

Fast Gaussian Blur

1.0.0

By Arjun Prashanth 20BCE1029

1 Report	1
1.1 Abstract	1
1.2 Dataset	1
1.3 Technologies Used -	2
1.4 Methodology	2
1.4.1 CPU -	2
1.4.2 GPU -	2
1.4.2.1 2D Kernel -	2
1.4.2.2 Separable Kernel (1D) -	2
1.5 Results -	3
1.5.1 Plots -	3
1.5.1.1 CPU -	3
1.5.1.2 GPU -	6
1.6 Inference	12
2 File Index	13
2.1 File List	13
3 File Documentation	15
3.1 blur.cu File Reference	15
3.1.1 Detailed Description	17
3.1.2 Macro Definition Documentation	17
3.1.2.1 PBSTR	17
3.1.2.2 PBWIDTH	17
3.1.2.3 SAFE_CALL	17
3.1.3 Function Documentation	18
3.1.3.1 add_value() [1/2]	18
3.1.3.2 add_value() [2/2]	18
3.1.3.3 call_gaussian_blur_1d()	19
3.1.3.4 call_gaussian_blur_2d()	19
3.1.3.5 gaussian_blur() [1/2]	19
3.1.3.6 gaussian_blur() [2/2]	20
3.1.3.7 gaussian_blur_exit()	20
3.1.3.8 gaussian_blur_init()	21
3.1.3.9 gaussian_blur_x()	21
3.1.3.10 gaussian_blur_y()	22
3.1.3.11 generate_gaussian_kernel_1d()	22
3.1.3.12 generate_gaussian_kernel_2d()	23
3.1.3.13 main()	23
3.1.3.14 multiply_value() [1/3]	23
3.1.3.15 multiply_value() [2/3]	24
3.1.3.16 multiply_value() [3/3]	24
3.1.3.17 printProgress()	25

3.1.3.18 set_value() [1/5]	25
3.1.3.19 set_value() [2/5]	25
3.1.3.20 set_value() [3/5]	26
3.1.3.21 set_value() [4/5]	26
3.1.3.22 set_value() [5/5]	26
3.1.3.23 stress_test()	27
3.1.3.24 subtract_value() [1/2]	27
3.1.3.25 subtract_value() [2/2]	27
3.1.4 Variable Documentation	28
3.1.4.1 ginput	28
3.1.4.2 goutput	28
3.2 blur.cu	28
3.3 main.cpp File Reference	33
3.3.1 Detailed Description	34
3.3.2 Macro Definition Documentation	34
3.3.2.1 PBSTR	34
3.3.2.2 PBWIDTH	34
3.3.3 Function Documentation	34
3.3.3.1 apply_convolution()	34
3.3.3.2 apply_convolution_multi_threaded()	35
3.3.3.3 apply_kernel()	35
3.3.3.4 apply_kernel_multithreaded()	36
3.3.3.5 generate_gaussian_kernel()	36
3.3.3.6 main()	36
3.3.3.7 printProgress()	37
3.3.3.8 stress_test()	37
3.4 main.cpp	37
Index	41

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 processing, 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 output 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 -

