

000  
001  
002  
003  
004  
005  
006  
007  
008  
009  
010  
011

## 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.

Anonymous CVPR submission

Paper ID \*\*\*\*

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:

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

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.

054  
055  
056  
057  
058  
059  
060  
061  
062  
063  
064  
065  
066  
067  
068  
069  
070  
071  
072  
073  
074  
075  
076  
077  
078  
079  
080  
081  
082  
083  
084  
085  
086  
087  
088  
089  
090  
091  
092  
093  
094  
095  
096  
097  
098  
099  
100  
101  
102  
103  
104  
105  
106  
107

108      The effects that can be achieved with Kernel  
 109      Image Processing are many; for simplicity we  
 110      consider the case of blurring.  
 111

## 112      2.1. Blurring

114      Blurring effect is given simply by taking as re-  
 115      sult pixel the average of input pixel and its neigh-  
 116      bour. More neighbours is taken and more blurred  
 117      is the result. There are mainly two way for blur-  
 118      ring effect: box blur and Gaussian blur.  
 119

### 121      2.1.1 Box Blur

123      This is the more naïve version: all the pixels con-  
 124      sidered have the same weight, that is the same im-  
 125      portance. The kernel simply consists in all equal  
 126      elements [2]:  
 127

$$128 \quad K = \frac{1}{d^2} \underbrace{\begin{pmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{pmatrix}}_d$$

### 135      2.1.2 Gaussian Blur

137      This is maybe the most used version of blur-  
 138      ring. This version assigns an higher weight  
 139      to pixel in the proximate vicinity of the pixel  
 140      in question. Mathematically it consists in con-  
 141      volving the image with a Gaussian function  
 142      (which 2-dimensional form is the product of  
 143      two 1-dimensional Gaussian function)  $G(x, y) =$   
 144       $\frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$  (see in [3]). In order to define a ker-  
 145      nel matrix we need a discrete approximation of  
 146      it, but the Gaussian function has infinite support,  
 147      so generally we truncate at some point (doing so  
 148      the sum of all elements of the kernel will not be  
 149      exactly 1, but almost 1).  
 150

153      Another way is that of approximating the Gaus-  
 154      sian function, which is the probability density  
 155      function of a Gaussian distribution, with the prob-  
 156      ability density function of the Binomial distribu-  
 157      tion (in [4]): indeed according to Central Limit  
 158      Theorem this is a good approximation (to tell the  
 159      truth better as the dimension of the kernel grows).  
 160      Here we consider this approximation.  
 161

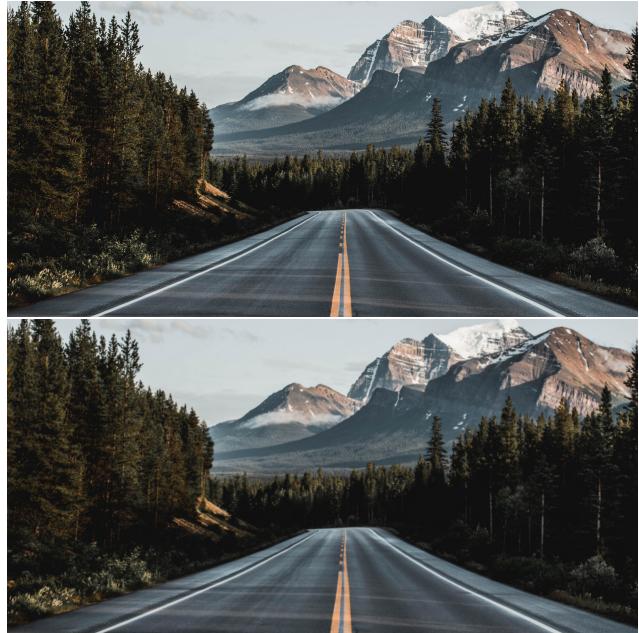


Figure 1. An image and its Gaussian blurred version (kernel dimension:  $25 \times 25$ )

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  $\omega$  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

