**1) Vtune profiling**

**save as matrix\_multiplication.cpp**

**matrix multipliation**

#include <iostream>

#include <vector>

#include <ctime>

#define N 1024 // Size of the matrix

// Function to perform matrix multiplication

void matrix\_multiply(const std::vector<std::vector<int>>& A, const std::vector<std::vector<int>>& B, std::vector<std::vector<int>>& C) {

for (int i = 0; i < N; ++i) {

for (int j = 0; j < N; ++j) {

C[i][j] = 0;

for (int k = 0; k < N; ++k) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

}

int main() {

std::vector<std::vector<int>> A(N, std::vector<int>(N, 1)); // Matrix A with all values 1

std::vector<std::vector<int>> B(N, std::vector<int>(N, 1)); // Matrix B with all values 1

std::vector<std::vector<int>> C(N, std::vector<int>(N)); // Result matrix C

clock\_t start\_time = clock();

matrix\_multiply(A, B, C);

clock\_t end\_time = clock();

double time\_taken = double(end\_time - start\_time) / CLOCKS\_PER\_SEC;

std::cout << "Matrix multiplication completed in " << time\_taken << " seconds." << std::endl;

return 0;

}

#include <iostream>

#include <vector>

#include <cstdlib> // For rand()

#include <ctime> // For time()

#include <omp.h> // For OpenMP

#define N 1024

using namespace std;

// Matrix multiplication with OpenMP

void matrix\_multiply(const vector<vector<int>>& A, const vector<vector<int>>& B, vector<vector<int>>& C) {

#pragma omp parallel for

for (int i = 0; i < N; ++i) {

for (int j = 0; j < N; ++j) {

C[i][j] = 0;

for (int k = 0; k < N; ++k) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

}

// Fill a matrix with random integers between 0 and 9

void fill\_matrix\_random(vector<vector<int>>& M) {

for (int i = 0; i < N; ++i) {

for (int j = 0; j < N; ++j) {

M[i][j] = rand() % 10; // Random value from 0 to 9

}

}

}

int main() {

srand(time(0)); // Seed the random number generator

vector<vector<int>> A(N, vector<int>(N));

vector<vector<int>> B(N, vector<int>(N));

vector<vector<int>> C(N, vector<int>(N));

fill\_matrix\_random(A);

fill\_matrix\_random(B);

double start\_time = omp\_get\_wtime();

matrix\_multiply(A, B, C);

double end\_time = omp\_get\_wtime();

cout << "Matrix multiplication completed in " << (end\_time - start\_time) << " seconds." << endl;

return 0;

}

**compile it using**

**1🡪**

g++ -O2 -g matrix\_multiplication.cpp -o matrix\_multiplication

2🡪  
g++ -fopenmp matrix\_mult.cpp -o matrix\_mult

**add the compiled file in configuration and run it and also memory consumption**

**2) vector addition using parallization approach**

**save file as vector\_addition.c**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 1000000 // Size of vectors

int main() {

// Allocate memory for vectors A, B, and C

int \*A = (int \*)malloc(N \* sizeof(int));

int \*B = (int \*)malloc(N \* sizeof(int));

int \*C = (int \*)malloc(N \* sizeof(int));

// Initialize vectors A and B with some values

for (int i = 0; i < N; i++) {

A[i] = i;

B[i] = 2 \* i;

}

// Start parallel region

#pragma omp parallel for

for (int i = 0; i < N; i++) {

C[i] = A[i] + B[i];

}

// Optionally, print the first 10 elements of the result vector to verify correctness

for (int i = 0; i < 10; i++) {

printf("C[%d] = %d\n", i, C[i]);

}

// Free allocated memory

free(A);

free(B);

free(C);

return 0;

}

**commands to run-**

gcc -fopenmp -o vector\_addition vector\_addition.c

./vector\_addition

**3)Histogam equilizaton using CUDA**

**install Cuda-**sudo apt update

sudo apt install pkg-config

sudo apt install libopencv-dev

sudo apt install nvidia-cuda-toolkit

sudo apt install gcc-9 g++-9

sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-9 50

**take one image and save it in file and add its path in code**

**file name- histogram\_equalization.cu**

take one image and save it in file

#define STB\_IMAGE\_IMPLEMENTATION

#define STB\_IMAGE\_WRITE\_IMPLEMENTATION

#include "stb\_image.h"

#include "stb\_image\_write.h"

