# GPU Assignment: Image processing

Stefano Sandonà

*Vrije Universiteit Amsterdam, Holland*

## 1 GPUs: NVIDIA GTX480

The aim of this assignment was to learn how to use many-core accelerators, GPUs in this particular case, to parallelize data-intensive code. All the implementations were written for the **NVIDIA GTX480**, using CUDA, a parallel computing platform and programming model invented by NVIDIA. Programming with CUDA, there is a straightforward mapping onto hardware, for this reason it is necessary to study the available HW before start developing an application. The architecture of the given accelerator is shown in Figure 1, its main characteristics and limits are shown in Table 1.


Figure 1: NVIDIA GTX480 Architecture

## 2 CImg

The image processing library used in this project was CImg, a small, modern and open-source toolkit developed for C++. CImg implements the RGB color model, an additive color model in which red, green, and blue light are added together in various ways to reproduce a broad array of colors. Each colored image of size *N*M* is composed by three parts (R,G,B) of the same size, so that *N*M*3* values are necessary to define an image. The Figure 2 shows an example of image composition.

| Microarchitecture | Fermi |
|---|---|
| Compute capability (version) | 2.0 |
| Maximum dimensionality of grid of thread blocks | 3 |
| Maximum x-dimension of a grid of thread blocks | 65535 |
| Maximum y-, or z-dimension of a grid of thread blocks | 65535 |
| Maximum dimensionality of thread block | 3 |
| Maximum x- or y-dimension of a block | 1024 |
| Maximum number of threads per block | 1024 |
| Cores per SM (warp size) | 32 |
| SM | 15 |
| Cores | 480 (32 * 15) |
| Maximum number of resident blocks per multiprocessor | 8 |
| Maximum number of resident warps per multiprocessor | 48 |
| Maximum number of resident threads per multiprocessor | 1536 (48 * 32) |
| Number of 32-bit registers per multiprocessor | 32K |
| Maximum amount of shared memory per multiprocessor | 48K |
| Theoretical Throughput | 1345 GFLOPS |
| Theoretical Bandwidth | 177.4 GB/s |

Table 1: NVIDIA GTX480 Specifications

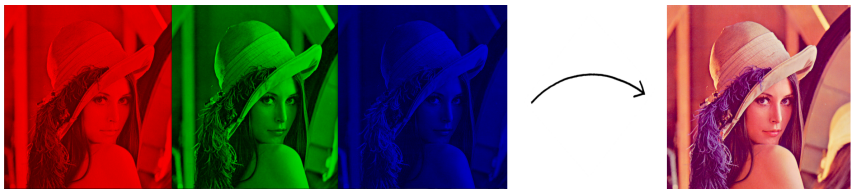


Figure 2: RGB model

# 3 The processing flow

Using CUDA there are two parts of the code: the device code, or GPU code, or the Kernel, that is a sequential program, write for one thread and execute for all and the HOST code, or CPU code, that is used to instantiate the grid, run the kernel, manage the memory. Figure 3 shows the processing flow of a CUDA application. In the particular case of image processing, everything starts from the CPU, that store the image from a file into a local buffer, allocates IN and OUT buffers on the GPU (*cudaMalloc*) and copy the image into the GPU's IN buffer (*cudaMemCpy*). After that, the CPU launches the GPU kernel with a defined grid configuration (*kernel_function «gridDim, blockDim»(params)*), that is executed by the GPU following the SIMT (Single Instruction, Multiple Threads) NVIDIA model. The threads are executed in parallel in each core, and they read the assigned part of IN data and generates the assigned part of OUT data. At the end, the results are copied out back to the CPU (*cudaMemCpy*) and the image is written to a file by the CPU.

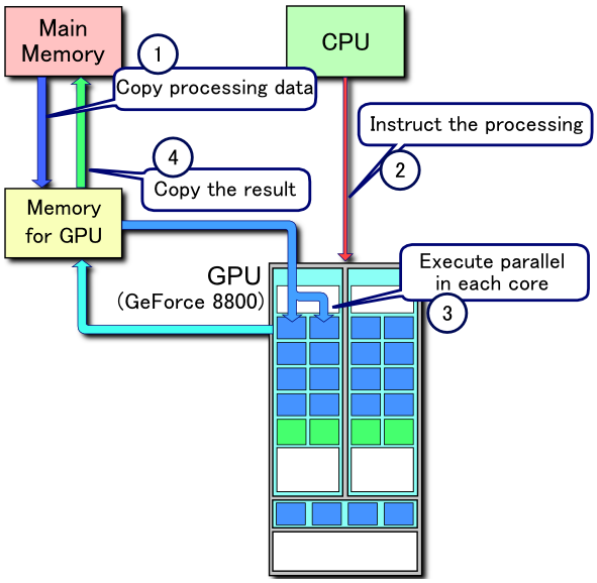
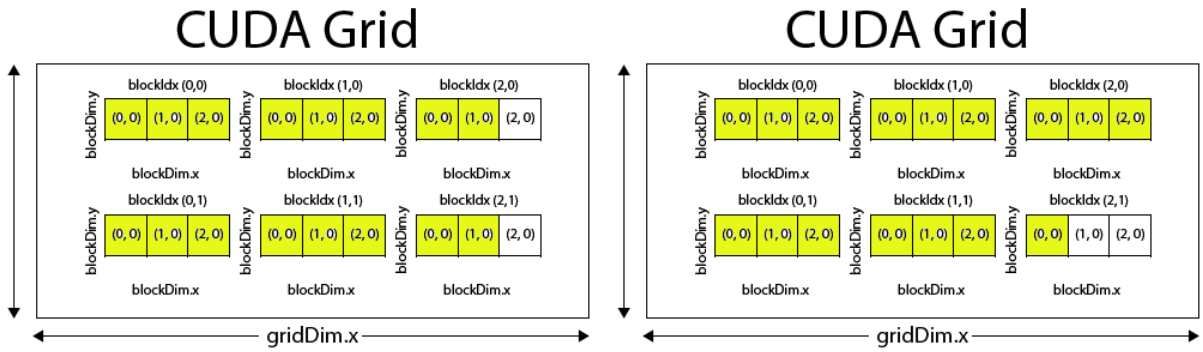



