RV COLLEGE OF ENGINEERING®
BENGALURU – 560059
(Autonomous Institution Affiliated to VTU, Belagavi)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

"Gaussian Blurring of an image"

PARALLEL ARCHITECTURE AND DISTRIBUTED PROGRAMMING LAB (18CS73)

OPEN ENDED EXPERIMENT REPORT

VII SEMESTER

2021-2022

Submitted by

Aravind A	**1RV18CS028**
Bhanu Prakash	**1RV18CS039**

Under the Guidance of

Prof. Anitha Sandeep
Department of CSE, R.V.C.E.,
Bengaluru - 560059

RV COLLEGE OF ENGINEERING®, BENGALURU - 560059
(Autonomous Institution Affiliated to VTU, Belagavi)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Certified that the **Open-Ended Experiment** titled "Gaussian Blurring of an image" has been carried out by **Aravind A(1RV18CS028) and Bhanu Prakash(1RV18CS039),** bonafide students of RV College of Engineering, Bengaluru, have submitted in partial fulfillment for the **Internal Assessment of Course: PARALLEL ARCHITECTURE AND DISTRIBUTED PROGRAMMING LAB (18CS73)** during the year 2021-2022. It is certified that all corrections/suggestions indicated for the Internal Assessment have been incorporated in the report.

Prof. Anitha Sandeep	**Dr. Ramakanth Kumar P**
Faculty Incharge,	Head of Department,
Department of CSE,	Department of CSE,
R.V.C.E., Bengaluru –59	R.V.C.E., Bengaluru–59

RV COLLEGE OF ENGINEERING® , BENGALURU - 560059
(Autonomous Institution Affiliated to VTU)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DECLARATION

We, **Aravind A(1RV18CS028) and Bhanu Prakash(1RV18CS039),** the students of Seventh Semester B.E., Computer Science and Engineering, R.V. College of Engineering, Bengaluru hereby declare that the mini-project titled **“**Gaussian Blurring of an image**”** has been carried out by us and submitted in partial fulfillment for the **Internal Assessment of Course: PARALLEL ARCHITECTURE AND DISTRIBUTED PROGRAMMING LAB (18CS73) - Open-Ended Experiment** during the year 2021-2022. We do declare that matter embodied in this report has not been submitted to any other university or institution for the award of any other degree or diploma.

Place: Bengaluru	**Aravind A**
Date:	**Bhanu Prakash**

Introduction

The goal of this project is to measure performance and scaling behaviour of four different Gaussian blurring implementations and compare them. The Gaussian blur implementations I will be measuring and comparing are single-pass 2D convolution algorithms:

1. Sequential
2. OpenACC
3. Naive CUDA
4. Shared Memory CUDA

I expected the shared memory CUDA implementation to perform the best out of all implementations as well as how it scales with input matrix sizes. I expected for the sequential implementation to perform the worst and for the OpenACC implementation to perform somewhere close to the naive CUDA implementation. For this project I analyzed the amount of time spent copying data to and from the device combined with the computation time of the kernel as a function of the input matrix dimension. In other words, I measured how fast each of these implementations ran over varying input matrix sizes.

2D Convolution

Convolution is the process of adding each element of the image to its local neighbors, weighted by the mask. This project focuses on Gaussian blurring specifically, but the way in which a Gaussian blur is applied to an image is through 2D convolution. For this project 3 different parallel implementations of a 2D convolution algorithm were written and one sequential implementation of the 2D convolution algorithm was written. A short recap of 2D convolution will be given in this section so that we can lead into the results and implications following this section more naturally.

Beginning with an image, we convert the image into a large matrix of decimal values. These decimal values represent the intensity of the pixels. Next, we create a “mask” or “kernel” which is usually a much smaller 3x3 - 9x9 matrix. Then we “convolve” the image matrix with the mask matrix; we overlay the mask matrix on the image matrix for every element in the image matrix, each time multiplying the mask elements by their overlaid partners in the image matrix and summing all products to form the new centre element. Finally, we normalize the values in the newly formed output matrix.