#include <iostream>

#include <cuda\_runtime.h>

#include <math.h>

#define HISTOGRAM\_SIZE 256

#define BLOCK\_SIZE 256

\_\_global\_\_ void computeHistogram(unsigned char \*d\_input, int \*d\_histogram, int width, int height, int channels) {

    int idx = blockIdx.x \* blockDim.x + threadIdx.x;

    if (idx < width \* height) {

        int pixel\_val = d\_input[idx \* channels];

        atomicAdd(&d\_histogram[pixel\_val], 1);

    }

}

\_\_global\_\_ void computeCDF(int \*d\_histogram, float \*d\_cdf, int total\_pixels) {

    \_\_shared\_\_ int temp[HISTOGRAM\_SIZE];

    int idx = threadIdx.x;

    if (idx < HISTOGRAM\_SIZE) {

        temp[idx] = d\_histogram[idx];

    }

    \_\_syncthreads();

    if (idx == 0) {

        float sum = 0;

        for (int i = 0; i < HISTOGRAM\_SIZE; ++i) {

            sum += temp[i];

            d\_cdf[i] = sum / total\_pixels;

        }

    }

}

\_\_global\_\_ void equalizeImage(unsigned char \*d\_input, unsigned char \*d\_output, float \*d\_cdf, int width, int height, int channels) {

    int idx = blockIdx.x \* blockDim.x + threadIdx.x;

    if (idx < width \* height) {

        for (int c = 0; c < channels; ++c) {

            unsigned char pixel = d\_input[idx \* channels + c];

            int equalized\_value = roundf(d\_cdf[pixel] \* 255.0f);

            d\_output[idx \* channels + c] = min(max(equalized\_value, 0), 255); // Clamp between 0 and 255

        }

    }

}

int main() {

    int width, height, channels;

    const char\* input\_image\_path = "image.jpg"; // Set the input image path

    unsigned char \*h\_input = stbi\_load(input\_image\_path, &width, &height, &channels, 0);

    if (h\_input == nullptr) {

        std::cerr << "Error loading image!" << std::endl;

        return -1;

    }

    int total\_pixels = width \* height;

    unsigned char \*h\_output = new unsigned char[total\_pixels \* channels];

    int \*h\_histogram = new int[HISTOGRAM\_SIZE](); // Initialize histogram with zeros

    float \*h\_cdf = new float[HISTOGRAM\_SIZE]; // CDF array

    unsigned char \*d\_input, \*d\_output;

    int \*d\_histogram;

    float \*d\_cdf;

    cudaMalloc((void\*\*)&d\_input, total\_pixels \* channels \* sizeof(unsigned char));

    cudaMalloc((void\*\*)&d\_output, total\_pixels \* channels \* sizeof(unsigned char));

    cudaMalloc((void\*\*)&d\_histogram, HISTOGRAM\_SIZE \* sizeof(int));

    cudaMalloc((void\*\*)&d\_cdf, HISTOGRAM\_SIZE \* sizeof(float));

    cudaMemcpy(d\_input, h\_input, total\_pixels \* channels \* sizeof(unsigned char), cudaMemcpyHostToDevice);

    // Initialize histogram on GPU/Device

    cudaMemset(d\_histogram, 0, HISTOGRAM\_SIZE \* sizeof(int));

    // CUDA events for timing

    cudaEvent\_t start, stop;

    float milliseconds = 0;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    // Record start time

    cudaEventRecord(start);

    int gridSize = (total\_pixels + BLOCK\_SIZE - 1) / BLOCK\_SIZE;

    computeHistogram<<<gridSize, BLOCK\_SIZE>>>(d\_input, d\_histogram, width, height, channels);

    cudaDeviceSynchronize();

    cudaMemcpy(h\_histogram, d\_histogram, HISTOGRAM\_SIZE \* sizeof(int), cudaMemcpyDeviceToHost);

    computeCDF<<<1, HISTOGRAM\_SIZE>>>(d\_histogram, d\_cdf, total\_pixels);

    cudaDeviceSynchronize();

    cudaMemcpy(h\_cdf, d\_cdf, HISTOGRAM\_SIZE \* sizeof(float), cudaMemcpyDeviceToHost);

    equalizeImage<<<gridSize, BLOCK\_SIZE>>>(d\_input, d\_output, d\_cdf, width, height, channels);

    cudaDeviceSynchronize();

    cudaMemcpy(h\_output, d\_output, total\_pixels \* channels \* sizeof(unsigned char), cudaMemcpyDeviceToHost);