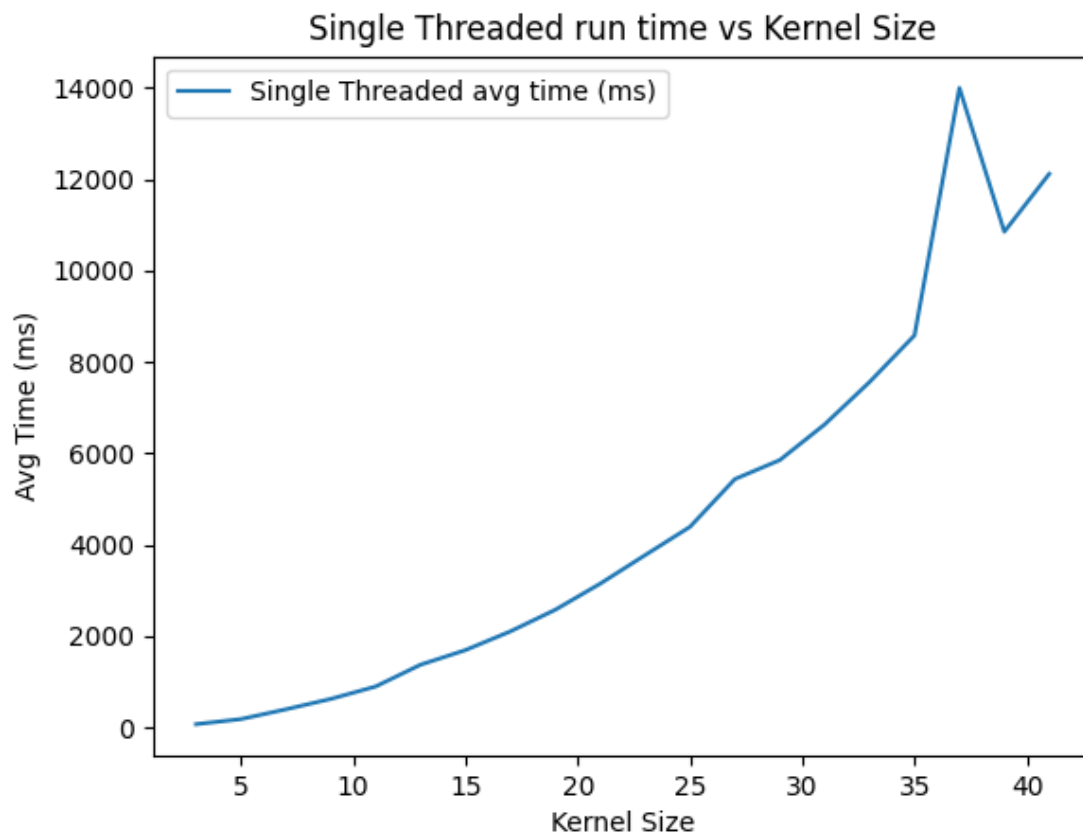


Figure 1.1 Gaussian Blur CPU Single Threaded

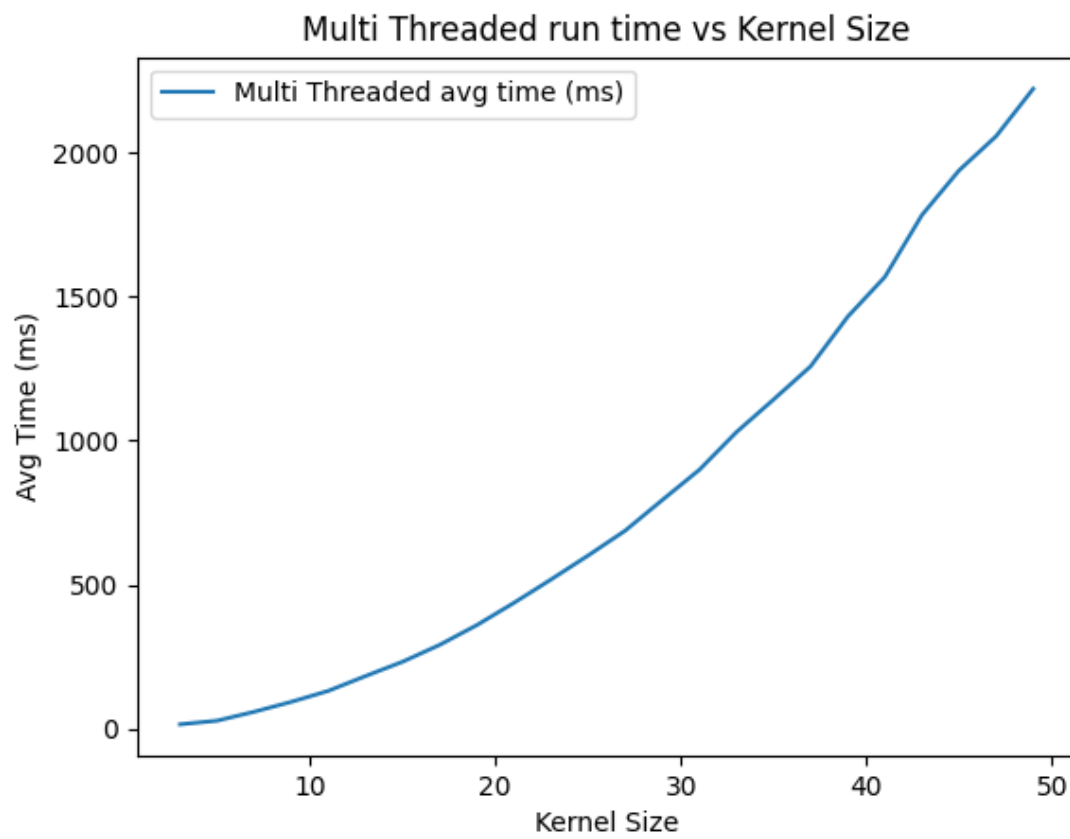


Figure 1.2 Gaussian Blur CPU Multiple Threaded

Comparing Single Threaded vs Multi Threaded

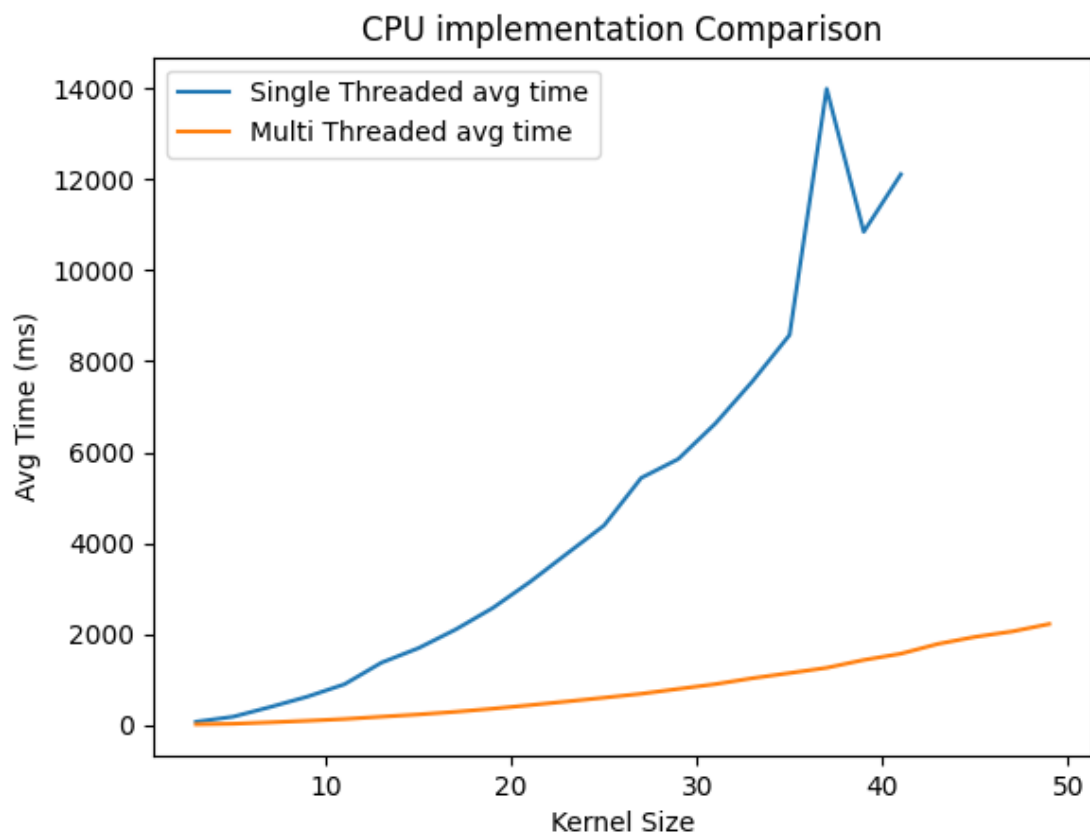


Figure 1.3 Gaussian Blur CPU Comparison

Average Speedup: 7.31 times

1.5.1.2 GPU -

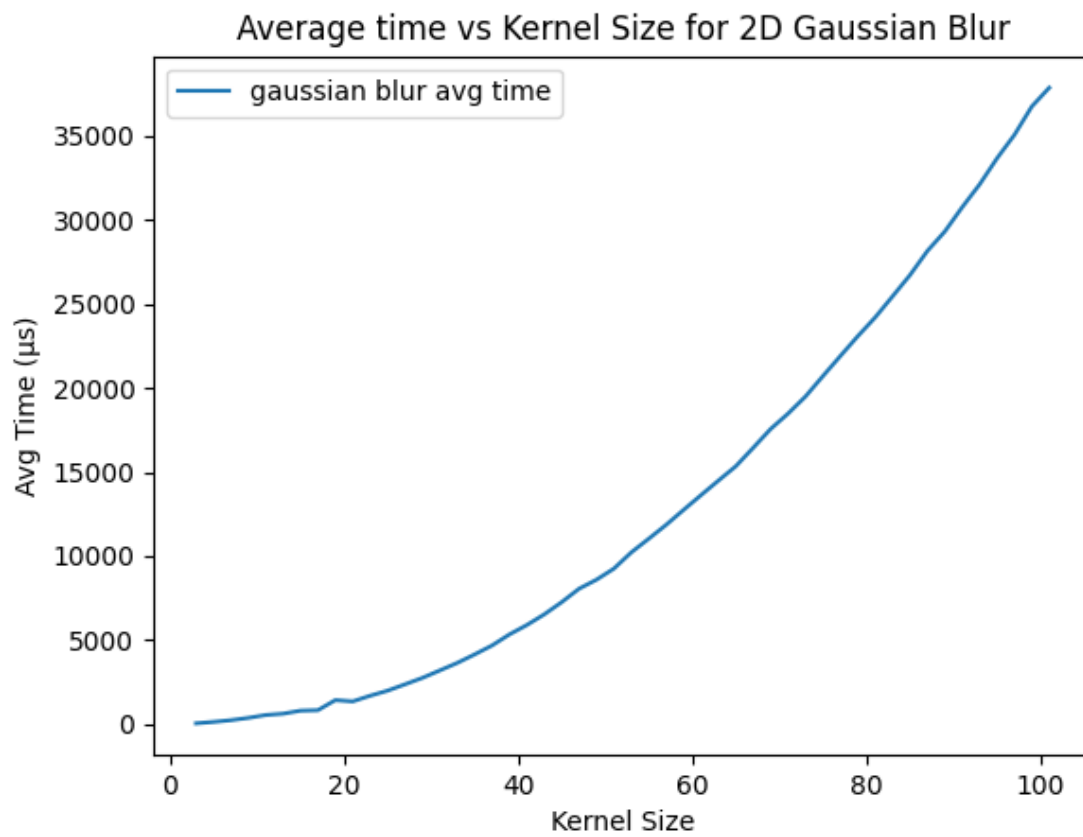


Figure 1.4 Gaussian Blur 2D kernel

Separable Kernel -

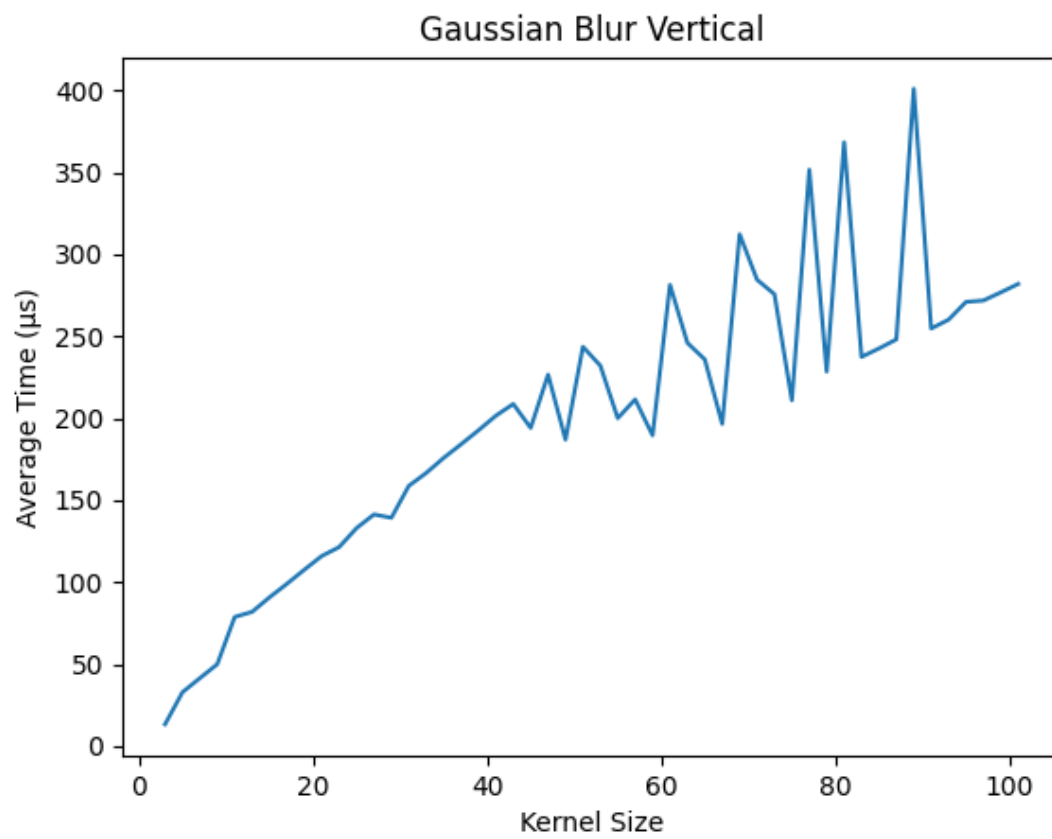


Figure 1.5 Gaussian Blur Separable Kernel Vertical

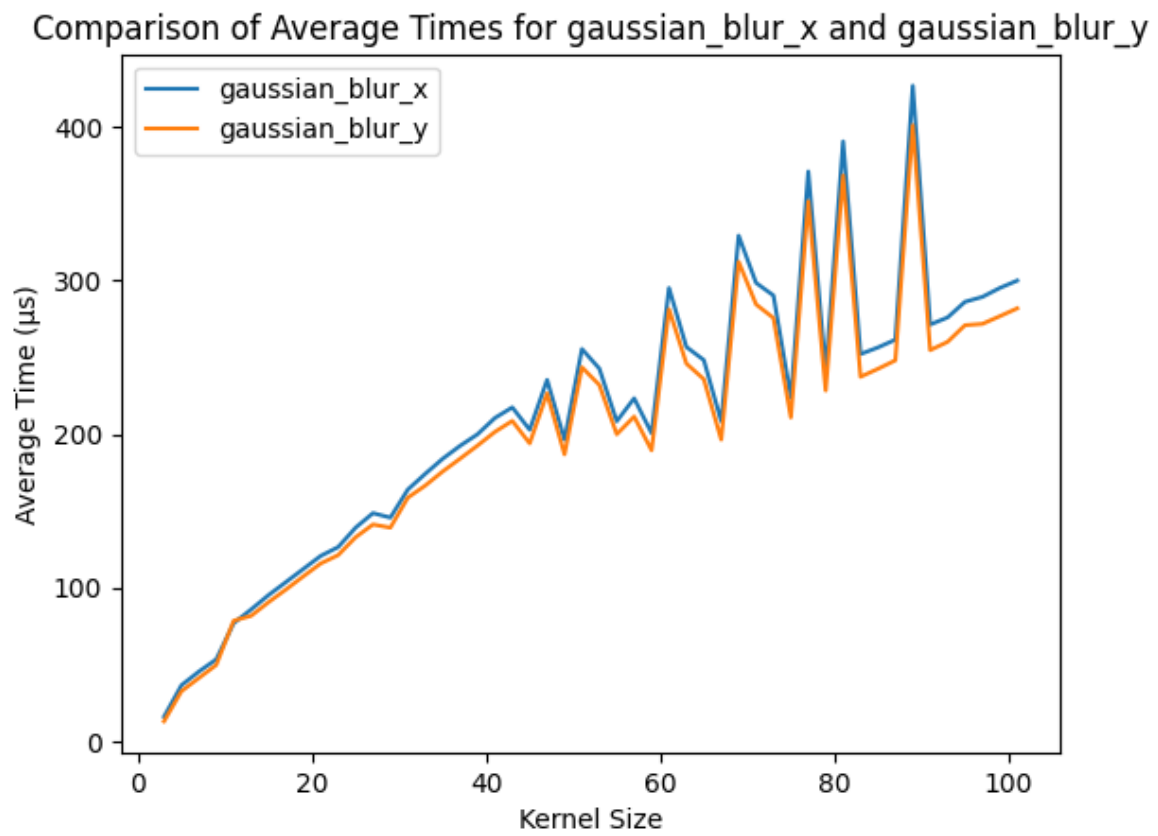


Figure 1.6 Gaussian Blur Separable kernel

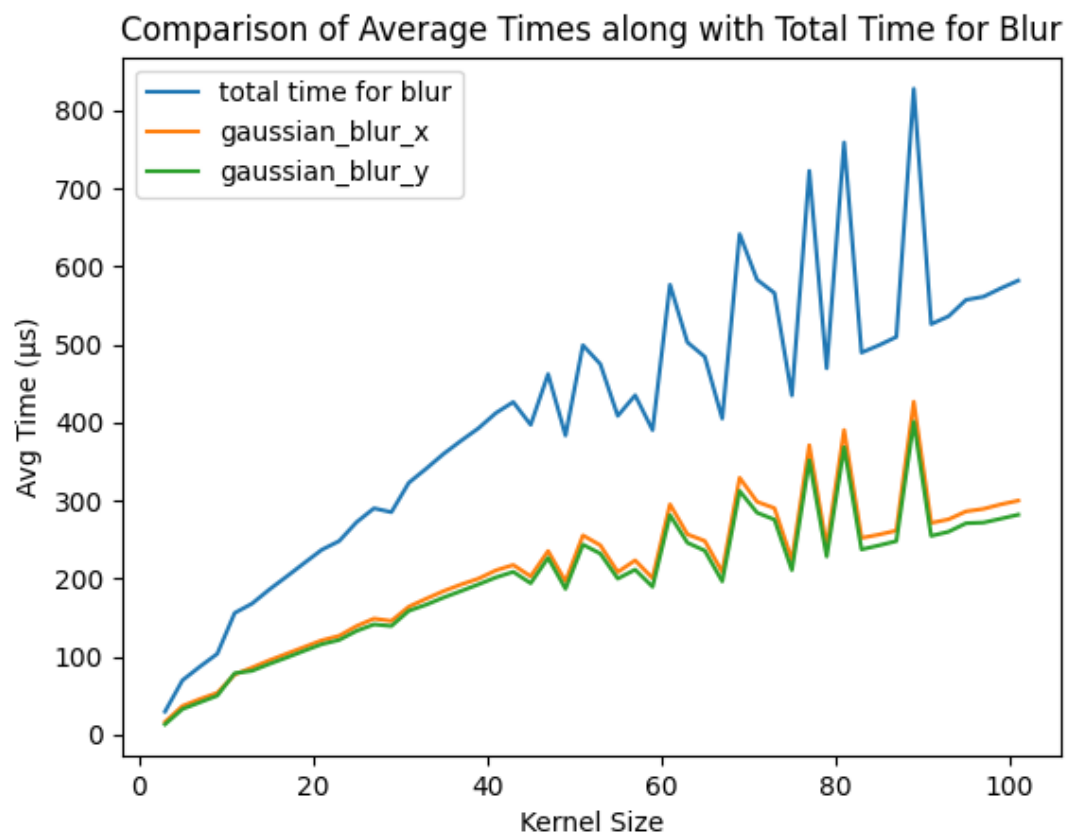


Figure 1.7 Gaussian Blur Separable kernel total

Comparing 2D Kernel vs Separable Kernel -

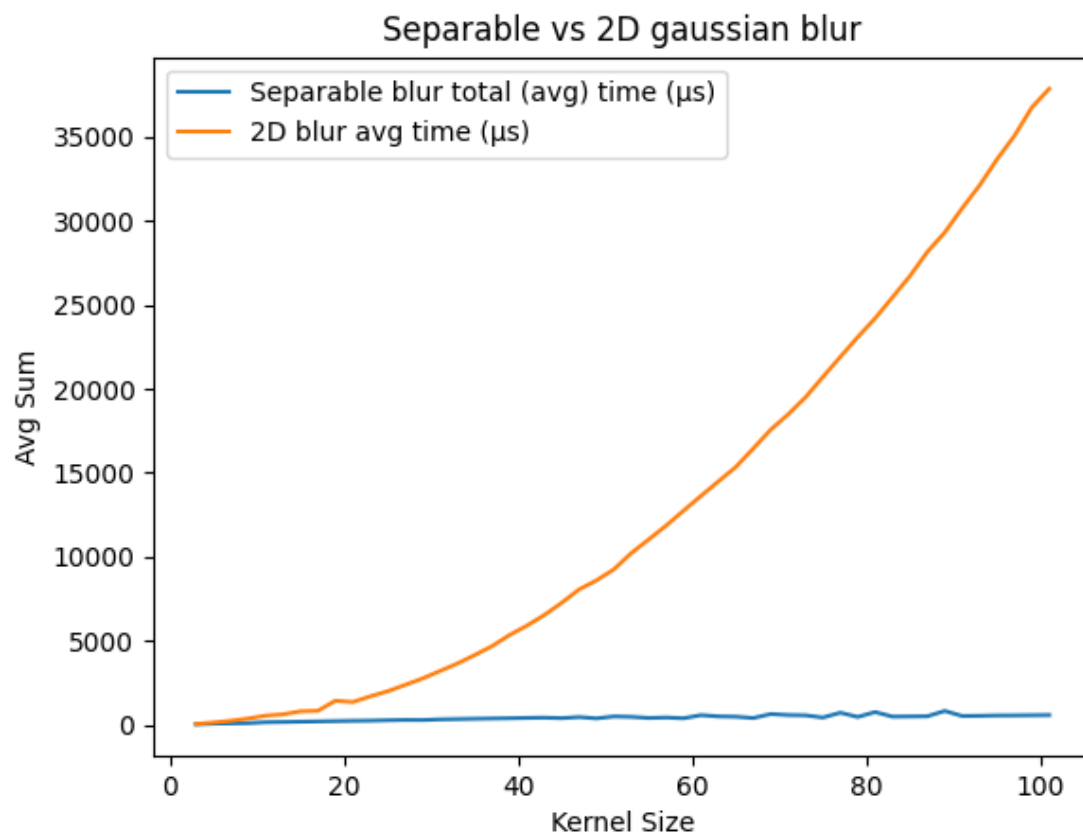


Figure 1.8 Gaussian Blur GPU Comparison

Average Speedup : 25.72 times

Comparing CPU vs GPU implementations -

Single Threaded vs Multi Threaded vs 2D Gaussian Blur vs Separable Gaussian

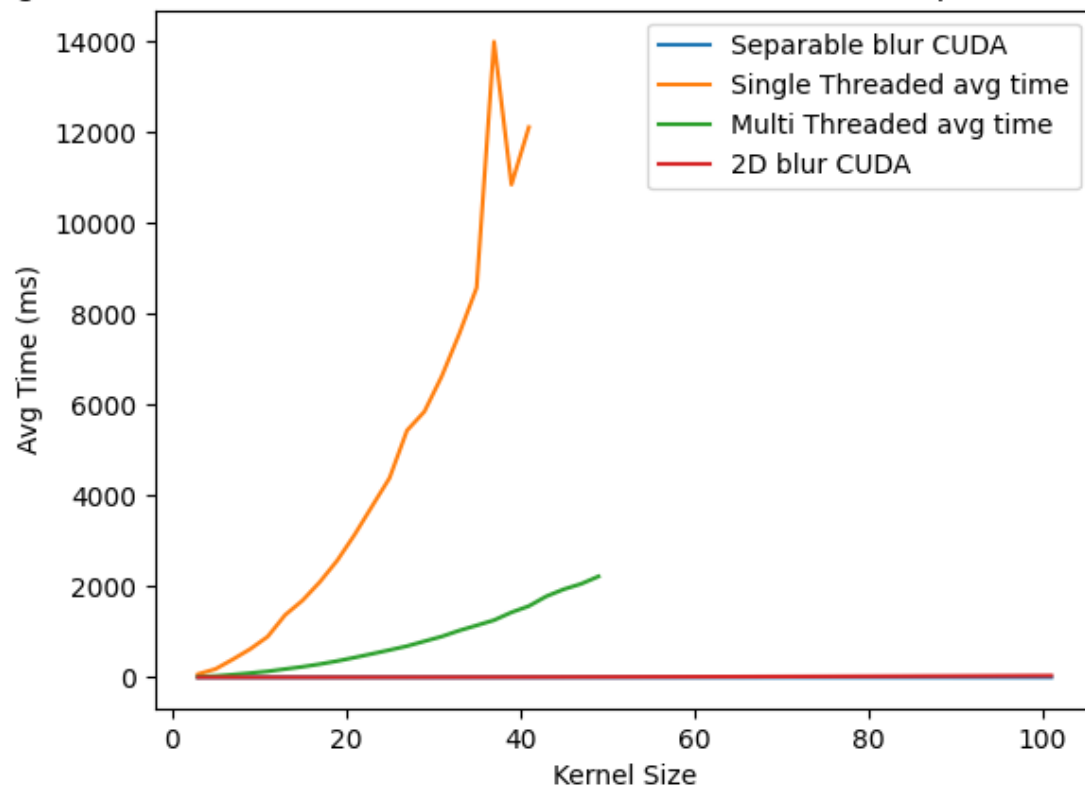


Figure 1.9 Gaussian Blur CPU vs GPU comparison

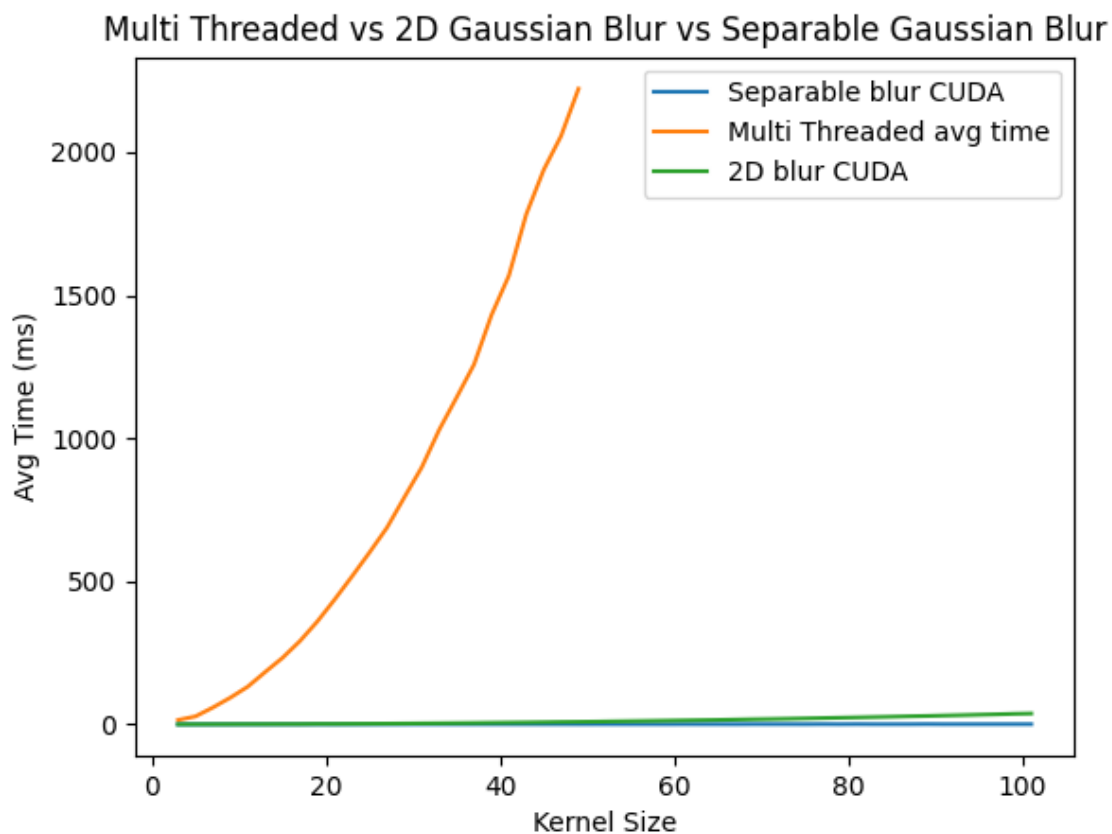


Figure 1.10 Gaussian Blur CPU vs GPU

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:

blur.cu	Convolution blurring in Nvidia CUDA	15
main.cpp	Gaussian blurring using CPU	33

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 flotat3
 - `__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

<i>call</i>	the CUDA function call
<i>msg</i>	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

<i>in1</i>	value 1
<i>in2</i>	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

<i>in1</i>	input 1
<i>in2</i>	input 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

<i>d_kernel</i>	the kernel, stored on GPU device memory