There are there are a few different algorithms for 2D convolution, though the main two have to do with whether or not the mask is “separable”. When a particular mask is separable then a two-pass algorithm is generally used, otherwise a single-pass algorithm is the easiest to implement. The single pass algorithm is what was described above and is what is used in this project for all parallel implementations. The two-pass algorithm is possible when the mask used is separable and is much faster than the single-pass algorithm. To implement it we decompose the mask matrix into two 1D factor matrices. Then we convolve the image with the both matrices (one horizontally and one vertically) where the order in which this happens does not matter and the output is equivalent to that in the single-pass. Lastly a mask is separable if all rows are multiples of each other.

Execution Environment

The execution environment used for both running and measuring these four implementations was Google Colaboratory. This system was used because it had the correct version of CUDA and contained a NVIDIA device capable of running OpenACC and CUDA programs.

Methodology

The process for creating these four implementations began with the naive CUDA implementations followed by the shared memory CUDA implementation. The sequential and OpenACC implementations came together last because the OpenACC version only needed a slightly modified sequential program to use of the pragma. For all implementations besides the sequential the outer loop was the sole focus of parallelization. The inner loops where the products of image and mask elements are summed were left sequential. Conceptually the naïve CUDA and OpenACC implementations are the same; they both attempt to parallelize the same thing with no extra additions to their algorithms. The shared memory CUDA implementation also parallelizes the outer loop, but in addition it makes use of shared memory and places the mask in constant memory.

Data collection on these implementations was accomplished by timing the total time taken to execute a call to the CUDA kernel. This includes both time spent copying data and time spent computing the convolution. For non-CUDA implementations I measured around what would be considered the equivalent of the kernel; for the OpenACC implementation that would be around the data initialization pragma, and for the sequential implementation that would be around the triple loop which forms the convolution computation. The actual measurements were taken by wrapping the aforementioned areas with start and end timers using the C++ Chrono library. Data from each measurement was written to a file unique to each implementation and later ran through a MATLAB script which formed the data into plots.

Wall clock time was used because it was the most straightforward method to measure all implementations. Data throughput was another option when it came to measuring techniques but it was not used for this project because I did not build the program to overlap the CUDA kernels, nor did I measure with consecutive data input in mind. In addition, the simplicity of wall clock measurement left ample time for me to become more familiar with much of the new content (e.g. CUDA, OpenACC, and the Gaussian blur algorithm and implementation.)

Result Analysis

The OpenACC implementation (left plot) and the shared memory CUDA (8x8 block) implementation (right plot) showed one of the largest disparities out of all of the programs. This is due to the shared memory CUDA implementation (8x8 block) performing 2nd best of all programs, and the OpenACC implementation performing the worst of all the programs.

While it isn’t surprising that the most heavily optimized CUDA implementation would perform well, it is surprising that a parallelized program would perform worse than the sequential despite the OpenACC compiler confirming there to be no errors or dependencies.

