Fast Gaussian Blur

1.0.0

By Arjun Prashanth 20BCE1029

Chapter 1

Report

1.1 Abstract

Gaussian Blurring has numerous applications in various fields, including gaming, photography, video processing, and medical imaging, to name a few. Here are some examples of how Gaussian Blurring is used in different areas:

Gaming: Gaussian Blurring is used in gaming to create realistic depth-of-field effects, which can simulate the way our eyes focus on objects in the real world. For example, in a first-person shooter game, blurring the background can make the player feel like they are focusing on a distant object. Another example is motion blur, which can simulate the way objects appear blurry when they are moving quickly.

Photography: Gaussian Blurring is commonly used in photography to remove noise from images and to create soft-focus effects. It can also be used to smooth out skin tones in portraits and to create a bokeh effect, where the background of an image appears out of focus.

Video processing: Gaussian Blurring is used in video processing to remove noise and to smooth out video frames. It is also used to create special effects, such as blurring out a face in a video to protect someone's identity.

Medical imaging: Gaussian Blurring is used in medical imaging to remove noise from images and to enhance the contrast between different structures in the image. For example, it can be used to enhance the edges of a tumor in a CT scan or MRI image.

However the time complexity $(O(nmk^2))$ of a naive implementation of Gaussian Blurring can be quite high, which can result in slow performance. This is because the process of blurring an image involves convolving the image with a Gaussian kernel, which requires performing a large number of calculations for each pixel in the image. Furthermore, the naive implementation requires applying the kernel to each pixel in the image, which can result in redundant calculations. This can lead to further slow-downs in performance, especially for larger images.

This project compares Gaussian Blurring implemented in both CPU and GPU architectures, and demonstrates the speed benefits of a separable 1D kernel versus a 2D kernel. Time complexity and performance measurements are used to analyse the differences between the two kernels on both the CPU and the GPU. Results show that a separable 1D kernel is faster than a 2D, and this proves that the speed benefits associated with a separable 1D kernel in image processing tasks extend across different hardware implementations.

1.2 Dataset

This project uses the standard dataset for image proessing, sourced from `here`. The dataset consists of 24 images, which include grayscale, 256 bit color and 512 bit color of each unique image.

1.3 Technologies Used -

1. OpenCV - For reading and writing images
2. OpenMP - For parallelizing the CPU implementation
3. CUDA - For parallelizing the GPU implementation

1.4 Methodology

1.4.1 CPU -

1. The 2D kernel is first generated and then applied onto the image as a convolution.
2. For parallelizing the CPU implementation, OpenMP `#pragma parallel for` compiler directive is used along with shared data to speed up. The separable kernel was not implemented for the CPU because, regardless of the implementation, the process is very slow as compared to GPU

1.4.2 GPU -

1.4.2.1 2D Kernel -

1. First the kernel is generated on the host (CPU).
2. Then the image is loaded on the host.
3. The image is then copied to the device (GPU) using OpenCV API.
4. The kernel is copied to the device using `cudaMemcpy()`.
5. An output image is created on the GPU with the help of OpenCV API.
6. The blocks and grids are created, and the Convolution kernel is called with the required parameters.
7. Synchronization of GPU threads takes place to ensure no race conditions during stress testing.
8. The output image is copied back to the host (CPU) using OpenCV API.
9. The GPU memory is freed using `cudaFree()` and the images are freed using OpenCV API.

1.4.2.2 Separable Kernel (1D) -

1. First a 1D kernel is generated on the host (CPU).
2. The image is loaded on the host and then uploaded to the device (GPU).
3. an outpuit image is created on the GPU with the help of OpenCV API.
4. first a convolution is applied along the horizontal axis (x-axis) on the original image. This output is stored in a temporary image.
5. Now a convolution is applied along the vertical axis (y-axis) on the temporary image. This output is stored in the final output image.
6. The final output image is downloaded to the host.
7. The GPU memory is freed using `cudaFree()` and the images are freed using OpenCV API.