    // Record stop time

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    // Calculate elapsed time

    cudaEventElapsedTime(&milliseconds, start, stop);

    std::cout << "Total execution time: " << milliseconds << " ms" << std::endl;

    if (stbi\_write\_jpg("output\_image.jpg", width, height, channels, h\_output, 90)) {

        std::cout << "Histogram equalization completed and saved as output\_image.jpg!" << std::endl;

    } else {

        std::cerr << "Error saving the image!" << std::endl;

    }

    // Free the Mem.

    stbi\_image\_free(h\_input);

    delete[] h\_output;

    delete[] h\_histogram;

    delete[] h\_cdf;

    cudaFree(d\_input);

    cudaFree(d\_output);

    cudaFree(d\_histogram);

    cudaFree(d\_cdf);

    cudaEventDestroy(start);

    cudaEventDestroy(stop);

    return 0;

}

//code is remaning  
#include <opencv2/opencv.hpp>

#include <iostream>

using namespace cv;

using namespace std;

void plotHistogram(const Mat &image, const string &windowName) {

    Mat grayImage;

    if (image.channels() == 3) {

        cvtColor(image, grayImage, COLOR\_BGR2GRAY);

    } else {

        grayImage = image;

    }

    int histSize = 256;

    float range[] = { 0, 256 };

    const float\* histRange = { range };

    Mat hist;

    calcHist(&grayImage, 1, 0, Mat(), hist, 1, &histSize, &histRange);

    normalize(hist, hist, 0, 255, NORM\_MINMAX);

    int histHeight = 400;

    int histWidth = 512;

    Mat histImage(histHeight, histWidth, CV\_8UC1, Scalar(255));

    for (int i = 0; i < histHeight; i += 50) {

        line(histImage, Point(0, i), Point(histWidth, i), Scalar(220), 1, 8);

    }

    for (int i = 0; i < histWidth; i += 50) {

        line(histImage, Point(i, 0), Point(i, histHeight), Scalar(220), 1, 8);

    }

    line(histImage, Point(40, histHeight - 30), Point(histWidth, histHeight - 30), Scalar(0), 2, 8); // X axis

    line(histImage, Point(40, 0), Point(40, histHeight), Scalar(0), 2, 8); // Y axis

    // Draw the histogram

    for (int i = 1; i < histSize; i++) {

        line(histImage, Point((i - 1) \* 2 + 40, histHeight - 30 - cvRound(hist.at<float>(i - 1))),

             Point(i \* 2 + 40, histHeight - 30 - cvRound(hist.at<float>(i))),

             Scalar(0), 2, 8, 0);

    }

    putText(histImage, "Pixel Intensity", Point(histWidth - 100, histHeight - 10),

            FONT\_HERSHEY\_SIMPLEX, 0.8, Scalar(0), 2, 8);  // X-axis label

    putText(histImage, "Frequency", Point(20, 30),

            FONT\_HERSHEY\_SIMPLEX, 0.8, Scalar(0), 2, 8);  // Y-axis label

    imshow(windowName, histImage);

}

int main() {

    // Start timer

    double startTime = (double)getTickCount();

    Mat image1 = imread("image.jpg", IMREAD\_COLOR);

    Mat image2 = imread("output\_image.jpg", IMREAD\_COLOR);

    if (image1.empty() || image2.empty()) {

        cout << "Could not open or find the images!" << endl;

        return -1;

    }

    // Time for loading images

    double loadTime = ((double)getTickCount() - startTime) / getTickFrequency();

    cout << "Time to load images: " << loadTime << " seconds" << endl;

    // Start plotting histograms

    startTime = (double)getTickCount();

    plotHistogram(image1, "Histogram of Image 1");

    plotHistogram(image2, "Histogram of Image 2");

    // Time for plotting histograms

    double plotTime = ((double)getTickCount() - startTime) / getTickFrequency();

    cout << "Time to plot histograms: " << plotTime << " seconds" << endl;

    waitKey(0);

    // Overall time

    double totalTime = ((double)getTickCount() - startTime) / getTickFrequency();

    cout << "Total execution time: " << totalTime << " seconds" << endl;

    return 0;

}

nvcc -o histogram\_equalization histogram\_equalization.cu -Xcompiler -std=c++14 -ccbin /usr/bin/g++-9 ‘pkg-config --cflags --libs opencv4’

