited to only PNG images, and the method in which it encodes the actual RBG data is abstracted. Hence manipulating this data can meddle with the contrast/RBG settings of the image. The image was first loaded onto a character vector and later to a 1D character array using the *encode* function.
Fist, a CPU based image augmentation (Vertical Flip and Horizontal Flip) was implemented where every iteration of pixel manipulation was run on just from CPU.
Image augmentation (Vertical Flip and Horizontal Flip) was then done using this header using a CUDA based implementation. For vertical flip, the pixel manipulation was done on a 1-D grid of blocks each with 32 threads, and the CUDA kernel took care of flipping the pixels. For horizontal flipping, image manipulation was done on a 2-D grid of blocks with 32 threads each, and the CUDA kernel took care of properly flipping the pixel from the right rows and columns. The major problem that was faced was that while flipping the pixels, the RBG proportions changed and hence the image color changed. This method of loading images was dropped and stbi was used in future experiments.

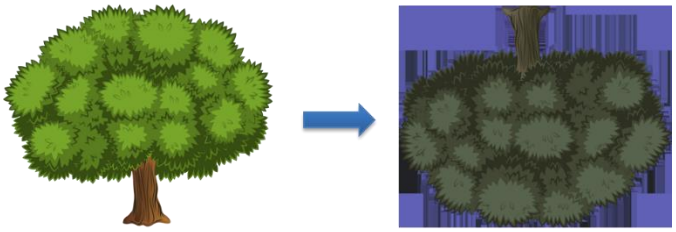

Figure 3 - Vertical Flip using LodePNG

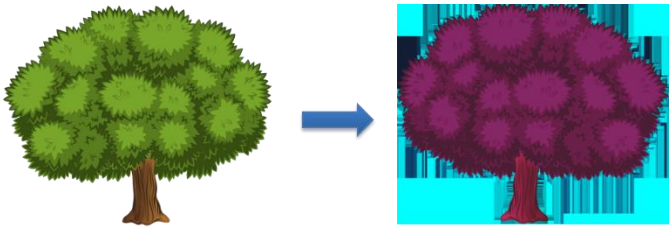

Figure 4 - Horizontal Flip using LodePNG

STB

Stb_image libraries have been used to read and write the images for CUDA based HPC implementation of image augmentation (Vertical Flip, Horizontal Flip, Rotate, Scaling).
STB_IMAGE_IMPLEMENTATION has to be defined the before including the stb_image.h header for reading the image and STB_IMAGE_WRITE_IMPLEMENTATION has to be defined for writing an image. It offers an in-built function stbi_load for loading an image which receives the image file path, width and number of channels as the arguments. This also gives the flexibility to load upto 3 channels for a 4 channel image. Similarly, stibi_image_write functions can be used to save back the image in the disk. This library offers much more flexibilty than the LodePNG as it supports multiple extensions of the image such as jpg,png etc and was also easier to do the image processing the loaded image and write back the processed image. The actual CUDA implementation for all the image augmentation variants utilized 2-D grid of blocks, and the CUDA kernel took care of manipulating the appropriate pixel based on the current executing thread.