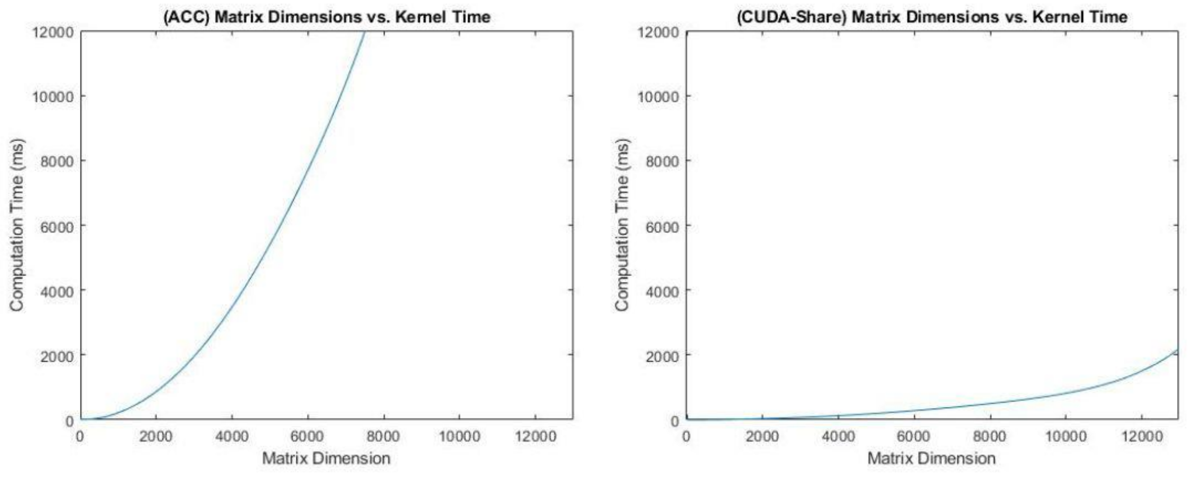

Figure 1 Comparison OpenACC vs SharedMemoryCUDA

We can see the direct comparison of the OpenACC (left plot) implementation to the sequential implementation (right plot) in the next figure. While only conjecture, it seems as if the OpenACC implementation leaves the upper limit of the plot at a larger angle than the sequential. So despite the sequential implementation performing better within the range of data collected, it's possible that the OpenACC implementation still scales better.

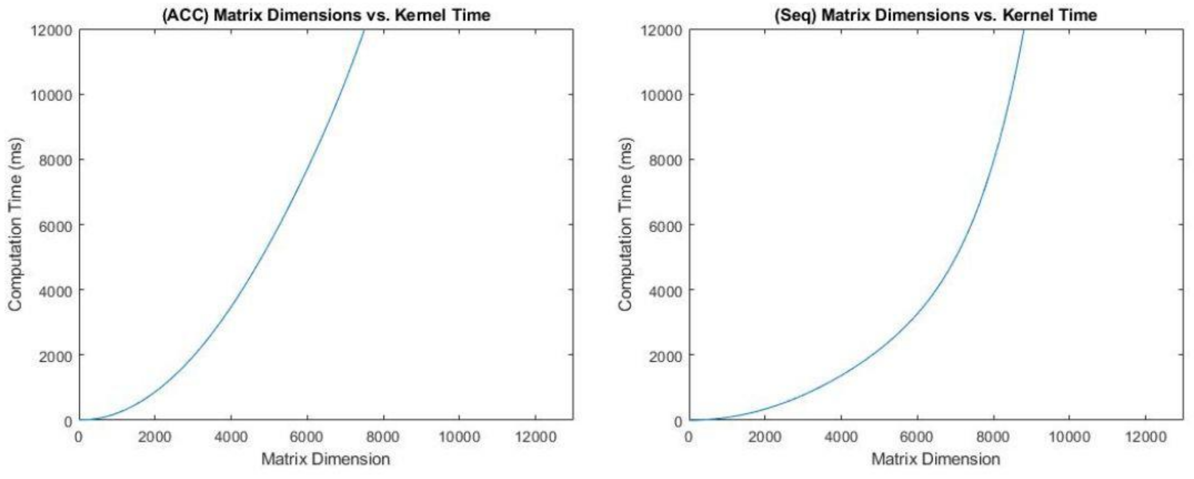

Figure 2 Comparison OpenACC vs Sequential

The next figure compares the performance of the 8x8 block, 5x5 mask naive (right plot) and shared memory(left plot) CUDA implementations. To paraphrase the K&H text, the difference in performance between these two implementations at smaller block and kernel sizes is bordering on negligible. In fact, it appears that the overhead of handling the shared memory is putting the naive implementation slightly ahead towards the larger input data sizes.