162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215



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);
```

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

327 The three external cycles are ordered so that the  
328 pixels are processed in the order they are memo-  
329 rized. Evidently this structure (possibly with mi-  
330 nor changes and optimization) is particularly suit-  
331 able for a parallelization with OpenMP.  
332

333 This code is available in [this GitHub repo](#).  
334

#### 335 4.1. More details

337 For the problem of Kernel Image Processing,  
338 or at least for this implementation, there's not  
339 need to time profile an execution because the al-  
340 gorithm is very simple and it is trivial to identify  
341 the hotspot: not only because the five nested loops  
342 are particularly heavy-weight from a computation  
343 point of view, but also because basically it is the  
344 only computation to be done in order to solve the  
345 problem.  
346

347 So I can skip the time profiling phase and jump  
348 directly to tests. In the reality before testing I took  
349 some time for profile the memory usage with Val-  
350 grind Memcheck in order to eliminate all the re-  
351 maining memory leaks.  
352

### 355 5. Parallel Solution: OpenMP

357 One of the pros of OpenMP is that the sequen-  
358 tial and the parallel code can be unified, indeed  
359 very often the parallelization consist of only some  
360 directives, that obviously are ignored in the se-  
361 quential case. As I said in the previous section,  
362 the structure of this code was very suitable for  
363 OpenMP parallelization, indeed I only need a sin-  
364 gle directive:  
365