Figure 3: CUDA processing flow

# 4 CUDA grid configuration

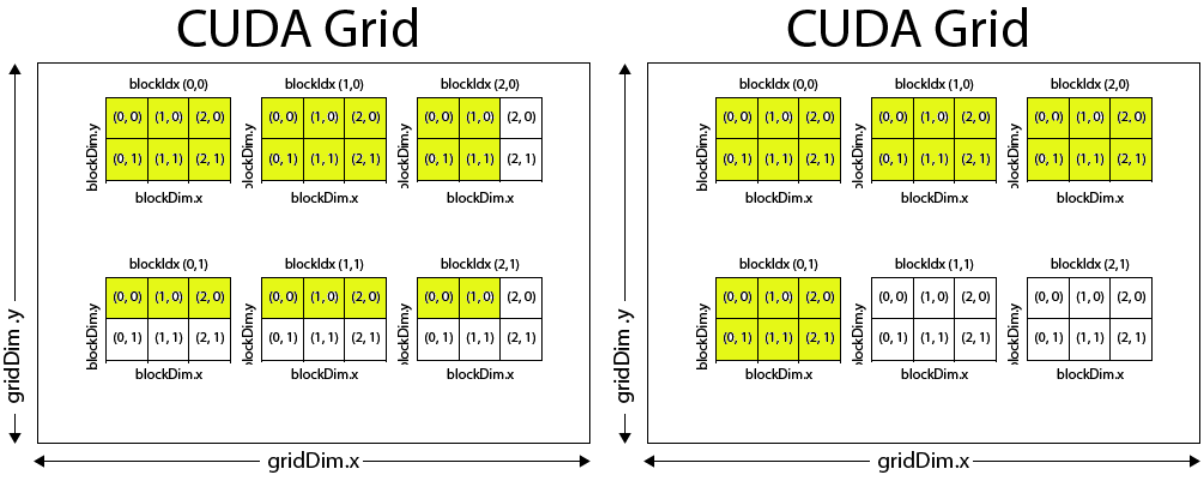
In CUDA, as mentioned before, there is a strinct mapping with the hardware, so that an hardware virtualization model is fixed with the concepts of thread, block and grid. Each **thread** executes the kernel code, running on one CUDA core. The threads are logically grouped into **thread blocks**, so that the threads of the same block will run on the same multiprocessor. The thread blocks are logically organized in a **Grid**, that represent the entire dataset. The blocks and the grid can be of 1D, 2D or 3D. The most important thing programming with GPUs, to make use of all their power, is to make them as busy as possible. Switching between concurrent warps has no overhead because registers and share memory are partitioned and not stored/restored, so that if one warp has to wait for example for a memory access and another warp is ready, there is a switch to hide the latency. The thread scheduling is really quick, and this allow the blocks to be swapped in and out really quickly. For this reason an high number of warps is needed and instantiating a grid, is good to have a number of blocks much bigger than the number of available multiprocessor and a block size that can be higher than the amount of cuda cores available per SM. Another interesting aspect to take into consideration, is the block size. The block is divided into warps, so that Threads 0..31 are part of Warp 0, Threads 32...63 are part of Warp 1 and so on. Considering this, it is good to have blocks with a size multiple of 32, so that no useless threads are launched. After certain tests, a block of size 256 revealed to be optimal. For what concerning the grid configuration, an image is a 2D structure and for this reason it is intuitive to set up also a 2D grid. By the way, setting up a 2D grid, there is not only one way to follow. For this particular project two possible block configurations were considered, 1D (1x256), 2D (32x8), 2D (32x16) and two possible grid configurations: dynamic and fixed. In the rest part of this document, **M1** indicates a dynamic grid of 1D blocks, in which the grid has a height of *image_ height*, a width of *(ceil(image_ width / 256))* and the threads on the right border if the image width is not a multiple of *block_ width* are idle (Figure 4a); **M2** indicates a dynamic grid of 1D blocks, in which the size of the grid is the same as M1, but the threads are consecutive and only the threads on the last block(s) are idle if the image's number of pixel is not a multiple of the size of the block (Figure 4b); **M3** indicates a dynamic grid of 2D blocks, in which the grid has a width of *ceil(image_ width / 16)*, a height of *ceil(image_ height / 16)* and the overflow is the same as M1 (Figure 5a); **M4** indicates a dynamic grid of 2D blocks, in which the size of the grid is the same as M3, but the oveflow is the same as M2 (Figure 5b).




(a) M1 - Right overflow

(b) M2 - Overflow at the end

Figure 4: 1D blocks, Kernel configurations, image of 16 pixels




(a) M3 - Right overflow

(b) M4 - overflow at the end

Figure 5: 2D blocks possible kernel configurations, image of 24 pixels

## 5 Coalesced memory access

One of the main bottlenecks with the GPUs are the global memory accesses that are expensive, for this reason it is better to maximize the use of bytes that travel from the DRAM to the Streaming Multiprocessor. CUDA uses a SIMT approach, in which all threads of a warp execute the same instruction, so that global memory accesses are effectuated "per warp". The threads in a warp (32) provide 32 addresses and the hardware converts these addresses into memory transactions. The memory is divided into regions of 128 bytes, so that bytes 0...127 are part of Region 0, bytes 128...255 are part of Region 1 and so on. The memory is accessed per region, that means if one thread wants the byte 0, the entire Region 0 is loaded. A kernel is correctly designed, if consecutive threads access consecutive memory addresses, so that all the requeted addresses fall on the same region and only one transaction is performed. Behaving in this "coalesced way" instead of having one access per thread, these accesses are grouped and the total memory overhead is reduced. Developing all the three algorithms this aspect was taken into consideration.

## 6 Algorithm 1: Grayscale Conversion and Darkening

From an RGB image, the output of this algorithm is a darker grayscale image. The gray value of a pixel is generated by weighting the three values (`0.3*R, 0.59*G, 0.11*B`) and then summing them together. To darken the obtained grayscale image, the final pixel value is multiplied by a constant (0.6). The Figure 6 shows an example of the result. The sequential algorithm, simply go through the entire image and computes for each pixel the corresponding value (Listing 1).

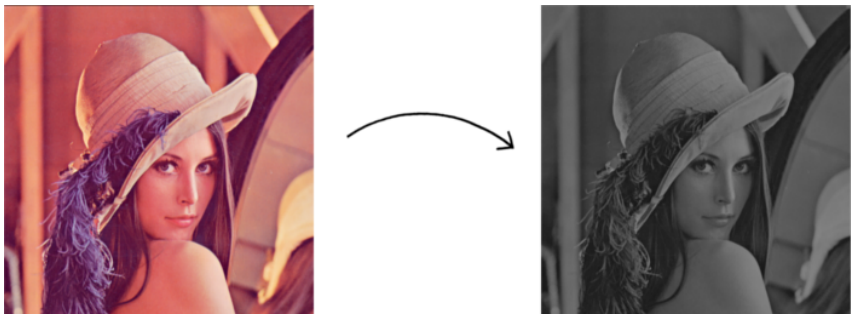


Figure 6: Grayscale Conversion and Darkening

```
//H=image_height, W=image_width
for ( int y = 0; y < H; y++ ) {
 for ( int x = 0; x < W; x++ ) {
  float grayPix = 0.0f;
  float r = static_cast< float >(inputImage[(y * W) + x]);
  float g = static_cast< float >(inputImage[(W * H) + (y * W) + x]);
  float b = static_cast< float >(inputImage[(2 * W * H) + (y * W) + x]);
  grayPix = ((0.3f * r) + (0.59f * g) + (0.11f * b));
  grayPix = (grayPix * 0.6f) + 0.5f;
  darkGrayImage[(y * W) + x] = static_cast< unsigned char >(grayPix);
 }
}
```

