PC-2020/21 Kernel Image Processing

Anonymous CVPR submission

Paper ID ****

Abstract

In this final project I have implemented a Kernel Image Processing in three different C-based version: a sequential one in pure C, a parallel one in C using the OpenMP framework for parallelism and another parallel one in the C extension CUDA-C. Then I tested all the versions processing large images in order to evaluating the speed-up.

Future Distribution Permission

The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

1. Introduction

In Image Processing one of the basic operations, or better transformations, is the filtering one. A filtering transformation can be performed by applying to the image a convolution with a kernel matrix: this operation is known as Kernel Image Processing (more details in section 2).

In this project I have implemented the Kernel Image Processing, focusing on the blurring operations.

In the following sections I will present my implementations, specifically:

- a first, very basic, one, sequential, in C;
- a second one exploiting the implicit parallelism paradigm given by the OpenMP framework (again based in C) and
- a last one, also parallel, but exploiting all the parallel power of GPU (maybe the best solution talking about Image Processing), developed in CUDA-C.

In the lasts sections I report the result of the performance tests, discussing the speed up and proposing my personal considerations about the efficiency of the solutions.

2. Kernel Image Processing

Kernel Image Processing [1] is used mainly to blurring, sharpening e edge detection operations (and a lot more). It consists essentially in a convolution between the image (obviously seen as a matrix) and a *kernel* matrix, also called *mask*. This matrix can theoretically have any dimension, but usually is squared (and from now on we will consider only this case). Mathematically speaking, said I the image and K the kernel of dimension d, the result P will be:

$$\begin{aligned} P(x,y) &= K * I(x,y) \\ &= \sum_{dx=-d}^{d} \sum_{dy=-d}^{d} K(dx,dy) I(x+dx, y+dy) \end{aligned}$$

or in terms of matrix:

$$P_{x,y} = \sum_{dx=-d}^{d} \sum_{dy=-d}^{d} K_{dx,dy} I_{x+dx,y+dy}$$

The kernel have to be *normalized* in order to preserve brightness. The normalization consists in dividing all the elements of the kernel by the sum of all of them; at the end the sum of all the elements will be one (for convenience in representation we leave unchanged the elements and we multiply all the matrix for the inverse of the sum of the elements). Doing so we obtain that the average pixel in the modified image is as bright as the average pixel in the original image.

CVPR
#****

The effects that can be achieved with Kernel Image Processing are many; for simplicity we consider the case of blurring.

2.1. Blurring

Blurring effect is given simply by taking as result pixel the average of input pixel and its neighbour. More neighbours is taken and more blurred is the result. There are mainly two way for blurring effect: box blur and Gaussian blur.

2.1.1 Box Blur

This is the more naïve version: all the pixels considered have the same weight, that is the same importance. The kernel simply consists in all equal elements [2]:

$$K = \frac{1}{d^2} \overbrace{\begin{pmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{pmatrix}}^{\text{d}}$$

2.1.2 Gaussian Blur

This is maybe the most used version of blurring. This version assigns an higher weight to pixel in the proximate vicinity of the pixel in question. Mathematically it consists in convolving the image with a Gaussian function (which 2-dimensional form is the product of two 1-dimensional Gaussian function) $G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$ (see in [3]). In order to define a kernel matrix we need a discrete approximation of it, but the Gaussian function has infinite support, so generally we truncate at some point (doing so the sum of all elements of the kernel will not be exactly 1, but almost 1).

Another way is that of approximating the Gaussian function, which is the probability density function of a Gaussian distribution, with the probability density function of the Binomial distribution (in [4]): indeed according to Central Limit Theorem this is a good approximation (to tell the truth better as the dimension of the kernel grows). Here we consider this approximation.

Figure 1. An image and its Gaussian blurred version (kernel dimension: 25×25)

The approximated kernel so will be the product of two Binomial distribution, whose (discrete) density function is a row of Pascal's triangle. So the d dimension approximated kernel K is:

$$K = \omega\omega^T$$

where ω is the d-th row of Pascal's Triangle (1-indexed). Obviously all multiplied by the inverse of the sum of the elements, that is $2^{2(d-1)}$ (proof in 8.1). for example, the kernel of dimension 5 will be:

$$K_5 = \frac{1}{2^{2\cdot 4}} \begin{pmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{pmatrix}^T = \frac{1}{256} \begin{pmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix}$$

Obviously this kernel is, by construction, separable, so it can be applied in a computationally convenient way first along a direction and then along the other; but here we consider the classic case.

