Blur Test

1.0.0

Generated by Doxygen 1.9.6

Chapter 1

Blur-Test

Testing out some blurs with opecv, OpenMP and CUDA

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 466 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 433 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 506 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 414 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 549 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 377 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 571 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 554 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;
```

```
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 potin
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         /* printf("%d\n", idx); */
00355         if (idx >= 0 && idx < width * height)
00356         {
00357             const float weight = kernel[i + radius];
00358             pixel = add_value(pixel, multiply_value(weight, input[idx]));
00359         }
00360     }
00361     set_value(pixel, output(y, x));
00362 }
00363
00376 template <typename T_in, typename T_out, typename F_cal>
```

```
00377 __global__ void gaussian_blur_y(float *kernel, int kernel_size,
00378                                  const cv::cuda::PtrStepSz<T_in> input,
00379                                  cv::cuda::PtrStepSz<T_out> output)
00380 {
00381     int x = blockIdx.x * blockDim.x + threadIdx.x;
00382     int y = blockIdx.y * blockDim.y + threadIdx.y;
00383     const int radius = kernel_size / 2;
00384     const int width = input.cols;
00385     const int height = input.rows;
00386 
00387     if (x >= input.cols || y >= input.rows) return;
00388 
00389     F_cal pixel;
00390     set_value(0, pixel);
00391     float weight_sum = 0;
00392     for (int i = -radius; i <= radius; i++)
00393     {
00394         int idx = (y + i) * width + x;
00395         if (idx >= 0 && idx < width * height)
00396         {
00397             float weight = kernel[i + radius];
00398             pixel = add_value(pixel, multiply_value(weight, input[idx]));
00399         }
00400     }
00401     set_value(pixel, output(y, x)); // output(y,x) = pixel;
00402 }
00403 
00413 template <typename... Ts>
00414 void gaussian_blur_exit(bool remove_globals, Ts &&...inputs)
00415 {
00416     if (remove_globals)
00417     {
00418         ginput.release();
00419         goutput.release();
00420     }
00421     ([&] { SAFE_CALL(cudaFree(inputs), "Unable to free"); }(), ...);
00422 }
00423 
00433 void call_gaussian_blur_2d(float *d_kernel, const int &n,
00434                            const cv::cuda::GpuMat &input,
00435                            cv::cuda::GpuMat &output)
00436 {
00437     CV_Assert(input.channels() == 1 || input.channels() == 3);
00438     const dim3 block(16, 16);
00439 
00440     // Calculate grid size to cover the whole image
00441     const dim3 grid(cv::cuda::device::divUp(input.cols, block.x),
00442                     cv::cuda::device::divUp(input.rows, block.y));
00443     if (input.channels() == 1)
00444     {
00445         gaussian_blur<uchar, uchar, float>
00446             <<<grid, block>>>(d_kernel, n, input, output);
00447         return;
00448     }
00449     else if (input.channels() == 3)
00450     {
00451         gaussian_blur<uchar3, uchar3, float3>
00452             <<<grid, block>>>(d_kernel, n, input, output);
00453     }
00454     cudaSafeCall(cudaGetLastError());
00455 }
00466 void call_gaussian_blur_1d(float *d_kernel, const int &n,
00467                            const cv::cuda::GpuMat &input,
00468                            cv::cuda::GpuMat &output)
00469 {
00470     CV_Assert(input.channels() == 1 || input.channels() == 3);
00471     const int block_size = 16;
00472     dim3 dimBlock(block_size, block_size);
00473     dim3 dimGrid(cv::cuda::device::divUp(input.cols, dimBlock.x),
00474                  cv::cuda::device::divUp(input.rows, dimBlock.y));
00475     cv::cuda::GpuMat temp = input.clone();
00476     // Apply the horizontal Gaussian blur
00477     if (input.channels() == 1)
00478     {
00479 
00480         gaussian_blur_x<uchar, uchar, float>
00481             <<<dimGrid, dimBlock>>>(d_kernel, n, input, temp);
00482         gaussian_blur_y<uchar, uchar, float>
00483             <<<dimGrid, dimBlock>>>(d_kernel, n, temp, output);
00484     }
00485     else if (input.channels() == 3)
00486     {
00487         gaussian_blur_x<uchar3, uchar3, float3>
00488             <<<dimGrid, dimBlock>>>(d_kernel, n, input, temp);
00489         gaussian_blur_y<uchar3, uchar3, float3>
00490             <<<dimGrid, dimBlock>>>(d_kernel, n, temp, output);
00491     }
```

