# ANSWER SHEET

## Core Algorithm Speedup Justification

Utilizing `gprof`, the runtime of functions was measured and detailed reports were generated to reveal the relationships between function calls as well as the time consumption of each function. The analysis highlighted that the `rotatePGM` function is the primary performance bottleneck, taking up 0.34 seconds of self-time, which amounts to 77.68% of the total runtime, as indicated by the Flat profile table1. In comparison, the `readPGM` and `writePGM` functions each account for about 10% of the execution time, with a self-time of 0.05 seconds for both.

Furthermore, the Call graph table2 shows that the `main` function calls `rotatePGM`, `readPGM`, and `writePGM`, with `rotatePGM` consuming the majority of the execution time. These findings suggest that optimizing the `rotatePGM` function is crucial for enhancing the program's performance since it is where the bulk of time is spent during the core algorithm of image rotation.

| % time | cumulative seconds | self seconds | calls | self ms/call | total ms/call | name |
|---|---|---|---|---|---|---|
| 77.68 | 0.34 | 0.34 | 1 | 341.78 | 341.78 | rotatePGM |
| 11.42 | 0.39 | 0.05 | 1 | 50.26 | 50.26 | readPGM |
| 11.42 | 0.44 | 0.05 | 1 | 50.26 | 50.26 | writePGM |

Table 1: Flat profile

| index | % time | self | children | called | name |
|---|---|---|---|---|---|
| | | | | | <spontaneous> |
| [1] | 100.0 | 0.00 | 0.44 | | main [1] |
| | | 0.34 | 0.00 | 1/1 | rotatePGM [2] |
| | | 0.05 | 0.00 | 1/1 | readPGM [3] |
| | | 0.05 | 0.00 | 1/1 | writePGM [4] |
| | | 0.34 | 0.00 | 1/1 | main [1] |
| [2] | 77.3 | 0.34 | 0.00 | 1 | rotatePGM [2] |
| | | 0.05 | 0.00 | 1/1 | main [1] |
| [3] | 11.4 | 0.05 | 0.00 | 1 | readPGM [3] |
| | | 0.05 | 0.00 | 1/1 | main [1] |
| [4] | 11.4 | 0.05 | 0.00 | 1 | writePGM [4] |

Table 2: Call graph

## Parallel implementations

# MPI

## rotate_mpi.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <mpi.h>

#define PI 3.1415926
#define FILENAME "im.pgm"

// Define a structure to store information about the PGM image
typedef struct {
    int width; // Width of the image
    int height; // Height of the image
    int maxval; // Maximum grayscale value of the image
    unsigned char* data; // Pixel data of the image stored in a one-dimensional array
} PGMImage;
```

```c
// Function to read a PGM image file and store its information in a PGMImage structure
void readPGM(PGMImage* image, const char* filename) {
    FILE* fp = fopen(filename, "r"); // Open the file in text mode
    if (fp == NULL) { // If the file opening fails, print an error message and exit the program
        perror("Cannot open file to read");
        exit(EXIT_FAILURE);
    }

    char ch; // Variable to store characters from the file
    int i; // Loop counter

    // Read the first line of the file and check if it is the P2 identifier
    if (fscanf(fp, "%c%c", &ch, &ch) != 2) {
        fprintf(stderr, "Error reading file header\n");
        fclose(fp);
        exit(EXIT_FAILURE);
    }
    if (ch != '2') { // If it is not the P2 identifier, print an error message and exit the program
        fprintf(stderr, "Not a valid P2 PGM file\n");
        exit(EXIT_FAILURE);
    }

    // Skip the newline character after the first line of the file
    fgetc(fp);

    // Skip comment lines in the file, if any
    while ((ch = fgetc(fp)) == '#') {
        while (fgetc(fp) != '\n');
    }

    // Put the last read character back into the file stream for later use with fscanf
    ungetc(ch, fp);

    // Read the width, height, and maximum grayscale value of the image from the file
    if (fscanf(fp, "%d%d%d", &image->width, &image->height, &image->maxval) != 3) {
        fprintf(stderr, "Error reading image size and maxval\n");
        fclose(fp);
        exit(EXIT_FAILURE);
    }

    // Dynamically allocate memory for storing the pixel data based on the width and height of the image
    image->data = (unsigned char*)malloc(image->width * image->height * sizeof(unsigned char));
    if (image->data == NULL) {
        fprintf(stderr, "Failed to allocate memory for pixel data\n");
        fclose(fp);
        exit(EXIT_FAILURE);
    }

    // Read the pixel data of the image from the file and store it in a one-dimensional array
    for (i = 0; i < image->width * image->height; i++) {
        if (fscanf(fp, "%hhu", &image->data[i]) != 1) {
            fprintf(stderr, "Error reading pixel data\n");
            fclose(fp);
            free(image->data);
            exit(EXIT_FAILURE);
        }
    }

    // Close the file
    fclose(fp);
}
```

```c
// Function to write a PGM image file, writing the information from a PGMImage structure to the file
void writePGM(PGMImage* image, const char* filename) {
    FILE* fp = fopen(filename, "w"); // Open the file in text mode
    if (fp == NULL) { // If the file opening fails, print an error message and exit the program
        perror("Cannot open file to write");
        exit(EXIT_FAILURE);
    }

    int i; // Loop counter

    // Write the P2 identifier to the first line of the file
    fprintf(fp, "P2\n");

    // Write the width and height of the image to the second line of the file
    fprintf(fp, "%d %d\n", image->width, image->height);

    // Write the maximum grayscale value of the image to the third line of the file
    fprintf(fp, "%d\n", image->maxval);

    // Write the pixel data of the image to the file, separating each value with a space
    for (i = 0; i < image->width * image->height; i++) {
        fprintf(fp, "%hhu ", image->data[i]);
    }

    // Close the file
    fclose(fp);
}
```

```c
void rotatePGM(PGMImage* src, PGMImage* dst, double angle, int rank, int size) {
    double radian = angle * PI / 180.0;
    double cosine = cos(radian);
    double sine = sin(radian);

    int ori_centre_x = src->width / 2;
    int ori_centre_y = src->height / 2;
    dst->width = (int)(fabs(src->width * cosine) + fabs(src->height * sine));
    dst->height = (int)(fabs(src->height * cosine) + fabs(src->width * sine));
    dst->maxval = src->maxval;

    if (rank == 0) {
        dst->data = (unsigned char*)malloc(dst->width * dst->height * sizeof(unsigned char));
        memset(dst->data, 0, dst->width * dst->height * sizeof(unsigned char));
    }

    int rows_per_process = (dst->height + size - 1) / size;
    int start_row = rank * rows_per_process;
    int end_row = (rank + 1) * rows_per_process;
    if (end_row > dst->height) end_row = dst->height;

    unsigned char* local_buffer = (unsigned char*)malloc(dst->width * rows_per_process * sizeof(unsigned char));
    memset(local_buffer, 0, dst->width * rows_per_process * sizeof(unsigned char));

    for (int i = start_row; i < end_row; ++i) {
        for (int j = 0; j < dst->width; ++j) {
            int ori_x = (int)((i - dst->height / 2) * cosine - (j - dst->width / 2) * sine) + ori_centre_x;
            int ori_y = (int)((i - dst->height / 2) * sine + (j - dst->width / 2) * cosine) + ori_centre_y;

            if (ori_x >= 0 && ori_x < src->width && ori_y >= 0 && ori_y < src->height) {
                local_buffer[(i - start_row) * dst->width + j] = src->data[ori_y * src->width + ori_x];
            }
        }
    }

    MPI_Gather(local_buffer, dst->width * rows_per_process, MPI_UNSIGNED_CHAR,
               dst->data, dst->width * rows_per_process, MPI_UNSIGNED_CHAR,
               0, MPI_COMM_WORLD);

    free(local_buffer);
}
```

```c
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc != 2) {
        if (rank == 0) {
            fprintf(stderr, "Usage: %s <rotation_angle>\n", argv[0]);
        }
        MPI_Finalize();
        return EXIT_FAILURE;
    }

    double angle = atof(argv[1]);
    PGMImage src, dst;

    if (rank == 0) {
        readPGM(&src, FILENAME);
    }

    MPI_Bcast(&src.width, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&src.height, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&src.maxval, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank != 0) {
        src.data = (unsigned char*)malloc(src.width * src.height * sizeof(unsigned char));
    }

    MPI_Bcast(src.data, src.width * src.height, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

    double start_time, end_time;
    if (rank == 0) {
        start_time = MPI_Wtime();
    }

    rotatePGM(&src, &dst, angle, rank, size);

    if (rank == 0) {
        end_time = MPI_Wtime();
        printf("Execution time: %f seconds\n", end_time - start_time);

        char new_filename[256];
        sprintf(new_filename, "rotated_%s", FILENAME);
        writePGM(&dst, new_filename);