2.2. Edge Handling

The convolution, when we consider a pixel close enough to the border of image (that is in the $\frac{d-1}{2}$-wide border), needs values from outside

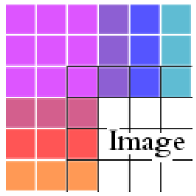

Figure 2. Edge handling behaviour in ”extend” method

the image. There are a variety of methods to handle this; here I have used the *extend* method. It consists in extending the nearest pixel as far as needed: conceptually the corner pixel is extended in 90° wedges, other edge pixels are extended in lines (like in figure 2).

3. Solution Implementation

In my opinion the most suitable language/framework for implementing Kernel Image Processing is CUDA since often it is the best working with image, where every pixel is independent and generally the dimension is really huge (e.g. for an image in 4K resolution and 16:10 ratio, the pixels amount to 10.5M).

Anyway without resorting to GPU, maybe the best choice is OpenMP thanks to its implicit parallelism paradigm.

Before to talk about the three specific implementations, let's take a look on some aspects that they have in common and that go beyond the scope of this project.

3.1. Image handling

The tool used for handling the images, intended as reading an image into a C struct and writing a C struct into an image, is a library called STD [5].

This library allow to import an image with the function `stbi_load()`. The input images are in JPEG format; I have wrapped this library function in the `loadJPEG()` function.

The output instead will be a PNG image, so I used the `stbi_write_png()` wrapped for convenience in the `savePNG()` function.

The C struct `Image` store the image in the classic way, that is an array of interleaved RGB pixels memorized as `unsigned char`.

3.2. Kernel storing

I have decided to store the kernel in his non normalized form, as matrix of integer coefficients and a weight (that obviously is the sum of all the elements). The reason is that, being the RGB substantially an integer, is more efficient to handle an integer along all the process. A simple aspect is that when I will implement the CUDA version (with my Compute Capability <6.0 GPU) I will be able to use the atomic operation, if needed. The other main reason is that in Gaussian kernel the normalized coefficients will become quickly very small, also under the float or even double precision.

Indeed for the tests I will have to reach non trivial kernel's dimension. But the non normalized elements grows very fast, so I decided to use an `unsigned long long int` for storing the coefficients. This allow to reach a Gaussian kernel of dimension 25, without overflow neither in the construction nor in the processing phase.

4. Sequential Solution

In this version there is this function:

```
Image *process(Image *img, Kernel *krn)
```

that from the input image and the kernel produces the output image. Its structure is trivial: 5 nested cycles. The first two cycles iterate on the pixels, the third iterates on the channels of each pixel; here is initialized a sum variable before the lasts two cycles that iterate on the elements of the kernel matrix. Inside them there are only some lines of code for edge handling and some other for operating the convolution, that is adding in the counter variable the value of the pixels' channels multiplied by the value of the kernel's elements.

```
for y from 1 to height
  for x from 1 to width
    for c in channels
      sum = 0;
        for i from 1 to d
          for j from 1 to d
            // edge handling
            sum += I(x,y)*K(i,j);
```

```
    sum = sum/K.weight;
    // store output pixel
```

The three external cycles are ordered so that the pixels are processed in the order they are memorized. Evidently this structure (possibly with minor changes and optimization) is particularly suitable for a parallelization with OpenMP.

This code is available in this GitHub repo.

4.1. More details

For the problem of Kernel Image Processing, or at least for this implementation, there's not need to time profile an execution because the algorithm is very simple and it is trivial to identify the hotspot: not only because the five nested loops are particularly heavy-weight from a computation point of view, but also because basically it is the only computation to be done in order to solve the problem.

So I can skip the time profiling phase and jump directly to tests. In the reality before testing I took some time for profile the memory usage with Valgrind Memcheck in order to eliminate all the remaining memory leaks.

5. Parallel Solution: OpenMP

One of the pros of OpenMP is that the sequential and the parallel code can be unified, indeed very often the parallelization consist of only some directives, that obviously are ignored in the sequential case. As I said in the previous section, the structure of this code was very suitable for OpenMP parallelization, indeed I only need a single directive:

```
#pragma omp parallel for
```

Clearly the parallelization hits the outermost cycle.

At this point I only needed to specify the variable sharing: input image, kernel and output image must be shared, indices and sum variables must be clearly private (we don't need to initialize them), and kernel radius firstprivate (because it has already been calculated).