1.5 Results -

1.5.1 Plots -

1.5.1.1 CPU -

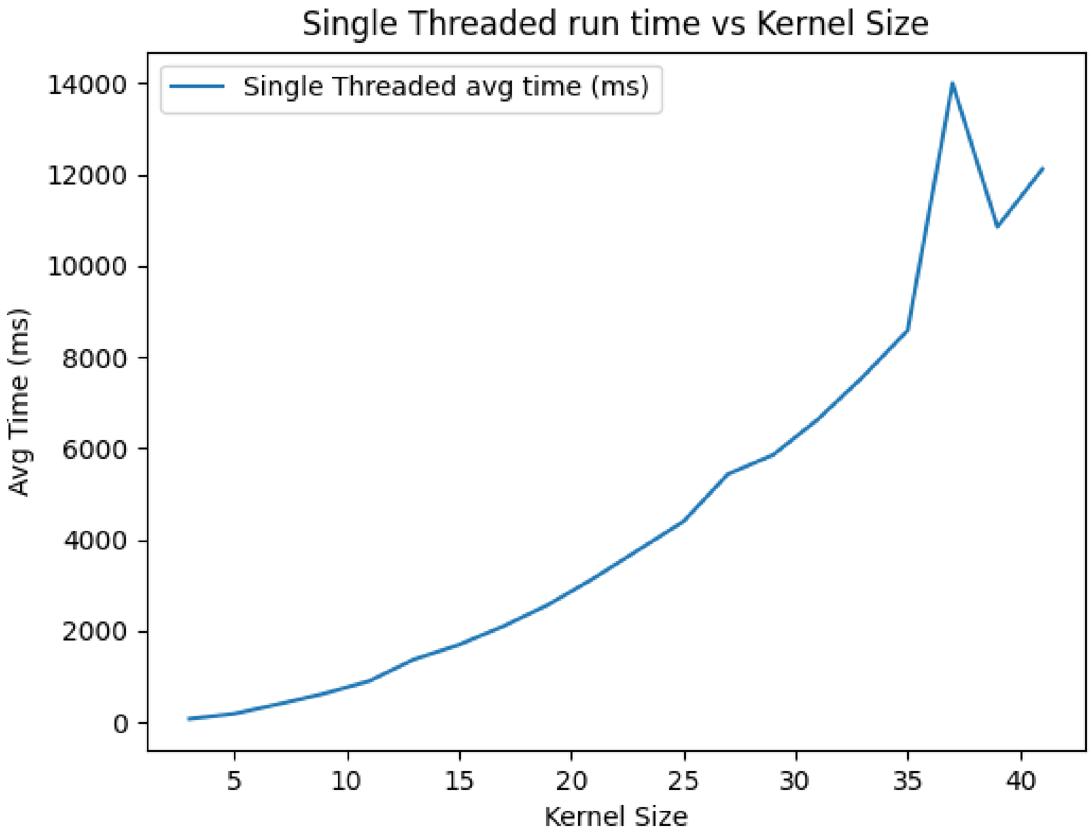

Figure 1.1 Gaussian Blur CPU Single Threaded

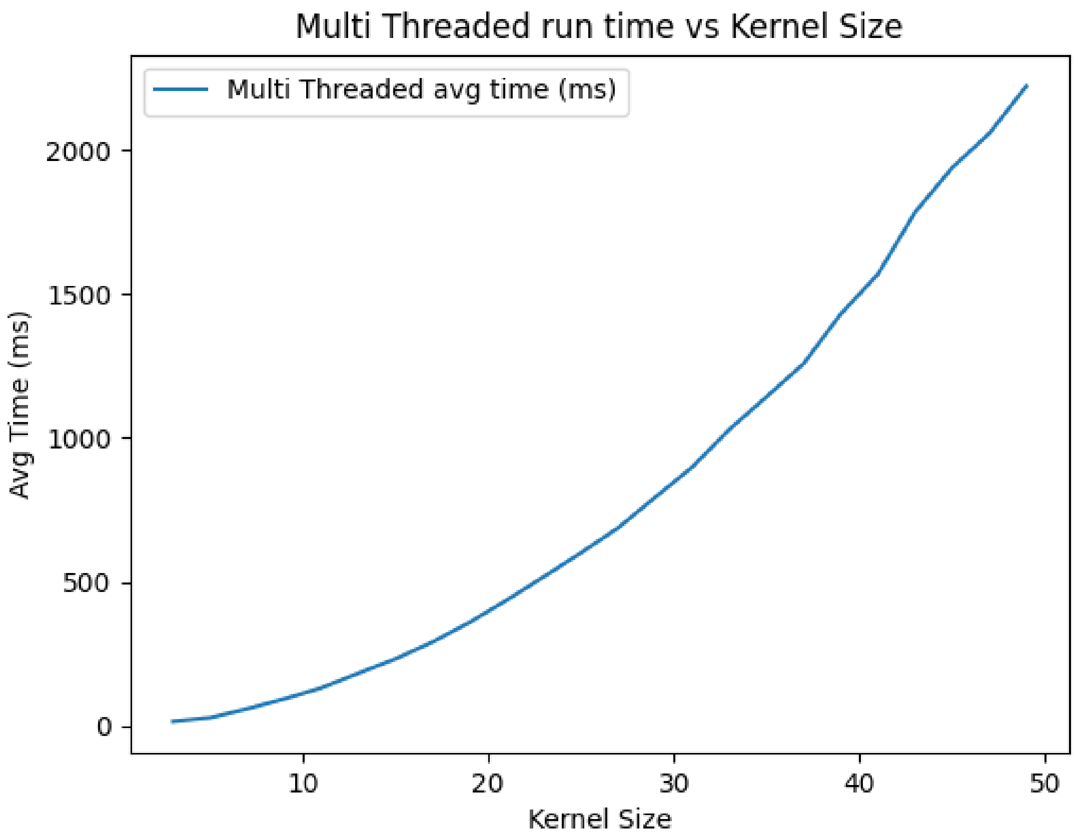

Figure 1.2 Gaussian Blur CPU Multiple Threaded

Comparing Single Threaded vs Multi Threaded

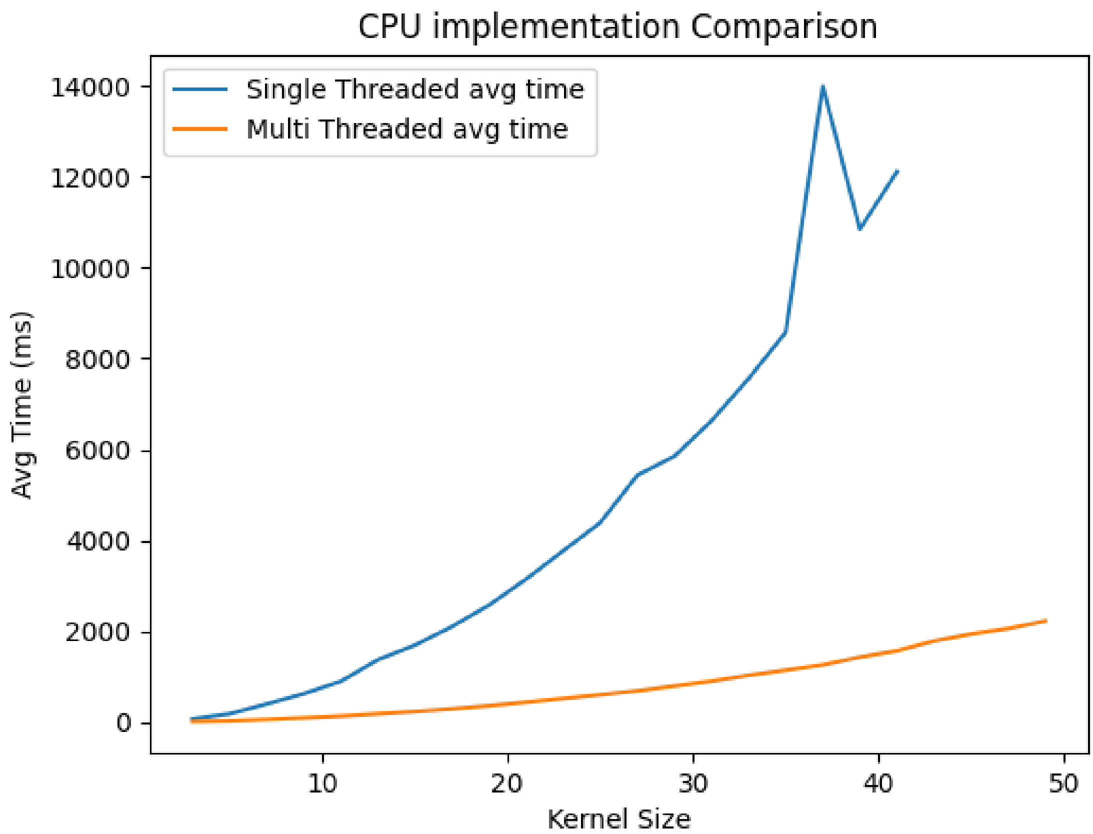

Figure 1.3 Gaussian Blur CPU Comparison

Average Speedup: 7.31 times

1.5.1.2 GPU -

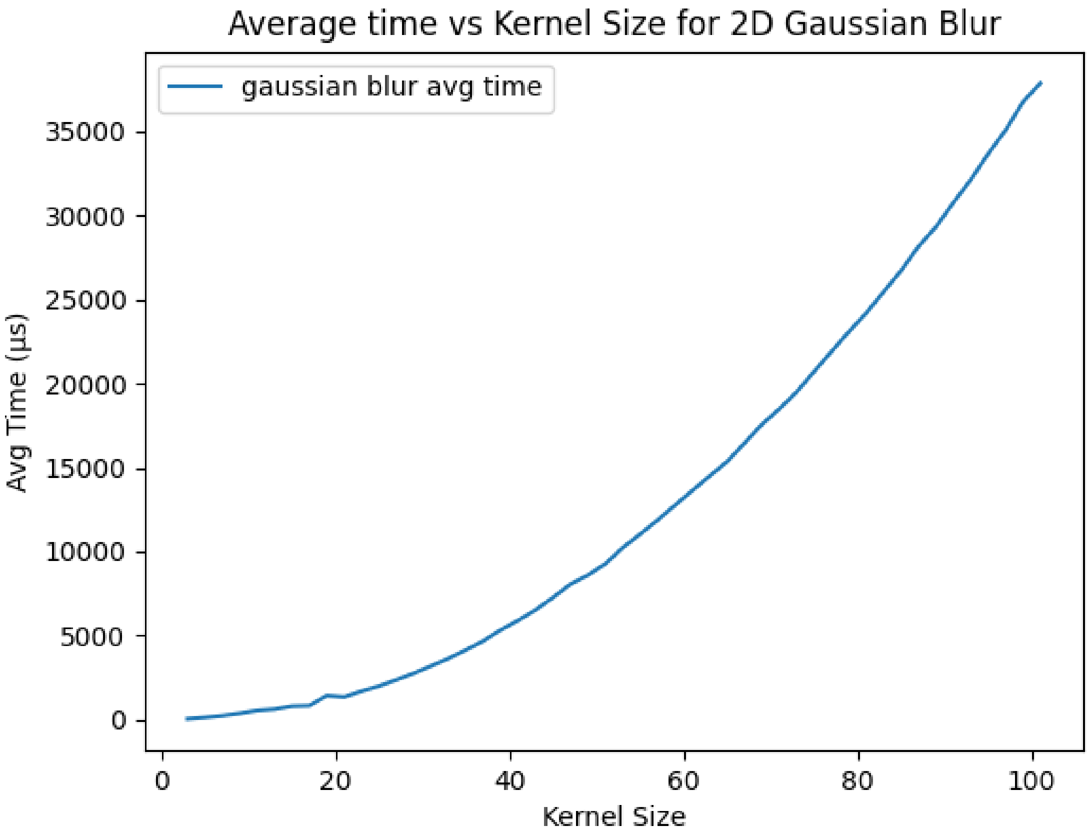

Figure 1.4 Gaussian Blur 2D kernel

Separable Kernel - 

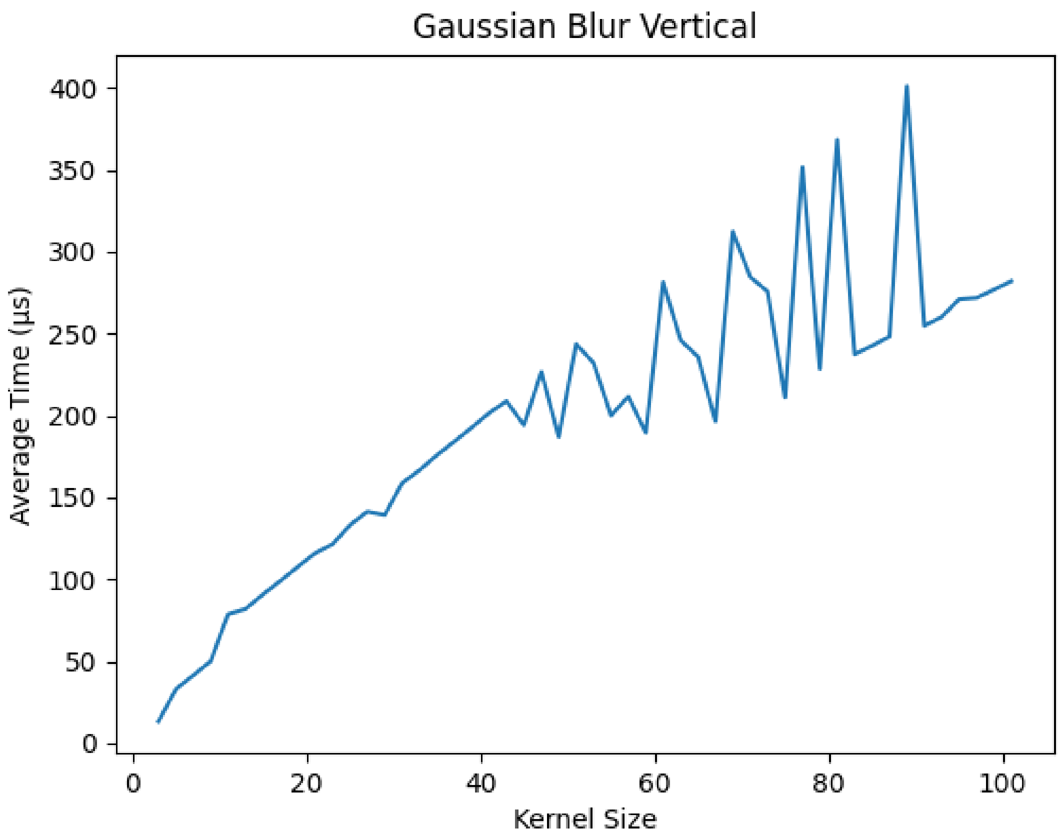

Figure 1.5 Gaussian Blur Separable Kernel Vertical

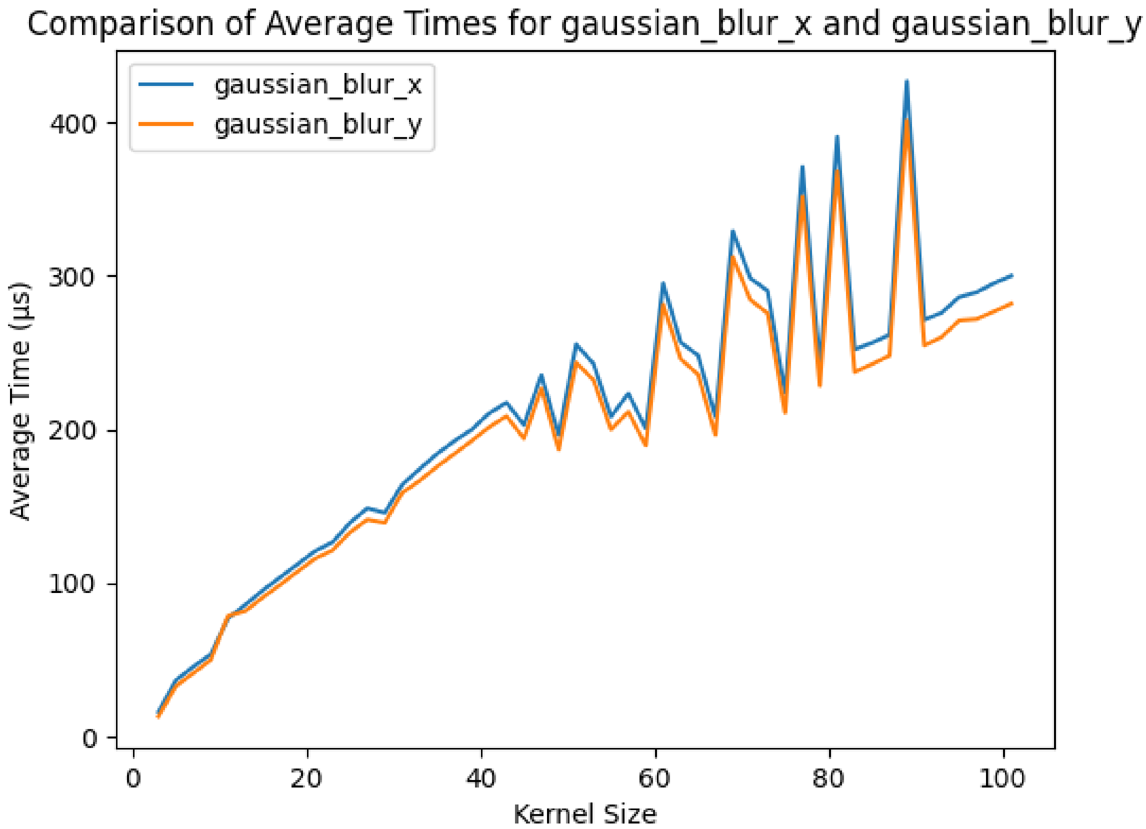

Figure 1.6 Gaussian Blur Separable kernel

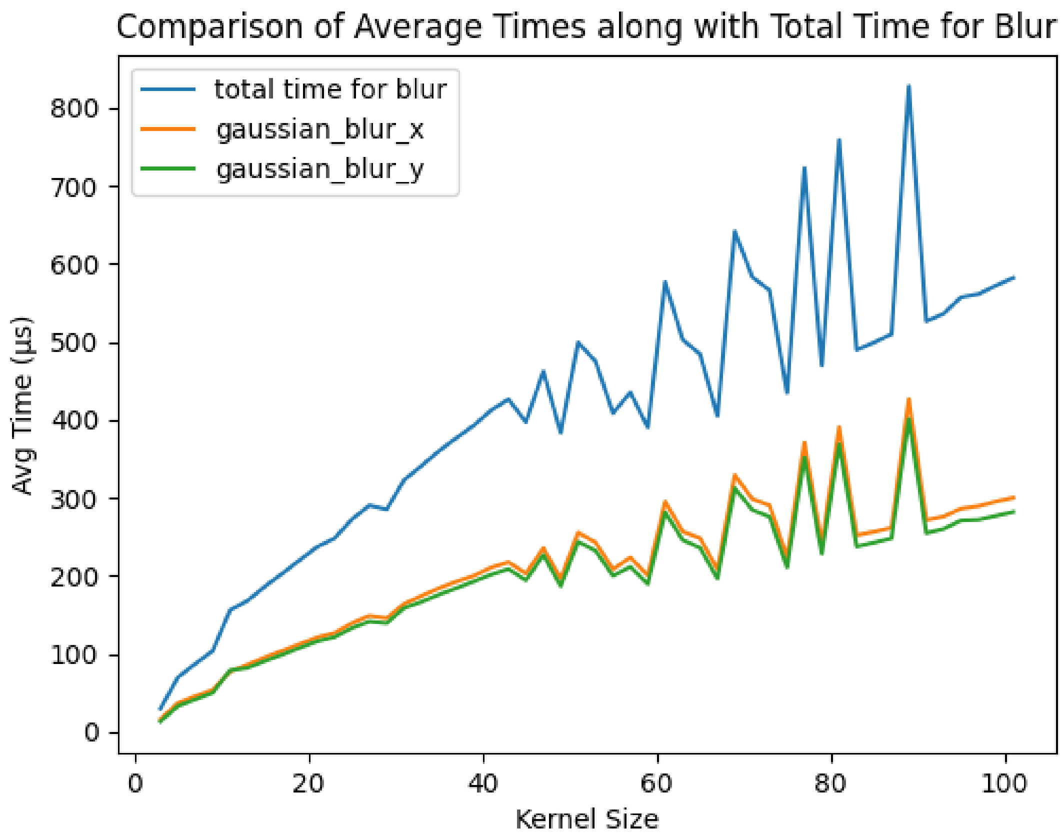

Figure 1.7 Gaussian Blur Separable kernel total

Comparing 2D Kernel vs Separable Kernel -

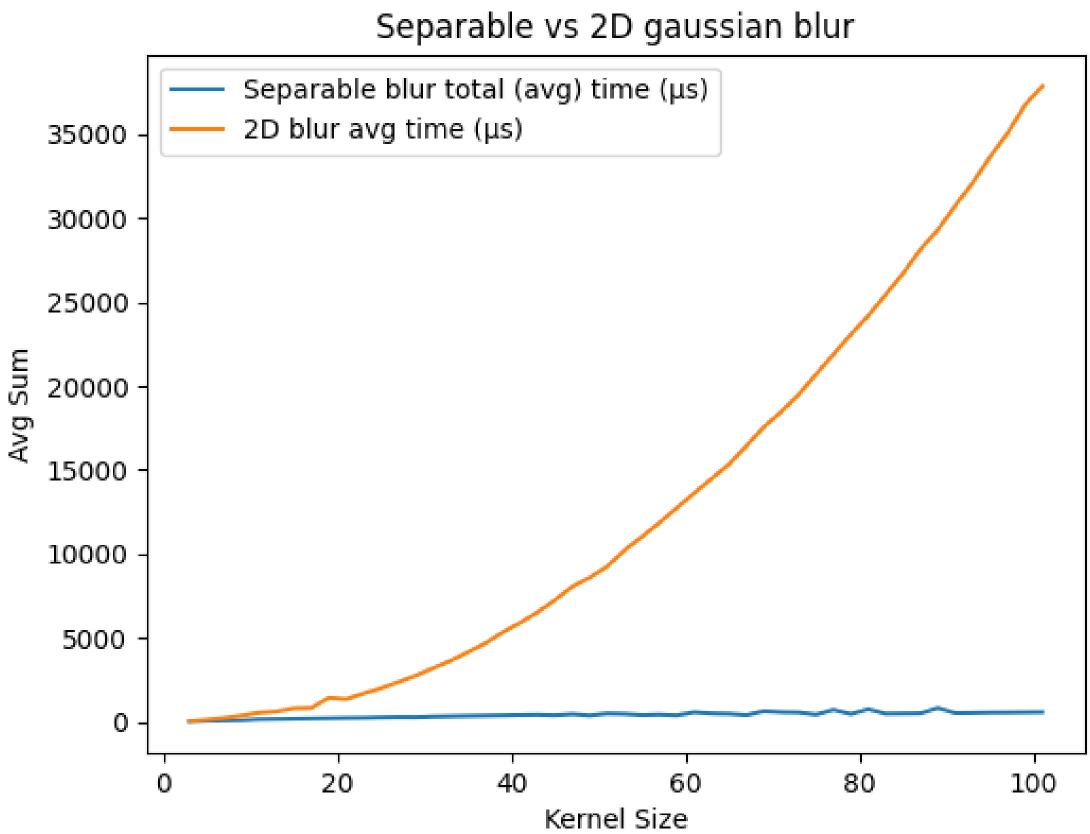

Figure 1.8 Gaussian Blur GPU Comparison

Average Speedup : 25.72 times

Comparing CPU vs GPU implementations -

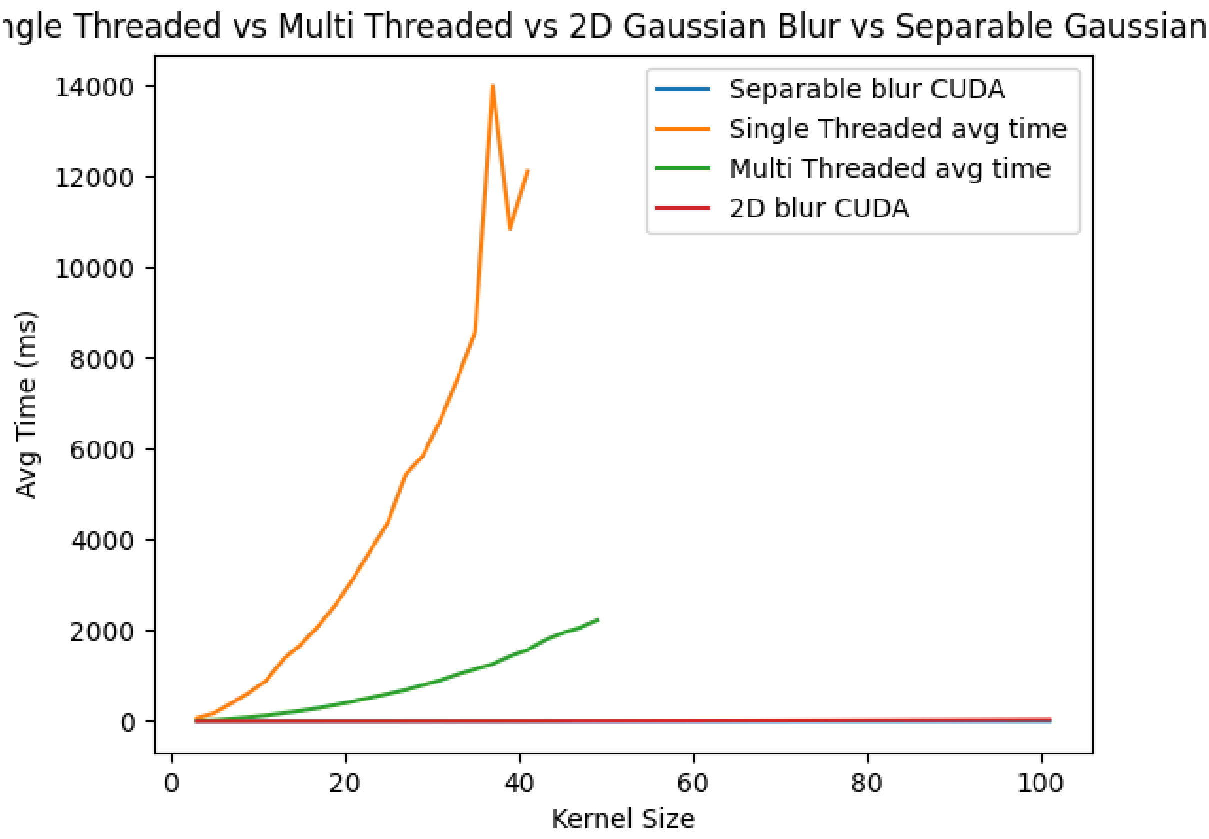

Figure 1.9 Gaussian Blur CPU vs GPU comparison

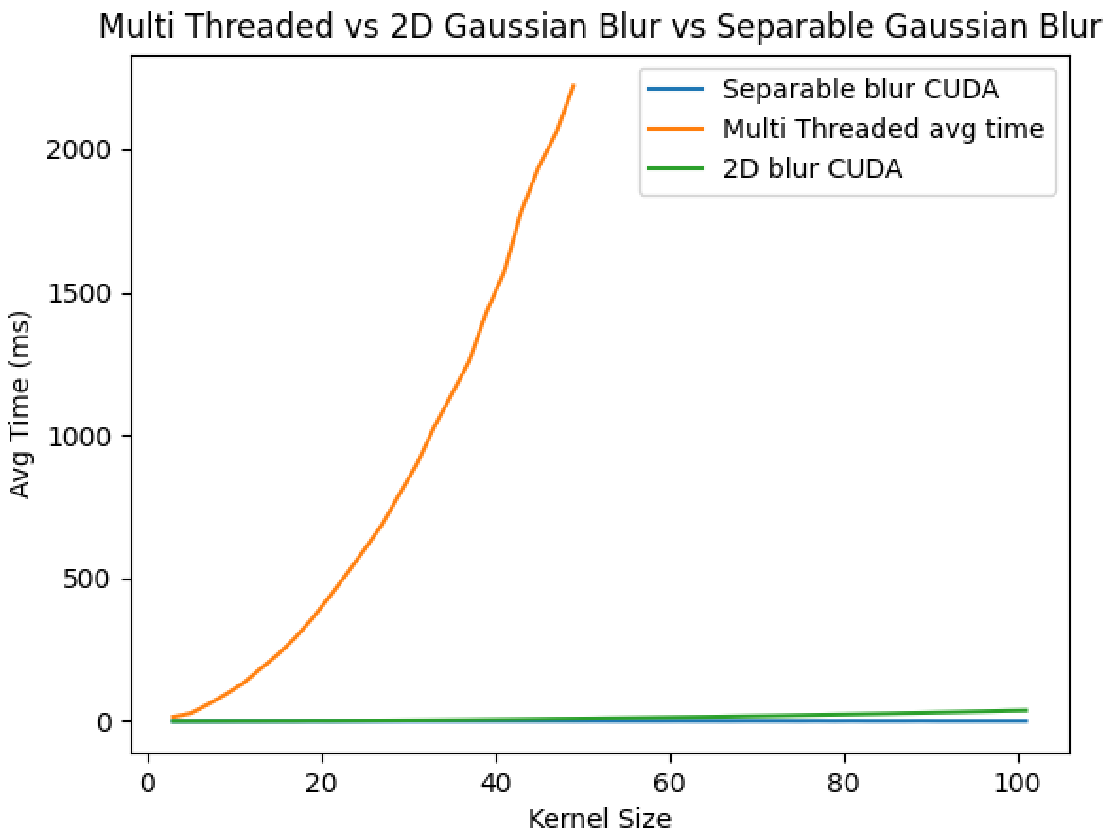

Figure 1.10 Gaussian Blur CPU vs GPU

Average Speedup : 2461.73 times

1.6 Inference

Gaussian Blurring is a popular technique used in image processing to remove noise and smooth an image. In this project, the technique is implemented using both CPU and GPU computing.

The CPU implementation first applies a 2D blur to the image, which is a computationally intensive task that involves convolving the image with a Gaussian kernel. This is done using a nested loop structure that applies the kernel to each pixel in the image.