<i>n</i>	the size of the kernel
<i>input</i>	the input image stored on the GPU
<i>output</i>	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

<i>d_kernel</i>	the kernel, stored on GPU device memory
<i>n</i>	the size of the kernel
<i>input</i>	the input image stored on the GPU
<i>output</i>	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

<i>input</i>	the input image stored on the CPU memory
<i>output</i>	the output image stored on the CPU memory
<i>n</i>	the size of the Gaussian kernel, defaults to 3
<i>sigma</i>	the standard deviation of the Gaussian kernel, defaults to 1.
<i>two↔ _d</i>	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 )
```

applies the gaussian blur convolution to the input image

Template Parameters

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

Parameters

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

<i>Ts</i>	
-----------	--

Parameters

<i>inputs</i>	varidaic list of resources
<i>remove_globals</i>	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

<i>input</i>	input image stored on the CPU
<i>output</i>	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 )
```

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

Template Parameters

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

Parameters

<i>kernel</i>	the kernel to apply the convolution
<i>kernel_size</i>	the dimension of the kernel
<i>input</i>	the input image
<i>output</i>	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 )
```

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

Template Parameters

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

Parameters

<i>kernel</i>	the kernel to apply the convolution
<i>kernel_size</i>	the dimension of the kernel
<i>input</i>	the input image