There are still two aspects to consider: the schedule and the number of threads. For the schedule, the most used option is the static one, so I will use this one (but then, in the tests I will validate this choice); about the number of threads we have to consider that this task is purely computational, so there should be a 1:1 ratio between threads and cores, as explained in [6]. I consider to put an extra thread to supply page fault events, but my test machines are equipped with Intel processors with Hyperthreading, so setting the number of threads equal to the number of virtual threads is sufficient to prevent page faults delays; anyway later we validate also this assumption with tests.

This code is available in this GitHub repo.

5.1. Further attempts

5.1.1 Reduction

The two internal cycles basically perform a summation: this is the case where a code can be parallelized with a reduction pattern. First of all I serialized the double cycle and then I applied an OpenMP reduction.

Anyway the performance was quite bad. Despite I instantiated the thread pool once outer all the cycles, evidently launching a reduction for every pixel is not so convenient because of overhead.

5.1.2 Serialized cycle over pixels

Obviously parallelizing more than one of the outermost cycle doesn't bring advantages because the only effects we obtain is overhead. But I thought that serializing the three cycles and parallelizing the single resulting cycle could have boost the performance.

Anyway my prediction was wrong: with a static schedule the performance was a bit worse, with a dynamic schedule the situation gets better, but does not improve the performance obtained with nested cycles. Effectively with a static schedule I obtain the same partition of pixels over threads (at least in their number), but with a dynamic schedule the partition can be adjusted to face the interruption of a thread by a system process (being using all the cores); but evidently not enough to improve the result obtained with the first version.

6. Parallel Solution: CUDA

From CUDA I expect a very high speedup: the problem is entirely embarrassingly parallel and on GPU is possible to have really huge number of threads, so it is possible to assign at each CUDA thread an element (i.e. a pixel). Also in this case it would be possible to parallelize at an "inner" level, but in this case would be even a worse idea, indeed so doing I will reduce the portion of parallelizable code wasting a lot of GPU computation power. So the parallelization has been done at pixel level.

The main focus in CUDA, in order to exploit all the available GPU power, will be the memory management. I have proceeded step by step from a first naïve version to a multi-optimized one.

All the code is available at this GitHub repo.

From now on, instead of the term *kernel* to indicate the matrix to convolve, I will use *mask* to avoid ambiguity with the CUDA kernel function.

6.1. 1st step: naïve version

It consists in the classic 5-step process:

1. Allocation of device variables: in this case input images and mask;
2. Copy Host-to-Device;
3. Kernel call;
4. Copy Device-to-Host: the result, i.e. the output image;
5. Free of device allocation.

The kernel simply consist in calculation of the pixel to process, with `ThreadIdx` and `BlockIdx`, and then the cycle on the channels and the convolution (if the thread correspond to a pixel inside the image) exactly as it is done in the serial version.

6.1.1 Block Width

An important choice is that of the block width. The number of threads in a block should be multiple of the warp size (32) and less than 1024, that is the hardware limit. Obviously the block has been made 2D for adapting to the input shape (the image). So the block width possible values are 8, 16, 24 and 32. I test all the values and that with high performance is 32, as I will discuss in the test section.

6.2. 2nd step: constant memory usage

Also in the naïve version I have included a cache optimization for the mask, passing to the kernel the pointer to the mask as `const __restrict__`.

Then at this point the mask is perfect for exploiting the advantage of constant memory, that offer an extreme caching policy, being exactly a constant during all the computation.

The constant variables have to be of a compile-time-known size, so I allocate the larger mask possible: `__constant__ KERNEL[25 * 25]`. Then obviously I copied in it only the actual mask.

6.3. 3rd step: shared memory usage

This step will bring the main improvement. At the actual point each thread reads d^2 pixels, of which $d^2 - d$ in common with the adjacent threads; it is a quite large number. The fact is that all this access are referred to global memory, so there is a huge waste of resources.

The goal is to use shared memory to reduce the number of reads in global memory. To do so I use the classic *tiling* approach: each block of threads cooperatively load from the global to the shared memory all the pixels that are used for the processing. Substantially it locally stores (per-block) a subimage of $w = block\_width + d - 1$ pixels per side, from which each thread reads the needed value without accessing to global memory.

The reduction of global memory accesses is important: from a minimum of 34.7 times for smaller mask to a maximum of 204 times for bigger mask (more details in 8.2).