        free(src.data);
        free(dst.data);
    }

    MPI_Finalize();
    return 0;
}
```

### Makefile

```
# Compiler settings - change these if necessary
MPICC=mpicc

# Target executable name
TARGET=rotate_mpi

# Source files
SRC=rotate_mpi.c

# Compiler flags
CFLAGS=-Wall

# Libraries
LIBS=-lm -lmpiP -lbfd -ldl

# mpiP library path
MPIPLIBPATH=/home/nvidia/Desktop/mpiP-3.5/mpiP-3.5/lib

# LDFLAGS for linking, include mpiP library path
LDFLAGS=-L$(MPIPLIBPATH)

# Rule to build the target
$(TARGET): $(SRC)
    $(MPICC) $(CFLAGS) $(LDFLAGS) -o $(TARGET) $(SRC) $(LIBS)

# Rule for cleaning the project
clean:
    rm -f $(TARGET)

# Phony targets for clean
.PHONY: clean
```

## CUDA

### rotate_cuda.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>

#define PI 3.1415926 // Define the value of pi
#define FILENAME "im.pgm" // Define the name of the PGM image file

// Define a structure to store information about the PGM image
typedef struct {
    int width; // Width of the image
    int height; // Height of the image
    int maxval; // Maximum grayscale value of the image
    unsigned char* data; // Pixel data of the image stored in a one-dimensional array
} PGMImage;
```

```c
// Function to read a PGM image file and store its information in a PGMImage structure
void readPGM(PGMImage* image, const char* filename) {
    FILE* fp = fopen(filename, "r"); // Open the file in text mode
    if (fp == NULL) { // If the file opening fails, print an error message and exit the program
        perror("Cannot open file to read");
        exit(EXIT_FAILURE);
    }

    char ch; // Variable to store characters from the file
    int i; // Loop counter

    // Read the first line of the file and check if it is the P2 identifier
    if (fscanf(fp, "%c%c", &ch, &ch) != 2) {
        fprintf(stderr, "Error reading file header\n");
        fclose(fp);
        exit(EXIT_FAILURE);
    }
    if (ch != '2') { // If it is not the P2 identifier, print an error
//message and exit the program
        fprintf(stderr, "Not a valid P2 PGM file\n");
        exit(EXIT_FAILURE);
    }

    // Skip the newline character after the first line of the file
    fgetc(fp);

    // Skip comment lines in the file, if any
    while ((ch = fgetc(fp)) == '#') {
        while (fgetc(fp) != '\n');
    }

    // Put the last read character back into the file stream for later use with fscanf
    ungetc(ch, fp);

    // Read the width, height, and maximum grayscale value of the image from the file
    if (fscanf(fp, "%d%d%d", &image->width, &image->height, &image->maxval) != 3) {
        fprintf(stderr, "Error reading image size and maxval\n");
        fclose(fp);
        exit(EXIT_FAILURE);
    }

    // Dynamically allocate memory for storing the pixel
//data based on the width and height of the image
    image->data = (unsigned char*)malloc(image->width * image->height * sizeof(unsigned char));
    if (image->data == NULL) {
        fprintf(stderr, "Failed to allocate memory for pixel data\n");
        fclose(fp);
        exit(EXIT_FAILURE);
    }

    // Read the pixel data of the image from the file and store it in a one-dimensional array
    for (i = 0; i < image->width * image->height; i++) {
        if (fscanf(fp, "%hhu", &image->data[i]) != 1) {
            fprintf(stderr, "Error reading pixel data\n");
            fclose(fp);
            free(image->data);
            exit(EXIT_FAILURE);
        }
    }

    // Close the file
    fclose(fp);
}
```

```
// Function to write a PGM image file, writing the information from a PGMImage
//structure to the file
void writePGM(PGMImage* image, const char* filename) {
    FILE* fp = fopen(filename, "w"); // Open the file in text mode
    if (fp == NULL) { // If the file opening fails,
//print an error message and exit the program
        perror("Cannot open file to write");
        exit(EXIT_FAILURE);
    }

    int i; // Loop counter

    // Write the P2 identifier to the first line of the file
    fprintf(fp, "P2\n");

    // Write the width and height of the image to the second line of the file
    fprintf(fp, "%d %d\n", image->width, image->height);

    // Write the maximum grayscale value of the image to the third line of the file
    fprintf(fp, "%d\n", image->maxval);

    // Write the pixel data of the image to the file, separating each value with a space
    for (i = 0; i < image->width * image->height; i++) {
        fprintf(fp, "%hhu ", image->data[i]);
    }

    // Close the file
    fclose(fp);
}

__global__ void rotatePGMKernel(unsigned char* src, unsigned char* dst, int srcWidth,
int srcHeight, int dstWidth, int dstHeight, double cosine, double sine,
double ori_centre_x, double ori_centre_y, double new_centre_x, double new_centre_y) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < dstWidth && y < dstHeight) {
        double sx = (y - new_centre_y) * sine + (x - new_centre_x) * cosine + ori_centre_x;
        double sy = (y - new_centre_y) * cosine - (x - new_centre_x) * sine + ori_centre_y;

        if (sx >= 0 && sx < srcWidth && sy >= 0 && sy < srcHeight) {
            int x1 = (int)(sx + 0.5);
            int y1 = (int)(sy + 0.5);
            dst[y * dstWidth + x] = src[y1 * srcWidth + x1];
        } else {
            dst[y * dstWidth + x] = 0;
        }
    }
}
```

```
void rotatePGM(PGMImage* src, PGMImage* dst, double angle) {
    // Create CUDA events for timing
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Start timing the rotatePGM function
    cudaEventRecord(start);

    double radian = angle * PI / 180;
    double cosine = cos(radian);
    double sine = sin(radian);

    double ori_centre_x = (src->width - 1) / 2.0;
    double ori_centre_y = (src->height - 1) / 2.0;

    dst->height = (int)ceil(fabs(src->height * cosine) + fabs(src->width * sine));
    dst->width = (int)ceil(fabs(src->width * cosine) + fabs(src->height * sine));

    double new_centre_x = (dst->width - 1) / 2.0;
    double new_centre_y = (dst->height - 1) / 2.0;

    dst->maxval = src->maxval;
    dst->data = (unsigned char*)malloc(dst->width * dst->height * sizeof(unsigned char));

    unsigned char *d_src, *d_dst;
    size_t srcSize = src->width * src->height * sizeof(unsigned char);
    size_t dstSize = dst->width * dst->height * sizeof(unsigned char);

    cudaMalloc(&d_src, srcSize);
    cudaMalloc(&d_dst, dstSize);

    cudaMemcpy(d_src, src->data, srcSize, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(32, 32);
    dim3 numBlocks((dst->width + 31) / 32, (dst->height + 31) / 32);
    rotatePGMKernel<<<numBlocks, threadsPerBlock>>>(d_src, d_dst, src->width,
src->height, dst->width, dst->height, cosine, sine, ori_centre_x,
ori_centre_y, new_centre_x, new_centre_y);

    cudaMemcpy(dst->data, d_dst, dstSize, cudaMemcpyDeviceToHost);

    cudaFree(d_src);
    cudaFree(d_dst);

    // Stop timing the rotatePGM function
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    // Calculate and print the elapsed time for rotatePGM function
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("rotatePGM execution time: %f ms\n", milliseconds);

    // Clean up CUDA events
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
}
```

```c
int main() {
    // Create CUDA events for timing the entire program
    cudaEvent_t startProgram, stopProgram;
    cudaEventCreate(&startProgram);
    cudaEventCreate(&stopProgram);

    // Start timing the entire program
    cudaEventRecord(startProgram);

    PGMImage src, dst;
    double angle = 45; // Example rotation angle, replace with desired value or argument

    // Read the source image
    readPGM(&src, FILENAME);

    // Perform the rotation
    rotatePGM(&src, &dst, angle);

    // Write the rotated image to a file
    writePGM(&dst, "rotated_im.pgm");

    // Free allocated memory
    free(src.data);
    free(dst.data);

    // Stop timing the entire program
    cudaEventRecord(stopProgram);
    cudaEventSynchronize(stopProgram);

    // Calculate and print the elapsed time for the entire program
    float totalMilliseconds = 0;
    cudaEventElapsedTime(&totalMilliseconds, startProgram, stopProgram);
    printf("Total program execution time: %f ms\n", totalMilliseconds);

    // Clean up CUDA events
    cudaEventDestroy(startProgram);
    cudaEventDestroy(stopProgram);

    return 0;
}
```

### Makefile

```
# Compiler settings
CC = nvcc
CFLAGS = -O2 -arch=sm_53

# Target executable name
TARGET=rotate_cuda

# Source file
SRC=rotate_cuda.cu

# Build target
all: $(TARGET)

# Compile the program
$(TARGET): $(SRC)
    $(CC) $(CFLAGS) $^ -o $@

# Clean up
clean:
    rm -f $(TARGET)
.PHONY: all clean
```

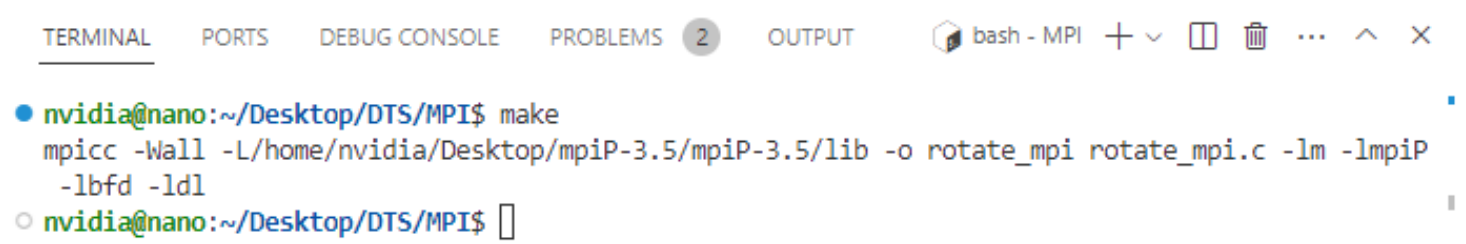
# Performance analysis

## MPI

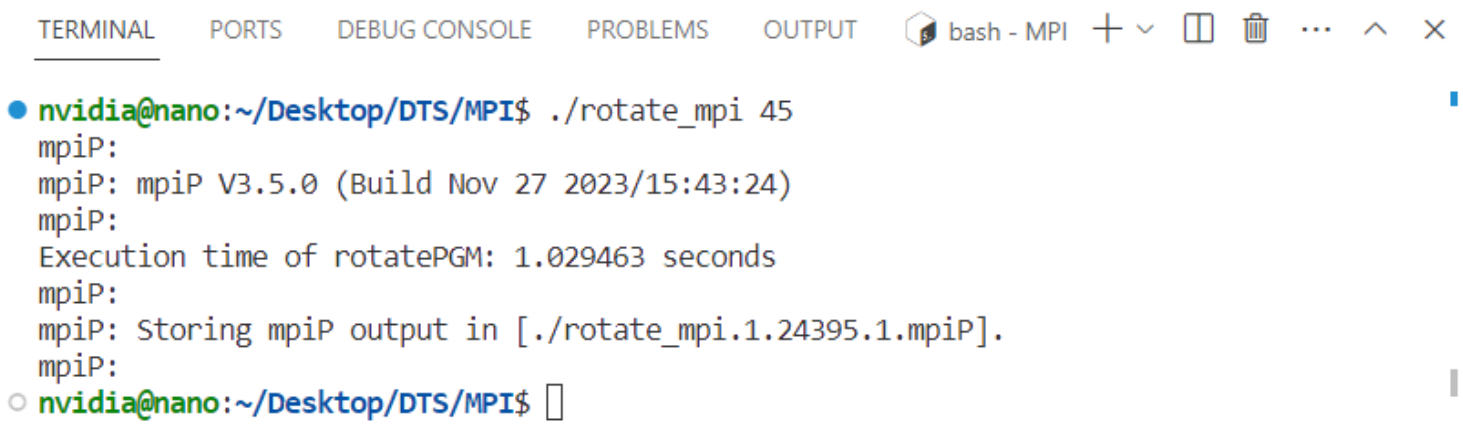
### Prerequisites

**Assumption:**
Assuming a rotation angle of 45 degrees.

```
make
```




```
./rotate_mpi 45
```




**Performance Analysis Tool:**

When conducting performance analysis of MPI parallel methods, **MpiP** was chosen over **gprof** as the primary tool. gprof is primarily designed for single-threaded applications, hence it has limitations in capturing the dynamics of multi-threaded or multi-process activities in parallel or distributed computing environments, especially in handling inter-process communication and synchronization issues. In contrast, **MpiP** (*MPI Profiling*) is a tool specifically developed for analyzing parallel programs based on the *Message Passing Interface (MPI)*, capable of capturing and analyzing in detail the communication and operations between processes. This is crucial for identifying performance bottlenecks in parallel programs, such as communication delays and load imbalances. Additionally, MpiP supports multi-process environments and can process and integrate a broader range of performance data, which gprof lacks. MpiP also provides customized reports specifically for parallel programs, focusing on key aspects of parallel performance, such as load balancing and communication efficiency, thus addressing areas where gprof falls short. Reports can be generated by running the followimg command in the terminal.

```
mpirun -np [number of threads] ./rotate-image 45
```

**Runtime Measurement:**

When measuring runtime, the MPI (Message Passing Interface) library function `MPI_Wtime` is used instead of the `ctime` function from the C standard library. Compared to `ctime`, `MPI_Wtime` measures wall time, which is the actual elapsed time from the start to the end of the program's execution. In contrast, `ctime` calculates the running time indirectly by obtaining the current calendar time, i.e., capturing the time points at the start and end of the program. Therefore, `MPI_Wtime` offers higher precision. Additionally, `MPI_Wtime` is integrated into parallel programs using MPI, providing a seamless user experience and avoiding the complexity of introducing external timing mechanisms, thus ensuring consistent time measurement.

**Device Parameter(CPU):**

The device used for running the code is **Jetson Nano**, a compact yet powerful computing device developed by **NVIDIA**, specifically designed for artificial intelligence and machine learning applications.

The Jetson Nano is based on the ARM architecture and is equipped with 4 *Cortex-A57* type CPU cores. It reaches a maximum CPU frequency of 1479 MHz and a minimum of 102 MHz, and has a substantial 2048K L2 cache, making it highly suitable for computationally intensive tasks. Below are the detailed CPU specifications of the device:

```
lscpu
```

Table 3: Jetson Nano CPU Specifications

| | |
|---|---|
| **Architecture:** | aarch64 |
| **Byte Order:** | Little Endian |
| **CPU(s):** | 4 |
| **On-line CPU(s) list:** | 0-3 |
| **Thread(s) per core:** | 1 |
| **Core(s) per socket:** | 4 |
| **Socket(s):** | 1 |
| **Vendor ID:** | ARM |
| **Model:** | 1 |
| **Model name:** | Cortex-A57 |
| **Stepping:** | r1p1 |
| **CPU max MHz:** | 1479.0000 |
| **CPU min MHz:** | 102.0000 |
| **BogoMIPS:** | 38.40 |
| **L1d cache:** | 32K |
| **L1i cache:** | 48K |
| **L2 cache:** | 2048K |
| **Flags:** | fp asimd evtstrm aes pmull sha1 sha2 crc32 |

The CPU of the device does not support hyper-threading technology, and each core can handle only one thread. Therefore, the number of threads equals the number of cores, which is 4. Consequently, when running applications, the specified number of threads should be 1, 2, 3, or 4. By running the followimg commands in the terminal respectively.