```
366 #pragma omp parallel for
```

368 Clearly the parallelization hits the outermost cy-  
369 cle.  
370

371 At this point I only needed to specify the vari-  
372 able sharing: input image, kernel and output im-  
373 age must be shared, indices and sum variables  
374 must be clearly private (we don't need to initial-  
375 ize them), and kernel radius firstprivate (because  
376 it has already been calculated).  
377

378 There are still two aspects to consider: the  
379 schedule and the number of threads. For the  
380 schedule, the most used option is the static one,  
381 so I will use this one (but then, in the tests I will  
382 validate this choice); about the number of threads  
383 we have to consider that this task is purely com-  
384 putational, so there should be a 1:1 ratio between  
385 threads and cores, as explained in [6]. I con-  
386 sider to put an extra thread to supply page fault  
387 events, but my test machines are equipped with  
388 Intel processors with Hyperthreading, so setting  
389 the number of threads equal to the number of vir-  
390 tual threads is sufficient to prevent page faults de-  
391 lays; anyway later we validate also this assump-  
392 tion with tests.  
393

394 This code is available in [this GitHub repo](#).  
395

#### 397 5.1. Further attempts

##### 398 5.1.1 Reduction

399 The two internal cycles basically perform a sum-  
400 mation: this is the case where a code can be par-  
401 allelized with a reduction pattern. First of all I  
402 serialized the double cycle and then I applied an  
403 OpenMP reduction.  
404

405 Anyway the performance was quite bad. De-  
406 spite I instantiated the thread pool once outer all  
407 the cycles, evidently launching a reduction for ev-  
408 ery pixel is not so convenient because of over-  
409 head.  
410

##### 412 5.1.2 Serialized cycle over pixels

414 Obviously parallelizing more than one of the out-  
415 ermost cycle doesn't bring advantages because  
416 the only effects we obtain is overhead. But I  
417 thought that serializing the three cycles and par-  
418 allelizing the single resulting cycle could have  
419 boost the performance.  
420

421 Anyway my prediction was wrong: with a  
422 static schedule the performance was a bit worse,  
423 with a dynamic schedule the situation gets bet-  
424 ter, but does not improve the performance ob-  
425 tained with nested cycles. Effectively with a static  
426 schedule I obtain the same partition of pixels over  
427 threads (at least in their number), but with a dy-  
428 namic schedule the partition is more balanced.  
429

namic 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  $w = \text{block\_width} + d - 1$  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

540 smaller mask to a maximum of 204 times for big-  
 541 ger mask (more details in 8.2).  
 542

543 One difference with respect to a common im-  
 544 plementation of tiling is that here mask size reach  
 545 uncommon value (25) and for high values the as-  
 546 sumption that each thread have to loads at most  
 547 two pixels doesn't hold anymore. So I don't have  
 548 implemented a classic two-batch loading, but a  
 549 multi-batch loading, with the number of batches  
 550 determined at runtime.  
 551

552 The edge handling is carried out at load time,  
 553 not at process time anymore.  
 554

555 The tile width is again 32, validated by test re-  
 556 sults.  
 557

558 A last note: again being the mask size  
 559 determined at runtime, I can't use a static  
 560 allocation for shared memory, so I resorted  
 561 to the dynamic one at kernel-launch moment:  
 562 `kernel<<<gridDim,blockDim,w*w>>>`.  
 563

### 564 6.3.1 Tile Width

565 Being completely changed the memory manage-  
 566 ment, it was not obvious that the previous optimal  
 567 block width will be the optimum also in this case.  
 568 I tested again the 8, 16, 24 and 32 size and indeed  
 569 16 turns out to be the optimum in this configura-  
 570 tion.  
 571

### 572 6.4. 4th step: pinned memory usage

573 As last step I have considered a more hidden  
 574 aspect. Let's take the case of the biggest image,  
 575  $8000 \times 6000$  pixels, each with 3 channels repre-  
 576 sented with a `char`: it is a total of 144MB of  