Listing 1: Darker Sequential code

## 6.1 Parallelization

## 6.2 One pixel per thread

After allocating into the GPU global memory some space to contain the input image (*3 * image_width * image_height * sizeof(unsigned char)*), copying the input image there and allocating some memory to content the output image (*image_width * image_height * sizeof(unsigned char)*), the kernel is ready to be launched. The GPU code is the same as the code content inside the loop of the sequential version (Listing 1), but instead of using the indexes of the loop to access the image pixels, it uses the index associated with the thread. In this first method, only one pixel is computed per thread. To exploit the coalesced memory access, two consecutive threads computes/accesses the values of two consecutive pixels. The 4 different grid configurations (explained in Section 4) were tested for this program, in order to establish the best. In Figure 7 are reported the obtained results. As shown methods M1 and M2 obtained the best results, so with unidimensional blocks the algorithm achieved better speedups. This is probably due to the fewest amount of operations to calculate the index of the pixel associated with each thread, because in this particular task there is no clear advantage on using 1D blocks instead of 2D blocks.

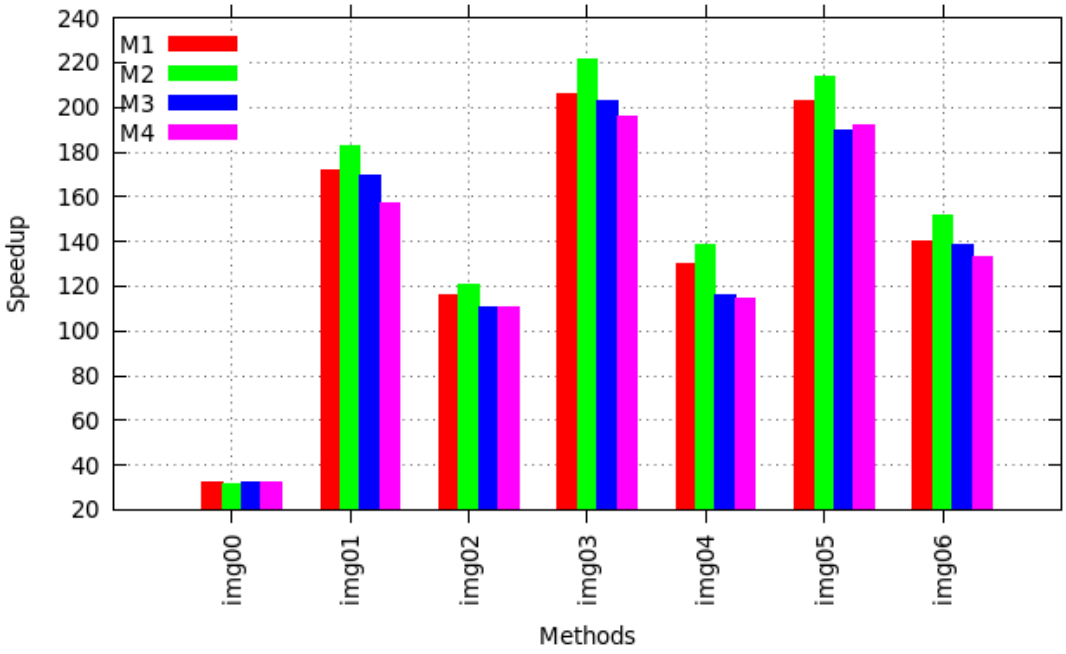



Figure 7: Speedups Comparison

### 6.3 Optimization - More pixels per thread

To exploit all the power of GPUs, as said, they have to be as busy as possible, because the warp scheduling is very fast and can hide the latency. Anyway, reducing the number of thread blocks by increasing the work per thread revealed to achieve better results. Lots of experiments were performed to establish the right number of pixels to compute by each thread, all taking into account the coalesced memory access. The grid was set up with a width of *ceil(width / 256)* and a height of *ceil(height / pixels_per_thread)* (unidimensional blocks). To maintain the coalescing, one thread computes the associated first pixel index with the formula shown in Listing 2 and then for the following pixels, it adds each time to this the total amount of pixels contained in the grid(*gridDim.x * blockDim.x * gridDim.y * blockDim.y*). The final code, is the one shown in Listing 3.

```
unsigned int pixel=((blockIdx.y * gridDim.x + blockIdx.x)
                    * blockDim.x) + threadIdx.x
```

Listing 2: Corresponding first pixel

```
for(i = ((blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x) + threadIdx.x;
    i < width * height;
    i += (gridDim.x * blockDim.x) * (gridDim.y * blockDim.y)) {
  float grayPix = 0.0f;
  float r = static_cast< float >(inputImage[i]);
  float g = static_cast< float >(inputImage[(width * height) + i]);
  float b = static_cast< float >(inputImage[(2 * width * height) + i]);
  grayPix = ((0.3f * r) + (0.59f * g) + (0.11f * b));
  grayPix = (grayPix * 0.6f) + 0.5f;
  outputDarkGrayImage[i] = static_cast< unsigned char >(grayPix);
}
```

Listing 3: Darker Parallel Code

Figure 8 represents the speedups achieved for different thread loads. After a certain number of pixels computed per thread, the performance of the program started to decrease, for this reason the idea to use a grid with a fixed size, instead of a dynamic size was not taken into consideration. Following this approach, with a number of 10 pixels per thread, the program obtained often the best results, so this is a good thread load to help the scheduler. This load was sufficient to both hide the latency and reduce

the warp scheduling overhead. Figure 9 shows the comparison between the best results obtained with a single pixel per thread and the results obtained with the new version. It is clear that the new version is an optimization, in fact for each image the speedup increased. This solution is the one adopted. In Table 2 and 3 are shown the detailed execution times and speedups achieved. Table 4 exposes the Achieved Throughput and Bandwidth. Table 5 exposes the differences obtained comparing the output of the sequential algorithm with output of the parallel algorithm. These differences could derive from a difference in floating point precision between the CPU and GPU, or from different ordering of floating point operations. In Section 9 is reported a detailed analysis using the NVIDIA Virtual Profiler.