To improve the performance of the CPU implementation, a multithreaded version using OpenMP is also implemented. This allows the kernel to be applied to multiple pixels simultaneously, reducing the total time taken to blur the image.

The GPU implementation uses CUDA to accelerate the Gaussian blurring process. The image is transferred to the GPU, and the kernel is applied to each pixel in parallel using a grid of threads. This greatly reduces the time taken to blur the image compared to the CPU implementation.

Finally, a separable kernel is used to further improve the performance of the GPU implementation. This involves breaking the Gaussian kernel into two 1D kernels, which can be applied separately in the x and y directions. This greatly reduces the number of computations required to apply the kernel, resulting in very high speedups.

In summary, Gaussian Blurring using CPU and GPU is a computationally intensive task that can benefit greatly from parallel processing. By implementing a multithreaded version using OpenMP and a GPU version using CUDA, the performance of the algorithm can be greatly improved. Using a separable kernel further improves the performance of the GPU implementation, resulting in very high speedups.

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

File Documentation

3.1 blur.cu File Reference

convolution blurring in Nvidia CUDA

```
#include <cstring>
#include <cuda_profiler_api.h>
#include <cuda_runtime.h>
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/core/cuda.hpp>
#include <opencv2/core/cuda/common.hpp>
#include <opencv2/core/matx.hpp>
#include <opencv2/cudaimgproc.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/opencv.hpp>