<i>output</i>	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

<i>kernel</i>	the array in which the weights are stored
<i>n</i>	the size of the kernel. a 1D kernel of length n is needed
<i>sigma</i>	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

<i>kernel</i>	the array in which the weights are stored
<i>n</i>	the size of the kernel, t.e. n x n kernel is needed
<i>sigma</i>	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

<i>val</i>	the value
<i>out</i>	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

<i>val</i>	the value
<i>out</i>	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

<i>val</i>	the value to set
<i>out</i>	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

<i>val</i>	The value
<i>out</i>	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	<i>val</i>	The value
	<i>out</i>	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	<i>in1</i>	Input 1
in	<i>in2</i>	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	<i>in1</i>	Input 1
in	<i>in2</i>	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;

```



```

00075     float p, q = 2.0 * sigma * sigma;
00076
00077     // Compute weights
00078     for (int i = 0; i < n; i++)
00079     {
00080         for (int j = 0; j < n; j++)
00081         {
00082             p = sqrt((i - mean) * (i - mean) + (j - mean) * (j - mean));
00083             kernel[i * n + j] = std::exp(-(p * p) / q) / (M_PI * q);
00084             sumOfWeights += kernel[i * n + j];
00085         }
00086     }
00087
00088     // Normalizing weights
00089     for (int i = 0; i < n; i++)
00090     {
00091         for (int j = 0; j < n; j++)
00092         {
00093             kernel[i * n + j] /= sumOfWeights;
00094         }
00095     }
00096 }
00097
00106 __host__ void generate_gaussian_kernel_1d(float *kernel, const int n,