```
00492     cudaSafeCall(cudaGetLastError());
00493 }
00506 __host__ void gaussian_blur(const cv::Mat &input, cv::Mat &output,
00507                             const int n = 3, const float sigma = 1.0,
00508                             bool two_d = true, bool remove_globals = true)
00509 {
00510     ginput.upload(input);
00511     std::vector<float> gauss_kernel_host;
00512     float *d_gauss_kernel;
00513     if (two_d)
00514     {
00515         gauss_kernel_host = std::vector<float>(n * n);
00516         generate_gaussian_kernel_2d(gauss_kernel_host.data(), n, sigma);
00517         cudaMalloc((void **)&d_gauss_kernel, n * n * sizeof(float));
00518         SAFE_CALL(cudaMemcpy(d_gauss_kernel, gauss_kernel_host.data(),
00519                              sizeof(float) * n * n, cudaMemcpyHostToDevice),
00520                   "Unable to copy kernel");
00521         /* cudaProfilerStart(); */
00522         call_gaussian_blur_2d(d_gauss_kernel, n, ginput, goutput);
00523         /* cudaProfilerStop(); */
00524     }
00525     else
00526     {
00527
00528         gauss_kernel_host = std::vector<float>(n);
00529         generate_gaussian_kernel_1d(gauss_kernel_host.data(), n, sigma);
00530         cudaMalloc((void **)&d_gauss_kernel, n * sizeof(float));
00531         SAFE_CALL(cudaMemcpy(d_gauss_kernel, gauss_kernel_host.data(),
00532                              sizeof(float) * n, cudaMemcpyHostToDevice),
00533                   "Unable to copy kernel");
00534         /* cudaProfilerStart(); */
00535         call_gaussian_blur_1d(d_gauss_kernel, n, ginput, goutput);
00536         /* cudaProfilerStop(); */
00537     }
00538     // goutput.upload(input);
00539     goutput.download(output);
00540     gaussian_blur_exit(remove_globals, d_gauss_kernel);
00541 }
00542
00549 void gaussian_blur_init(const cv::Mat &input, cv::Mat &output)
00550 {
00551     ginput.create(input.rows, input.cols, input.type());
00552     goutput.create(output.rows, output.cols, output.type());
00553 }
00554 void stress_test(const int &n, const bool &two_d)
00555 {
00556     std::cout « "Kernel size: " « n « std::endl;
00557     const std::string path = "../images/peppers_color.tif";
00558     cv::Mat input = cv::imread(path, 1);
00559     auto output = input.clone();
00560     gaussian_blur_init(input, output);
00561     for (int i = 0; i < 100; i++)
00562     {
00563         printProgress((float)i / 100);
00564         gaussian_blur(input, output, n, 1.7, two_d, false);
00565     }
00566     std::cout « std::endl;
00567     ginput.release();
00568     goutput.release();
00569     return;
00570 }
00571 int main(int argc, char **argv)
00572 {
00573
00574     if (argc < 3)
00575     {
00576         printf("usage: Blur_Test <kernel_size> <Image_Path> [<Output_Path>]\n");
00577         return -1;
00578     }
00579     std::string mTitle = "Display Image";
00580     cv::Mat input;
00581     int n = atoi(argv[1]);
00582     if (strncmp(argv[2], "stress2d", 8) == 0)
00583     {
00584         stress_test(n, true);
00585         return 0;
00586     }
00587     else if (strncmp(argv[2], "stress1d", 8) == 0)
00588     {
00589         stress_test(n, false);
00590         return 0;
00591     }
00592     input = cv::imread(argv[2], 1);
00593     if (!input.data)
00594     {
00595         printf("No image data \n");
00596         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);
    /* namedWindow("gauss", cv::WINDOW_AUTOSIZE); */