#include <stdio.h>
```

Macros

- #define PBSTR "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||"
- #define PBWIDTH 60
- #define SAFE_CALL(call, msg) _safe_cuda_call((call), (msg), __FILE__, __LINE__)

  a macro for sage calling CUDA functions

Functions

- void printProgress (double percentage)
- __host__ void generate_gaussian_kernel_2d (float ∗kernel, const int n, const float sigma=1)

  generate the gaussian kernel with given kernel size and standard deviation
- __host__ void generate_gaussian_kernel_1d (float ∗kernel, const int n, const float sigma=1)

  generate a 1D gaussian kernel
- __device__ __forceinline__ void set_value (const int &val, uchar &out)

  Sets the value of a uchar type.
- __device__ __forceinline__ void set_value (const float &val, float &out)

set the value for a floating point type.

- __device__ __forceinline__ void set_value (const float &val, float3 &out)

    set the value for a float3 tupe. All the 3 fields will have the value `val`

- __device__ __forceinline__ void set_value (const float3 &val, uchar3 &out)

    set the value for a unsigned char3 type with a flot3 type

- __device__ __forceinline__ void set_value (const int &val, uchar3 &out)

    Sets the value of a uchar3 type.

- __device__ __forceinline__ uchar3 subtract_value (uchar3 in1, uchar3 in2)

    Subtraction for uchar3 types.

- __device__ __forceinline__ float3 add_value (float3 in1, float3 in2)

    add two values and return it

- __device__ __forceinline__ float add_value (float in1, float in2)

    add two floating point values

- __device__ __forceinline__ uchar subtract_value (uchar in1, uchar in2)

    Subtraction for uchar types.