00107                                           const float sigma = 1)
00108 {
00109     // Calculate the values of the kernel
00110     float sum = 0.0f;
00111     for (int i = 0; i < n; i++)
00112     {
00113         float x = i - (n - 1) / 2.0f;
00114         kernel[i] = std::exp(-x * x / (2 * sigma * sigma));
00115         sum += kernel[i];
00116     }
00117
00118     // Normalize the kernel so that its sum equals 1
00119     for (int i = 0; i < n; i++)
00120     {
00121         kernel[i] /= sum;
00122     }
00123 }
00124
00131 __device__ __forceinline__ void set_value(const int &val, uchar &out)
00132 {
00133     out = val;
00134 }
00135
00142 __device__ __forceinline__ void set_value(const float &val, float &out)
00143 {
00144     out = val;
00145 };
00146
00154 __device__ __forceinline__ void set_value(const float &val, float3 &out)
00155 {
00156     out.x = val, out.y = val, out.z = val;
00157 }
00158
00165 __device__ __forceinline__ void set_value(const float3 &val, uchar3 &out)
00166 {
00167     out.x = val.x;
00168     out.y = val.y;
00169     out.z = val.z;
00170 }
00177 __device__ __forceinline__ void set_value(const int &val, uchar3 &out)
00178 {
00179     out.x = val;
00180     out.y = val;
00181     out.z = val;
00182 }
00183
00192 __device__ __forceinline__ uchar3 subtract_value(uchar3 in1, uchar3 in2)
00193 {
00194     uchar3 out;
00195     out.x = in1.x - in2.x;
00196     out.y = in1.y - in2.y;
00197     out.z = in1.z - in2.z;
00198     return out;
00199 }
00200
00208 __device__ __forceinline__ float3 add_value(float3 in1, float3 in2)
00209 {
00210     return {in1.x + in2.x, in1.y + in2.y, in1.z + in2.z};
00211 }
00212
00220 __device__ __forceinline__ float add_value(float in1, float in2)
00221 {
00222     return in1 + in2;