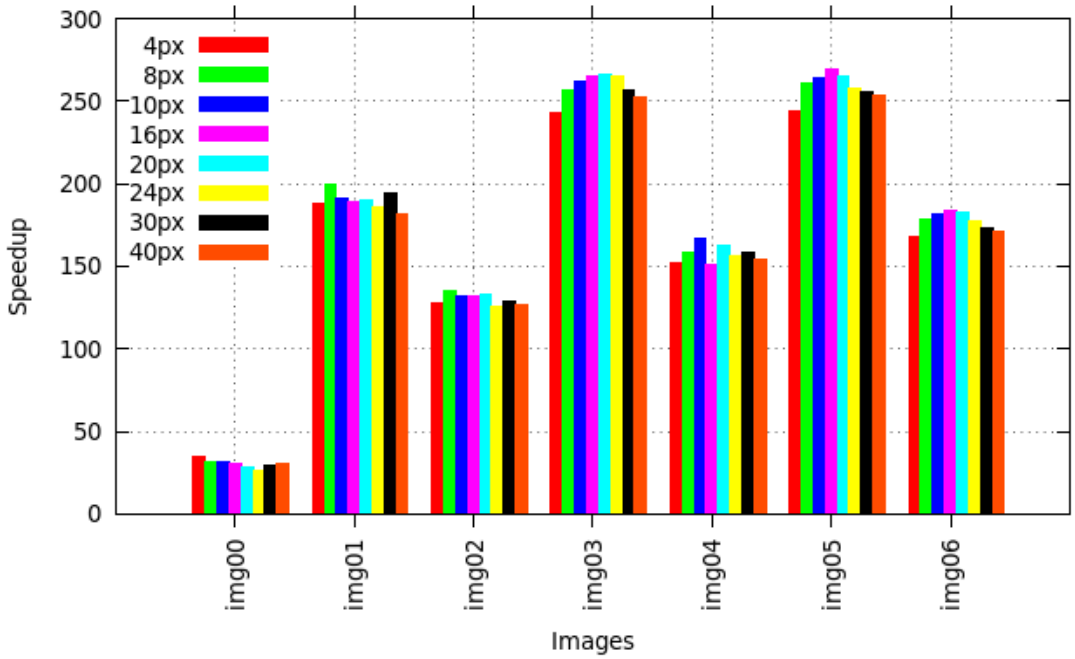



Figure 8: Speedup with more different number of pixels per thread

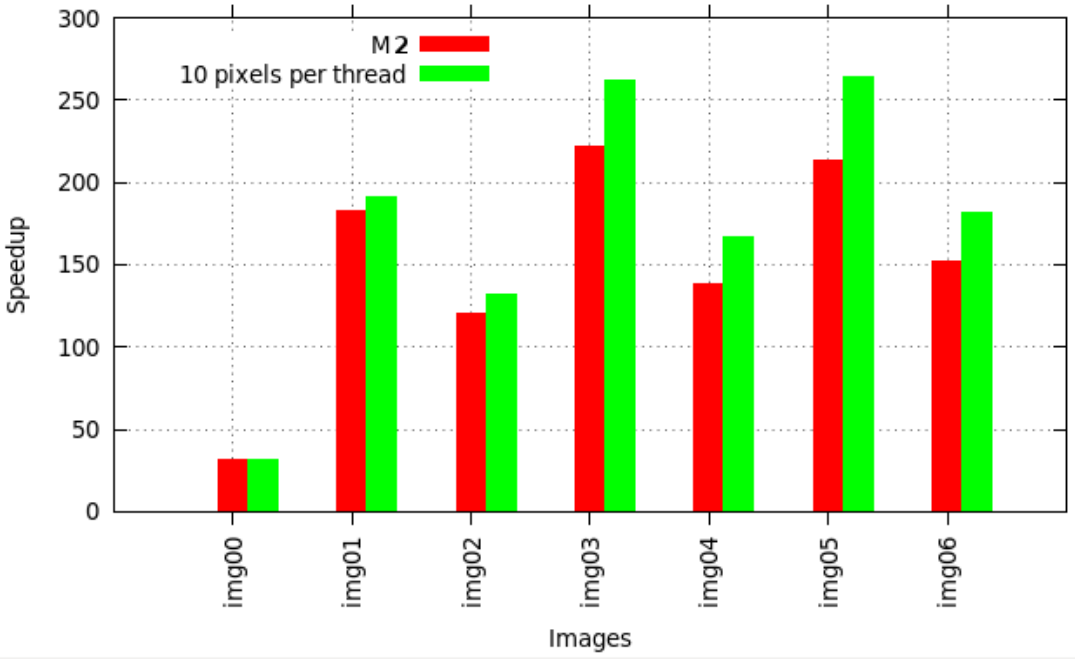



Figure 9: Solutions comparison

| | Sequential | M2 | 10 pixels per thread |
|---|---|---|---|
| **img00** | 0.002416 | 0.000076 | 0,000076 |
| **img01** | 0.036925 | 0.000202 | 0,000193 |
| **img02** | 0.033726 | 0.000279 | 0,000255 |
| **img03** | 0.340821 | 0.001537 | 0,001299 |
| **img04** | 0.151429 | 0.001091 | 0,000907 |
| **img05** | 0.445976 | 0.002090 | 0,001692 |
| **img06** | 0.588814 | 0.003880 | 0,003239 |

Table 2: Execution Times

| | M2 | 10 pixels per thread |
|---|---|---|
| **img00** | 31.79 | 31.79 |
| **img01** | 182.80 | 191.32 |
| **img02** | 120.88 | 132.26 |
| **img03** | 221.74 | 262.37 |
| **img04** | 138.80 | 166.96 |
| **img05** | 213.39 | 263.58 |
| **img06** | 151.76 | 181.79 |

Table 3: Speedups

| | Ach. Throughput (GFLOPS/s) | Ach. Bandwidth (GB/s) | Compute utilization (%) | Bandwidth utilization (%) |
|---|---|---|---|---|
| **Img00** | 24,14 | 13,80 | 1,80 | 7,78 |
| **Img01** | 95,08 | 54,33 | 7,07 | 30,63 |
| **Img02** | 101,20 | 57,83 | 7,52 | 32,60 |
| **Img03** | 136,02 | 77,72 | 10,11 | 43,81 |
| **Img04** | 129,48 | 73,99 | 9,63 | 41,71 |
| **Img05** | 137,11 | 78,35 | 10,19 | 44,16 |
| **Img06** | 141,63 | 80,93 | 10,53 | 45,62 |

Table 4: Achieved Throughput and Bandwidth

| | img00 | img01 | img02 | img03 | img04 | img05 | img06 |
|---|---|---|---|---|---|---|---|
| **Pixels above threshold** | 111 | 231 | 135 | 735 | 1503 | 9 | 4206 |