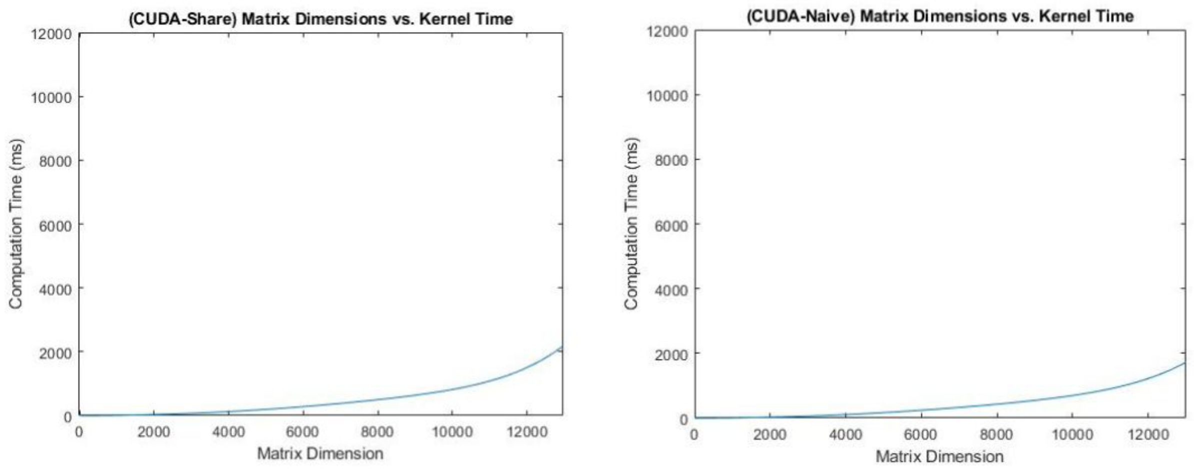

Figure 3 Comparison SharedMemoryCUDA vs NaiveCUDA with 8x8 block size and 5x5 mask size

This becomes more apparent when viewing their comparison again, but with block size 64x64 and mask size 9x9. In the following figure we see the shared memory (left plot) and naive (right plot) CUDA implementations taking very different paths as the input data sizes grow. In short, the increased global memory fetches begin to cause significant congestion for the naïve implementation, whereas the elements loaded into shared memory for the shared memory implementation are finally showing their worth.

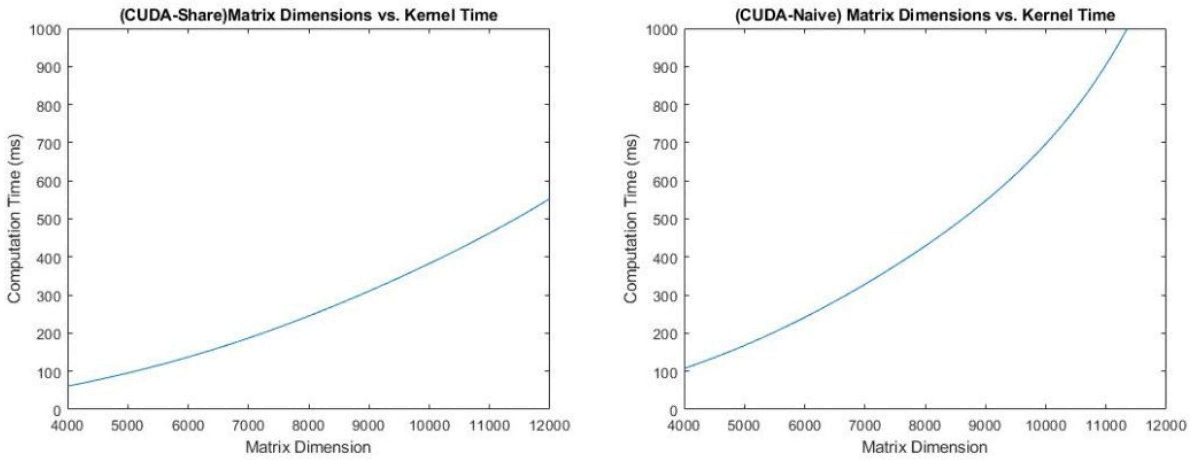

Figure 4 Comparison SharedMemoryCUDA vs NaiveCUDA with block size 64x64 and mask size 9x9

Another exciting result, shown in the plot below, was the comparison between the 64x64 block, 9x9 mask CUDA implementations (left plot) and the 8x8 block, 5x5 mask CUDA implementations (right plot). Keep an eye on the y-axis, despite how the plots look the numbers are going to be what's important here. The left plot representing the larger block and mask size shows the computation time reaching ~600 ms at an input data size of 12000x12000. The right plot representing the smaller block and mask size shows the computation time reaching ~1500 ms at an input data size of 12000x12000. Comparing the raw data confirmed that this pattern was the case for almost all input data sizes besides the very small ones. The larger block and mask size roughly cut in half the computation time needed to complete a Gaussian blur.