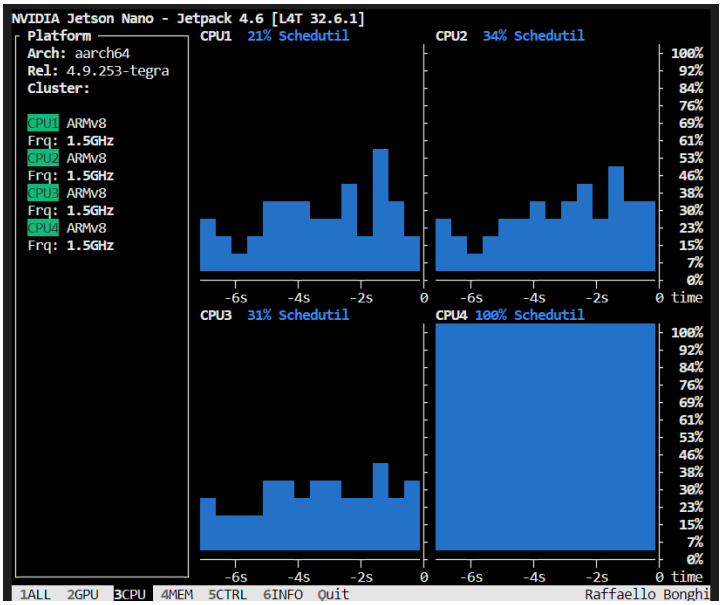
```
mpirun -np 1 ./rotate_mpi 45
```

```
mpirun -np 2 ./rotate_mpi 45
```

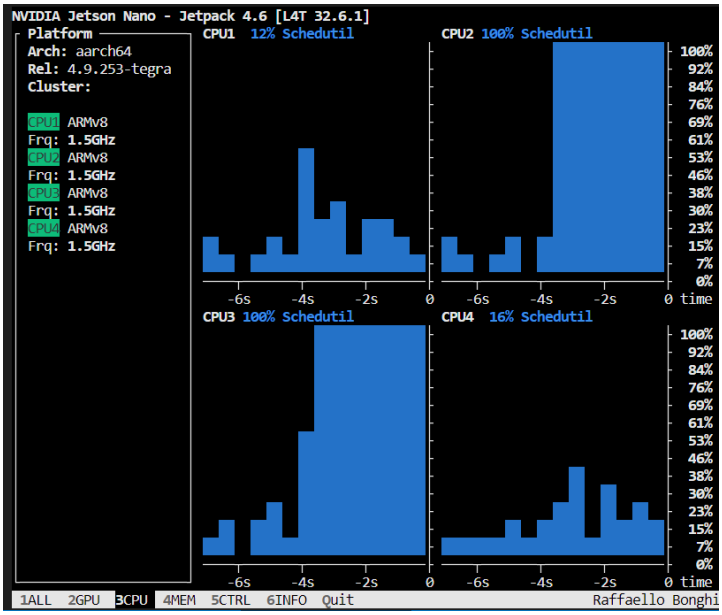
```
mpirun -np 3 ./rotate_mpi 45
```

```
mpirun -np 4 ./rotate_mpi 45
```

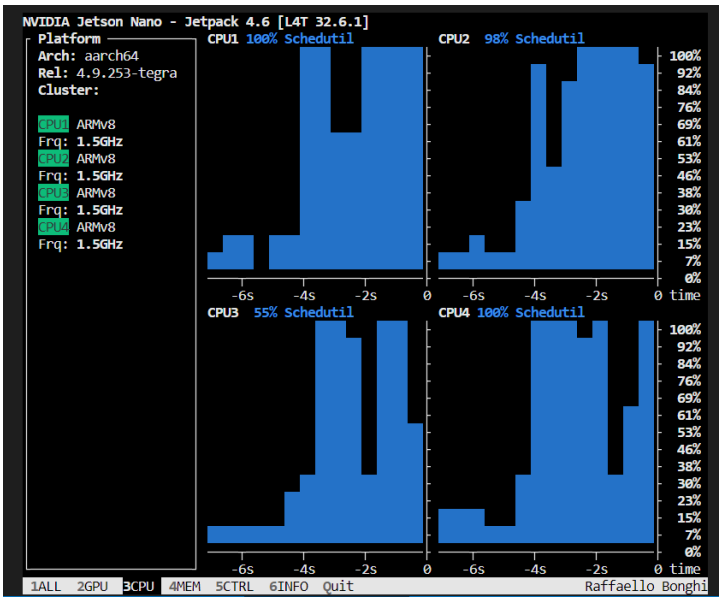
**CPU Core Utilization Across Different Threads Monitored Using jtop**



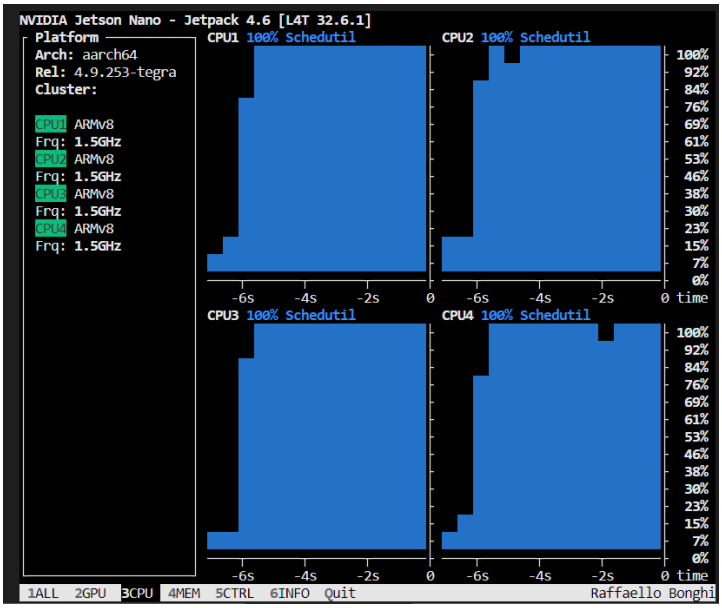

(a) The number of threads is 1




(b) The number of threads is 2




(c) The number of threads is 3




(d) The number of threads is 4

## Results

### Image Post-Rotation via MPI Parallelization:

**MpiP:**

| Task | AppTime | MPITime | MPI% |
|---|---|---|---|
| 0 | 16 | 0.000172 | 0.00 |
| * | 16 | 0.000172 | 0.00 |

Table 4: MPI Time in seconds.

| Call | Site | Time | App% | MPI% | Count | COV |
|---|---|---|---|---|---|---|
| Bcast | 1 | 0.172 | 0.00 | 100.00 | 4 | 0.00 |

Table 5: Aggregate Time in milliseconds.

| Call | Site | Count | Total | Avrg |
|---|---|---|---|---|
| Bcast | 1 | 4 | 1.6e+07 | 4e+06 |

Table 6: Aggregate Sent Message Size in bytes.

| Name | Site | Rank | Count | Max | Mean | Min | App% | MPI% |
|---|---|---|---|---|---|---|---|---|
| Bcast | 1 | 0 | 4 | 0.149 | 0.043 | 0.00672 | 0.00 | 100.00 |
| Bcast | 1 | * | 4 | 0.149 | 0.043 | 0.00672 | 0.00 | 100.00 |

Table 7: Callsite Time Statistics in milliseconds.

Table 8: The number of threads is 1

| Task | AppTime | MPITime | MPI% |
|---|---|---|---|
| 0 | 15.4 | 0.013 | 0.08 |
| 1 | 5.71 | 5.33 | 93.34 |
| * | 21.1 | 5.34 | 25.30 |

Table 9: MPI Time in seconds.

| Call | Site | Time | App% | MPI% | Count | COV |
|---|---|---|---|---|---|---|
| Bcast | 1 | 5.34e+03 | 25.30 | 100.00 | 8 | 1.41 |

Table 10: Aggregate Time in milliseconds.

| Call | Site | Count | Total | Avrg |
|---|---|---|---|---|
| Bcast | 1 | 8 | 3.2e+07 | 4e+06 |

Table 11: Aggregate Sent Message Size in bytes.

| Name | Site | Rank | Count | Max | Mean | Min | App% | MPI% |
|---|---|---|---|---|---|---|---|---|
| Bcast | 1 | 0 | 4 | 12.8 | 3.26 | 0.00859 | 0.08 | 100.00 |
| Bcast | 1 | 1 | 4 | 5.31e+03 | 1.33e+03 | 0.00995 | 93.34 | 100.00 |
| Bcast | 1 | * | 8 | 5.31e+03 | 667 | 0.00859 | 25.30 | 100.00 |

Table 12: Callsite Time Statistics in milliseconds.

Table 13: The number of threads is 2

| Task | AppTime | MPITime | MPI% |
|---|---|---|---|
| 0 | 15.2 | 0.0259 | 0.17 |
| 1 | 5.59 | 5.33 | 95.45 |
| 2 | 5.59 | 5.33 | 95.46 |
| * | 26.4 | 10.7 | 40.51 |

Table 14: MPI Time in seconds.

| Call | Site | Time | App% | MPI% | Count | COV |
|---|---|---|---|---|---|---|
| Bcast | 1 | 1.07e+04 | 40.51 | 100.00 | 12 | 0.86 |

Table 15: Aggregate Time in milliseconds.

| Call | Site | Count | Total | Avrg |
|---|---|---|---|---|
| Bcast | 1 | 12 | 4.8e+07 | 4e+06 |

Table 16: Aggregate Sent Message Size in bytes.

| Name | Site | Rank | Count | Max | Mean | Min | App% | MPI% |
|---|---|---|---|---|---|---|---|---|
| Bcast | 1 | 0 | 4 | 25.6 | 6.46 | 0.00943 | 0.17 | 100.00 |
| Bcast | 1 | 1 | 4 | 5.31e+03 | 1.33e+03 | 0.0101 | 95.45 | 100.00 |
| Bcast | 1 | 2 | 4 | 5.31e+03 | 1.33e+03 | 0.00906 | 95.46 | 100.00 |
| Bcast | 1 | * | 12 | 5.31e+03 | 891 | 0.00906 | 40.51 | 100.00 |

Table 17: Callsite Time Statistics in milliseconds.

Table 18: The number of threads is 3

| Task | AppTime | MPITime | MPI% |
|---|---|---|---|
| 0 | 15.5 | 0.0406 | 0.26 |
| 1 | 5.59 | 5.4 | 96.59 |
| 2 | 5.6 | 5.4 | 96.51 |
| 3 | 5.6 | 5.4 | 96.40 |
| * | 32.3 | 16.2 | 50.34 |

Table 19: MPI Time in seconds.

| Call | Site | Time | App% | MPI% | Count | COV |
|---|---|---|---|---|---|---|
| Bcast | 1 | 1.62e+04 | 50.34 | 100.00 | 16 | 0.66 |

Table 20: Aggregate Time in milliseconds.

| Call | Site | Count | Total | Avrg |
|---|---|---|---|---|
| Bcast | 1 | 16 | 6.4e+07 | 4e+06 |

Table 21: Aggregate Sent Message Size in bytes.

| Name | Site | Rank | Count | Max | Mean | Min | App% | MPI% |
|---|---|---|---|---|---|---|---|---|
| Bcast | 1 | 0 | 4 | 39.8 | 10.2 | 0.0157 | 0.26 | 100.00 |
| Bcast | 1 | 1 | 4 | 5.36e+03 | 1.35e+03 | 0.0318 | 96.59 | 100.00 |
| Bcast | 1 | 2 | 4 | 5.36e+03 | 1.35e+03 | 0.0321 | 96.51 | 100.00 |
| Bcast | 1 | 3 | 4 | 5.36e+03 | 1.35e+03 | 0.0284 | 96.40 | 100.00 |
| Bcast | 1 | * | 16 | 5.36e+03 | 1.02e+03 | 0.0157 | 50.34 | 100.00 |

Table 22: Callsite Time Statistics in milliseconds.