One difference with respect to a common implementation of tiling is that here mask size reach uncommon value (25) and for high values the assumption that each thread have to loads at most two pixels doesn't hold anymore. So I don't have implemented a classic two-batch loading, but a multi-batch loading, with the number of batches determined at runtime.

The edge handling is carried out at load time, not at process time anymore.

The tile width is again 32, validated by test results.

A last note: again being the mask size determined at runtime, I can't use a static allocation for shared memory, so I resorted to the dynamic one at kernel-launch moment: `kernel<<<gridDim,blockDim,w*w>>>`.

6.3.1 Tile Width

Being completely changed the memory management, it was not obvious that the previous optimal block width will be the optimum also in this case. I tested again the 8, 16, 24 and 32 size and indeed 16 turns out to be the optimum in this configuration.

6.4. 4th step: pinned memory usage

As last step I have considered a more hidden aspect. Let's take the case of the biggest image, 8000×6000 pixels, each with 3 channels represented with a `char`: it is a total of 144MB of memory of array. Not so much, but it starts to be a considerable amount.

On a machine with reduced amount of memory can be present a delay during the phase of copying the data from the host to the device if the memory page containing that data has been swapped to disk. To prevent this I used pinned memory for storing the images data.

7. Tests and Results

The execution time obviously grows as image and kernel dimensions grow. I have considered five different dimensions for input images: 4000×2000, 5000×3000, 6000×4000, 7000×5000 and 8000×6000 for a total of, respectively, 8M, 15M, 24M, 35M and 48M of pixels. I tested every dimension for different kernel dimensions: 7, 13, 19 and 25[1].

The execution time does not depend on the specific input image because the number of operations is constant. However in order to avoid result's alterations due to casual fluctuations I run the test (for every input) 3 times, seizing the opportunity to process different images. I have summarized the results in Tables 1 and 2, where I report mean execution time, speedup and efficiency.

A last note: the tests has been executed compiling the code in release mode; this aspect has allowed to save around 33% of execution time.

Input		sequential C	parallel C + OpenMP		
		T_S	T_2	S_2	E_2
4K	$d=7$	$6.1s$	$2.7s$	2.26	1.13
	$d=13$	$19.7s$	$8.5s$	2.32	1.16
	$d=19$	$45.1s$	$17.7s$	**2.55**	**1.28**
	$d=25$	$75.8s$	$31.8s$	2.38	1.19
5K	$d=7$	$11.4s$	$5.0s$	2.28	1.14
	$d=13$	$36.9s$	$15.9s$	2.32	1.16
	$d=19$	$84.7s$	$33.2s$	**2.55**	**1.28**
	$d=25$	$142.4s$	$59.8s$	2.38	1.19
6K	$d=7$	$18.2s$	$8.0s$	2.28	1.14
	$d=13$	$58.9s$	$25.5s$	2.31	1.16
	$d=19$	$135.3s$	$53.1s$	**2.55**	**1.28**
	$d=25$	$227.6s$	$95.5s$	2.38	1.19
7K	$d=7$	$26.6s$	$11.7s$	2.27	1.14
	$d=13$	$86.3s$	$37.2s$	2.32	1.16
	$d=19$	$197.4s$	$77.5s$	**2.55**	**1.28**
	$d=25$	$332.4s$	$139.0s$	2.39	1.20
8K	$d=7$	$35.1s$	$16.0s$	2.19	1.10
	$d=13$	$108.2s$	$51.0s$	2.12	1.06