```

```

00223 }
00224
00233 __device__ __forceinline__ uchar subtract_value(uchar in1, uchar in2)
00234 {
00235     return in1 - in2;
00236 }
00245 __device__ __forceinline__ float3 multiply_value(const float &x,
00246                                                    const uchar3 &y)
00247 {
00248     return {x * (float)y.x, x * (float)y.y, x * (float)y.z};
00249 }
00250
00259 __device__ __forceinline__ float3 multiply_value(const float &x,
00260                                                    const float3 &y)
00261 {
00262     return {x * (float)y.x, x * (float)y.y, x * (float)y.z};
00263 }
00264
00272 __device__ __forceinline__ float multiply_value(const float &x, const uchar &y)
00273 {
00274     return x * (float)y;
00275 }
00276
00289 template <typename T_in, typename T_out, typename F_cal>
00290 __global__ void gaussian_blur(const float *kernel, int n,
00291                               const cv::cuda::PtrStepSz<T_in> input,
00292                               cv::cuda::PtrStepSz<T_out> output)
00293 {
00294     // calculate the x & y position of the current image pixel
00295     const int x = blockIdx.x * blockDim.x + threadIdx.x;
00296     const int y = blockIdx.y * blockDim.y + threadIdx.y;
00297
00298     if (x >= input.cols || y >= input.rows) return;
00299
00300     const int mid = n / 2;
00301     F_cal sum;
00302     set_value(0, sum);
00303     // synchronize all the threads till this point
00304     __syncthreads();
00305
00306     // loop over the n x n neighborhood of the current pixel
00307     for (int i = 0; i < n; i++)
00308     {
00309         for (int j = 0; j < n; j++)
00310         {
00311             int y_idx = y + i - mid;
00312             int x_idx = x + j - mid;
00313             if (y_idx > input.rows || x_idx > input.cols) continue;
00314             const float kernel_val = kernel[(n - i - 1) * n + (n - j - 1)];
00315             sum =
00316                 add_value(sum, multiply_value(kernel_val, input(y_idx, x_idx)));
00317         }
00318     }
00319     T_out result;
00320     set_value(sum, result);
00321     output(y, x) = result;
00322 }
00335 template <typename T_in, typename T_out, typename F_cal>
00336 __global__ void gaussian_blur_x(float *kernel, int kernel_size,
00337                                 const cv::cuda::PtrStepSz<T_in> input,
00338                                 cv::cuda::PtrStepSz<T_out> output)
00339 {
00340     int x = blockIdx.x * blockDim.x + threadIdx.x;
00341     int y = blockIdx.y * blockDim.y + threadIdx.y;
00342     const int radius = kernel_size / 2;
00343     const int width = input.cols;
00344     const int height = input.rows;
00345
00346     if (x >= input.cols || y >= input.rows) return;
00347
00348     F_cal pixel;
00349     set_value(0, pixel);
00350
00351     for (int i = -radius; i <= radius; i++)
00352     {
00353         int idx = y * width + (x + i);
00354         if (idx >= 0 && idx < width * height)
00355         {
00356             const float weight = kernel[i + radius];
00357             pixel = add_value(pixel, multiply_value(weight, input[idx]));
00358         }
00359     }
00360     set_value(pixel, output(y, x));
00361 }
00362
00375 template <typename T_in, typename T_out, typename F_cal>
00376 __global__ void gaussian_blur_y(float *kernel, int kernel_size,

```

```

00377                                     const cv::cuda::PtrStepSz<T_in> input,
00378                                     cv::cuda::PtrStepSz<T_out> output)
00379 {
00380     int x = blockIdx.x * blockDim.x + threadIdx.x;
00381     int y = blockIdx.y * blockDim.y + threadIdx.y;
00382     const int radius = kernel_size / 2;
00383     const int width = input.cols;
00384     const int height = input.rows;
00385
00386     if (x >= input.cols || y >= input.rows) return;
00387
00388     F_cal pixel;
00389     set_value(0, pixel);
00390     float weight_sum = 0;
00391     for (int i = -radius; i <= radius; i++)
00392     {
00393         int idx = (y + i) * width + x;
00394         if (idx >= 0 && idx < width * height)
00395         {
00396             float weight = kernel[i + radius];
00397             pixel = add_value(pixel, multiply_value(weight, input[idx]));
00398         }
00399     }
00400     set_value(pixel, output(y, x)); // output(y,x) = pixel;
00401 }
00402
00412 template <typename... Ts>
00413 void gaussian_blur_exit(bool remove_globals, Ts &&...inputs)
00414 {
00415     if (remove_globals)
00416     {
00417         ginput.release();
00418         goutput.release();
00419     }
00420     ([&] { SAFE_CALL(cudaFree(inputs), "Unable to free"); }(), ...);
00421 }
00422
00432 void call_gaussian_blur_2d(float *d_kernel, const int &n,
00433                           const cv::cuda::GpuMat &input,
00434                           cv::cuda::GpuMat &output)
00435 {
00436     CV_Assert(input.channels() == 1 || input.channels() == 3);
00437     const dim3 block(16, 16);
00438
00439     // Calculate grid size to cover the whole image
00440     const dim3 grid(cv::cuda::device::divUp(input.cols, block.x),
00441                    cv::cuda::device::divUp(input.rows, block.y));
00442     if (input.channels() == 1)
00443     {
00444         gaussian_blur<uchar, uchar, float>
00445             <<grid, block>>(d_kernel, n, input, output);
00446         return;
00447     }
00448     else if (input.channels() == 3)
00449     {
00450         gaussian_blur<uchar3, uchar3, float3>
00451             <<grid, block>>(d_kernel, n, input, output);
00452     }
00453     cudaSafeCall(cudaGetLastError());
00454 }
00465 void call_gaussian_blur_1d(float *d_kernel, const int &n,
00466                           const cv::cuda::GpuMat &input,
00467                           cv::cuda::GpuMat &output)
00468 {
00469     CV_Assert(input.channels() == 1 || input.channels() == 3);
00470     const int block_size = 16;
00471     dim3 dimBlock(block_size, block_size);
00472     dim3 dimGrid(cv::cuda::device::divUp(input.cols, dimBlock.x),
00473                 cv::cuda::device::divUp(input.rows, dimBlock.y));
00474     cv::cuda::GpuMat temp = input.clone();
00475     // Apply the horizontal Gaussian blur
00476     if (input.channels() == 1)
00477     {
00478         gaussian_blur_x<uchar, uchar, float>
00479             <<dimGrid, dimBlock>>(d_kernel, n, input, temp);
00480         gaussian_blur_y<uchar, uchar, float>
00481             <<dimGrid, dimBlock>>(d_kernel, n, temp, output);
00482     }
00483     else if (input.channels() == 3)
00484     {
00485         gaussian_blur_x<uchar3, uchar3, float3>
00486             <<dimGrid, dimBlock>>(d_kernel, n, input, temp);
00487         gaussian_blur_y<uchar3, uchar3, float3>
00488             <<dimGrid, dimBlock>>(d_kernel, n, temp, output);
00489     }
00490     cudaSafeCall(cudaGetLastError());
00491 }