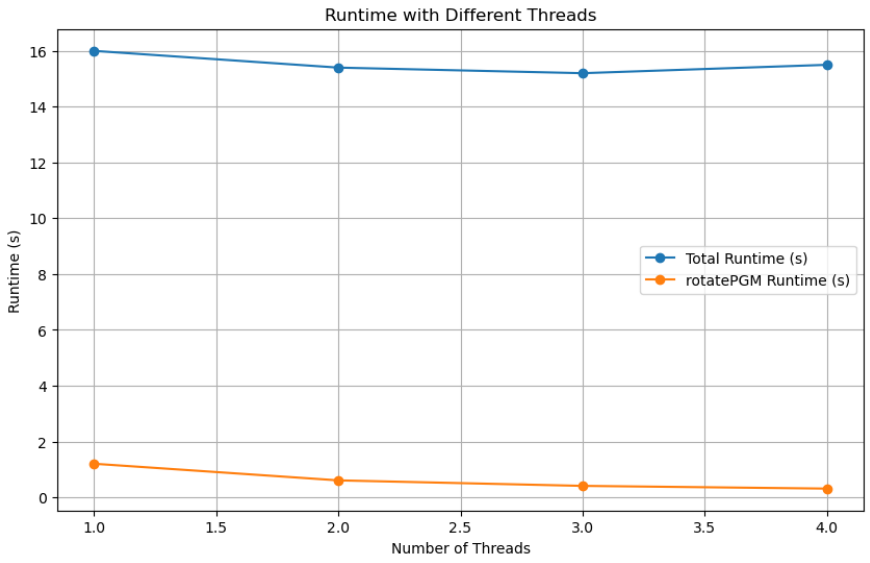
Table 23: The number of threads is 4

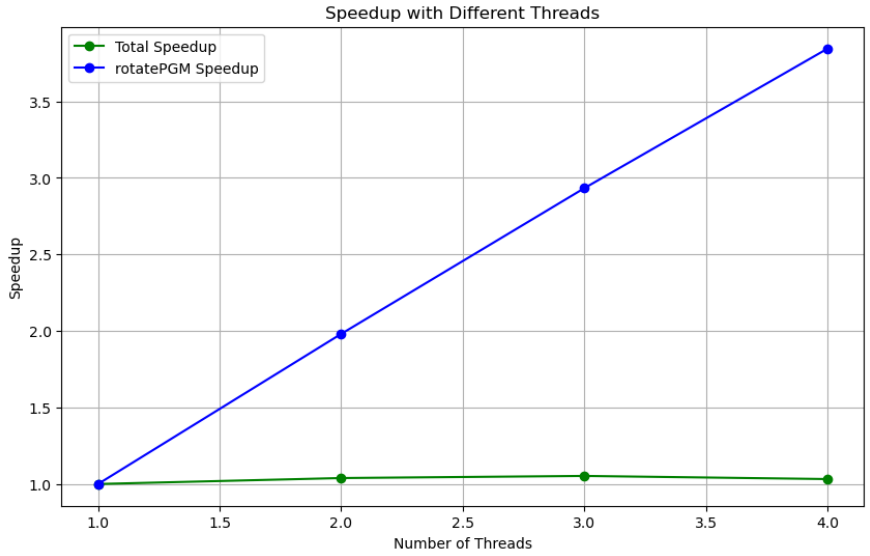
**MPI's timing functions(MPI_Wtime):**

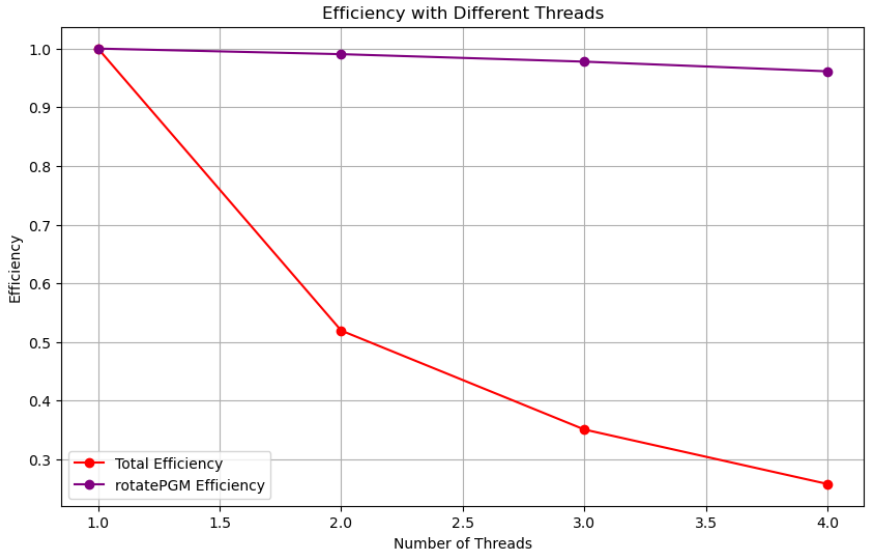
| Number of Threads | Total Program Runtime (s) | rotatePGM Runtime (s) |
|---|---|---|
| 1 | 16 | 1.204163 |
| 2 | 15.4 | 0.607861 |
| 3 | 15.2 | 0.410521 |
| 4 | 15.5 | 0.313175 |

Table 24: MPI Runtime with Different Threads

**Analysis**






# CUDA

## Prerequisites

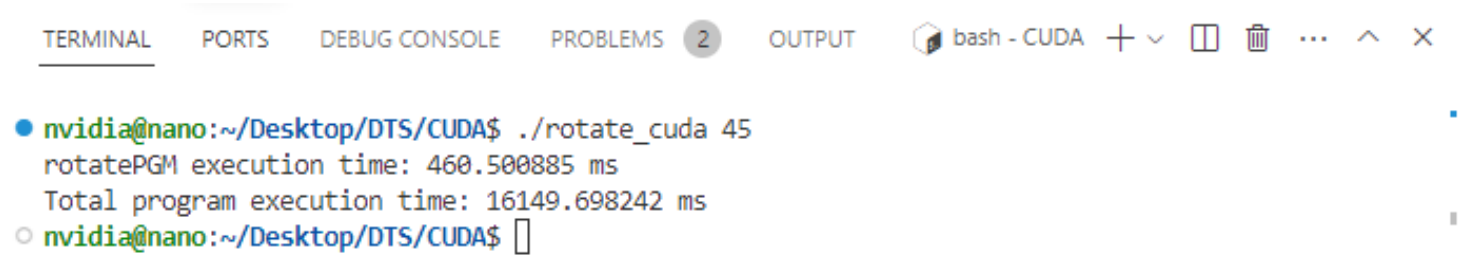
### Assumption:

Assuming a rotation angle of 45 degrees.

```
make
```

```
TERMINAL  PORTS  DEBUG CONSOLE  PROBLEMS 2  OUTPUT  bash - CUDA
nvidia@nano:~/Desktop/DTS/CUDA$ make
nvcc -O2 -arch=sm_53 rotate_cuda.cu -o rotate_cuda
nvidia@nano:~/Desktop/DTS/CUDA$
```

```
./rotate_cuda 45
```