- __device__ __forceinline__ float3 multiply_value (const float &x, const uchar3 &y)

    multiplication for float and uchar3 types. Multiply each filed in uchar3 with the float value and return a flolat3

- __device__ __forceinline__ float3 multiply_value (const float &x, const float3 &y)

    multiplication for float and float3 types. Multiply each filed in uchar3 with the float value and return a float3

- __device__ __forceinline__ float multiply_value (const float &x, const uchar &y)

    multiplication for float and uchar4 types

- template<typename T_in , typename T_out , typename F_cal >
  __global__ void gaussian_blur (const float *kernel, int n, const cv::cuda::PtrStepSz< T_in > input, cv::cuda::PtrStepSz< T_out > output)

    applys the gaussian blur convolution to the input image

- template<typename T_in , typename T_out , typename F_cal >
  __global__ void gaussian_blur_x (float *kernel, int kernel_size, const cv::cuda::PtrStepSz< T_in > input, cv::cuda::PtrStepSz< T_out > output)

    applys the gaussian blur convolution to the input image along the x-axis

- template<typename T_in , typename T_out , typename F_cal >
  __global__ void gaussian_blur_y (float *kernel, int kernel_size, const cv::cuda::PtrStepSz< T_in > input, cv::cuda::PtrStepSz< T_out > output)

    applys the gaussian blur convolution to the input image along the y-axis

- template<typename... Ts>
  void gaussian_blur_exit (bool remove_globals, Ts &&...inputs)

    free all the GPU resources

- void call_gaussian_blur_2d (float *d_kernel, const int &n, const cv::cuda::GpuMat &input, cv::cuda::GpuMat &output)

    calls the gaussian_blur function appropriately based on the type of image

- void call_gaussian_blur_1d (float *d_kernel, const int &n, const cv::cuda::GpuMat &input, cv::cuda::GpuMat &output)

    calls the separable gaussian_blur function appropriately based on the type of image

- __host__ void gaussian_blur (const cv::Mat &input, cv::Mat &output, const int n=3, const float sigma=1.0, bool two_d=true, bool remove_globals=true)

    the gaussian blur function which runs on the HOST CPU. It calls the `call_gaussian_blur` *function after initialization of the appropriate values and kernel.*

- void gaussian_blur_init (const cv::Mat &input, cv::Mat &output)

    initialization for gaussian blurring operation

- void stress_test (const int &n, const bool &two_d)
- int main (int argc, char **argv)

Variables

- cv::cuda::GpuMat ginput
- cv::cuda::GpuMat goutput

3.1.1 Detailed Description

convolution blurring in Nvidia CUDA

Author

Arjun31415

Definition in file blur.cu.

3.1.2 Macro Definition Documentation

3.1.2.1 PBSTR

```
#define PBSTR "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||"
```

Definition at line 23 of file blur.cu.

3.1.2.2 PBWIDTH

```
#define PBWIDTH 60
```

Definition at line 24 of file blur.cu.

3.1.2.3 SAFE_CALL

```
#define SAFE_CALL(
              call,
              msg ) _safe_cuda_call((call), (msg), __FILE__, __LINE__)
```

a macro for sage calling CUDA functions

Parameters

call	the CUDA function call
msg	user specified message

Definition at line 60 of file blur.cu.

3.1.3 Function Documentation

3.1.3.1 add_value() [1/2]

```
__device__ __forceinline__ float add_value (
             float in1,
             float in2 )
```

add two floating point values

Parameters

in1	value 1
in2	value 2

Returns

the sum

Definition at line 220 of file blur.cu.

3.1.3.2 add_value() [2/2]

```
__device__ __forceinline__ float3 add_value (
             float3 in1,
             float3 in2 )
```

add two values and return it

Parameters

in1	input 1
in2	intput 2

Returns

returns the added value

Definition at line 208 of file blur.cu.

3.1.3.3 call_gaussian_blur_1d()

```
void call_gaussian_blur_1d (
            float * d_kernel,
            const int & n,
            const cv::cuda::GpuMat & input,
            cv::cuda::GpuMat & output )
```

calls the separable gaussian_blur function appropriately based on the type of image

Parameters

d_kernel	the kernel, stored on GPU device memory
n	the size of the kernel
input	the input image stored on the GPU
output	the output image stored on the GPU

Definition at line 465 of file blur.cu.

3.1.3.4 call_gaussian_blur_2d()

```
void call_gaussian_blur_2d (
            float * d_kernel,
            const int & n,
            const cv::cuda::GpuMat & input,
            cv::cuda::GpuMat & output )
```

calls the gaussian_blur function appropriately based on the type of image

Parameters

d_kernel	the kernel, stored on GPU device memory
n	the size of the kernel
input	the input image stored on the GPU
output	the output image stored on the GPU

Definition at line 432 of file blur.cu.

3.1.3.5 gaussian_blur() [1/2]

```
__host__ void gaussian_blur (
            const cv::Mat & input,
            cv::Mat & output,
            const int n = 3,
            const float sigma = 1.0,
```

```
bool two_d = true,
bool remove_globals = true )
```

the gaussian blur function which runs on the HOST CPU. It calls the `call_gaussian_blur` function after initialization of the appropriate values and kernel.

Parameters

input	the input image stored on the CPU memory
output	the output image stored on the CPU memory
n	the size of the Gaussian kernel, defaults to 3
sigma	the standard deviation of the Gaussian kernel, defaults to 1.
two←_d	whether to use the 2D gaussian blur kernel or two separable 1D gaussian blur kernels, defaults to true

Definition at line 505 of file blur.cu.

3.1.3.6 gaussian_blur() [2/2]

```
template<typename T_in , typename T_out , typename F_cal >
__global__ void gaussian_blur (
            const float * kernel,
            int n,
            const cv::cuda::PtrStepSz< T_in > input,
            cv::cuda::PtrStepSz< T_out > output )
```

applys the gaussian blur convolution to the input image

Template Parameters

t_in	the type of input image, i.e uchar for black and white, uchar3 for rgb, float3 etc
t_out	the type of output image
f_cal	the type for calculating intermediate sums and products

Parameters

kernel	the kernel to apply the convolution
n	the dimension of the kernel $(n \times n)$
input	the input image
output	the output image

Definition at line 290 of file blur.cu.

3.1.3.7 gaussian_blur_exit()

```
template<typename... Ts>
void gaussian_blur_exit (
```

```
            bool remove_globals,
            Ts &&... inputs )
```

free all the GPU resources

Template Parameters

Ts	

Parameters

inputs	varidaic list of resources
remove_globals	if true, removes the global variables, otherwise not, if Not then user has to handle the removal of the global variables and freeing the GPU memory

Definition at line 413 of file blur.cu.

3.1.3.8 gaussian_blur_init()

```
void gaussian_blur_init (
            const cv::Mat & input,
            cv::Mat & output )
```

initialization for gaussian blurring operation

Parameters

input	input image stored on the CPU
output	output image stored on the CPU

Definition at line 543 of file blur.cu.

3.1.3.9 gaussian_blur_x()

```
template<typename T_in , typename T_out , typename F_cal >
__global__ void gaussian_blur_x (
            float * kernel,
            int kernel_size,
            const cv::cuda::PtrStepSz< T_in > input,
            cv::cuda::PtrStepSz< T_out > output )
```

applys the gaussian blur convolution to the input image along the x-axis

Template Parameters

t_in	the type of input image, i.e uchar for black and white, uchar3 for rgb, float3 etc
t_out	the type of output image
f_cal	the type for calculating intermediate sums and products

Parameters

kernel	the kernel to apply the convolution
kernel_size	the dimension of the kernel
input	the input image
output	the output image

Definition at line 336 of file blur.cu.

3.1.3.10 gaussian_blur_y()

```
template<typename T_in , typename T_out , typename F_cal >
__global__ void gaussian_blur_y (
            float * kernel,
            int kernel_size,
            const cv::cuda::PtrStepSz< T_in > input,
            cv::cuda::PtrStepSz< T_out > output )
```

applys the gaussian blur convolution to the input image along the y-axis

Template Parameters

t_in	the type of input image, i.e uchar for black and white, uchar3 for rgb, float3 etc
t_out	the type of output image
f_cal	the type for calculating intermediate sums and products

Parameters

kernel	the kernel to apply the convolution
kernel_size	the dimension of the kernel
input	the input image
output	the output image

Definition at line 376 of file blur.cu.

3.1.3.11 generate_gaussian_kernel_1d()

```
__host__ void generate_gaussian_kernel_1d (
            float * kernel,
            const int n,
            const float sigma = 1 )
```

generate a 1D gaussian kernel

Parameters

kernel	the array in which the weights are stored
n	the size of the kernel. a 1D kernel of length n is needed
sigma	the standard deviation of the kernel

Returns

Definition at line 106 of file blur.cu.

3.1.3.12 generate_gaussian_kernel_2d()

```
__host__ void generate_gaussian_kernel_2d (
            float * kernel,
            const int n,
            const float sigma = 1 )
```

generate the gaussian kernel with given kernel size and standard deviation

Parameters

kernel	the array in which the weights are stored
n	the size of the kernel, t.e. n x n kernel is needed
sigma	the standard deviation

Definition at line 70 of file blur.cu.

3.1.3.13 main()

```
int main (
            int argc,
            char ** argv )
```

Definition at line 565 of file blur.cu.

3.1.3.14 multiply_value() [1/3]

```
__device__ __forceinline__ float3 multiply_value (
            const float & x,
            const float3 & y )
```

multiplication for float and float3 types. Multiply each filed in uchar3 with the float value and return a float3

Parameters

x	Input 1
y	Input 2

Returns

value after multiplication

Definition at line 259 of file blur.cu.

3.1.3.15 multiply_value() [2/3]

```
__device__ __forceinline__ float multiply_value (
            const float & x,
            const uchar & y )
```

multiplication for float and uchar4 types

Parameters

x	Input 1
y	Input 2

Returns

x*y

Definition at line 272 of file blur.cu.