./histogram\_equilization

**4)Image compression ith NVPROF**

take one image and save it in file

#include <iostream>

#include <opencv2/opencv.hpp>

#include <cuda\_runtime.h>

// CUDA kernel for simple image compression (dummy logic)

\_\_global\_\_ void compressImageKernel(unsigned char \*image, int width, int height) {

int idx = threadIdx.x + blockIdx.x \* blockDim.x;

if (idx < width \* height) {

image[idx] = image[idx] / 2; // Example compression (not real compression)

}

}

// Host function to launch the kernel

void compressImage(cv::Mat &inputImage) {

unsigned char \*d\_image;

size\_t imageSize = inputImage.total() \* inputImage.elemSize();

cudaMalloc((void\*\*)&d\_image, imageSize);

cudaMemcpy(d\_image, inputImage.data, imageSize, cudaMemcpyHostToDevice);

int threadsPerBlock = 256;

int numBlocks = (inputImage.total() + threadsPerBlock - 1) / threadsPerBlock;

compressImageKernel<<<numBlocks, threadsPerBlock>>>(d\_image, inputImage.cols, inputImage.rows);

cudaDeviceSynchronize();

cudaMemcpy(inputImage.data, d\_image, imageSize, cudaMemcpyDeviceToHost);

cudaFree(d\_image);

}

int main() {

// Load grayscale image

std::string imagePath = "img1.jpeg";

cv::Mat inputImage = cv::imread(imagePath, cv::IMREAD\_GRAYSCALE);

if (inputImage.empty()) {

std::cerr << "❌ Error: Could not load image at path: " << imagePath << std::endl;

return -1;

}

std::cout << "✅ Image loaded successfully. Size: "

<< inputImage.cols << "x" << inputImage.rows << std::endl;

// Compress image using CUDA

compressImage(inputImage);

// Save output image

std::string outputPath = "compressed\_image.jpg";

cv::imwrite(outputPath, inputImage);

std::cout << "✅ Compressed image saved to: " << outputPath << std::endl;

return 0;

}

7)OpenACC computing PI

save file as- montecarlo\_pi.c

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main() {

int N = 10000000;

int count = 0;

#pragma omp parallel for reduction(+:count)

for (int i = 0; i < N; i++) {

// Simple deterministic pseudo-random numbers based on index

float x = (float)(i \* 123456789 % 100000) / 100000.0f;

float y = (float)(i \* 362436069 % 100000) / 100000.0f;

if (x \* x + y \* y <= 1.0f) {

count++;

}

}

float pi = 4.0f \* count / N;

printf("Estimated Pi = %f\n", pi);

return 0;

}

**run commands-**

gcc -fopenmp -O2 montecarlo\_pi.c -o pi

./pi

9)**OneAPI SYCL Computing PI with Monte Carlo**

**Installation of oneapi ->**

**1. install ->https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit-download.html**

**2. chmod +x l\_BaseKit\_p\_2025.0.0.xxx\_offline.sh**

**3. ./l\_BaseKit\_p\_2025.0.0.xxx\_offline.sh**

**4. source /home/it/intel/oneapi/setvars.sh**

**5. which dpcpp**

// montecarlo\_pi\_sycl.cpp

#include <CL/sycl.hpp>

#include <iostream>

#include <random>

using namespace sycl;

int main() {

const size\_t total\_points = 1 << 24;

size\_t inside\_count = 0;

queue q{default\_selector{}};

std::cout << "Running on " << q.get\_device().get\_info<info::device::name>() << "\n";

buffer<size\_t> count\_buf(&inside\_count, 1);

q.submit([&](handler& h) {

auto count\_acc = count\_buf.get\_access<access::mode::atomic>(h);

h.parallel\_for(range<1>{total\_points}, [=](id<1> i) {

size\_t seed = i[0];

float x = (float)((seed \* 16807) % 10000) / 10000.0f;

float y = (float)(((seed + 1) \* 16807) % 10000) / 10000.0f;

if (x\*x + y\*y <= 1.0f) {

count\_acc[0].fetch\_add(1);

}

});

}).wait();

float pi = 4.0f \* inside\_count / total\_points;

std::cout << "Estimated Pi = " << pi << std::endl;

return 0;

}

**Compile command : dpcpp montecarlo\_pi\_sycl.cpp -o montecarlo\_pi**

**Execution command: ./montecarlo\_pi**