**MPI**

**Code parallelization with MPI, OpenMP and CUDA.**

**INDEX**

[1 Introduction 3](#_Toc58190599)

[2 About the code 4](#_Toc58190600)

[3 Section A – Decomposition 5](#_Toc58190601)

[4 Section B – Assignment 7](#_Toc58190602)

[4.1 CUDA 7](#_Toc58190603)

[4.2 MPI 7](#_Toc58190604)

[4.3 OpenMP 8](#_Toc58190605)

[5 Section C – Orchestration 9](#_Toc58190606)

[5.1 Input Image 9](#_Toc58190607)

[5.2 Output images 10](#_Toc58190608)

[6 Section D – Mapping 11](#_Toc58190609)

[1 Results 12](#_Toc58190610)

[1.1 Sequential Result 12](#_Toc58190611)

[1.2 Parallel Result 13](#_Toc58190612)

[1.3 Comparatives by nodes 14](#_Toc58190613)

[1.4 Global Comparisons 15](#_Toc58190614)

[1.5 Acceleration 15](#_Toc58190615)

[2 Conclusions 16](#_Toc58190616)

# Introduction

The X-ray images, scientific images, or satellite images the majority of the time lack details for such images contrast enhancement is a commonly used operation in image processing. Sometimes pictures are overexposed or underexposed to light, it is also a useful technique for enhancing details.

The objective of this work is to develop a contrast enhancement application on a colored picture that uses acceleration over MPI, OpenMP, and CUDA. The comparison among the time taken to colored images contrast enhancement on MPI, OpenMP, and CUDA will be drawn and analyze.

# About the code

The code of the parallelization is based on four phases: decomposition, allocation, orchestration, and distribution. Further elaboration on these four phases will be done in their respective sections.

# Section A – Decomposition

In this code, we used OpenMP, MPI, and CUDA for parallelism. In OpenMP, Parallelism occurs where every parallel thread has access to all of the data. In the case of MPI, parallelism is executed by allocating each process its isolated space memory. Meanwhile, CUDA does parallel computing at the expense of GPU power.

The parallel decomposition is done by sequential code block is used in “for loop”.