3.1.3.16 multiply_value() [3/3]

```
__device__ __forceinline__ float3 multiply_value (
            const float & x,
            const uchar3 & y )
```

multiplication for float and uchar3 types. Multiply each filed in uchar3 with the float value and return a flolat3

Parameters

x	Input 1
y	Input 2

Returns

value after multiplication

Definition at line 245 of file blur.cu.

3.1.3.17 printProgress()

```
void printProgress (
            double percentage )
```

Definition at line 25 of file blur.cu.

3.1.3.18 set_value() [1/5]

```
__device__ __forceinline__ void set_value (
            const float & val,
            float & out )
```

set the value for a floating point type.

Parameters

val	the value
out	the output

Definition at line 142 of file blur.cu.

3.1.3.19 set_value() [2/5]

```
__device__ __forceinline__ void set_value (
            const float & val,
            float3 & out )
```

set the value for a float3 tupe. All the 3 fields will have the value `val`

Parameters

val	the value
out	the output

Definition at line 154 of file blur.cu.

3.1.3.20 set_value() [3/5]

```
__device__ __forceinline__ void set_value (
            const float3 & val,
            uchar3 & out )
```

set the value for a unsigned char3 type with a flot3 type

Parameters

val	the value to set
out	the ouput

Definition at line 165 of file blur.cu.

3.1.3.21 set_value() [4/5]

```
__device__ __forceinline__ void set_value (
            const int & val,
            uchar & out )
```

Sets the value of a uchar type.

Parameters

val	The value
out	The output

Definition at line 131 of file blur.cu.

3.1.3.22 set_value() [5/5]

```
__device__ __forceinline__ void set_value (
            const int & val,
            uchar3 & out )
```

Sets the value of a uchar3 type.

Parameters

in	*val*	The value
	out	The output

Definition at line 177 of file blur.cu.

3.1.3.23 stress_test()

```
void stress_test (
            const int & n,
            const bool & two_d )
```

Definition at line 548 of file blur.cu.

3.1.3.24 subtract_value() [1/2]

```
__device__ __forceinline__ uchar subtract_value (
            uchar in1,
            uchar in2 )
```

Subtraction for uchar types.

Parameters

in	*in1*	Input 1
in	*in2*	Input 2

Returns

Output

Definition at line 233 of file blur.cu.

3.1.3.25 subtract_value() [2/2]

```
__device__ __forceinline__ uchar3 subtract_value (
            uchar3 in1,
            uchar3 in2 )
```

Subtraction for uchar3 types.

Parameters

in	*in1*	Input 1
in	*in2*	Input 2

Returns

Output

Definition at line 192 of file blur.cu.

3.1.4 Variable Documentation

3.1.4.1 ginput

```
cv::cuda::GpuMat ginput
```

Definition at line 20 of file blur.cu.

3.1.4.2 goutput

```
cv::cuda::GpuMat goutput
```

Definition at line 20 of file blur.cu.