    imshow(mTitle, input);
    /* imshow("gaussian", output); */
    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 241 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     /* for (auto i = 0; i < num_runs; i++)
00209         std::cout « run_times[i] « "\t"; */
00210     sort(run_times.begin(), run_times.end());
00211     double avg = std::accumulate(run_times.begin(), run_times.end(), 0.0) /
00212                  run_times.size();
00213     double median =
00214         ((run_times.size() % 2 == 0) ? (run_times[run_times.size() / 2 - 1] +
00215                                         run_times[run_times.size() / 2]) /
00216                                            2
00217                                      : run_times[run_times.size() / 2]);
00218
00219     std::cout « "\nMax(ms)\t\tAvg(ms)\t\tMedian(ms)\t\tMin(ms)\n";
00220     std::cout « run_times.back() « "\t\t" « avg « "\t\t" « median
00221
00222               « "\t\t" « run_times.front() « "\n";
00223     std::fstream file(fname, std::ios::in | std::ios_base::app);
00224     if (file.tellg() == 0)
00225     {
00226         // write the headers if the file is empty
00227         file « "KERNEL_SIZE,MAX_RUN_TIME,MIN_RUN_TIME,AVG_RUN_TIME,MEDIAN_RUN_"
00228                "TIME"
00229              « std::endl;
00230     }
00231     file « n « "," « run_times.back() « "," « run_times.front() « ","
00232          « avg « "," « median « std::endl;
00233
00234     file.close();
00235
00236     /* setenv("MAX_RUN_TIME", std::to_string(run_times.back()).c_str(), 1);
00237     setenv("MIN_RUN_TIME", std::to_string(run_times.front()).c_str(), 1);
00238     setenv("AVG_RUN_TIME", std::to_string(avg).c_str(), 1);
00239     setenv("MEDIAN_RUN_TIME", std::to_string(median).c_str(), 1); */
00240 }
00241 int main(int argc, char **argv)
00242 {
00243     if (argc < 3)
00244     {
00245         printf("usage: Blur_Test <kernel_size> <Image_Path> [<Output_Path>]\n");
00246         return -1;
00247     }
00248     int n = atoi(argv[1]);
00249     if (strncmp(argv[2], "stressm", 7) == 0)
00250     {
00251         stress_test(n, true);
00252         return 0;
00253     }
00254     else if (strncmp(argv[2], "stress", 6) == 0)
00255     {
00256         stress_test(n, false);
00257         return 0;
00258     }
00259
00260     std::vector<std::vector<float» gauss_kernel(n, std::vector<float>(n));
00261     generate_gaussian_kernel(gauss_kernel, n, 1.6);
00262
00263     std::string mTitle = "Display Image";
00264     Mat image;
00265     image = imread(argv[2], 1);
00266     if (!image.data)
00267     {
00268         printf("No image data \n");
00269         return -1;
00270     }
```

```
00271     namedWindow(mTitle, WINDOW_AUTOSIZE);
00272     auto new_img = image.clone();
00273     // namedWindow("gauss", WINDOW_AUTOSIZE);
00274     apply_kernel_multithreaded(gauss_kernel, image, new_img);
00275     imshow(mTitle, image);
00276     imshow("gaussian", new_img);
00277     if (argc >= 4) imwrite(argv[3], new_img);
00278     do
00279     {
00280
00281         auto k = waitKey(500);
00282         if (k == 27)
00283         {
00284             cv::destroyAllWindows();
00285             return 0;
00286         }
00287         if (cv::getWindowProperty(mTitle, WND_PROP_VISIBLE) == 0)
00288         {
00289             return 0;
00290             break;
00291         }
00292
00293     } while (true);
00294     return 0;
00295 }
```

Index