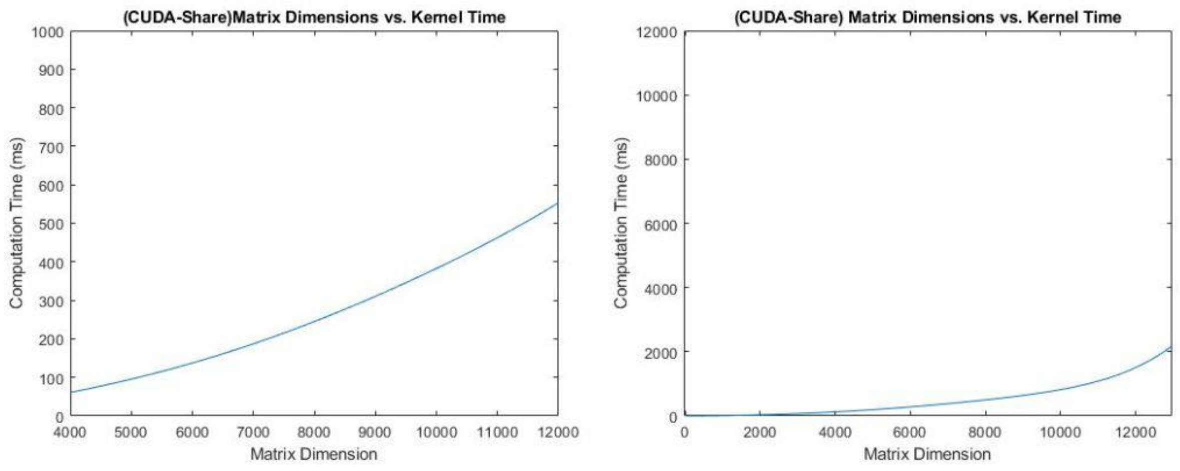

Figure 5 Comparison SharedMemoryCUDA with 64x64 block, 9x9 mask and 8x8 block, 5x5 mask

Most of the unexpected results during this project came from the OpenACC implementation. It is likely that the measurements seen from this implementation were the result of some subtle or strange interaction with how I structured the sequential program. While I would not consider this to be an error (as verified by the OpenACC compiler), I would note that there is likely some small OpenACC-specific changes that could be made by more expert eyes to vastly improve performance. In other words since 2D convolution is parallelizable there must be a way to produce some speedup. What's interesting about this outcome is that it demonstrates the subtleties involved in creating an optimal parallel program with OpenACC. Though more nuanced than it originally seems, it is still much easier to handle than a CUDA program written from scratch. In terms of positive surprises, the naive and shared memory CUDA implementations landed at practically equal performance when executed with a smaller block size. While at the much larger 64x64 block size, the shared memory implementation jumped ahead. This is inline with what the textbook states, nevertheless it is always confounding for a time sink like the shared memory implementation to not perform any better than the sequential implementation.

With a much larger block and kernel size there is more congestion from the fetches in global memory so the shared memory helps to alleviate that traffic and speed up the overall process, hence the shared memory implementation doing much better at the larger block size.

Conclusion and Future Enhancements

To recap what was covered in the previous segments:

● The naive and shared memory CUDA implementations perform and scale similarly at smaller block and kernel sizes.
● At much larger block and kernel sizes the shared memory CUDA implementation not only outperformed the naive CUDA implementation at all input sizes, but scaled better too.
● Both CUDA implementations performance significantly better than the sequential and OpenACC implementations. This was despite the lengthy copy times involved and no kernel overlapping.
● The OpenACC implementation performed slightly worse than the sequential implementation.