**Sequential Code:**

# Usage

# First Convert the picture in in.pgm/in.ppm

convert highres.jpg in.pgm

convert highres.jpg in.ppm

# For Compilation of Makefile

Make

# For Execution of Sequential Code

make run

**OpenMP Code:**

# Usage

# First Convert the picture in in.pgm/in.ppm

convert highres.jpg in.pgm

convert highres.jpg in.ppm

# For Compilation make

make

# For Execution

make run

# Compare the execution time with the sequential code

**MPI Code:**

# Usage

# First Convert the picture in in.pgm/in.ppm

convert highres.jpg in.pgm

convert highres.jpg in.ppm

# For Compilation make

make

# For Execution

make run

# Compare the execution time with the sequential code

**CUDA Code:**

# Usage

# First Convert the picture in in.pgm/in.ppm

convert highres.jpg in.pgm

convert highres.jpg in.ppm

# For Compilation make

make

# For Execution CUDA

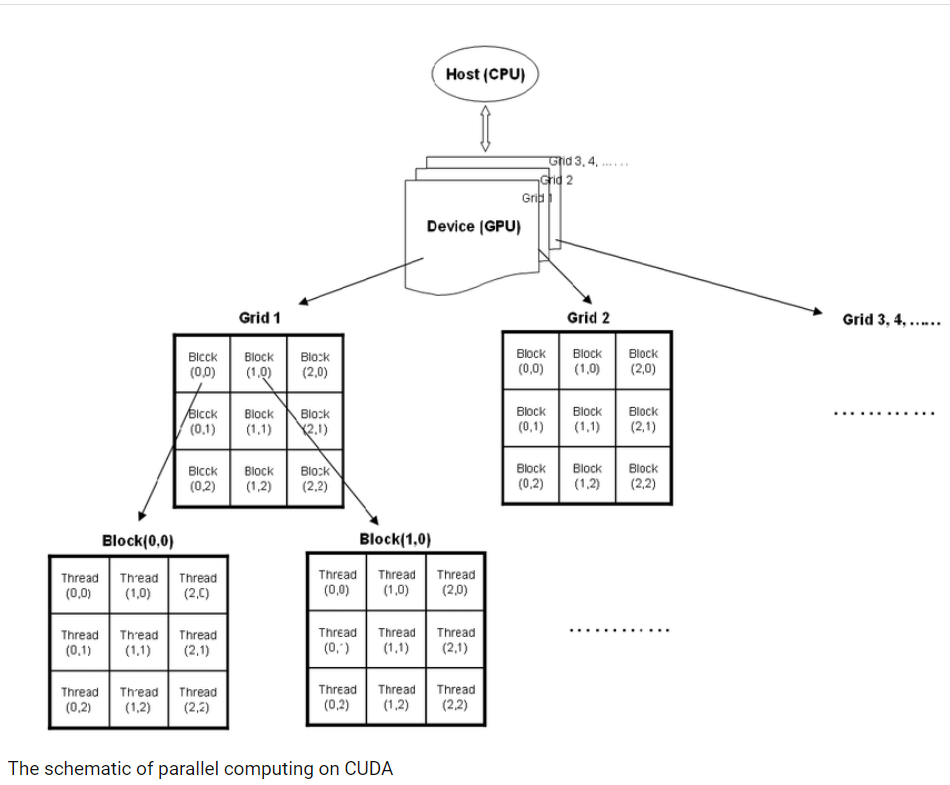
make run

# Compare the execution time with the sequential code

# Section B – Assignment

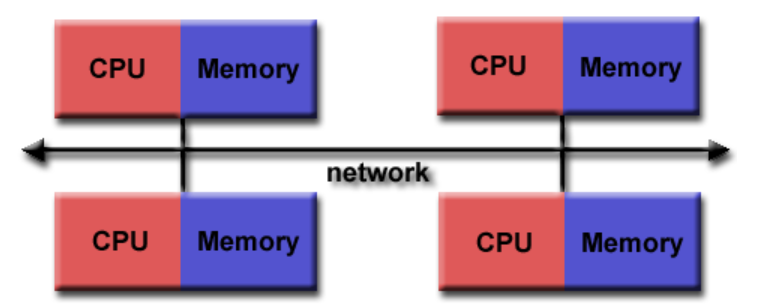
## CUDA

CUDA is a parallel computing platform and programming model developed by Nvidia for general computing on its GPUs (graphics processing units).



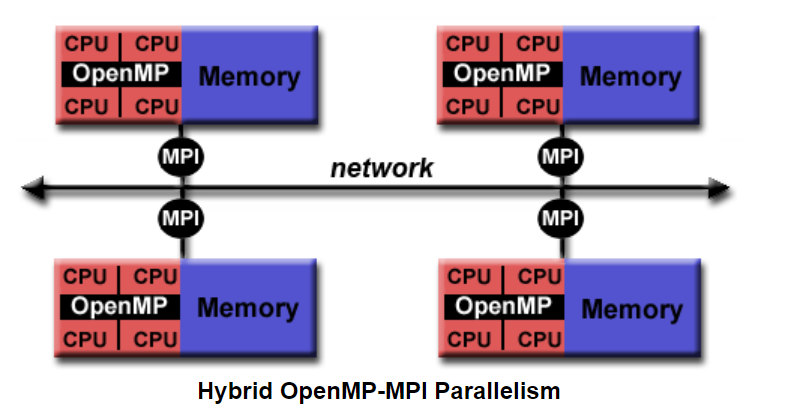
## MPI

MPI is specifically used to allow applications to run in parallel across several separate computers connected by a network.



## OpenMP

OpenMP is designed for shared memory parallel programming, it is largely limited to **single-node** parallelism. Typically, the number of processing elements (cores) on a node determines how much parallelism can be implemented.



# Section C – Orchestration

A sequential code is running for parallel distribution among the different processes.

Firstly the images are converted from jpeg to in. pgm/in.ppm using the “convert” function.

Compilation is made using the ‘make’ function and for execution ‘run’ function is used.

## Input Image



## Output images



1 After HSL ppm



2 After YUV ppm

# Section D – Mapping

“Make” function is used for mapping the picture. It returns the output images.

# Results

The following are the results of sequential, OpenMP, MPI, and CUDA respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Time (sec) | Sequential | OpenMP | MPI | CUDA |
| HSL Time | 6.0206 | 5.4383 | 5.1242 | 0.7464 |
| YUV Time | 2.8499 | 9.9670 | 2.6457 | 0.8402 |
| Total Run Time | 11.9564 | 19.0380 | 5.7924 | 5.2189 |

## Sequential Result

We can see that the YUV image took less time for enhancement than the HSL color-coded image. The time difference of 4 seconds is visible in this chart.

## Parallel Result

In the case of OpenMP, the processing time is increasing exponentially. We can see that the least time taken is almost 4 seconds and the color code is HSL.

In contrast to the OpenMP, the color code YUV takes lesser time than HSL. The YUV run time is less by 3 seconds than HSL.

In the case of CUDA, The run time for HSL and YUV color-coded image enhancements is almost the same. The run time is less than 1 second.

## Comparatives by nodes

For sequential, OpenMP, and MPI we used Intel Core i7 7th generation system, 2 Cores, and 4 threads. The clock speed was 2.80 GHz. The number of processes running while execution are 259.

In the case of CUDA, NVIDIA Tesla T4 GPU was used with 2560 cores and the bus width of 256 bits.

## Global Comparisons

This Histogram shows that the least runtime taken for the image enhancement process is in CUDA, followed up by the MPI. The surprising observation is that the runtime for OpenMP is more by a large number than the Sequential processing.

## Acceleration

# Conclusions

The output images show that enhancement in the HSL and YUV images are quite visible. The details that can be seen in the output images speak for themselves. When coming to the comparison among different parallel computing platform deployed we can surely say that CUDA takes the least time. This Optimal Time was achieved at expense of the higher power of GPU.

The comparison among the platform using CPU clearly states that the least time was taken by MPI and on contrary the most time-consuming platform was OpenMP.