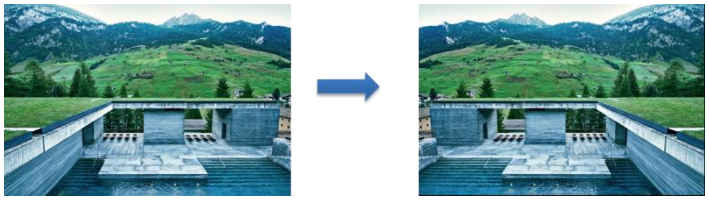

Figure 5 - Horizontal Flip using STBI

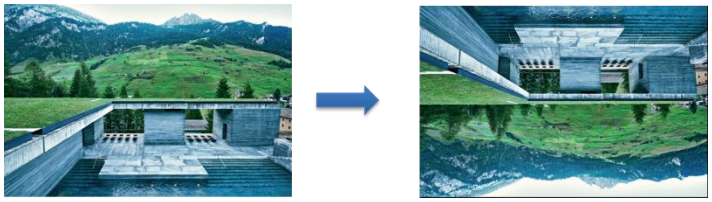

Figure 6 - Vertical Flip using STBI

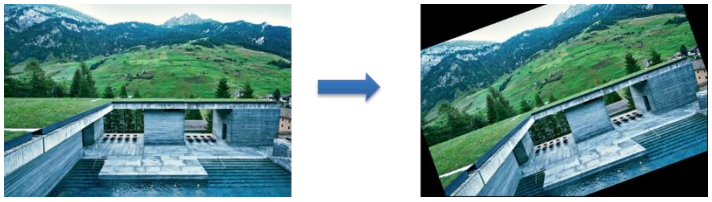

Figure 7 - Image Rotation using STBI

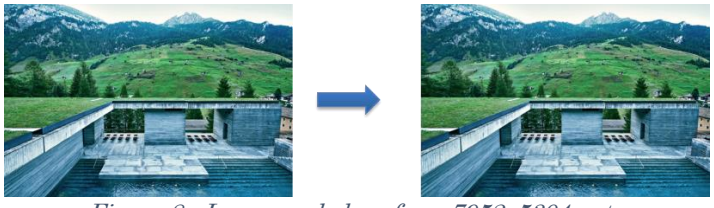

Figure 8 - Image scaled-up from 7952x5304px to 31808x21216px

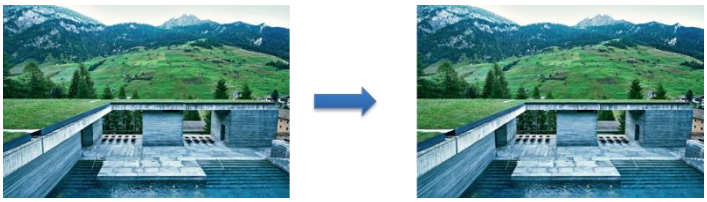

Figure 9 - Image scaled-down from 7952x5304px to 3976x2652px

C – CODE BUILD AND CONTINOUS INTEGRATION (CI)

For efficiently maintaining the codes, its versions, builds, and for continuous integration, GitHub repo was utilized. Repo Link: **https://github.com/prakash90om/MLOps_HPC_Image_Processing**.

For CI, GitHub Actions was used. A single workflow is incorporated and it is triggered when any changes are pushed to the repo. Inside the workflow a single job (build) is being triggered which runs all the mentioned steps, such as cloning the repository to the runner using "actions/checkout@v3", installing Cuda setup and then building the code.

For code compilation and build, Makefile was used, which has a set of rules added, so that different portion of files are compiled with those rules. A rule generally looks like this: *target: prerequisites*, then commands. So, to build/compile/clean commands like: *make cuda (for CUDA files to build), make clean (to clean the build folder)* can be used etc.

SECTION III – RESULTS

This section captures the results from different experiments tried out.

OpenMP based image augmentation was one on **7952x5304 px** image of size ~17Mb. Below is the execution time on CPU implementation v/s OpenMP implementation and the speedup achieved. The time taken below purely denotes the time to manipulate the image.

	CPU Time	4 Threads	8 Threads	16 Threads	32 Threads	Maximum Speedup
Horizontal Flip	545.84ms	263.53ms	143.52ms	87.80ms	69.846ms	7.814
Vertical Flip	392.24ms	182.30ms	110.16ms	65.203ms	51.772	7.576

Figure 10 - Computation Time Comparison for sequential v/s OpenMP based implementation

CUDA based Image Augmentation using LodePNG was done on a **8000x6612 px** image of size **5.5 Mb**. Below is the execution time on CPU implementation v/s GPU implementation and the speedup achieved for 32 threads. The time taken below purely denotes the time to manipulate the image.

	CPU Time	GPU Time	Speedup
Vertical Flip	2517.747ms	21.9754 ms	114.5711
Horizontal Flip	5965.787 ms	21.6209 ms	275.9268

Figure 11 - Computation Time comparison for CPU v/s GPU in LodePNG based CUDA implementation

CUDA based Image Augmentation using STBI was done on a **7952x5304 px** image of size ~**17 Mb**. Below is the execution time on CPU implementation v/s GPU implementation and the speedup achieved for 32 threads. The time taken below purely denotes the time to manipulate the image.

	CPU Time	GPU Time	Speedup
Vertical Flip	274.303ms	0.648ms	423.307
Horizontal Flip	440.6 ms	5.779ms	76.2416
Image Scaling up	2746.7ms	7.409ms	370.725
Image Scaling Down	143.397ms	0.383ms	374.405
Image Rotation	694.323ms	0.879ms	789.901

Figure 12 - Computation Time comparison for CPU v/s GPU in STBI based CUDA implementation

SECTION IV –CONCLUSION

In conclusion, we can see that a HPC based implementation offers a large speedup compared to a CPU based implementation. The sheer processing power that multiprocessor workstations offer can be leveraged to speed-up image processing applications.

SECTION V –REFERENCES

1. LodePNG Documentation: https://lodev.org/lodepng
2. OpenMP Documentation: https://www.openmp.org/spec-html/5.0/openmp.html
3. GitHub Actions: https://docs.github.com/en/actions
4. CUDA: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
5. Project GitHub Repo: https://github.com/prakash90om/MLOps_HPC_Image_Processing
6. STB Image: https://github.com/nothings/stb