The clearest takeaway from these measurements is the substantial performance boost resulting solely from increasing block size in the CUDA implementations. Both naive and shared memory implementations experienced a reduction in computation time upon raising block size from 8x8 to 64x64 equivalent to ~2X speedup (this can be inferred from the plots). Doing this is not always easy though as raising the block size to this extent excludes smaller input data; with the maximum number of resident blocks per SM being a minimum of 16 on NVIDIA devices this amounts to a minimum of 256x256 input data size (and this only grows for the rest of the devices where maximum resident blocks per SM reaches 32!)

The other priority takeaway is the improved scaling of the shared memory CUDA implementation using block size 64x64 and mask size 9x9 vs. block size 8x8 and mask size 5x5. Referencing the K&H textbook, the image array access reduction ratio for an 8x8 block paired with a 5x5 mask is a low 11.1. Moving up to a 64x64 block paired with a 9x9 mask, the text shows the image array access reduction ratio at 64, this is a substantial increase from the former. In other words, the shared memory implementation only becomes useful if given a larger block and kernel size. While this is certainly backed by the text, this is also evident as shown by the aforementioned plots. The shared memory implementation with 64x64 block size rises in computation time at a much steadier rate than its naive counterpart.

Future Enhancements

There are two improvements I would like to see in future work on this project. Most important would be the lack of NVIDIA resources used in debugging errors and measuring performance. NVIDIA provides tools like CUDA-GDB for solving errors in CUDA programs and Nsight Visual Studio for monitoring and measuring analytics. With more time these tools could provide essential information about the CUDA implementations. Likewise for the OpenACC implementation; the OpenACC compiler comes with a variety of tools made for the purpose of measuring performance and providing feedback. But these tools, much like CUDA and OpenACC, are practically languages in and of themselves. Becoming proficient in these resources will require far more time than what is available for side projects during an academic quarter.
Data throughput is another concept I would enjoy using more in this project. While in their current state the CUDA implementations would not showcase useful measurements in data throughput because only one kernel is executed at a time with no overlapping. It is stated in the K&H text and mentioned in class that GPUs hide their transfer latency by overlapping kernels.
Overlapping kernels allows back-to-back execution which increases data throughput when running multiple iterations as was done in this project to obtain the data over thousands of input data sizes. I believe this type of measurement would showcase GPUs in the environment they shine most in.