Table 1. Time, speedup and efficiency for all test cases in OpenMP

[1] 8K images tested only for littler kernel dimensions

Input		sequential C	parallel CUDA-C							
			naïve		constant		shared		constant+shared	
		T_S	T	S	T	S	T	S	T	S
4K	$d = 7$	$6.1s$	$0.15s$	40.7	$0.16s$	38.1	$0.14s$	43.6	$0.13s$	46.9
	$d = 13$	$19.7s$	$0.38s$	51.8	$0.39s$	50.5	$0.27s$	73.0	$0.25s$	78.8
	$d = 19$	$45.1s$	$0.74s$	60.9	$0.79s$	57.1	$0.46s$	98.0	$0.45s$	100.2
	$d = 25$	$75.8s$	$1.24s$	61.1	$1.32s$	57.4	$0.75s$	101.0	$0.72s$	105.3
5K	$d = 7$	$11.4s$	$0.28s$	40.7	$0.28s$	40.7	$0.25s$	45.6	$0.23s$	49.6
	$d = 13$	$36.9s$	$0.70s$	52.7	$0.74s$	49.9	$0.50s$	73.8	$0.47s$	78.5
	$d = 19$	$84.7s$	$1.40s$	60.5	$1.47s$	57.6	$0.87s$	97.4	$0.84s$	100.8
	$d = 25$	$142.4s$	$2.33s$	61.1	$2.47s$	57.7	$1.40s$	101.7	$1.35s$	105.5
6K	$d = 7$	$18.2s$	$0.44s$	41.4	$0.45s$	40.4	$0.39s$	47.7	$0.35s$	52.0
	$d = 13$	$58.9s$	$1.12s$	52.6	$1.18s$	49.9	$0.80s$	73.6	$0.75s$	78.5
	$d = 19$	$135.3s$	$2.22s$	60.9	$2.33s$	58.1	$1.38s$	98.0	$1.34s$	101.0
	$d = 25$	$227.6s$	$3.68s$	61.8	$3.94s$	57.8	$2.22s$	102.5	$2.16s$	105.3
7K	$d = 7$	$26.6s$	$0.63s$	42.2	$0.64s$	41.6	$0.55s$	48.4	$0.51s$	52.2
	$d = 13$	$86.3s$	$1.63s$	52.9	$1.70s$	50.8	$1.17s$	73.8	$1.09s$	79.2
	$d = 19$	$197.4s$	$3.23s$	61.1	$3.39s$	58.2	$2.02s$	97.7	$1.93s$	101.2
	$d = 25$	$332.4s$	$5.36s$	62.0	$5.74s$	57.9	$3.23s$	102.9	$3.13s$	106.2
8K	$d = 7$	$35.1s$	$0.86s$	40.8	$0.87s$	40.3	$0.75s$	46.8	$0.69s$	50.9
	$d = 13$	$108.2s$	$2.23s$	48.5	$2.33s$	46.4	$1.59s$	68.1	$1.49s$	72.6

Table 2. Time and speedup for all test cases in CUDA

7.1. OpenMP

7.1.1 Main tests

As I said this tests has been run with static schedule and 4 threads. The results highlight that this solution exploits very well the Hyperthreading, indeed the speedup is always fairly greater than 2, with peak of 2.55, for an efficiency of 1.28. This is a great result, but again is not all thanks to OpenMP, but the presence of Hyperthreading is also relevant.

7.1.2 Optimal parallelism degree and schedule tests

From previous results seems that the speedup depends on the kernel's dimension (i.e. the dimension of the two inner cycles) and not on the image dimension, so I run this test only on a single image dimension, the middle one, averaging between kernel dimension. I changed the number of threads and the schedule type, and the results are those in figures 3 and 4. The most efficient schedule is the static one by far and the optimal number of threads is 4 (although in the static schedule case we obtain exactly the same speedup also with 2 threads), as predicted.