Table 5: Differences between the Sequential and the Parallel output

# 7 Algorithm 2: Histogram Computation

From an RGB image, the output of this algorithm is a grayscale image with the relative histogram of 256 possible values of gray. The histogram measures how often a value of gray is used in an image. The sequential algorithm, simply go through the entire image, computing for each pixel the corresponding gray value and incrementing the corresponding counter (Listing 4). The Figure 10 shows an example of the result.

```
//H=image_height , W=image_width
for ( int y = 0; y < H; y++ ) {
 for ( int x = 0; x < W; x++ ) {
  float grayPix = 0.0f;
  float r = static_cast< float >(inputImage[(y * W) + x]);
  float g = static_cast< float >(inputImage[(W * H) + (y * W) + x]);
  float b = static_cast< float >(inputImage[(2 * W * H) + (y * W) + x]);
  grayPix = ((0.3f * r) + (0.59f * g) + (0.11f * b)) + 0.5f;
```

```
    grayImage[(y * W) + x] = static_cast< unsigned char >(grayPix);
    histogram[static_cast< unsigned int >(grayPix)] += 1;
  }
}
```

Listing 4: Sequential code

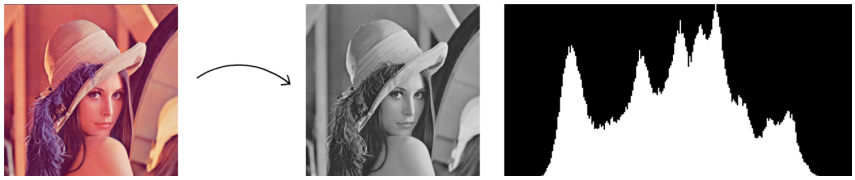


Figure 10: Histogram Computation

## 7.1 Parallelization

## 7.2 One pixel per thread

After copying the input image and the initial empty histogram into the GPU global memory and allocating some memory to content the output image, the kernel is ready to be launched. As for the previous algorithm, the GPU code is almost the same as the code content inside the loop of the sequential version (Figure 4), but instead of using the indexes of the loop to access the image pixels, it uses the index associated with the thread. Also in this case, to exploit the coalesced memory access, two consecutive threads computes/accesses the values of two consecutive pixels. The interesting aspect of computing histograms in parallel, is that it is possible that two threads read at the same time the same value of gray for a pixel, so that at the same time they try to increment the corresponding histogram bin. For a correct execution, to avoid wrong updates, the increment has to be an atomic operation, so that only one thread at the time will modify the bin value. The logic behind an atomic operation is that each thread "locks" the variable, modify it, and "unlocks" it, so that the other threads that try to modify the same variable has to wait that this will be "unlocked". The worst case, in the histogram scenario, is a monochrome image (only one color), in which all the threads try to modify the same bin, which implies a long waiting queue of threads. The general idea to have high performance, is to reduce the number of conflicts at the minimum. Also in this algorithm, the speedup is not related to the shape of the block, in fact the performance of the algorithm using the atomics depends on the image. Dealing with an image with horizontal stripes of different colors of 1 pixel, for example, with a 1D block (1x256) there will be all the 256 threads that try to increment the same histogram bin. In the same scenario, using a 2D block (8x32) there will be only 32 threads that try to increment the same histogram value. If the image instead, is composed of vertical stripes of 32 pixels wide, it happens the opposite, the 1D approach will obtain more performance. For these reasons, for the initial tests, to remove the fortuity, the given images were substituted with images of the same sizes but composed by only one color (monochrome). Different implementations were tested. The first approach consisted of using only the histogram array stored in the global memory and modifying it with atomic add operations (Figure 11). The second approach consisted of using an additional histogram array stored in shared memory, performing the atomic add operations there and at the end, assigning a certain bin to a thread and reducing the shared histograms in the global one (Figure 12). Doing this, not all the launched threads will compete for the same bin, but only the threads in the same block during the computation and the threads to which is assigned the same bin during the reduction phase. Figure 13 shows the obtained results for the two approaches. From this image it is clear that the second approach drastically increased the speedups and that the block configuration (1D or 2D) doesn't affect the performance.

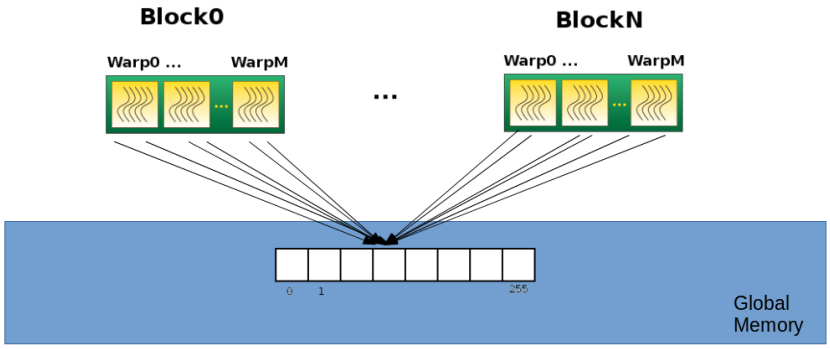



Figure 11: Global histogram - Concurrency

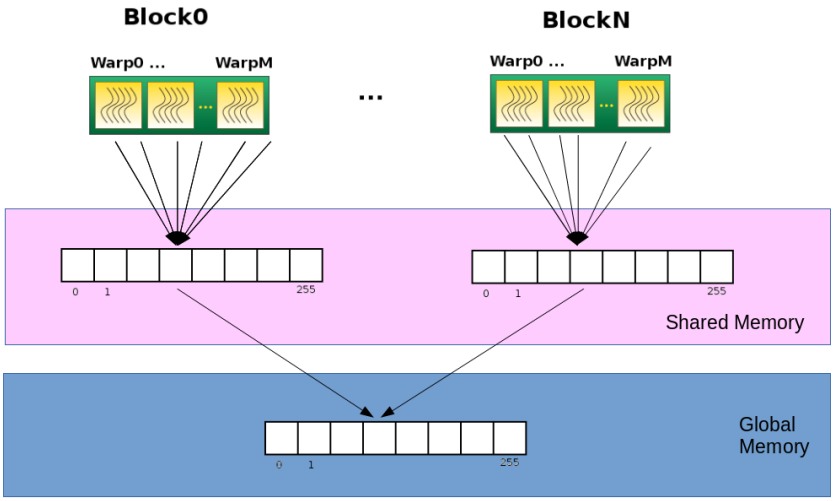



Figure 12: Shared histogram - Concurrency

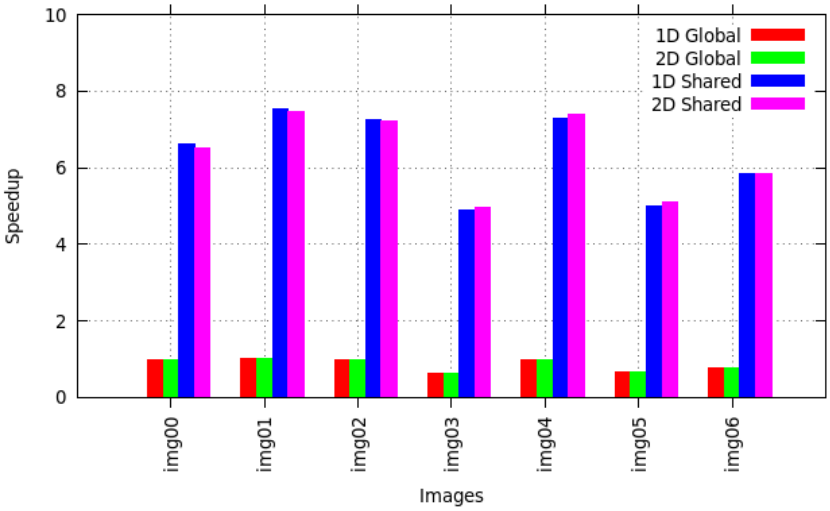



Figure 13: Global Atomics vs Shared Atomics (Monochrome images)