```

```

00492 }
00505 __host__ void gaussian_blur(const cv::Mat &input, cv::Mat &output,
00506                             const int n = 3, const float sigma = 1.0,
00507                             bool two_d = true, bool remove_globals = true)
00508 {
00509     ginput.upload(input);
00510     std::vector<float> gauss_kernel_host;
00511     float *d_gauss_kernel;
00512     if (two_d)
00513     {
00514         gauss_kernel_host = std::vector<float>(n * n);
00515         generate_gaussian_kernel_2d(gauss_kernel_host.data(), n, sigma);
00516         cudaMalloc((void **)&d_gauss_kernel, n * n * sizeof(float));
00517         SAFE_CALL(cudaMemcpy(d_gauss_kernel, gauss_kernel_host.data(),
00518                             sizeof(float) * n * n, cudaMemcpyHostToDevice),
00519                 "Unable to copy kernel");
00520         call_gaussian_blur_2d(d_gauss_kernel, n, ginput, goutput);
00521     }
00522     else
00523     {
00524
00525         gauss_kernel_host = std::vector<float>(n);
00526         generate_gaussian_kernel_1d(gauss_kernel_host.data(), n, sigma);
00527         cudaMalloc((void **)&d_gauss_kernel, n * sizeof(float));
00528         SAFE_CALL(cudaMemcpy(d_gauss_kernel, gauss_kernel_host.data(),
00529                             sizeof(float) * n, cudaMemcpyHostToDevice),
00530                 "Unable to copy kernel");
00531         call_gaussian_blur_1d(d_gauss_kernel, n, ginput, goutput);
00532     }
00533     goutput.download(output);
00534     gaussian_blur_exit(remove_globals, d_gauss_kernel);
00535 }
00536
00543 void gaussian_blur_init(const cv::Mat &input, cv::Mat &output)
00544 {
00545     ginput.create(input.rows, input.cols, input.type());
00546     goutput.create(output.rows, output.cols, output.type());
00547 }
00548 void stress_test(const int &n, const bool &two_d)
00549 {
00550     std::cout << "Kernel size: " << n << std::endl;
00551     const std::string path = "../images/peppers_color.tif";
00552     cv::Mat input = cv::imread(path, 1);
00553     auto output = input.clone();
00554     gaussian_blur_init(input, output);
00555     for (int i = 0; i < 100; i++)
00556     {
00557         printProgress((float)i / 100);
00558         gaussian_blur(input, output, n, 1.7, two_d, false);
00559     }
00560     std::cout << std::endl;
00561     ginput.release();
00562     goutput.release();
00563     return;
00564 }
00565 int main(int argc, char **argv)
00566 {
00567
00568     if (argc < 3)
00569     {
00570         printf("usage: Blur_Test <kernel_size> <Image_Path> [<Output_Path>]\n");
00571         return -1;
00572     }
00573     std::string mTitle = "Display Image";
00574     cv::Mat input;
00575     int n = atoi(argv[1]);
00576     if (strcmp(argv[2], "stress2d", 8) == 0)
00577     {
00578         stress_test(n, true);
00579         return 0;
00580     }
00581     else if (strcmp(argv[2], "stress1d", 8) == 0)
00582     {
00583         stress_test(n, false);
00584         return 0;
00585     }
00586     input = cv::imread(argv[2], 1);
00587     if (!input.data)
00588     {
00589         printf("No image data \n");
00590         return -1;
00591     }
00592     auto output = input.clone();
00593
00594     // Call the wrapper function
00595     gaussian_blur_init(input, output);
00596     gaussian_blur(input, output, n, 1.7, 0);

```

```

00597
00598     // Show the input and output
00599     cv::imshow("Output", output);
00600
00601     // Wait for key press
00602     cv::waitKey();
00603     namedWindow(mTitle, cv::WINDOW_AUTOSIZE);
00604     imshow(mTitle, input);
00605     if (argc >= 4) imwrite(argv[3], output);
00606     do
00607     {
00608
00609         auto k = cv::waitKey(500);
00610         if (k == 27)
00611         {
00612             cv::destroyAllWindows();
00613             return 0;
00614         }
00615         if (cv::getWindowProperty(mTitle, cv::WND_PROP_VISIBLE) == 0) return 0;
00616     } while (true);
00617     return 0;
00618
00619 }
00620 }

```

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

<i>kernel</i>	the convolution kernel
<i>original_img</i>	the original image
<i>new_img</i>	the output image
<i>r</i>	the row number of the current pixel
<i>c</i>	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

<i>kernel</i>	the convolution kernel
<i>original_img</i>	the original image
<i>new_img</i>	the output image
<i>r</i>	the row number of the current pixel
<i>c</i>	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

<i>kernel</i>	the convolution kernel
<i>original_img</i>	the original image
<i>new_img</i>	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

<i>kernel</i>	the convolution kernel
<i>original_img</i>	the original_img
<i>new_img</i>	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

<i>kernel</i>	the kernel to be populated
<i>n</i>	the size of the kernel, the kernel must be of size $n * n$
<i>sigma</i>	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;
```

```

00063     }
00064 }
00065 }
00076 void apply_convolution(const std::vector<std::vector<float>> &kernel,
00077                       const Mat &original_img, Mat &new_img, const int &r,
00078                       const int &c)
00079 {
00080     const size_t n = kernel.size();
00081     assert(n % 2 == 1);
00082     assert(n == kernel[0].size());
00083     const size_t mid = n / 2;
00084     new_img.at<Vec3b>(r, c) = {0, 0, 0};
00085     for (int i = 0; i < n; i++)
00086     {
00087         for (int j = 0; j < n; j++)
00088         {
00089             if (r - mid + i >= 0 && r - mid + i < original_img.rows &&
00090                 c - mid + j >= 0 && c - mid + j < original_img.cols)
00091                 new_img.at<Vec3b>(r, c) +=
00092                     kernel[n - i - 1][n - j - 1] *
00093                     original_img.at<Vec3b>(r - mid + i, c - mid + j);
00094         }
00095     }
00096 }
00106 void apply_convolution_multi_threaded(
00107     const std::vector<std::vector<float>> &kernel, const Mat &original_img,
00108     Mat &new_img, const int &r, const int &c)
00109 {
00110     const size_t n = kernel.size();
00111     assert(n % 2 == 1);
00112     assert(n == kernel[0].size());
00113     const size_t mid = n / 2;
00114     new_img.at<Vec3b>(r, c) = {0, 0, 0};
00115     #pragma omp parallel for shared(r, c, original_img, new_img, kernel)
00116     for (int i = 0; i < n; i++)
00117     {
00118         #pragma omp parallel for shared(r, c, original_img, new_img, kernel)
00119         for (int j = 0; j < n; j++)
00120         {
00121             if (r - mid + i >= 0 && r - mid + i < original_img.rows &&
00122                 c - mid + j >= 0 && c - mid + j < original_img.cols)
00123                 new_img.at<Vec3b>(r, c) +=
00124                     kernel[n - i - 1][n - j - 1] *
00125                     original_img.at<Vec3b>(r - mid + i, c - mid + j);
00126         }
00127     }
00128 }
00136 void apply_kernel(const std::vector<std::vector<float>> &kernel,
00137                  const Mat &original_img, Mat &new_img)
00138 {
00139     for (int i = 0; i < original_img.rows; i++)
00140     {
00141         for (int j = 0; j < original_img.cols; j++)
00142         {
00143             apply_convolution(kernel, original_img, new_img, i, j);
00144         }
00145     }
00146 }
00155 void apply_kernel_multithreaded(const std::vector<std::vector<float>> &kernel,
00156                                const Mat &original_img, Mat &new_img)
00157 {
00158     #pragma omp barrier
00159     #pragma omp parallel for shared(original_img, new_img, kernel)
00160     for (int i = 0; i < original_img.rows; i++)
00161     {
00162         #pragma omp parallel for shared(original_img, new_img, kernel)
00163         for (int j = 0; j < original_img.cols; j++)
00164         {
00165             apply_convolution(kernel, original_img, new_img, i, j);
00166         }
00167     }
00168     #pragma omp barrier
00169 }
00170 void stress_test(const int &n, const bool &multi = true)
00171 {
00172     std::cout << "Stress testing" << std::endl;
00173     const std::string path = "../images/peppers_color.tif";
00174     auto image = imread(path, 1);
00175     auto new_img = image.clone();
00176     std::vector<std::vector<float>> gauss_kernel(n, std::vector<float>(n));
00177     generate_gaussian_kernel(gauss_kernel, n, 1.6);
00178     std::vector<double> run_times;
00179     const int num_runs = 20;
00180     std::string fname;
00181     if (!multi)
00182     {
00183         fname = "profile_single_threaded.csv";

```