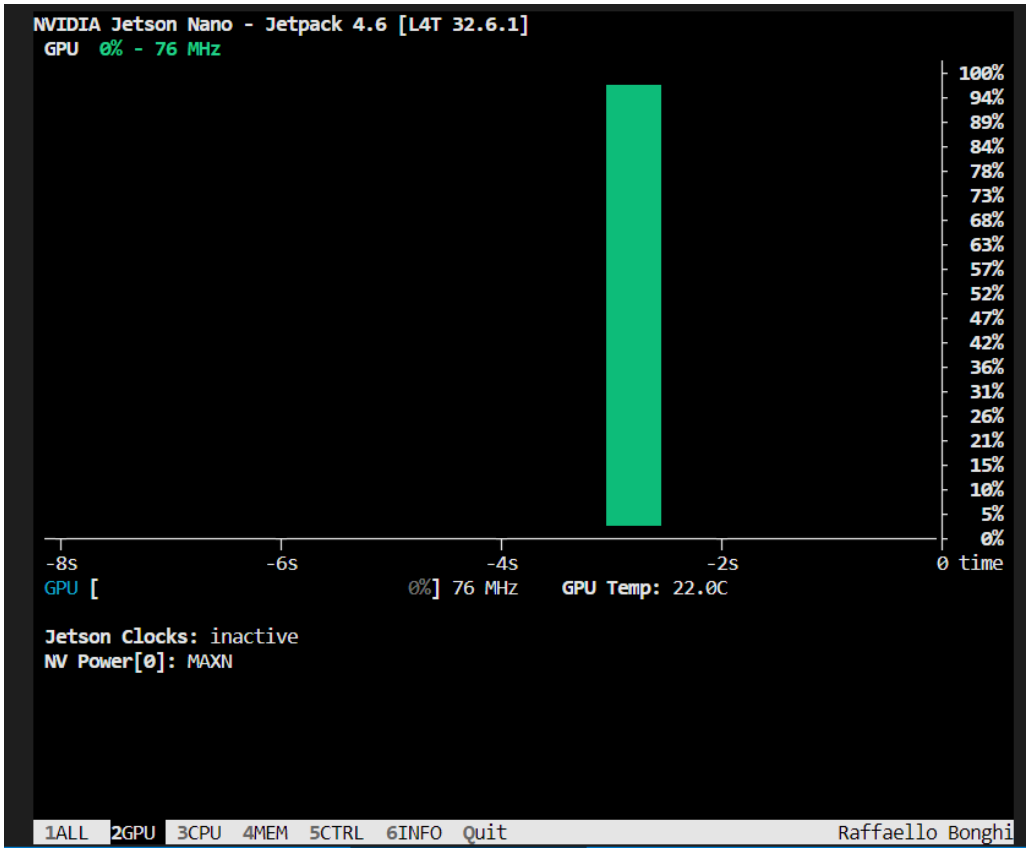



**Performance Analysis Tool:**

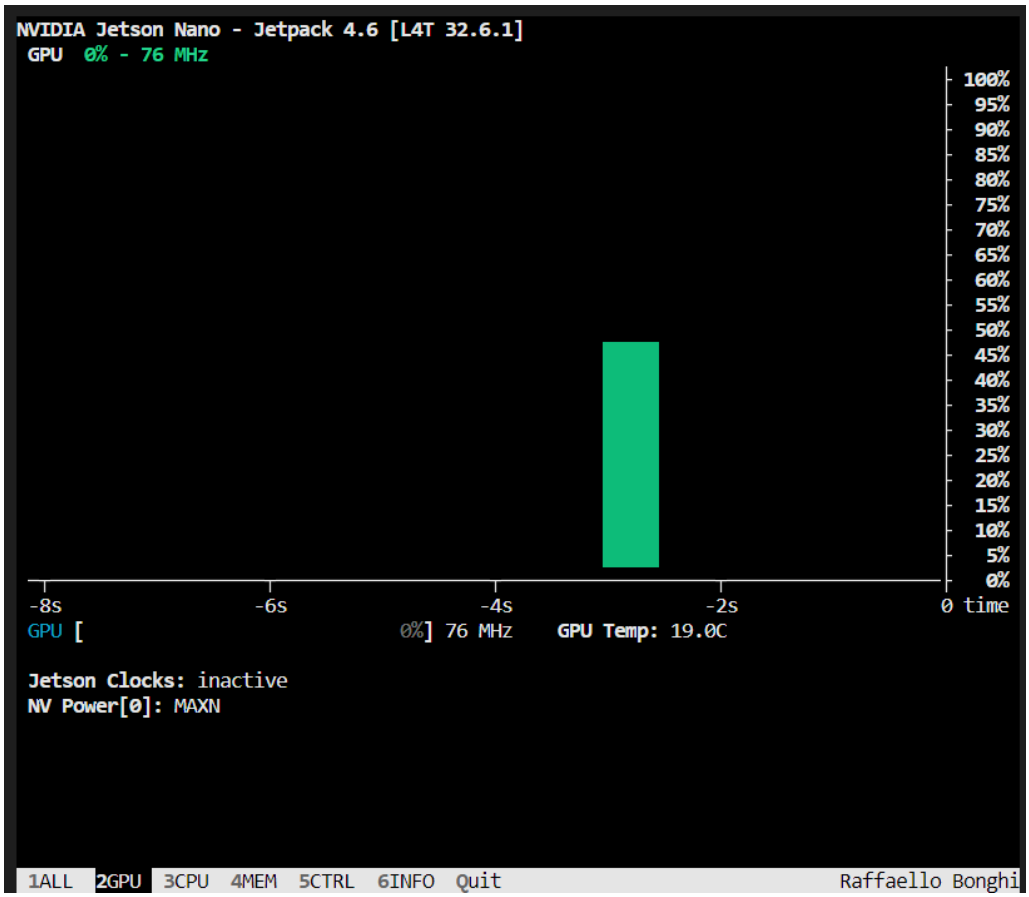
When conducting performance analysis of parallel computing implemented with CUDA code, **NVIDIA Nsight Systems** was chosen over other general profiling tools, primarily due to its significant advantages in capturing the complex dynamics of multi-threaded and multi-process activities in GPU-accelerated computing. It provides an in-depth analysis of GPU execution, including kernel launches, execution configurations, and memory usage.By executing the following command in the terminal, a performance analysis report can be generated:

```
sudo nsys profile --stats=true ./rotate_cuda 45
```


**GPU Utilization Under Different Thread Counts as Monitored by jtop**



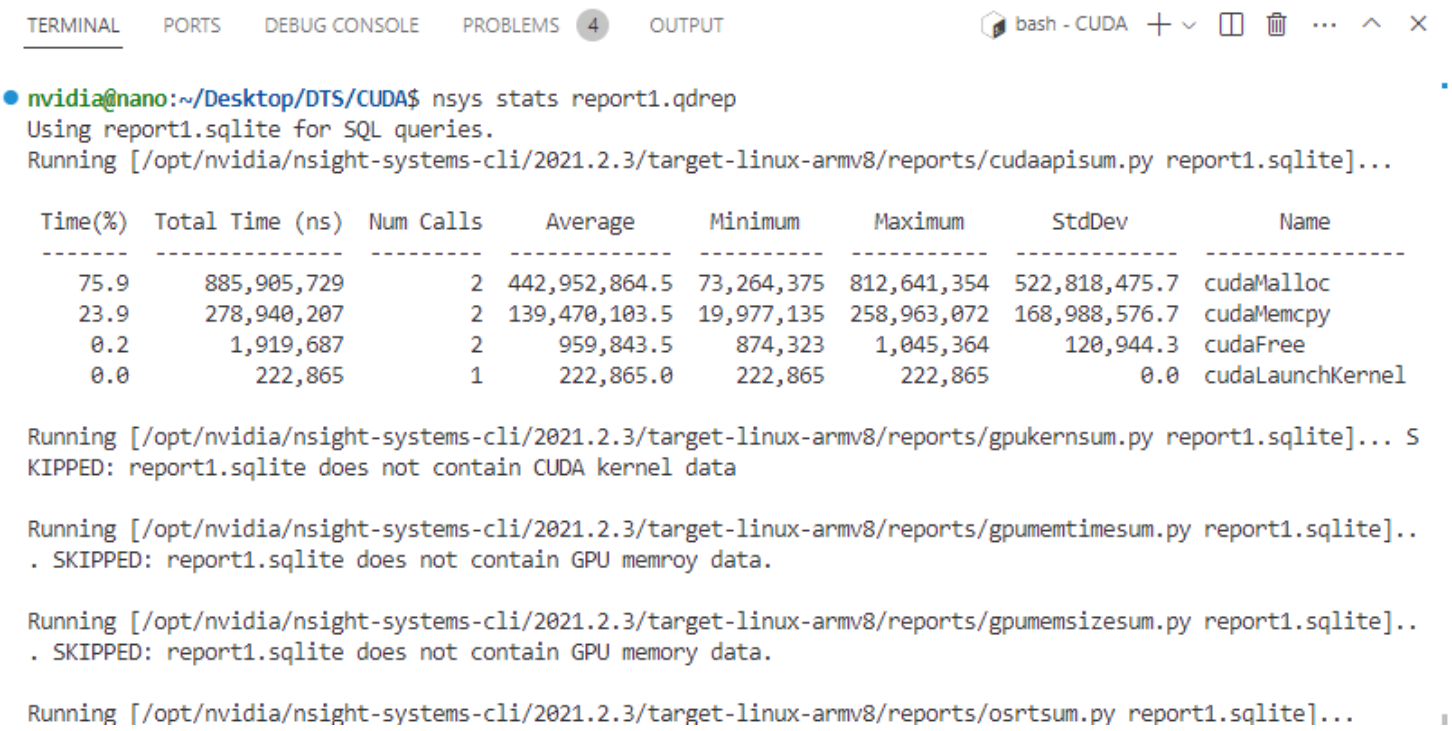

4*4 threads



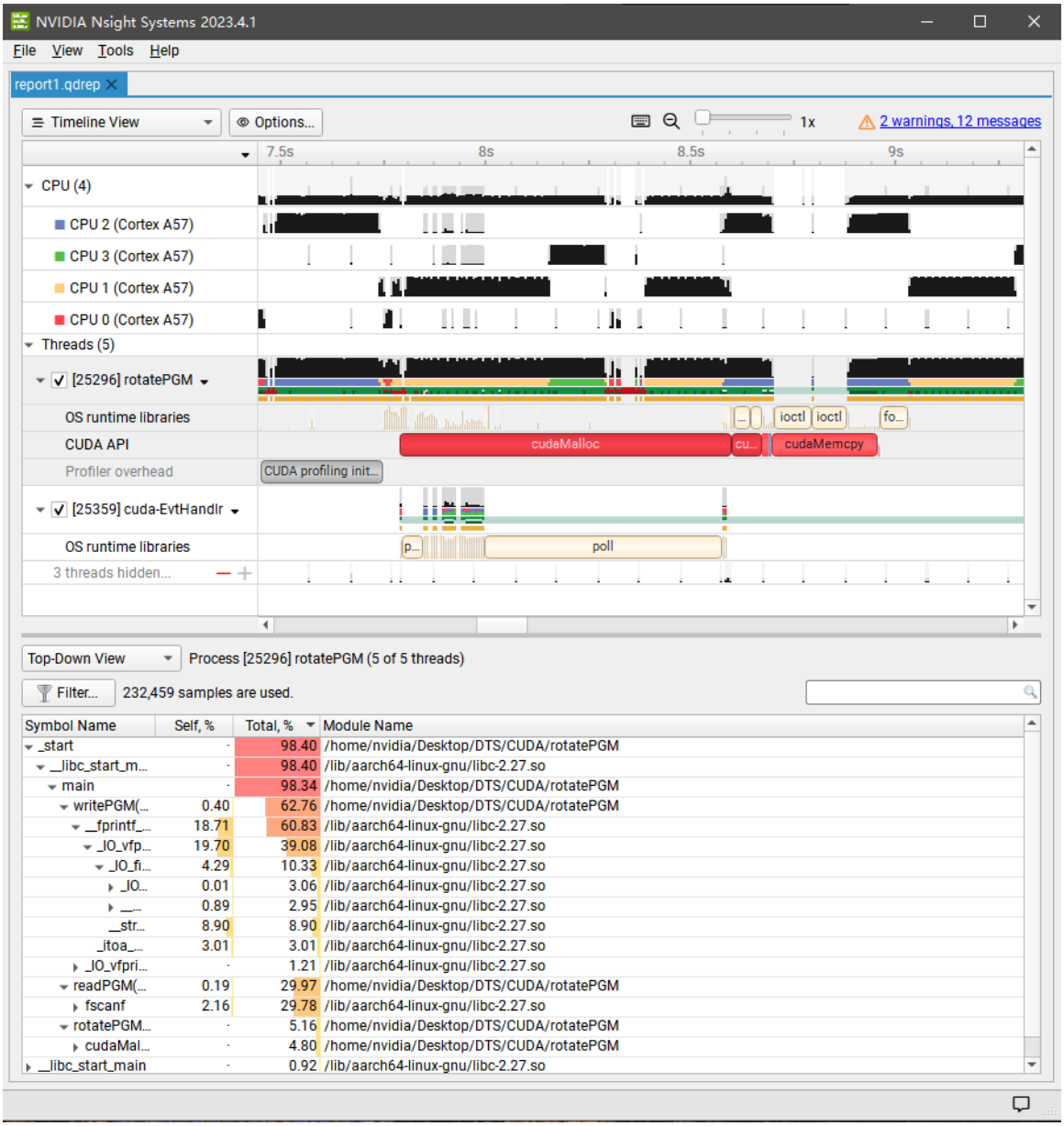

32*32threads

After execution, this will create a `report.qdrep` and `report.sqlite` file. However, these files cannot be directly viewed, especially the `report.qdrep` file, which is a file format specially developed by NVIDIA. Thus, it cannot be opened with standard file viewers or editors. By executing:

```
nsys stats reportX.qdrep
```




The performance data from the file can be extracted and output as key performance statistics in text format. Additionally, analysis can also be performed using the graphical user interface (GUI) of NVIDIA Nsight Systems, which offers intuitive visualization. In the GUI, the timeline view integrates CPU and GPU activities, facilitating the precise identification of performance bottlenecks across CPU and GPU operations. The lower part is the advanced analysis feature, effectively tracking CUDA API calls, load balancing, and resource utilization.




**Runtime Measurement:**

When measuring execution time, it is preferable to use the CUDA time measurement function `cudaEventElapsedTime` over the C language standard library's `ctime`. This is because, in comparison to `ctime`, CUDA's time measurement methods are optimized specifically for GPU-accelerated computation, providing higher precision and suitability. `cudaEventElapsedTime` measures the wall time on the GPU, which is the actual elapsed time from the start to the end of a CUDA program or kernel execution, accurately reflecting the actual time taken by GPU operations. `ctime` indirectly calculates the runtime by obtaining the current calendar time, which may not be as precise when measuring GPU-accelerated computations and is more suitable for measuring application execution time on the CPU.Furthermore, using CUDA time measurement functions ensures that the measurement process is tightly integrated with the CUDA programming model and GPU execution environment, avoiding the complexity of introducing external time measurement mechanisms in GPU-accelerated applications. This ensures more consistent time measurements.