An additional "privatization" of the histogram could be performed. Instead of having one histogram per block, it is possible to assign one histogram per warp (Figure 14), so that the concurrency at the end is only among the 32 warp threads during the computation and between the threads to which is assigned the same bin during the reducing phase. Figure 15 shows the performance of this last method compared to the previous. The number of concurrent histogram modifications was reduced, so that better speedups

were achieved.

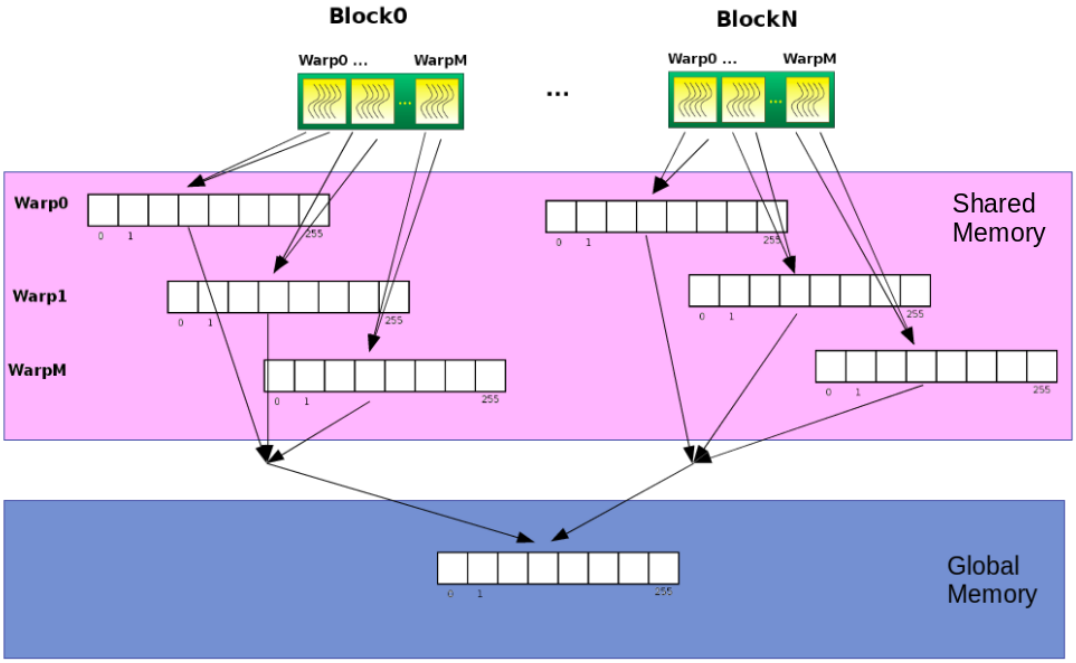



Figure 14: Shared Warps histogram - Concurrency

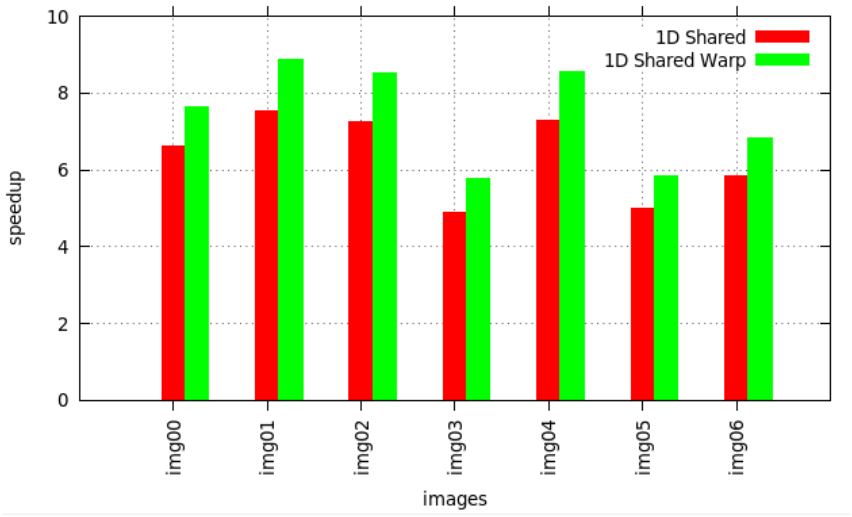



Figure 15: Shared Atomics with Warps vs Shared Atomics (Monochrome images)

After a testing phase with monochrome images, the tests were performed with the real given images. Figure 16 represents the obtained results. As it is possible to see, in this case the warp solution is much worse than the shared version, that means for standard images, the time spent to reduce the single warp histograms is bigger than the time spent to compute the atomic operations.