3.2 blur.cu

Go to the documentation of this file.

```
00001
00007 #include <cstring>
00008 #undef __noinline__
00009 #include <cuda_profiler_api.h>
00010 #include <cuda_runtime.h>
00011 #include <iostream>
00012 #include <opencv2/core/core.hpp>
00013 #include <opencv2/core/cuda.hpp>
00014 #include <opencv2/core/cuda/common.hpp>
00015 #include <opencv2/core/matx.hpp>
00016 #include <opencv2/cudaimgproc.hpp>
00017 #include <opencv2/highgui.hpp>
00018 #include <opencv2/opencv.hpp>
00019 #include <stdio.h>
00020 cv::cuda::GpuMat ginput, goutput;
00021
00022 // Progress Bar STRing
00023 #define PBSTR "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||"
00024 #define PBWIDTH 60
00025 void printProgress(double percentage)
00026 {
00027     int val = (int)(percentage * 100);
00028     int lpad = (int)(percentage * PBWIDTH);
00029     int rpad = PBWIDTH - lpad;
00030     printf("\r%3d%% [%.*s%*s]", val, lpad, PBSTR, rpad, "");
00031     fflush(stdout);
00032 }
00033
00043 static inline void _safe_cuda_call(cudaError err, const char *msg,
00044                                    const char *file_name, const int line_number)
00045 {
00046     if (err != cudaSuccess)
00047     {
00048         fprintf(stderr, "%s\n\nFile: %s\n\nLine Number: %d\n\nReason: %s\n",
00049                 msg, file_name, line_number, cudaGetErrorString(err));
00050         std::cin.get();
00051         exit(EXIT_FAILURE);
00052     }
00053 }
00060 #define SAFE_CALL(call, msg) _safe_cuda_call((call), (msg), __FILE__, __LINE__)
00061
00070 __host__ void generate_gaussian_kernel_2d(float *kernel, const int n,
00071                                           const float sigma = 1)
00072 {
00073     int mean = n / 2;
00074     float sumOfWeights = 0;
```

```
    float p, q = 2.0 * sigma * sigma;

    // Compute weights
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            p = sqrt((i - mean) * (i - mean) + (j - mean) * (j - mean));
            kernel[i * n + j] = std::exp((-(p * p) / q)) / (M_PI * q);
            sumOfWeights += kernel[i * n + j];
        }
    }

    // Normalizing weights
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            kernel[i * n + j] /= sumOfWeights;
        }
    }
}

__host__ void generate_gaussian_kernel_1d(float *kernel, const int n,
                                          const float sigma = 1)
{
    // Calculate the values of the kernel
    float sum = 0.0f;
    for (int i = 0; i < n; i++)
    {
        float x = i - (n - 1) / 2.0f;
        kernel[i] = std::exp(-x * x / (2 * sigma * sigma));
        sum += kernel[i];
    }

    // Normalize the kernel so that its sum equals 1
    for (int i = 0; i < n; i++)
    {
        kernel[i] /= sum;
    }
}

__device__ __forceinline__ void set_value(const int &val, uchar &out)
{
    out = val;
}

__device__ __forceinline__ void set_value(const float &val, float &out)
{
    out = val;
};

__device__ __forceinline__ void set_value(const float &val, float3 &out)
{
    out.x = val, out.y = val, out.z = val;
}

__device__ __forceinline__ void set_value(const float3 &val, uchar3 &out)
{
    out.x = val.x;
    out.y = val.y;
    out.z = val.z;
}
__device__ __forceinline__ void set_value(const int &val, uchar3 &out)
{
    out.x = val;
    out.y = val;
    out.z = val;
}

__device__ __forceinline__ uchar3 subtract_value(uchar3 in1, uchar3 in2)
{
    uchar3 out;
    out.x = in1.x - in2.x;
    out.y = in1.y - in2.y;
    out.z = in1.z - in2.z;
    return out;
}

__device__ __forceinline__ float3 add_value(float3 in1, float3 in2)
{
    return {in1.x + in2.x, in1.y + in2.y, in1.z + in2.z};
}

__device__ __forceinline__ float add_value(float in1, float in2)
{
    return in1 + in2;
```

```
}

__device__ __forceinline__ uchar subtract_value(uchar in1, uchar in2)
{
    return in1 - in2;
}
__device__ __forceinline__ float3 multiply_value(const float &x,
                                                 const uchar3 &y)
{
    return {x * (float)y.x, x * (float)y.y, x * (float)y.z};
}

__device__ __forceinline__ float3 multiply_value(const float &x,
                                                 const float3 &y)
{
    return {x * (float)y.x, x * (float)y.y, x * (float)y.z};
}

__device__ __forceinline__ float multiply_value(const float &x, const uchar &y)
{
    return x * (float)y;
}

template <typename T_in, typename T_out, typename F_cal>
__global__ void gaussian_blur(const float *kernel, int n,
                              const cv::cuda::PtrStepSz<T_in> input,
                              cv::cuda::PtrStepSz<T_out> output)
{
    // calculate the x & y position of the current image pixel
    const int x = blockIdx.x * blockDim.x + threadIdx.x;
    const int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= input.cols || y >= input.rows) return;

    const int mid = n / 2;
    F_cal sum;
    set_value(0, sum);
    // synchronize all the threads till this potin
    __syncthreads();

    // loop over the n x n neighborhood of the current pixel
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            int y_idx = y + i - mid;
            int x_idx = x + j - mid;
            if (y_idx > input.rows || x_idx > input.cols) continue;
            const float kernel_val = kernel[(n - i - 1) * n + (n - j - 1)];
            sum =
                add_value(sum, multiply_value(kernel_val, input(y_idx, x_idx)));
        }
    }
    T_out result;
    set_value(sum, result);
    output(y, x) = result;
}
template <typename T_in, typename T_out, typename F_cal>
__global__ void gaussian_blur_x(float *kernel, int kernel_size,
                                const cv::cuda::PtrStepSz<T_in> input,
                                cv::cuda::PtrStepSz<T_out> output)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    const int radius = kernel_size / 2;
    const int width = input.cols;
    const int height = input.rows;

    if (x >= input.cols || y >= input.rows) return;

    F_cal pixel;
    set_value(0, pixel);

    for (int i = -radius; i <= radius; i++)
    {
        int idx = y * width + (x + i);
        if (idx >= 0 && idx < width * height)
        {
            const float weight = kernel[i + radius];
            pixel = add_value(pixel, multiply_value(weight, input[idx]));
        }
    }
    set_value(pixel, output(y, x));
}

template <typename T_in, typename T_out, typename F_cal>
__global__ void gaussian_blur_y(float *kernel, int kernel_size,
```

```
                                 const cv::cuda::PtrStepSz<T_in> input,
                                 cv::cuda::PtrStepSz<T_out> output)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    const int radius = kernel_size / 2;
    const int width = input.cols;
    const int height = input.rows;

    if (x >= input.cols || y >= input.rows) return;

    F_cal pixel;
    set_value(0, pixel);
    float weight_sum = 0;
    for (int i = -radius; i <= radius; i++)
    {
        int idx = (y + i) * width + x;
        if (idx >= 0 && idx < width * height)
        {
            float weight = kernel[i + radius];
            pixel = add_value(pixel, multiply_value(weight, input[idx]));
        }
    }
    set_value(pixel, output(y, x)); // output(y,x) = pixel;
}

template <typename... Ts>
void gaussian_blur_exit(bool remove_globals, Ts &&...inputs)
{
    if (remove_globals)
    {
        ginput.release();
        goutput.release();
    }
    ([&] { SAFE_CALL(cudaFree(inputs), "Unable to free"); }(), ...);
}

void call_gaussian_blur_2d(float *d_kernel, const int &n,
                           const cv::cuda::GpuMat &input,
                           cv::cuda::GpuMat &output)
{
    CV_Assert(input.channels() == 1 || input.channels() == 3);
    const dim3 block(16, 16);

    // Calculate grid size to cover the whole image
    const dim3 grid(cv::cuda::device::divUp(input.cols, block.x),
                    cv::cuda::device::divUp(input.rows, block.y));
    if (input.channels() == 1)
    {
        gaussian_blur<uchar, uchar, float>
            «<grid, block»>(d_kernel, n, input, output);
        return;
    }
    else if (input.channels() == 3)
    {
        gaussian_blur<uchar3, uchar3, float3>
            «<grid, block»>(d_kernel, n, input, output);
    }
    cudaSafeCall(cudaGetLastError());
}
void call_gaussian_blur_1d(float *d_kernel, const int &n,
                           const cv::cuda::GpuMat &input,
                           cv::cuda::GpuMat &output)
{
    CV_Assert(input.channels() == 1 || input.channels() == 3);
    const int block_size = 16;
    dim3 dimBlock(block_size, block_size);
    dim3 dimGrid(cv::cuda::device::divUp(input.cols, dimBlock.x),
                 cv::cuda::device::divUp(input.rows, dimBlock.y));
    cv::cuda::GpuMat temp = input.clone();
    // Apply the horizontal Gaussian blur
    if (input.channels() == 1)
    {

        gaussian_blur_x<uchar, uchar, float>
            «<dimGrid, dimBlock»>(d_kernel, n, input, temp);
        gaussian_blur_y<uchar, uchar, float>
            «<dimGrid, dimBlock»>(d_kernel, n, temp, output);
    }
    else if (input.channels() == 3)
    {
        gaussian_blur_x<uchar3, uchar3, float3>
            «<dimGrid, dimBlock»>(d_kernel, n, input, temp);
        gaussian_blur_y<uchar3, uchar3, float3>
            «<dimGrid, dimBlock»>(d_kernel, n, temp, output);
    }
    cudaSafeCall(cudaGetLastError());
```

```
}
__host__ void gaussian_blur(const cv::Mat &input, cv::Mat &output,
                            const int n = 3, const float sigma = 1.0,
                            bool two_d = true, bool remove_globals = true)
{
    ginput.upload(input);
    std::vector<float> gauss_kernel_host;
    float *d_gauss_kernel;
    if (two_d)
    {
        gauss_kernel_host = std::vector<float>(n * n);
        generate_gaussian_kernel_2d(gauss_kernel_host.data(), n, sigma);
        cudaMalloc((void **)&d_gauss_kernel, n * n * sizeof(float));
        SAFE_CALL(cudaMemcpy(d_gauss_kernel, gauss_kernel_host.data(),
                             sizeof(float) * n * n, cudaMemcpyHostToDevice),
                  "Unable to copy kernel");
        call_gaussian_blur_2d(d_gauss_kernel, n, ginput, goutput);
    }
    else
    {

        gauss_kernel_host = std::vector<float>(n);
        generate_gaussian_kernel_1d(gauss_kernel_host.data(), n, sigma);
        cudaMalloc((void **)&d_gauss_kernel, n * sizeof(float));
        SAFE_CALL(cudaMemcpy(d_gauss_kernel, gauss_kernel_host.data(),
                             sizeof(float) * n, cudaMemcpyHostToDevice),
                  "Unable to copy kernel");
        call_gaussian_blur_1d(d_gauss_kernel, n, ginput, goutput);
    }
    goutput.download(output);
    gaussian_blur_exit(remove_globals, d_gauss_kernel);
}

void gaussian_blur_init(const cv::Mat &input, cv::Mat &output)
{
    ginput.create(input.rows, input.cols, input.type());
    goutput.create(output.rows, output.cols, output.type());
}
void stress_test(const int &n, const bool &two_d)
{
    std::cout « "Kernel size: " « n « std::endl;
    const std::string path = "../images/peppers_color.tif";
    cv::Mat input = cv::imread(path, 1);
    auto output = input.clone();
    gaussian_blur_init(input, output);
    for (int i = 0; i < 100; i++)
    {
        printProgress((float)i / 100);
        gaussian_blur(input, output, n, 1.7, two_d, false);
    }
    std::cout « std::endl;
    ginput.release();
    goutput.release();
    return;
}
int main(int argc, char **argv)
{

    if (argc < 3)
    {
        printf("usage: Blur_Test <kernel_size> <Image_Path> [<Output_Path>]\n");
        return -1;
    }
    std::string mTitle = "Display Image";
    cv::Mat input;
    int n = atoi(argv[1]);
    if (strncmp(argv[2], "stress2d", 8) == 0)
    {
        stress_test(n, true);
        return 0;
    }
    else if (strncmp(argv[2], "stress1d", 8) == 0)
    {
        stress_test(n, false);
        return 0;
    }
    input = cv::imread(argv[2], 1);
    if (!input.data)
    {
        printf("No image data \n");
        return -1;
    }
    auto output = input.clone();

    // Call the wrapper function
    gaussian_blur_init(input, output);
    gaussian_blur(input, output, n, 1.7, 0);
```

```

    // Show the input and output
    cv::imshow("Output", output);

    // Wait for key press
    cv::waitKey();
    namedWindow(mTitle, cv::WINDOW_AUTOSIZE);
    imshow(mTitle, input);
    if (argc >= 4) imwrite(argv[3], output);
    do
    {

        auto k = cv::waitKey(500);
        if (k == 27)
        {
            cv::destroyAllWindows();
            return 0;
        }
        if (cv::getWindowProperty(mTitle, cv::WND_PROP_VISIBLE) == 0) return 0;

    } while (true);

    return 0;
}
```

3.3 main.cpp File Reference

gaussian blurring using CPU

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <numeric>
#include <omp.h>
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/opencv.hpp>
#include <string>
#include <vector>
```

Macros

- #define PBSTR "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||"
- #define PBWIDTH 60

Functions

- void printProgress (double percentage)
- void generate_gaussian_kernel (std::vector< std::vector< float > > &kernel, const int n, const float sigma=1)

  Generate a 2D gaussian kernel.
- void apply_convolution (const std ::vector< std::vector< float > > &kernel, const Mat &original_img, Mat &new_img, const int &r, const int &c)

  apply a convolution kernel to a pixel
- void apply_convolution_multi_threaded (const std::vector< std::vector< float > > &kernel, const Mat &original_img, Mat &new_img, const int &r, const int &c)

  apply a convolution kernel to a pixel using multiple threads (OMP)
- void apply_kernel (const std::vector< std::vector< float > > &kernel, const Mat &original_img, Mat &new_img)

apply a convolution kernel to the entire image

- void apply_kernel_multithreaded (const std::vector< std::vector< float > > &kernel, const Mat &original_↩ img, Mat &new_img)

  apply a convolution kernel to the entire image using multiple threads (OMP)

- void stress_test (const int &n, const bool &multi=true)
- int main (int argc, char **argv)

3.3.1 Detailed Description

gaussian blurring using CPU

Author

Arjun31415

Definition in file main.cpp.

3.3.2 Macro Definition Documentation

3.3.2.1 PBSTR

```
#define PBSTR "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||"
```

Definition at line 20 of file main.cpp.

3.3.2.2 PBWIDTH

```
#define PBWIDTH 60
```

Definition at line 21 of file main.cpp.

3.3.3 Function Documentation

3.3.3.1 apply_convolution()

```
void apply_convolution (
            const std ::vector< std::vector< float > > & kernel,
            const Mat & original_img,
            Mat & new_img,
            const int & r,
            const int & c )
```

apply a convolution kernel to a pixel

Parameters

kernel	the convolution kernel
original_img	the original image
new_img	the output image
r	the row number of the current pixel
c	the column number of the current pixel

Definition at line 76 of file main.cpp.

3.3.3.2 apply_convolution_multi_threaded()

```
void apply_convolution_multi_threaded (
            const std::vector< std::vector< float > > & kernel,
            const Mat & original_img,
            Mat & new_img,
            const int & r,
            const int & c )
```

apply a convolution kernel to a pixel using multiple threads (OMP)

Parameters

kernel	the convolution kernel
original_img	the original image
new_img	the output image
r	the row number of the current pixel
c	the column number of the current pixel

Definition at line 106 of file main.cpp.

3.3.3.3 apply_kernel()

```
void apply_kernel (
            const std::vector< std::vector< float > > & kernel,
            const Mat & original_img,
            Mat & new_img )
```

apply a convolution kernel to the entire image

Parameters

kernel	the convolution kernel
original_img	the original image
new_img	the output image

Definition at line 136 of file main.cpp.

3.3.3.4 apply_kernel_multithreaded()

```
void apply_kernel_multithreaded (
            const std::vector< std::vector< float > > & kernel,
            const Mat & original_img,
            Mat & new_img )
```

apply a convolution kernel to the entire image using multiple threads (OMP)

Parameters

kernel	the convolution kernel
original_img	the original_img
new_img	the output image

Definition at line 155 of file main.cpp.

3.3.3.5 generate_gaussian_kernel()

```
void generate_gaussian_kernel (
            std::vector< std::vector< float > > & kernel,
            const int n,
            const float sigma = 1 )
```

Generate a 2D gaussian kernel.

Parameters

kernel	the kernel to be populated
n	the size of the kernel, the `kernel` must be of size $n ∗ n$
sigma	the standard deviation of the gaussian kernel

Definition at line 39 of file main.cpp.

3.3.3.6 main()

```
int main (
            int argc,
            char ** argv )
```

Definition at line 234 of file main.cpp.

3.3.3.7 printProgress()

```
void printProgress (
            double percentage )
```

Definition at line 23 of file main.cpp.

3.3.3.8 stress_test()

```
void stress_test (
            const int & n,
            const bool & multi = true )
```

Definition at line 170 of file main.cpp.

3.4 main.cpp

Go to the documentation of this file.