**Device Parameter(GPU):**

The GPU of the Jetson Nano utilizes the NVIDIA Maxwell architecture with 128 cores. It offers a floating-point performance of 472 GFLOPs (FP16), demonstrating its capability in handling computationally intensive tasks, particularly in AI and machine learning applications.

```
#include <cuda_runtime.h>
#include <stdio.h>

int main() {
    int device;
    cudaDeviceProp properties;

    cudaGetDevice(&device);

    cudaGetDeviceProperties(&properties, device);

    printf("Device Name : %s\n", properties.name);
    printf("Total Global Memory : %zu\n", properties.totalGlobalMem);
    printf("Shared Memory Per Block : %zu\n", properties.sharedMemPerBlock);
    printf("Registers Per Block : %d\n", properties.regsPerBlock);
    printf("Warp Size : %d\n", properties.warpSize);
    printf("Max Threads Per Block : %d\n", properties.maxThreadsPerBlock);
    printf("Total Constant Memory : %zu\n", properties.totalConstMem);
    printf("Max Threads Dimensions : x = %d, y = %d, z = %d\n",
           properties.maxThreadsDim[0], properties.maxThreadsDim[1], properties.maxThreadsDim[2]);
    printf("Max Grid Dimensions : x = %d, y = %d, z = %d\n",
           properties.maxGridSize[0], properties.maxGridSize[1], properties.maxGridSize[2]);

    return 0;
}
```

| Property | Value |
|---|---|
| Device Name | NVIDIA Tegra X1 |
| Total Global Memory | 4156661760 bytes (3.87 GB) |
| Shared Memory Per Block | 49152 bytes (48 KB) |
| Registers Per Block | 32768 |
| Warp Size | 32 |
| Max Threads Per Block | 1024 |
| Total Constant Memory | 65536 bytes (64 KB) |
| Max Threads Dimensions (x, y, z) | 1024, 1024, 64 |
| Max Grid Dimensions (x, y, z) | 2147483647, 65535, 65535 |

Table 25: Specifications of NVIDIA GPU

In CUDA, when setting block sizes, it is important to ensure that the width and height of the image to be processed are divisible by the block size for optimal performance. For an image with dimensions of 4000x4000, considering that the maximum number of threads per block is 1024 and must be greater than 1, it is preferable to choose multiples of 8 as the block size. Therefore, block sizes of 4, 8, 16, and 32 have been used. The number of threads equals the square of the block size, resulting in 16, 64, 256, and 1024 threads respectively. In CUDA, the number of threads cannot be specified via the command line; it can

only be set by modifying the CUDA source code file.The CUDA code to set the threads per block and number of blocks is shown below:

```
dim3 threadsPerBlock(block_size, block_size);
dim3 numBlocks((dst->width + block_size - 1) / block_size,
               (dst->height + block_size - 1) / block_size);
```

For instance:

```
dim3 threadsPerBlock(4, 4);
dim3 numBlocks((dst->width + 3) / 4, (dst->height + 3) / 4);
```

```
dim3 threadsPerBlock(8, 8);
dim3 numBlocks((dst->width + 7) / 8, (dst->height + 7) / 8);
```

```
dim3 threadsPerBlock(16, 16);
dim3 numBlocks((dst->width + 15) / 16, (dst->height + 15) / 16);
```

```
dim3 threadsPerBlock(32, 32);
dim3 numBlocks((dst->width + 31) / 32, (dst->height + 31) / 32);
```

After each modification, save the file and recompile using `make`.

**Results**

**Image Post-Rotation via CUDA Parallelization:**

**NVIDIA System Profiler (NSight Systems):**

Table 26: CUDA Profiling Data for 4*4 Threads

Table 27: CUDA API Statistics

| Name | Time(%) | Total Time (ns) | Num Calls | Average | Minimum | Maximum | StdDev |
|---|---|---|---|---|---|---|---|
| cudaMemcpy | 47.6 | 523,040,417 | 2 | 261,520,208.5 | 21,884,063 | 501,156,354 | 338,896,687.0 |
| cudaEventCreate | 46.1 | 506,808,437 | 4 | 126,702,109.3 | 4,010 | 506,776,666 | 253,383,038.0 |
| cudaMalloc | 4.5 | 49,060,520 | 2 | 24,530,260.0 | 15,861,979 | 33,198,541 | 12,258,800.6 |
| cudaEventRecord | 1.7 | 18,221,562 | 4 | 4,555,390.5 | 59,531 | 9,002,344 | 4,993,653.2 |
| cudaFree | 0.2 | 1,789,636 | 2 | 894,818.0 | 813,125 | 976,511 | 115,531.3 |
| cudaLaunchKernel | 0.0 | 290,573 | 1 | 290,573.0 | 290,573 | 290,573 | 0.0 |
| cudaEventDestroy | 0.0 | 113,803 | 4 | 28,450.8 | 3,385 | 93,230 | 43,352.5 |
| cudaEventSynchronize | 0.0 | 29,948 | 2 | 14,974.0 | 14,219 | 15,729 | 1,067.7 |

Table 28: Operating System Runtime API Statistics

| Name | Time(%) | Total Time (ns) | Num Calls | Average | Minimum | Maximum | StdDev |
|---|---|---|---|---|---|---|---|
| sem_timedwait | 94.8 | 22,785,827,598 | 228 | 99,937,840.3 | 60,360,782 | 105,897,916 | 2,660,697.9 |
| ioctl | 2.5 | 602,917,282 | 1,343 | 448,933.2 | 2,084 | 103,890,677 | 5,784,655.8 |
| poll | 2.1 | 499,646,558 | 331 | 1,509,506.2 | 4,688 | 390,081,354 | 21,521,558.6 |
| fopen | 0.3 | 73,570,206 | 17 | 4,327,659.2 | 6,042 | 73,237,239 | 17,757,603.4 |
| fclose | 0.2 | 54,857,083 | 6 | 9,142,847.2 | 3,906 | 54,821,094 | 22,377,679.6 |
| write | 0.0 | 3,505,939 | 331 | 10,592.0 | 2,760 | 79,479 | 8,710.0 |
| sched_yield | 0.0 | 2,505,162 | 972 | 2,577.3 | 990 | 95,573 | 8,163.2 |
| read | 0.0 | 1,103,963 | 334 | 3,305.3 | 1,563 | 25,625 | 2,186.8 |
| mmap | 0.0 | 575,467 | 16 | 35,966.7 | 3,854 | 112,604 | 28,554.1 |
| sem_wait | 0.0 | 349,272 | 5 | 69,854.4 | 39,531 | 123,334 | 39,116.3 |
| pthread_create | 0.0 | 243,177 | 2 | 121,588.5 | 96,250 | 146,927 | 35,834.1 |
| open | 0.0 | 227,187 | 11 | 20,653.4 | 8,541 | 32,448 | 8,883.0 |
| fgets | 0.0 | 207,135 | 3 | 69,045.0 | 57,656 | 75,052 | 9,868.1 |
| fgetc | 0.0 | 203,074 | 9 | 22,563.8 | 2,396 | 37,969 | 11,907.1 |
| pipe2 | 0.0 | 51,771 | 3 | 17,257.0 | 11,615 | 28,333 | 9,592.7 |
| connect | 0.0 | 38,541 | 1 | 38,541.0 | 38,541 | 38,541 | 0.0 |
| open64 | 0.0 | 26,197 | 1 | 26,197.0 | 26,197 | 26,197 | 0.0 |
| fcntl | 0.0 | 17,971 | 11 | 1,633.7 | 990 | 2,084 | 396.3 |
| socket | 0.0 | 14,844 | 1 | 14,844.0 | 14,844 | 14,844 | 0.0 |
| munmap | 0.0 | 11,980 | 2 | 5,990.0 | 3,907 | 8,073 | 2,945.8 |
| fwrite | 0.0 | 8,593 | 1 | 8,593.0 | 8,593 | 8,593 | 0.0 |

Table 29: The number of threads is 8*8

Table 30: CUDA API Statistics

| Name | Time(%) | Total Time (ns) | Num Calls | Average | Minimum | Maximum | StdDev |
|---|---|---|---|---|---|---|---|
| cudaEventCreate | 58.2 | 605,249,374 | 4 | 151,312,343.5 | 4,010 | 605,218,385 | 302,604,027.8 |
| cudaMemcpy | 34.6 | 359,902,292 | 2 | 179,951,146.0 | 20,719,063 | 339,183,229 | 225,188,171.3 |
| cudaMalloc | 5.3 | 55,162,291 | 2 | 27,581,145.5 | 15,374,427 | 39,787,864 | 17,262,906.9 |
| cudaEventRecord | 1.6 | 17,000,936 | 4 | 4,250,234.0 | 36,718 | 8,452,604 | 4,752,015.8 |
| cudaFree | 0.2 | 1,695,521 | 2 | 847,760.5 | 636,042 | 1,059,479 | 299,415.2 |
| cudaLaunchKernel | 0.0 | 258,073 | 1 | 258,073.0 | 258,073 | 258,073 | 0.0 |
| cudaEventSynchronize | 0.0 | 98,646 | 2 | 49,323.0 | 46,406 | 52,240 | 4,125.3 |
| cudaEventDestroy | 0.0 | 26,719 | 4 | 6,679.8 | 2,604 | 10,990 | 4,247.0 |

Table 31: Operating System Runtime API Statistics

| Name | Time(%) | Total Time (ns) | Num Calls | Average | Minimum | Maximum | StdDev |
|---|---|---|---|---|---|---|---|
| sem_timedwait | 95.4 | 22,838,416,651 | 229 | 99,731,077.1 | 18,838,021 | 100,200,833 | 5,369,072.9 |
| poll | 2.5 | 598,364,120 | 331 | 1,807,746.6 | 4,688 | 490,653,177 | 27,011,933.1 |
| ioctl | 1.9 | 446,915,496 | 1,333 | 335,270.4 | 2,031 | 103,754,115 | 4,519,298.8 |
| fclose | 0.2 | 55,211,511 | 6 | 9,201,918.5 | 4,218 | 55,120,521 | 22,495,435.4 |
| write | 0.0 | 3,181,770 | 331 | 9,612.6 | 2,761 | 81,459 | 8,680.6 |
| sched_yield | 0.0 | 2,714,582 | 962 | 2,821.8 | 1,041 | 394,687 | 14,577.5 |
| read | 0.0 | 1,156,502 | 334 | 3,462.6 | 1,510 | 32,448 | 3,351.4 |
| fopen | 0.0 | 875,312 | 17 | 51,488.9 | 5,312 | 453,750 | 110,129.2 |
| mmap | 0.0 | 642,136 | 16 | 40,133.5 | 4,323 | 126,407 | 34,039.2 |
| sem_wait | 0.0 | 544,323 | 5 | 108,864.6 | 38,177 | 176,250 | 49,418.3 |

Table 32: The number of threads is 16*16

Table 33: CUDA API Statistics

| Name | Time(%) | Total Time (ns) | Num Calls | Average | Minimum | Maximum | StdDev |
|---|---|---|---|---|---|---|---|
| cudaEventCreate | 51.1 | 524,364,844 | 4 | 131,091,211.0 | 4,532 | 524,329,687 | 262,158,984.2 |
| cudaMemcpy | 41.6 | 426,396,042 | 2 | 213,198,021.0 | 20,849,584 | 405,546,458 | 272,021,768.3 |
| cudaMalloc | 5.3 | 54,712,605 | 2 | 27,356,302.5 | 16,229,740 | 38,482,865 | 15,735,335.6 |
| cudaEventRecord | 1.7 | 17,186,981 | 4 | 4,296,745.3 | 74,063 | 8,398,542 | 4,697,333.1 |
| cudaFree | 0.2 | 1,766,875 | 2 | 883,437.5 | 706,979 | 1,059,896 | 249,550.0 |
| cudaLaunchKernel | 0.0 | 493,906 | 1 | 493,906.0 | 493,906 | 493,906 | 0.0 |
| cudaEventSynchronize | 0.0 | 233,073 | 2 | 116,536.5 | 13,854 | 219,219 | 145,215.0 |
| cudaEventDestroy | 0.0 | 28,594 | 4 | 7,148.5 | 3,490 | 11,511 | 4,032.3 |