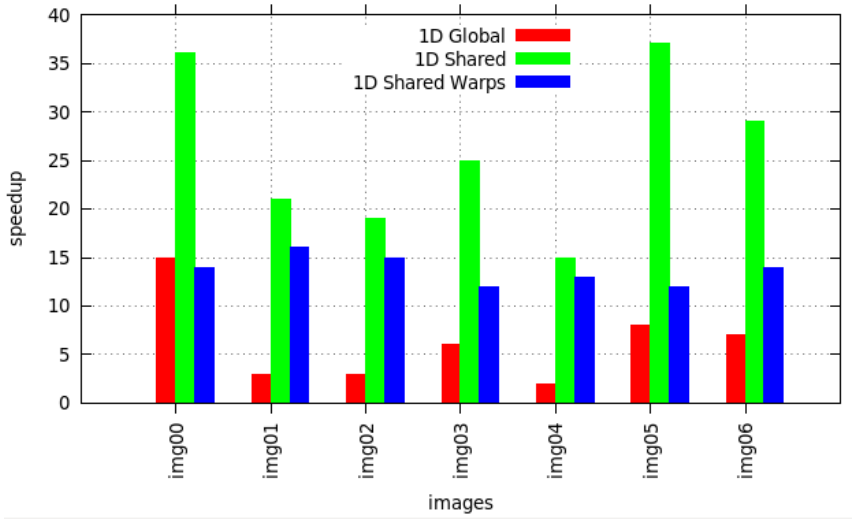



Figure 16: Global Atomics vs Shared Atomics vs Shared Atomics with Warps

### 7.3 Optimization - More pixels per thread

As for the previous algorithm, some additional tests were performed assigning the computation of more pixels per thread. The last two approaches (Shared Atomics and Shared Atomics with Warps) were improved to test this solution and both on the real images obtained better results. This is due to a reduction of the scheduling load and on the reduction of global histogram reductions. Different work loads were assigned to each thread and measured. A dynamic grid and fixed grid configurations were tested and Figures 17, 18, 19 and 20 show the obtained results. Figure 21 compares the best obtained configurations for the two approaches. As shown, the second approach (Shared Atomics with Warps), differently than the one pixel per thread version, obtains for almost all the images the best results. This result could surprise, but means that the decrement on the total number of reduction operations is critical for this approach. For the small images (e.g. Img00) the time spent for the final reduction remains high in comparison with the total computation due to the low amount of pixels to compute. The final adopted solution is the Shared Atomics with Warps with 30pxs computed per threads, because it is good also for small images (not a big amount of useless launched blocks) and the difference with the 60x45 grid version is not so evident. Tables 6 and 7 show the detailed obtained results. Table 8 shows the differences obtained comparing the output of the sequential algorithm with output of the parallel algorithm.

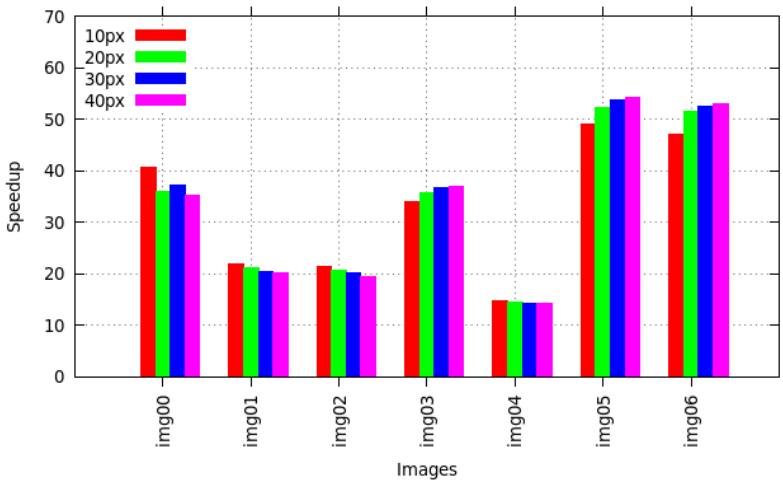



Figure 17: Shared Atomics, 1D blocks - Different number of assigned pixels per thread

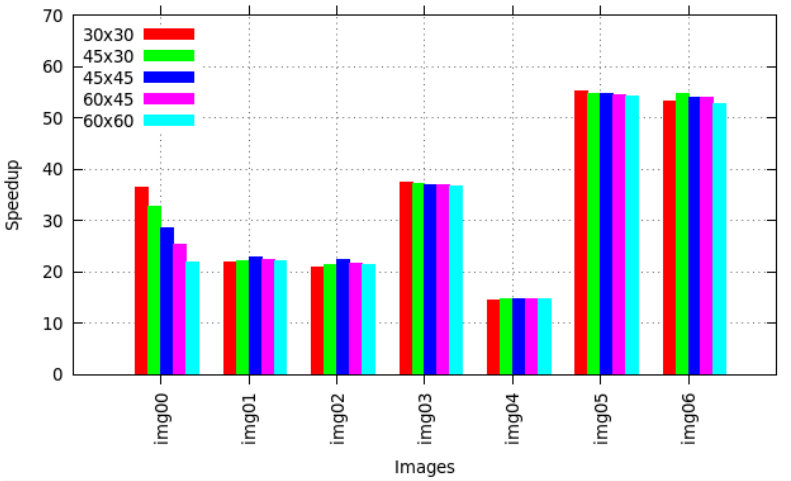



Figure 18: Shared Atomics, 1D blocks - Fixed grid

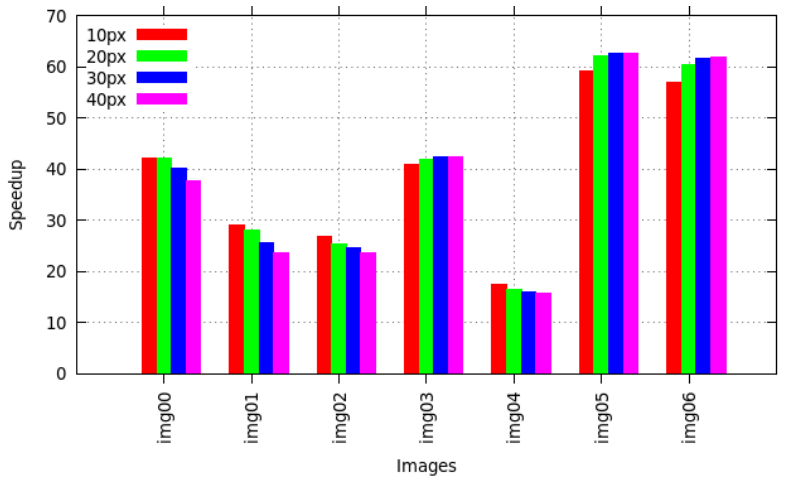



Figure 19: Shared Atomics with Warps, 1D blocks - Different number of assigned pixels per thread