```

00184
00185     for (int i = 0; i < num_runs; i++)
00186     {
00187         printProgress((float)i / num_runs);
00188         auto start = std::chrono::high_resolution_clock::now();
00189         apply_kernel(gauss_kernel, image, new_img);
00190         auto end = std::chrono::high_resolution_clock::now();
00191         std::chrono::duration<double, std::milli> duration_ms = end - start;
00192         run_times.push_back(duration_ms.count());
00193     }
00194 }
00195 else
00196 {
00197     fname = "profile_multi_threaded.csv";
00198     for (int i = 0; i < num_runs; i++)
00199     {
00200         printProgress((float)i / num_runs);
00201         auto start = std::chrono::high_resolution_clock::now();
00202         apply_kernel_multithreaded(gauss_kernel, image, new_img);
00203         auto end = std::chrono::high_resolution_clock::now();
00204         std::chrono::duration<double, std::milli> duration_ms = end - start;
00205         run_times.push_back(duration_ms.count());
00206     }
00207 }
00208 sort(run_times.begin(), run_times.end());
00209 double avg = std::accumulate(run_times.begin(), run_times.end(), 0.0) /
00210     run_times.size();
00211 double median =
00212     ((run_times.size() % 2 == 0) ? (run_times[run_times.size() / 2 - 1] +
00213         run_times[run_times.size() / 2]) /
00214         2
00215         : run_times[run_times.size() / 2]);
00216
00217 std::cout << "\nMax(ms)\t\tAvg(ms)\t\tMedian(ms)\t\tMin(ms)\n";
00218 std::cout << run_times.back() << "\t\t" << avg << "\t\t" << median
00219
00220     << "\t\t" << run_times.front() << "\n";
00221 std::fstream file(fname, std::ios::in | std::ios_base::app);
00222 if (file.tellg() == 0)
00223 {
00224     // write the headers if the file is empty
00225     file << "KERNEL_SIZE,MAX_RUN_TIME,MIN_RUN_TIME,AVG_RUN_TIME,MEDIAN_RUN_"
00226         << "TIME"
00227         << std::endl;
00228 }
00229 file << n << ", " << run_times.back() << ", " << run_times.front() << ", "
00230     << avg << ", " << median << std::endl;
00231
00232 file.close();
00233 }
00234 int main(int argc, char **argv)
00235 {
00236     if (argc < 3)
00237     {
00238         printf("usage: Blur_Test <kernel_size> <Image_Path> [<Output_Path>]\n");
00239         return -1;
00240     }
00241     int n = atoi(argv[1]);
00242     if (strcmp(argv[2], "stressm", 7) == 0)
00243     {
00244         stress_test(n, true);
00245         return 0;
00246     }
00247     else if (strcmp(argv[2], "stress", 6) == 0)
00248     {
00249         stress_test(n, false);
00250         return 0;
00251     }
00252
00253     std::vector<std::vector<float>> gauss_kernel(n, std::vector<float>(n));
00254     generate_gaussian_kernel(gauss_kernel, n, 1.6);
00255
00256     std::string mTitle = "Display Image";
00257     Mat image;
00258     image = imread(argv[2], 1);
00259     if (!image.data)
00260     {
00261         printf("No image data \n");
00262         return -1;
00263     }
00264     namedWindow(mTitle, WINDOW_AUTOSIZE);
00265     auto new_img = image.clone();
00266     apply_kernel_multithreaded(gauss_kernel, image, new_img);
00267     imshow(mTitle, image);
00268     imshow("gaussian", new_img);
00269     if (argc >= 4) imwrite(argv[3], new_img);
00270     do

```

```
00271     {
00272
00273         auto k = waitKey(500);
00274         if (k == 27)
00275         {
00276             cv::destroyAllWindows();
00277             return 0;
00278         }
00279         if (cv::getWindowProperty(mTitle, WND_PROP_VISIBLE) == 0)
00280         {
00281             return 0;
00282             break;
00283         }
00284     } while (true);
00285     return 0;
00286 }
00287 }
```

Index

- add_value
 - blur.cu, [18](#)
- apply_convolution
 - main.cpp, [34](#)
- apply_convolution_multi_threaded
 - main.cpp, [35](#)
- apply_kernel
 - main.cpp, [35](#)
- apply_kernel_multithreaded
 - main.cpp, [36](#)
- blur.cu, [15](#)
 - add_value, [18](#)
 - call_gaussian_blur_1d, [18](#)
 - call_gaussian_blur_2d, [19](#)
 - gaussian_blur, [19](#), [20](#)
 - gaussian_blur_exit, [20](#)
 - gaussian_blur_init, [21](#)
 - gaussian_blur_x, [21](#)
 - gaussian_blur_y, [22](#)
 - generate_gaussian_kernel_1d, [22](#)
 - generate_gaussian_kernel_2d, [23](#)
 - ginput, [28](#)
 - goutput, [28](#)
 - main, [23](#)
 - multiply_value, [23](#), [24](#)
 - PBSTR, [17](#)
 - PBWIDTH, [17](#)
 - printProgress, [25](#)
 - SAFE_CALL, [17](#)
 - set_value, [25](#), [26](#)
 - stress_test, [26](#)
 - subtract_value, [27](#)
- call_gaussian_blur_1d
 - blur.cu, [18](#)
- call_gaussian_blur_2d
 - blur.cu, [19](#)
- gaussian_blur
 - blur.cu, [19](#), [20](#)
- gaussian_blur_exit
 - blur.cu, [20](#)
- gaussian_blur_init
 - blur.cu, [21](#)
- gaussian_blur_x
 - blur.cu, [21](#)
- gaussian_blur_y
 - blur.cu, [22](#)
- generate_gaussian_kernel

- main.cpp, [36](#)
- generate_gaussian_kernel_1d
 - blur.cu, [22](#)
- generate_gaussian_kernel_2d
 - blur.cu, [23](#)
- ginput
 - blur.cu, [28](#)
- goutput
 - blur.cu, [28](#)
- main
 - blur.cu, [23](#)
 - main.cpp, [36](#)
- main.cpp, [33](#)
 - apply_convolution, [34](#)
 - apply_convolution_multi_threaded, [35](#)
 - apply_kernel, [35](#)
 - apply_kernel_multithreaded, [36](#)
 - generate_gaussian_kernel, [36](#)
 - main, [36](#)
 - PBSTR, [34](#)
 - PBWIDTH, [34](#)
 - printProgress, [36](#)
 - stress_test, [37](#)
- multiply_value
 - blur.cu, [23](#), [24](#)
- PBSTR
 - blur.cu, [17](#)
 - main.cpp, [34](#)
- PBWIDTH
 - blur.cu, [17](#)
 - main.cpp, [34](#)
- printProgress
 - blur.cu, [25](#)
 - main.cpp, [36](#)
- SAFE_CALL
 - blur.cu, [17](#)
- set_value
 - blur.cu, [25](#), [26](#)
- stress_test
 - blur.cu, [26](#)
 - main.cpp, [37](#)
- subtract_value
 - blur.cu, [27](#)