Table 34: Operating System Runtime API Statistics

| Name | Time(%) | Total Time (ns) | Num Calls | Average | Minimum | Maximum | StdDev |
|---|---|---|---|---|---|---|---|
| sem_timedwait | 92.9 | 23,390,401,127 | 234 | 99,958,979.2 | 70,564,167 | 100,169,948 | 1,929,991.1 |
| fclose | 2.6 | 657,785,050 | 6 | 109,630,841.7 | 3,958 | 657,748,281 | 268,521,609.1 |
| ioctl | 2.1 | 524,152,494 | 1,340 | 391,158.6 | 2,083 | 103,798,594 | 5,121,757.8 |
| poll | 2.1 | 517,646,819 | 331 | 1,563,887.7 | 4,584 | 397,244,531 | 21,980,096.5 |
| fopen | 0.3 | 73,415,625 | 17 | 4,318,566.2 | 6,198 | 73,030,677 | 17,706,720.9 |
| write | 0.0 | 2,996,917 | 331 | 9,054.1 | 2,708 | 88,021 | 7,886.5 |
| sched_yield | 0.0 | 2,697,296 | 948 | 2,845.2 | 1,093 | 513,281 | 18,048.3 |
| read | 0.0 | 1,038,435 | 334 | 3,109.1 | 1,510 | 32,864 | 2,381.4 |
| mmap | 0.0 | 650,782 | 16 | 40,673.9 | 4,531 | 106,979 | 29,380.7 |
| fgetc | 0.0 | 357,295 | 16 | 22,330.9 | 3,073 | 38,230 | 7,821.6 |
| sem_wait | 0.0 | 332,501 | 5 | 66,500.2 | 29,063 | 119,219 | 44,147.1 |
| pthread_create | 0.0 | 227,447 | 2 | 113,723.5 | 89,739 | 137,708 | 33,919.2 |
| open | 0.0 | 207,135 | 11 | 18,830.5 | 8,854 | 39,479 | 10,390.5 |
| fgets | 0.0 | 187,030 | 3 | 62,343.3 | 52,447 | 72,760 | 10,166.5 |
| open64 | 0.0 | 45,000 | 1 | 45,000.0 | 45,000 | 45,000 | 0.0 |
| pthread_mutex_trylock | 0.0 | 37,083 | 1 | 37,083.0 | 37,083 | 37,083 | 0.0 |
| fcntl | 0.0 | 37,032 | 10 | 3,703.2 | 1,198 | 22,240 | 6,521.2 |
| pipe2 | 0.0 | 34,896 | 3 | 11,632.0 | 10,365 | 12,448 | 1,112.3 |
| connect | 0.0 | 21,198 | 1 | 21,198.0 | 21,198 | 21,198 | 0.0 |
| pthread_cond_broadcast | 0.0 | 20,157 | 2 | 10,078.5 | 1,094 | 19,063 | 12,706.0 |
| socket | 0.0 | 16,979 | 1 | 16,979.0 | 16,979 | 16,979 | 0.0 |
| munmap | 0.0 | 13,542 | 2 | 6,771.0 | 3,906 | 9,636 | 4,051.7 |
| fwrite | 0.0 | 8,802 | 1 | 8,802.0 | 8,802 | 8,802 | 0.0 |

Table 35: CUDA Profiling Data for 32*32 Threads

Table 36: CUDA API Statistics

| Name | Time(%) | Total Time (ns) | Num Calls | Average | Minimum | Maximum | StdDev |
|---|---|---|---|---|---|---|---|
| cudaMemcpy | 49.5 | 554,140,313 | 2 | 277,070,156.5 | 21,839,688 | 532,300,625 | 360,950,390.1 |
| cudaEventCreate | 44.5 | 497,964,010 | 4 | 124,491,002.5 | 4,219 | 497,903,281 | 248,941,520.0 |
| cudaMalloc | 4.3 | 47,683,126 | 2 | 23,841,563.0 | 15,702,865 | 31,980,261 | 11,509,857.1 |
| cudaEventRecord | 1.6 | 17,361,875 | 4 | 4,340,468.8 | 37,344 | 8,449,271 | 4,734,284.4 |
| cudaFree | 0.2 | 1,939,323 | 2 | 969,661.5 | 912,917 | 1,026,406 | 80,248.8 |
| cudaLaunchKernel | 0.0 | 231,563 | 1 | 231,563.0 | 231,563 | 231,563 | 0.0 |
| cudaEventDestroy | 0.0 | 75,625 | 4 | 18,906.3 | 3,230 | 58,958 | 26,811.3 |
| cudaEventSynchronize | 0.0 | 28,697 | 2 | 14,348.5 | 14,114 | 14,583 | 331.6 |

Table 37: Operating System Runtime API Statistics

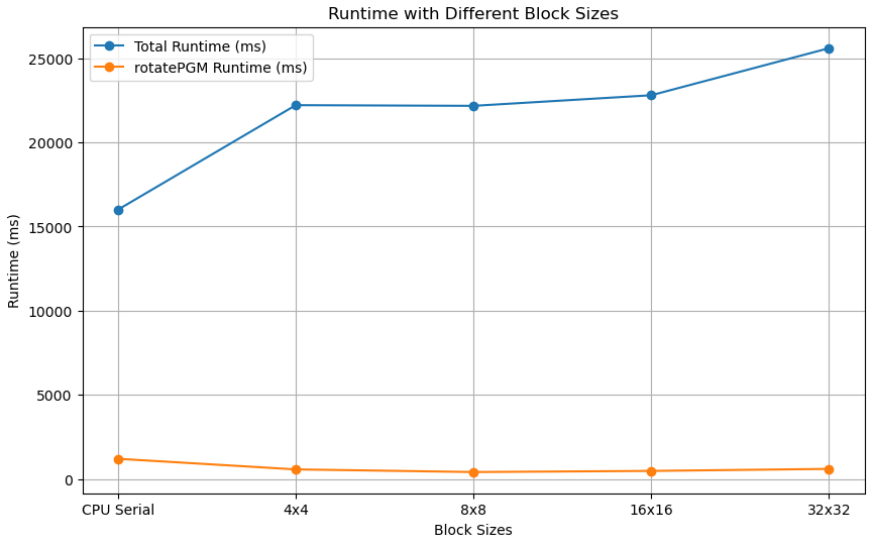
| Name | Time(%) | Total Time (ns) | Num Calls | Average | Minimum | Maximum | StdDev |
|---|---|---|---|---|---|---|---|
| sem_timedwait | 93.5 | 26,153,489,420 | 262 | 99,822,478.7 | 31,565,990 | 100,267,708 | 4,233,127.9 |
| fclose | 2.3 | 636,497,084 | 6 | 106,082,847.3 | 4,011 | 636,461,562 | 259,831,444.3 |
| ioctl | 2.2 | 619,550,498 | 1,343 | 461,318.3 | 1,927 | 103,958,958 | 5,846,918.9 |
| poll | 1.8 | 491,383,028 | 331 | 1,484,540.9 | 4,583 | 389,529,896 | 21,456,281.5 |
| fopen | 0.3 | 74,438,648 | 17 | 4,378,744.0 | 5,416 | 74,123,177 | 17,972,734.6 |
| write | 0.0 | 3,123,226 | 331 | 9,435.7 | 2,760 | 75,781 | 8,709.3 |
| sched_yield | 0.0 | 2,785,621 | 747 | 3,729.1 | 1,041 | 122,605 | 10,060.7 |
| read | 0.0 | 1,226,621 | 334 | 3,672.5 | 1,406 | 97,760 | 6,153.5 |
| mmap | 0.0 | 593,595 | 16 | 37,099.7 | 4,062 | 102,865 | 27,484.8 |
| sem_wait | 0.0 | 414,582 | 5 | 82,916.4 | 31,406 | 138,437 | 49,621.6 |
| fgetc | 0.0 | 319,946 | 14 | 22,853.3 | 3,490 | 40,678 | 9,539.5 |
| pthread_create | 0.0 | 191,716 | 11 | 17,428.7 | 8,958 | 30,520 | 7,829.3 |
| open | 0.0 | 188,594 | 2 | 94,297.0 | 90,469 | 98,125 | 5,413.6 |
| fgets | 0.0 | 185,053 | 3 | 61,684.3 | 36,250 | 74,896 | 22,032.3 |
| pipe2 | 0.0 | 55,417 | 3 | 18,472.3 | 12,136 | 29,323 | 9,441.0 |
| connect | 0.0 | 40,417 | 1 | 40,417.0 | 40,417 | 40,417 | 0.0 |
| fcntl | 0.0 | 35,052 | 11 | 3,186.5 | 1,198 | 20,157 | 5,636.5 |
| open64 | 0.0 | 25,313 | 1 | 25,313.0 | 25,313 | 25,313 | 0.0 |
| socket | 0.0 | 16,198 | 1 | 16,198.0 | 16,198 | 16,198 | 0.0 |
| munmap | 0.0 | 14,427 | 2 | 7,213.5 | 6,041 | 8,386 | 1,658.2 |
| fwrite | 0.0 | 9,323 | 1 | 9,323.0 | 9,323 | 9,323 | 0.0 |
| fflush | 0.0 | 1,198 | 1 | 1,198.0 | 1,198 | 1,198 | 0.0 |
| pthread_cond_broadcast | 0.0 | 1,094 | 1 | 1,094.0 | 1,094 | 1,094 | 0.0 |

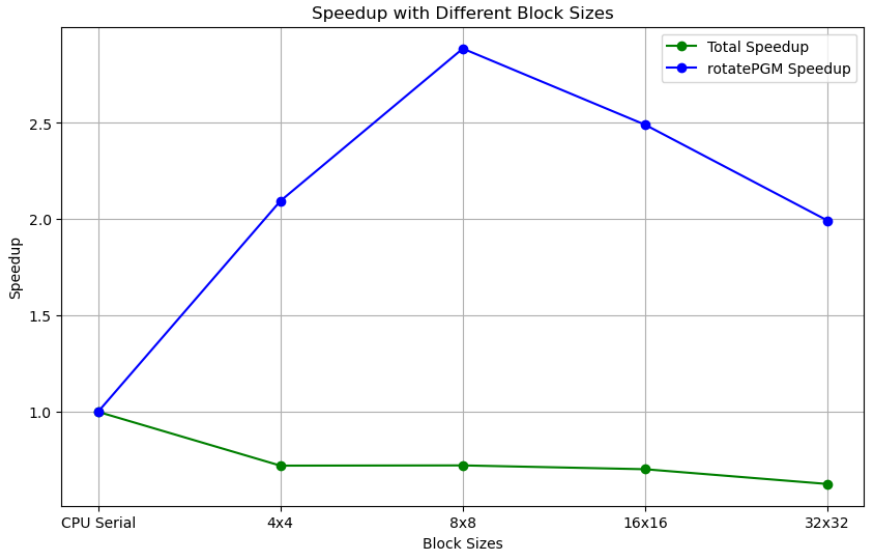
**CUDA's event-based timing mechanism:**