 577 memory of array. Not so much, but it starts to be  
 578 a considerable amount.  
 579

580 On a machine with reduced amount of memory  
 581 can be present a delay during the phase of copying  
 582 the data from the host to the device if the mem-  
 583 ory page containing that data has been swapped  
 584 to disk. To prevent this I used pinned memory for  
 585 storing the images data.  
 586

## 587 7. Tests and Results

588 The execution time obviously grows as image  
 589 and kernel dimensions grow. I have considered  
 590

Input	$T_S$	sequential	parallel			594
		C	C + OpenMP		$E_2$	
4K	$d = 7$	6.1s	2.7s	2.26	1.13	595
	$d = 13$	19.7s	8.5s	2.32	1.16	596
	$d = 19$	45.1s	17.7s	<b>2.55</b>	<b>1.28</b>	597
	$d = 25$	75.8s	31.8s	2.38	1.19	598
5K	$d = 7$	11.4s	5.0s	2.28	1.14	599
	$d = 13$	36.9s	15.9s	2.32	1.16	600
	$d = 19$	84.7s	33.2s	<b>2.55</b>	<b>1.28</b>	601
	$d = 25$	142.4s	59.8s	2.38	1.19	602
6K	$d = 7$	18.2s	8.0s	2.28	1.14	603
	$d = 13$	58.9s	25.5s	2.31	1.16	604
	$d = 19$	135.3s	53.1s	<b>2.55</b>	<b>1.28</b>	605
	$d = 25$	227.6s	95.5s	2.38	1.19	606
7K	$d = 7$	26.6s	11.7s	2.27	1.14	607
	$d = 13$	86.3s	37.2s	2.32	1.16	608
	$d = 19$	197.4s	77.5s	<b>2.55</b>	<b>1.28</b>	609
	$d = 25$	332.4s	139.0s	2.39	1.20	610
8K	$d = 7$	35.1s	16.0s	2.19	1.10	611
	$d = 13$	108.2s	51.0s	2.12	1.06	612

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

563 five different dimensions for input images:  $4000 \times$   
 564  $2000$ ,  $5000 \times 3000$ ,  $6000 \times 4000$ ,  $7000 \times 5000$   
 565 and  $8000 \times 6000$  for a total of, respectively, 8M,  
 566 15M, 24M, 35M and 48M of pixels. I tested every  
 567 dimension for different kernel dimensions: 7, 13,  
 568 19 and 25<sup>1</sup>.

569 The execution time does not depend on the spe-  
 570 cific input image because the number of opera-  
 571 tions is constant. However in order to avoid re-  
 572 sult's alterations due to casual fluctuations I run  
 573 the test (for every input) 3 times, seizing the op-  
 574 portunity to process different images. I have sum-  
 575 marized the results in Tables 1 and 2, where I re-  
 576 port mean execution time, speedup and efficiency.  
 577

578 A last note: the tests has been executed com-  
 579 piling the code in release mode; this aspect has  
 580 allowed to save around 33% of execution time.  
 581

582 <sup>1</sup>8K images tested only for littler kernel dimensions  
 583

Input	sequential C	parallel CUDA-C								
		naïve		constant		shared		constant+shared		
		$T_S$	$T$	$S$	$T$	$S$	$T$	$S$	$T$	
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

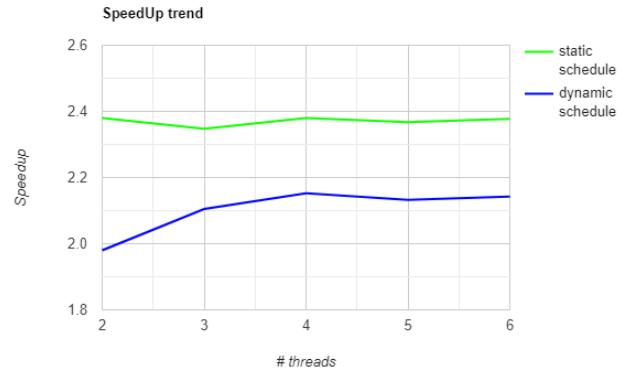


Figure 3. Speedup trend varying the number of threads

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.

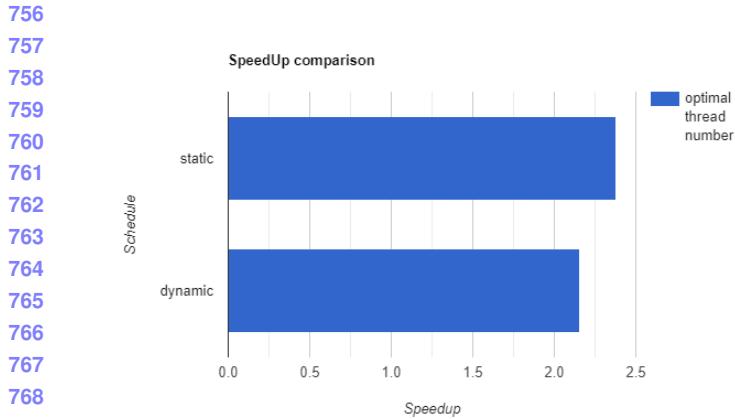


Figure 4. Speedup comparison varying schedule

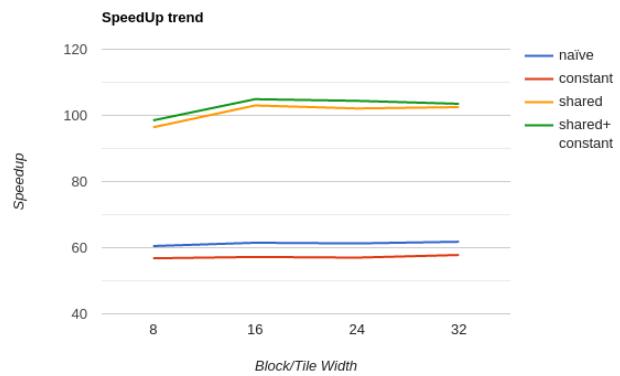


Figure 5. Speedup comparison varying block/tile width

## 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).

### 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.

#### 7.2.2 Pinned memory

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. Effectively profiling the CUDA execution with nvprof (results [here](#)) we can see that at GPU memory level there is a small improvement: the time passed on copying hot-to-device went from 275ms to 253ms (cumulatively for 6 calling to processing) and the throughput from a maximum of 1.56GB/s to a maximum of 1.59GB/s.

Evidently this very small improvement was not sufficient to supply at the greater instantiation time, but certainly pinned memory can do the difference in a system with limited memory amount, giving the chance to exploit at all the GPU power available

## References

- [1] *Kernel image processing*, Wikipedia. [Online]. Available: <https://en.wikipedia.org/wiki/>

810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863

[Kernel \\_ \(image \\_ processing\)](#) (visited on Oct. 15, 2020).

- [2] *Box blur*, Wikipedia. [Online]. Available: [https://en.wikipedia.org/wiki/Box\\_blur](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](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); 918  
919  
920  
921  
922
  - Ubuntu 20.10. 923  
924
  - NVIDIA Toolkit 11.2 925  
926