```
00001
00007 #include <cmath>
00008 #include <fstream>
00009 #include <iostream>
00010 #include <numeric>
00011 #include <omp.h>
00012 #include <opencv2/core.hpp>
00013 #include <opencv2/highgui.hpp>
00014 #include <opencv2/imgcodecs.hpp>
00015 #include <opencv2/opencv.hpp>
00016 #include <string>
00017 #include <vector>
00018 using namespace cv;
00019
00020 #define PBSTR "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||"
00021 #define PBWIDTH 60
00022
00023 void printProgress(double percentage)
00024 {
00025     int val = (int)(percentage * 100);
00026     int lpad = (int)(percentage * PBWIDTH);
00027     int rpad = PBWIDTH - lpad;
00028     printf("\r%3d%% [%.*s%*s]", val, lpad, PBSTR, rpad, "");
00029     fflush(stdout);
00030 }
00031
00039 void generate_gaussian_kernel(std::vector<std::vector<float» &kernel,
00040                               const int n, const float sigma = 1)
00041 {
00042     int mean = n / 2;
00043     float sumOfWeights = 0;
00044     float p, q = 2.0 * sigma * sigma;
00045
00046     // Compute weights
00047     for (int i = 0; i < n; i++)
00048     {
00049         for (int j = 0; j < n; j++)
00050         {
00051             p = sqrt((i - mean) * (i - mean) + (j - mean) * (j - mean));
00052             kernel[i][j] = std::exp((-(p * p) / q)) / (M_PI * q);
00053             sumOfWeights += kernel[i][j];
00054         }
00055     }
00056
00057     // Normalizing weights
00058     for (int i = 0; i < n; i++)
00059     {
00060         for (int j = 0; j < n; j++)
00061         {
00062             kernel[i][j] /= sumOfWeights;
```

```
        }
    }
}
void apply_convolution(const std ::vector<std::vector<float» &kernel,
                       const Mat &original_img, Mat &new_img, const int &r,
                       const int &c)
{
    const size_t n = kernel.size();
    assert(n % 2 == 1);
    assert(n == kernel[0].size());
    const size_t mid = n / 2;
    new_img.at<Vec3b>(r, c) = {0, 0, 0};
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (r - mid + i >= 0 && r - mid + i < original_img.rows &&
                c - mid + j >= 0 && c - mid + j < original_img.cols)
                new_img.at<Vec3b>(r, c) +=
                    kernel[n - i - 1][n - j - 1] *
                    original_img.at<Vec3b>(r - mid + i, c - mid + j);
        }
    }
}
void apply_convolution_multi_threaded(
    const std::vector<std::vector<float» &kernel, const Mat &original_img,
    Mat &new_img, const int &r, const int &c)
{
    const size_t n = kernel.size();
    assert(n % 2 == 1);
    assert(n == kernel[0].size());
    const size_t mid = n / 2;
    new_img.at<Vec3b>(r, c) = {0, 0, 0};
#pragma omp parallel for shared(r, c, original_img, new_img, kernel)
    for (int i = 0; i < n; i++)
    {
#pragma omp parallel for shared(r, c, original_img, new_img, kernel)
        for (int j = 0; j < n; j++)
        {
            if (r - mid + i >= 0 && r - mid + i < original_img.rows &&
                c - mid + j >= 0 && c - mid + j < original_img.cols)
                new_img.at<Vec3b>(r, c) +=
                    kernel[n - i - 1][n - j - 1] *
                    original_img.at<Vec3b>(r - mid + i, c - mid + j);
        }
    }
}
void apply_kernel(const std::vector<std::vector<float» &kernel,
                  const Mat &original_img, Mat &new_img)
{
    for (int i = 0; i < original_img.rows; i++)
    {
        for (int j = 0; j < original_img.cols; j++)
        {
            apply_convolution(kernel, original_img, new_img, i, j);
        }
    }
}
void apply_kernel_multithreaded(const std::vector<std::vector<float» &kernel,
                                const Mat &original_img, Mat &new_img)
{
#pragma omp barrier
#pragma omp parallel for shared(original_img, new_img, kernel)
    for (int i = 0; i < original_img.rows; i++)
    {
#pragma omp parallel for shared(original_img, new_img, kernel)
        for (int j = 0; j < original_img.cols; j++)
        {
            apply_convolution(kernel, original_img, new_img, i, j);
        }
    }
#pragma omp barrier
}
void stress_test(const int &n, const bool &multi = true)
{
    std::cout « "Stress testing" « std::endl;
    const std::string path = "../images/peppers_color.tif";
    auto image = imread(path, 1);
    auto new_img = image.clone();
    std::vector<std::vector<float» gauss_kernel(n, std::vector<float>(n));
    generate_gaussian_kernel(gauss_kernel, n, 1.6);
    std::vector<double> run_times;
    const int num_runs = 20;
    std::string fname;
    if (!multi)
    {
        fname = "profile_single_threaded.csv";
```

```
        for (int i = 0; i < num_runs; i++)
        {
            printProgress((float)i / num_runs);
            auto start = std::chrono::high_resolution_clock::now();
            apply_kernel(gauss_kernel, image, new_img);
            auto end = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double, std::milli> duration_ms = end - start;
            run_times.push_back(duration_ms.count());
        }
    }
    else
    {
        fname = "profile_multi_threaded.csv";
        for (int i = 0; i < num_runs; i++)
        {
            printProgress((float)i / num_runs);
            auto start = std::chrono::high_resolution_clock::now();
            apply_kernel_multithreaded(gauss_kernel, image, new_img);
            auto end = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double, std::milli> duration_ms = end - start;
            run_times.push_back(duration_ms.count());
        }
    }
    sort(run_times.begin(), run_times.end());
    double avg = std::accumulate(run_times.begin(), run_times.end(), 0.0) /
                 run_times.size();
    double median =
        ((run_times.size() % 2 == 0) ? (run_times[run_times.size() / 2 - 1] +
                                        run_times[run_times.size() / 2]) /
                                           2
                                     : run_times[run_times.size() / 2]);

    std::cout « "\nMax(ms)\t\tAvg(ms)\t\tMedian(ms)\t\tMin(ms)\n";
    std::cout « run_times.back() « "\t\t" « avg « "\t\t" « median

              « "\t\t" « run_times.front() « "\n";
    std::fstream file(fname, std::ios::in | std::ios_base::app);
    if (file.tellg() == 0)
    {
        // write the headers if the file is empty
        file « "KERNEL_SIZE,MAX_RUN_TIME,MIN_RUN_TIME,AVG_RUN_TIME,MEDIAN_RUN_"
                "TIME"
             « std::endl;
    }
    file « n « "," « run_times.back() « "," « run_times.front() « ","
         « avg « "," « median « std::endl;

    file.close();
}
int main(int argc, char **argv)
{
    if (argc < 3)
    {
        printf("usage: Blur_Test <kernel_size> <Image_Path> [<Output_Path>]\n");
        return -1;
    }
    int n = atoi(argv[1]);
    if (strncmp(argv[2], "stressm", 7) == 0)
    {
        stress_test(n, true);
        return 0;
    }
    else if (strncmp(argv[2], "stress", 6) == 0)
    {
        stress_test(n, false);
        return 0;
    }

    std::vector<std::vector<float» gauss_kernel(n, std::vector<float>(n));
    generate_gaussian_kernel(gauss_kernel, n, 1.6);

    std::string mTitle = "Display Image";
    Mat image;
    image = imread(argv[2], 1);
    if (!image.data)
    {
        printf("No image data \n");
        return -1;
    }
    namedWindow(mTitle, WINDOW_AUTOSIZE);
    auto new_img = image.clone();
    apply_kernel_multithreaded(gauss_kernel, image, new_img);
    imshow(mTitle, image);
    imshow("gaussian", new_img);
    if (argc >= 4) imwrite(argv[3], new_img);
    do
```

```
    {

        auto k = waitKey(500);
        if (k == 27)
        {
            cv::destroyAllWindows();
            return 0;
        }
        if (cv::getWindowProperty(mTitle, WND_PROP_VISIBLE) == 0)
        {
            return 0;
            break;
        }

    } while (true);
    return 0;
}
```

Index