Appendix

Appendix A: Output

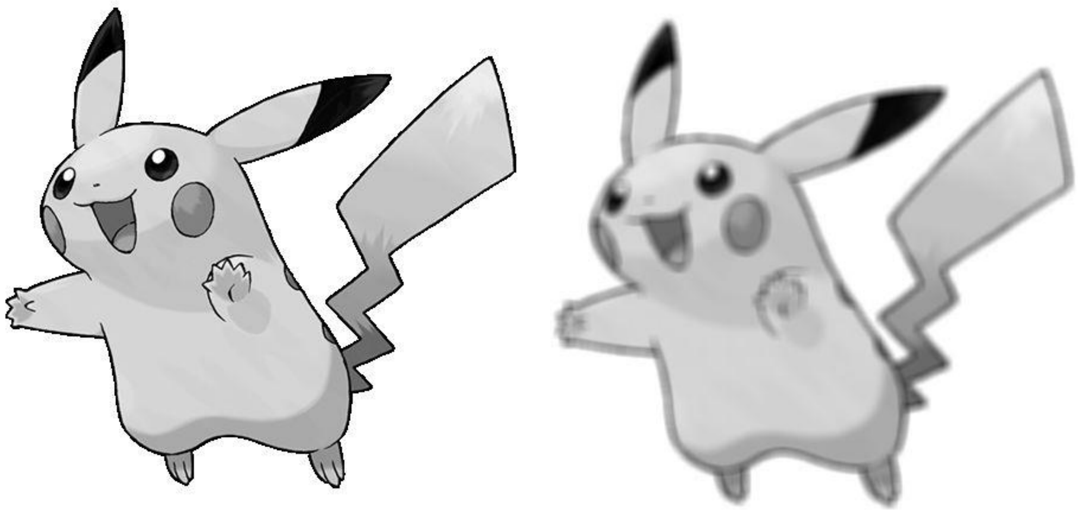

Figure 6 Original Image and Blurred Image

Appendix B: Code

Sequential Code

```
#include<opencv2/imgproc.hpp>
#include<opencv2/highgui.hpp>
#include<iostream>
#include <chrono>
using namespace std::chrono;

using namespace cv;
using namespace std;

int main(){
    //Read image
    Mat image = imread("Pikachu.jpg",IMREAD_GRAYSCALE);

    //check if image exits
    if(image.empty()){
        cout<<"can not find image"<<endl;
        return 0;
    }
    Mat result1, result2;

    //Apply gaussian filter with kernel size 5
    auto start = high_resolution_clock::now();
    GaussianBlur(image, result1, Size(7, 7), 0, 0);
    auto stop = high_resolution_clock::now();

    imwrite("out_seq.jpg",result1);

    auto duration = duration_cast<milliseconds>(stop - start);
    cout << (double)duration.count()/1000 << endl;

    return 0;
}
```