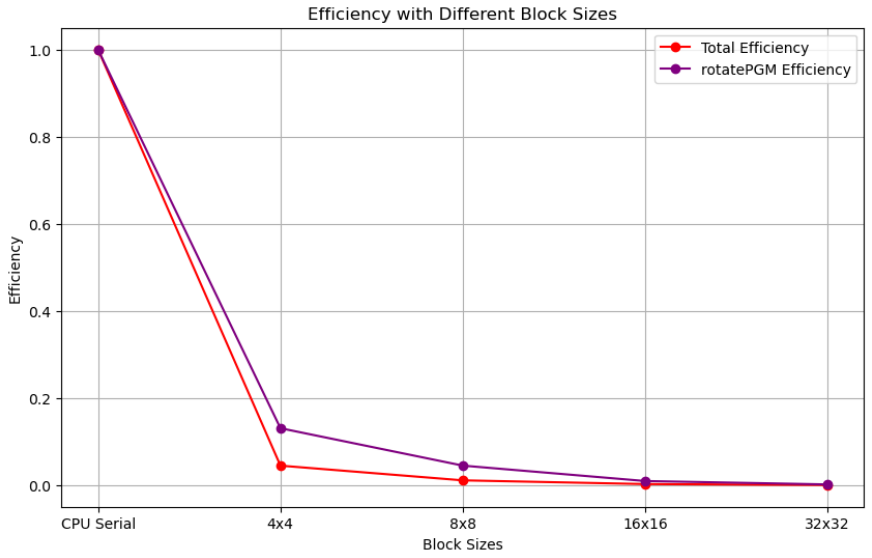
| Number of Threads | Total Program Runtime (ms) | rotatePGM Runtime (ms) |
|---|---|---|
| CPU Serial | 16000 | 1204.163 |
| 4x4 | 22206.0332 | 574.802673 |
| 8x8 | 22165.83594 | 417.248169 |
| 16x16 | 22794.43555 | 483.537292 |
| 32x32 | 25589.49219 | 604.250671 |

Table 38: CUDA Runtime with Different Block Sizes

## Analysis


Runtime with Different Block Sizes
Total Runtime (ms)
rotatePGM Runtime (ms)
Runtime (ms)
25000
20000
15000
10000
5000
0
CPU Serial
4x4
8x8
16x16
32x32
Block Sizes


Speedup with Different Block Sizes
Total Speedup
rotatePGM Speedup
Speedup
2.5
2.0
1.5
1.0
CPU Serial
4x4
8x8
16x16
32x32
Block Sizes


Efficiency with Different Block Sizes
Total Efficiency
rotatePGM Efficiency
Efficiency
1.0
0.8
0.6
0.4
0.2
0.0
CPU Serial
4x4
8x8
16x16
32x32
Block Sizes

## Compare and Contrast

For the specific task of image rotation, CUDA emerges as the preferred parallel technology for the task of image rotation on Jetson Nano. This is attributed to the GPU performance of Jetson Nano and CUDA's specialized optimization for NVIDIA GPU architecture. CUDA's parallel processing capabilities are particularly well-suited for rapidly performing image-related operations, including image rotation, with a relatively straightforward debugging and performance optimization process on a single device. In contrast, MPI would require additional installation and configuration for MPI-related libraries (MpiP), making it less optimal for this specific use case in a single-node environment like Jetson Nano.

**Performance Assessment**

*CUDA:* The CUDA framework exhibits excellent acceleration ratios for the specific task of image rotation, leveraging the parallel processing capabilities and the multitude of threads available on the GPU. However, due to the large number of computing units constrained by power consumption, size, and heat, each core's computational power is relatively weak, resulting in lower efficiency for CUDA's parallel method. In contrast, the MPI parallel method shows better efficiency due to the CPU's design for handling complex logic and sequential tasks, emphasizing the power and efficiency of each core.

*MPI:* MPI does not exhibit the same level of performance advantages for image rotation tasks, particularly in a single-node environment like Jetson Nano, where image processing can generally be efficiently completed.

**Hardware Limitations**

The NVIDIA Tegra X1 GPU on the Jetson Nano, with 3.87 GB of total global memory, provides ample cache for handling large image data. The 48 KB shared memory per block and 32768 registers per block allow for efficient image processing within each block, reducing global memory access and enhancing efficiency. However, the CPU of Jetson Nano, with only four ARM Cortex-A57 cores and no hyper-threading support, has limited parallel processing capabilities.

**Ease of Debugging and Testing**

The JetPack SDK provides a comprehensive solution for the Jetson module, including a full set of environments from the operating system to development tools. This makes CUDA parallel computing on the Jetson Nano straightforward, with no need to build the environment from scratch. The integrated rich libraries and tools, including those for GPU computing, multimedia, graphics, and computer vision, along with the graphical performance analysis tool (Nsight System), facilitate better data visualization and performance bottleneck identification. On the other hand, MPI requires more manual setup for configuration and debugging, and additional steps for visualization.

**Communication and Synchronization**

In a single GPU environment, communication and synchronization are highly efficient, making the processing of image data straightforward without the need for extensive cross-node communication. However, the actual performance might be less than the theoretical performance due to the need for coordination with the CPU, limited by the GPU's memory and processing capacity. In contrast, MPI's communication overhead might become a performance bottleneck, especially in single-node environments like the Jetson

Nano.

# Reflection

## Installation and Configuration of mpiP

While executing the command `mpirun -np [number of threads] ./rotate-image 45` in the terminal, the dynamic library `libmpiP.so` could not be loaded. It was discovered that mpiP was not installed. The latest version of mpiP was obtained from the official website: *https://software.llnl.gov/mpiP/*. According to the website instructions, the source was downloaded from GitHub:

```
wget https://github.com/LLNL/mpiP/releases/download/3.5/mpip-3.5.tgz
```

The downloaded file was then extracted:

```
tar -xzvf mpip-3.5.tgz
```

Following this, mpiP was compiled and installed:

```
cd mpip-3.5
./configure
```

During the configure process, the script checked for the required libraries. Missing libraries were installed using `sudo apt-get install` on Ubuntu:

```
make
make install
```

After successful installation, the command `mpirun -np [number of threads] ./rotate-image 45` still returned an error indicating that the `libmpiP.so` library could not be found. To resolve this, the environment variable was set in the `~/.bashrc` file.

The `~/.bashrc` file was opened using vim:

```
vim ~/.bashrc
```

In vim, 'i' was pressed to enter insert mode, and the following line was appended at the end of the file:

```
export LD_LIBRARY_PATH=/path/to/mpip-3.5/lib:$LD_LIBRARY_PATH
```

*/path/to/mpip-3.5/lib* is */home/nvidia/Desktop/mpiP-3.5/mpiP-3.5/lib*

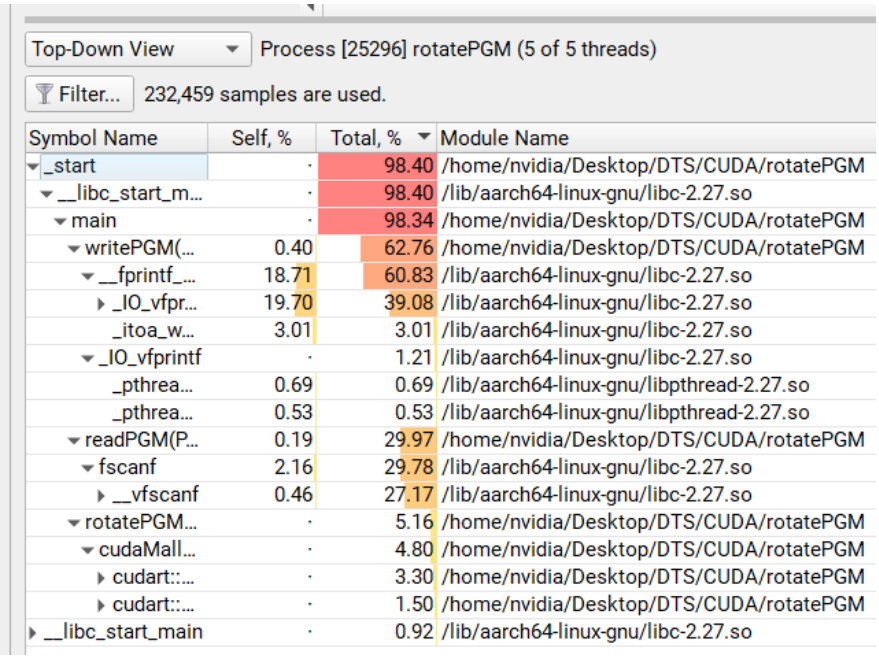
The file was saved and exited using `:wq`. Finally, the changes were applied with:

```
source ~/.bashrc
```

## Change in the Method of Calculating Program Speedup and Efficiency

After initial testing, it was found that the overall speedup and efficiency of the program did not meet the expected targets. In fact, with the MPI approach, as the number of threads increased, the total runtime of the program became longer, with a speedup less than 1 for more than two threads. The preliminary assumption was that communication issues between different processes caused a performance bottleneck. However, after reviewing the mpiP performance report, it was found that communication time accounted for about 30% of the total process time, which is relatively normal considering the large size of the images to be rotated.

Subsequently, the focus shifted to the hypothesis that the bottleneck was primarily in the reading and writing of images. To verify this hypothesis, the NVIDIA nsight system tool was employed for performance analysis. Its excellent visualization capabilities revealed that, in the three main functions – readPGM, writePGM, and rotatePGM – readPGM and writePGM accounted for nearly 90% of the time. Therefore, it was decided to analyze only the parallel part of the program, rotatePGM, for performance.




| Symbol Name | Self, % | Total, % | Module Name |
|---|---|---|---|
| _start | · | 98.40 | /home/nvidia/Desktop/DTS/CUDA/rotatePGM |
| __libc_start_m... | · | 98.40 | /lib/aarch64-linux-gnu/libc-2.27.so |
| main | · | 98.34 | /home/nvidia/Desktop/DTS/CUDA/rotatePGM |
| writePGM(... | 0.40 | 62.76 | /home/nvidia/Desktop/DTS/CUDA/rotatePGM |
| __fprintf_... | 18.71 | 60.83 | /lib/aarch64-linux-gnu/libc-2.27.so |
| _IO_vfpr... | 19.70 | 39.08 | /lib/aarch64-linux-gnu/libc-2.27.so |
| _itoa_w... | 3.01 | 3.01 | /lib/aarch64-linux-gnu/libc-2.27.so |
| _IO_vfprintf | · | 1.21 | /lib/aarch64-linux-gnu/libc-2.27.so |
| _pthrea... | 0.69 | 0.69 | /lib/aarch64-linux-gnu/libpthread-2.27.so |
| _pthrea... | 0.53 | 0.53 | /lib/aarch64-linux-gnu/libpthread-2.27.so |
| readPGM(P... | 0.19 | 29.97 | /home/nvidia/Desktop/DTS/CUDA/rotatePGM |
| fscanf | 2.16 | 29.78 | /lib/aarch64-linux-gnu/libc-2.27.so |
| __vfscanf | 0.46 | 27.17 | /lib/aarch64-linux-gnu/libc-2.27.so |
| rotatePGM... | · | 5.16 | /home/nvidia/Desktop/DTS/CUDA/rotatePGM |
| cudaMall... | · | 4.80 | /home/nvidia/Desktop/DTS/CUDA/rotatePGM |
| cudart::... | · | 3.30 | /home/nvidia/Desktop/DTS/CUDA/rotatePGM |
| cudart::... | · | 1.50 | /home/nvidia/Desktop/DTS/CUDA/rotatePGM |
| __libc_start_main | · | 0.92 | /lib/aarch64-linux-gnu/libc-2.27.so |

## SSH Remote Connection and Passwordless Login Configuration

When using SSH to remotely connect to a Jetson Nano, the necessity of entering the password each time when opening VS Code significantly reduced work efficiency. This issue was resolved by generating an SSH key and adding its public counterpart to the Jetson Nano for passwordless login.

First, a private and public key pair was generated. On the client side (Windows), the ‘cmd‘ was opened by pressing Win + R and typing ‘cmd‘. The following command was then executed in the cmd:

```
ssh-keygen -t rsa -b 4096
```

Here, ‘-t‘ specifies the type of the encryption algorithm, RSA in this case, and ‘-b‘ specifies the bit length of the key, using 4096 bits to ensure a relatively strong key.

The ‘id_rsa.pub‘ file, which was generated at the specified address, was then opened and its contents (‘ssh-rsa XXXXXXXX‘) were copied. These contents were added to the ‘authorized_keys‘ file on the server side, the Jetson Nano (Linux). After remotely logging into the Jetson Nano via cmd, the following command was executed:

```
echo ''ssh-rsa XXXXXXXX'' >> ~/.ssh/authorized_keys
```