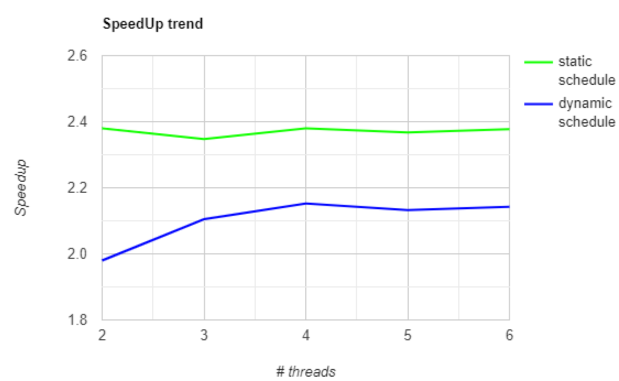

Figure 3. Speedup trend varying the number of threads

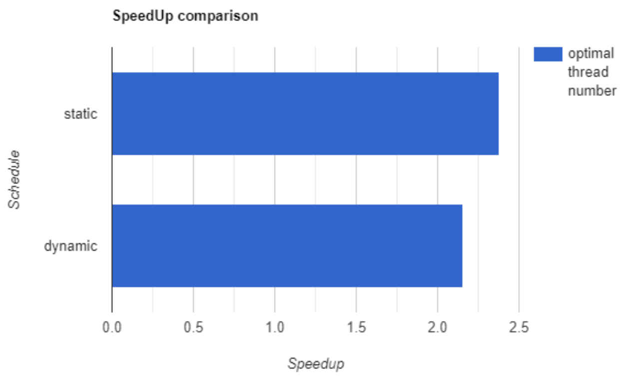

Figure 4. Speedup comparison varying schedule

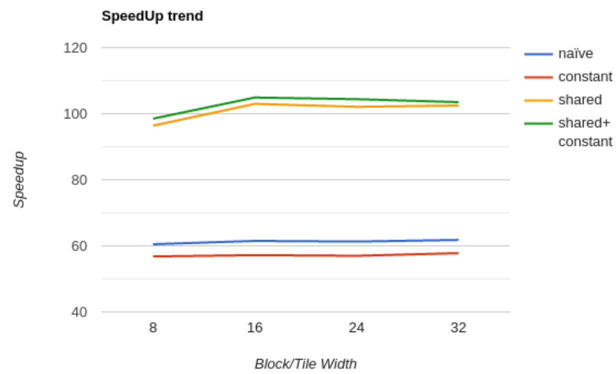

Figure 5. Speedup comparison varying block/tile width

7.2. CUDA

As expected I obtain a speed up of several dozens, up to over one hundred. But let's examine the results more deeply.

A first aspect to discuss is that the speed up grows exclusively with mask size: this is obvious, having a thread per pixel, the only difference is how many operations has to do a single thread, i.e. how big is the mask.

With the naïve version I achieved speed up from 40.7 to 62.0, the version with shared memory improve this results and amplifies the difference between minimum and maximum, from 43.6 up to 102.9. This is exactly what I expected.

The constant memory bring a weird improvement: using only the constant memory I have a little degradation with respect to the naïve version, but using it in combination with shared memory I obtain an improvement (again little) with respect to the only shared memory version (speed up from 46.9 to 106.2).

Last note about the pinned memory: its usage bring only a little degradation, but that is easily explainable, indeed my machine has sufficient memory to not swap that area of memory on the disk (especially under the test conditions), so the only effect was a little instantiation overhead.

7.2.1 Optimal Block Width

As I said before I tested each version with block-tile of dimension 8, 16, 24 and 32. For the non shared memory versions, the difference between all values is minimal (as shown in figure 5) with 32 as optimal value. The shared memory versions instead with size 8 shown a big difference, in negative, with other sizes, which instead are very close each other; in that case the optimal value is 16.

Given the big difference in speedup between smaller and larger mask size, I tested the middle sized image only with the bigger mask possible.

References

[1] *Kernel image processing*, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Kernel_(image_processing) (visited on Oct. 15, 2020).

[2] *Box blur*, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Box_blur (visited on Oct. 15, 2020).

[3] *Gaussian blur*, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Gaussian_blur (visited on Oct. 15, 2020).

[4] *Gaussian bluring*. [Online]. Available: https://theailearner.com/2019/05/06/gaussian-blurring/ (visited on Oct. 15, 2020).

[5] *Stb image library*. [Online]. Available: https://github.com/nothings/stb (visited on Sep. 15, 2020).

[6] N. Matloff, *Programming on Parallel Machines: GPU, Multicore, Clusters and More*. University of California, 2012, ch. 3.

8. Math details

8.1. Gaussian Blur Kernel Approximation

The approximated Gaussian kernel K is obtained element wise in this way: $K_{i,j} = a_i \cdot a_j$. The sum of all elements so will be

$$\sum_{i=0}^{d} \sum_{j=0}^{d} a_i a_j = \sum_{i=0}^{d} a_i \sum_{j=0}^{d} a_j = 2^{d-1} \cdot 2^{d-1} = 2^{2(d-1)}$$

being the sum of the elements of the d-th Pascal's triangle's row 2^{d-1}.

8.2. Reduction of global memory accesses

The number of global reads without the use of shared memory is $block\_width^2 \cdot d^2$, while using the shared memory they are only $(block\_width + d - 1)^2$. The ratio in the smaller mask case is $\frac{32^2 \cdot 7^2}{38^2} = 34.7$, while in the larger case $\frac{32^2 \cdot 25^2}{56^2} = 204.1$.

9. Additional Notes

9.1. Technical Specification

Main tests were run on/with:

- Intel Core i7-6500U (2.5 GHz up to 3.1 GHz, 2 cores, 4 threads, 4 MB of cache);
- 8 GB of RAM;
- NVIDIA GeForce 940MX (Maxwell architecture, Compute Capability 5.0, 512 CUDA cores, 4 SMM, 2GB DDR5 memory, 40.10 GB/s bandwidth);
- Ubuntu 20.10.
- NVIDIA Toolkit 11.2