Parallelized Code with Shared-Memory CUDA

```
#include <device_launch_parameters.h>
#include <cuda_runtime.h>
#include <opencv2/opencv.hpp>
#include <iostream>
#include <cmath>
#include <sys/time.h>
using namespace cv;
using namespace std;
struct timeval timer_start, timer_end;
#pragma once
#ifdef __INTELLISENSE__
        void __syncthreads();
#endif

#define MAX_KERNEL_WIDTH 441
__constant__ double K[MAX_KERNEL_WIDTH];

__global__ void Gaussian(double*, double*, int, int, int, int);
```

```
__host__ void generateGaussian(vector<double>&, int, int);
__host__ void errCatch(cudaError_t);
template<typename T> size_t vBytes(vector<T>&);

int main() {
	vector<double> hIn, hKernel, hOut;
	double* dIn, * dOut;
	int inCols, inRows;
	int kDim, kRadius;
	int outCols, outRows;
	int max = 0;
	double bw = 8;

	/*
	 * Load image into OpenCV matrix and transfer to vector as linearized matrix
	*/

	Mat image = imread("/content/GaussianBlur-CUDA/Images/Pikachu.jpg", IMREAD_GRAYSCALE);
	if (!image.data || !image.isContinuous()) {
		cout << "Could not open image file." << endl;
		exit(EXIT_FAILURE);
	}
	hIn.assign(image.data, image.data + image.total());
	inCols = image.cols;
	inRows = image.rows;
	hOut.resize(inCols * inRows, 0);

	/*
	 * Set mask dimensions and determine whether image and masks dimensions are compatible
	*/

	kDim = 5; // Kernel is square and odd in dimension, should be variable at some point
	if ((inRows < 2 * kDim + 1) || (inCols < 2 * kDim + 1)) {
		cout << "Image is too small to apply kernel effectively." << endl;
		exit(EXIT_FAILURE);
	}
	kRadius = floor(kDim / 2.0); // Radius of odd kernel doesn't consider middle index
	hKernel.resize(pow(kDim, 2), 0);

	gettimeofday(&timer_start, NULL);

	generateGaussian(hKernel, kDim, kRadius);

	// Trim output matrix to account for kernel size
	outCols = inCols - (kDim-1);
	outRows = inRows - (kDim-1);

	/*
	 * Device matrices allocation and copying
	*/

	errCatch(cudaMalloc((void**)& dIn, vBytes(hIn)));
	errCatch(cudaMemcpy(dIn, hIn.data(), vBytes(hIn), cudaMemcpyHostToDevice));
	errCatch(cudaMalloc((void**)& dOut, vBytes(hOut)));
	errCatch(cudaMemcpy(dOut, hOut.data(), vBytes(hOut), cudaMemcpyHostToDevice));
	errCatch(cudaMemcpyToSymbol(K, hKernel.data(), vBytes(hKernel)));

	/*
	 * Kernel configuration and launch
	*/
```

```
            int bwHalo = bw + (kDim-1); // Increase number of threads per block to account for halo cells
            dim3 dimBlock(bwHalo, bwHalo);
            dim3 dimGrid(ceil(inCols / bw), ceil(inRows / bw));
            Gaussian <<<dimGrid, dimBlock, bwHalo*bwHalo*sizeof(double)>>>(dIn, dOut, kDim, inCols,
outCols, outRows);
            errCatch(cudaDeviceSynchronize());
            errCatch(cudaMemcpy(hOut.data(), dOut, vBytes(hOut), cudaMemcpyDeviceToHost));
            errCatch(cudaDeviceSynchronize());

            /*
             * Normalizing output matrix values
            */

            for (auto& value : hOut)
                        max = (value > max) ? value : max;
            for (auto& value : hOut)
                        value = (value * 255) / max;

            /*
             * Converting output matrix to OpenCV Mat type
            */

            vector<int> toInt(hOut.begin(), hOut.end()); // Converting from double to integer matrix
            Mat blurImg = Mat(toInt).reshape(0, inRows);
            blurImg.convertTo(blurImg, CV_8UC1);
            Mat cropImg = blurImg(Rect(0, 0, outCols, outRows));

            /*
             * Display blurred image
            */


            gettimeofday(&timer_end, NULL);
            imwrite("out_cuda.png",cropImg);

            double time_spent = timer_end.tv_sec - timer_start.tv_sec +
                  (timer_end.tv_usec - timer_start.tv_usec) / 1000000.0;

            cout << "Time for paralel computation section: "<< time_spent << "  seconds." << endl;

            image.release();
            errCatch(cudaFree(dIn));
            errCatch(cudaFree(dOut));

            exit(EXIT_SUCCESS);
}

// CUDA kernel, it performs the image convolution
__global__
void Gaussian(double* In, double* Out, int kDim, int inWidth, int outWidth, int outHeight) {
            extern __shared__ double loadIn[];

            // trueDim is tile dimension without halo cells
            int trueDimX = blockDim.x - (kDim-1);
            int trueDimY = blockDim.y - (kDim-1);

            // trueDim used in place of blockDim so Grid step/stride does not consider halo cells
            int col = (blockIdx.x * trueDimX) + threadIdx.x;
            int row = (blockIdx.y * trueDimY) + threadIdx.y;
```

```
    if (col < outWidth && row < outHeight) { // Filter out-of-bounds threads

        // Load input tile into shared memory for the block
        loadIn[threadIdx.y * blockDim.x + threadIdx.x] = In[row * inWidth + col];
        __syncthreads();

        if (threadIdx.y < trueDimY && threadIdx.x < trueDimX) { // Filter extra threads used for halo
cells
            double acc = 0;
            for (int i = 0; i < kDim; ++i)
                for (int j = 0; j < kDim; ++j)
                    acc += loadIn[(threadIdx.y + i) * blockDim.x + (threadIdx.x + j)] *
K[(i * kDim) + j];
            Out[row * inWidth + col] = acc;
        }
    } else
        loadIn[threadIdx.y * blockDim.x + threadIdx.x] = 0.0;
}

// This function takes a linearized matrix in the form of a vector and
// calculates elements according to the 2D Gaussian distribution
__host__
void generateGaussian(vector<double> & K, int dim, int radius) {
    double stdev = 1.5;
    double pi = 355.0 / 113.0;
    double constant = 1.0 / (2.0 * pi * pow(stdev, 2));

    for (int i = -radius; i < radius + 1; ++i)
        for (int j = -radius; j < radius + 1; ++j)
            K[(i + radius) * dim + (j + radius)] = constant * (1 / exp((pow(i, 2) + pow(j, 2)) / (2 *
pow(stdev, 2))));
}

// Catches errors returned from CUDA functions
__host__
void errCatch(cudaError_t err) {
    if (err != cudaSuccess) {
        cout << cudaGetErrorString(err) << endl;
        exit(EXIT_FAILURE);
    }
}

// Returns the size in bytes of any type of vector
template<typename T>
size_t vBytes(vector<T> & v) {
    return sizeof(T)* v.size();
}
```