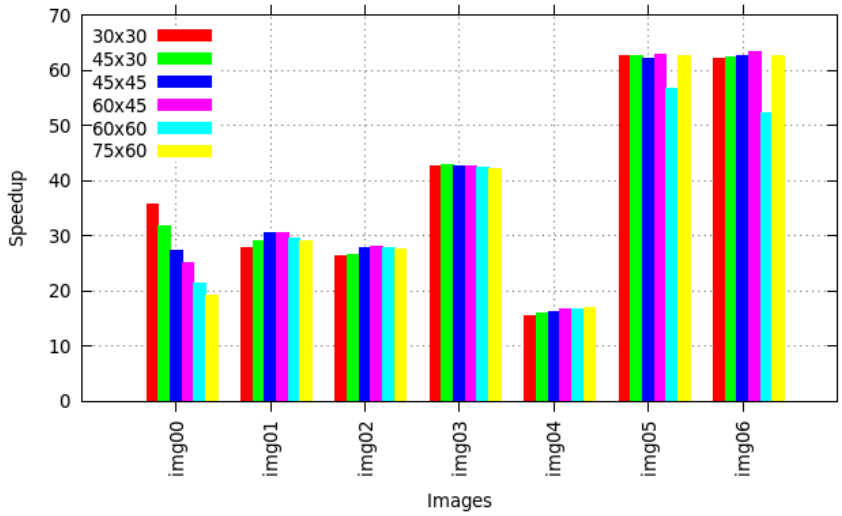



Figure 20: Shared Atomics with Warps, 1D blocks - Fixed grid

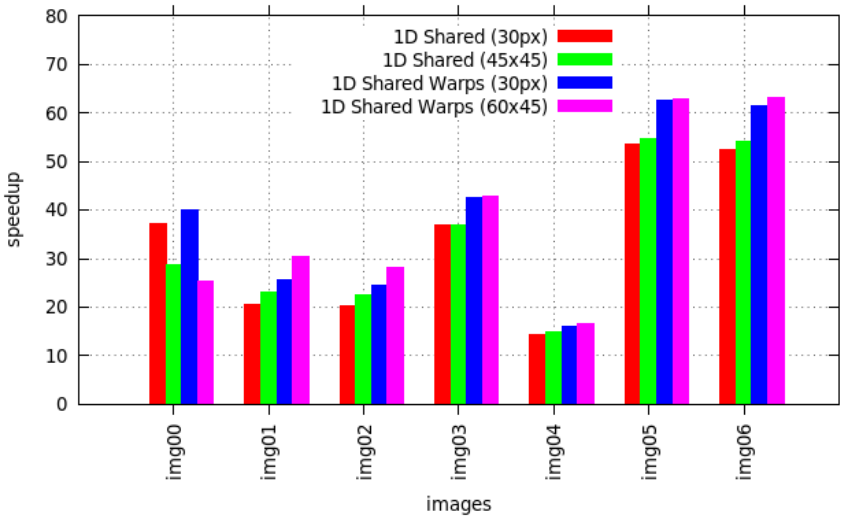



Figure 21: Best solutions comparison

| | **Sequential** | **30px** | **65x40** |
|---|---|---|---|
| **Img00** | 0,003208 | 0,000080 | 0,000127 |
| **Img01** | 0,032379 | 0,001264 | 0,001062 |
| **Img02** | 0,043784 | 0,001778 | 0,001553 |
| **Img03** | 0,200681 | 0,004724 | 0,004698 |
| **Img04** | 0,197973 | 0,012325 | 0,011871 |
| **Img05** | 0,267590 | 0,004278 | 0,004260 |
| **Img06** | 0,616924 | 0,010026 | 0,009756 |

Table 6: Execution Times

| | **30px** | **65x40** |
|---|---|---|
| **Img00** | 40.10 | 25.26 |
| **Img01** | 25.62 | 30.49 |
| **Img02** | 24.63 | 28.19 |
| **Img03** | 42.48 | 42.72 |
| **Img04** | 16.06 | 16.68 |
| **Img05** | 62.55 | 62.81 |
| **Img06** | 61.53 | 63.24 |

Table 7: Speedups

| | **img00** | **img01** | **img02** | **img03** | **img04** | **img05** | **img06** |
|---|---|---|---|---|---|---|---|
| **Pixels above threshold** | 324 | 12 | 0 | 24 | 0 | 12 | 48 |

Table 8: Differences between the Sequential and the Parallel output

# 8 Algorithm 3: Smoothing

Smoothing is the process of removing noise from an image by the means of statistical analysis. To remove the noise, each point is replaced by a weighted average of its neighbours. In this way small-scale structures are removed from the image. In this case a two-dimensional 5-point triangular smooth filter was used. The Figure 22 shows an example of the result.

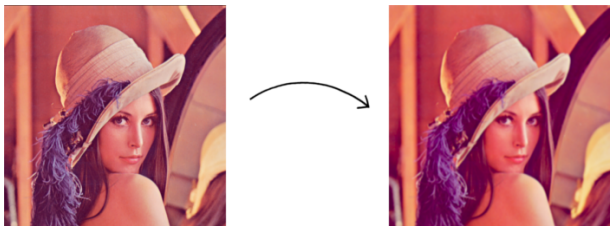
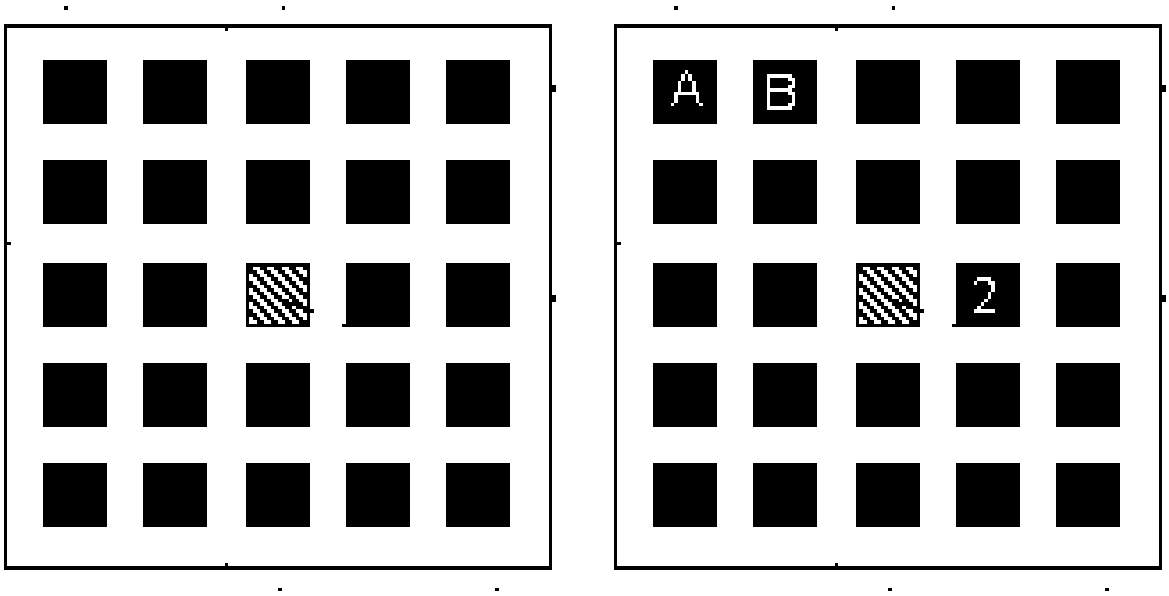


Figure 22: Smoothing

## 8.1 Parallelization

This particular algorithm deals with square areas of the image (filter), so that using 2D blocks, the threads can efficiently share memory and prevent a lot of global memory accesses.




(a) No square grid with 1D blocks

(b) Square grid with 1D blocks

Figure 23: Possible kernel configurations

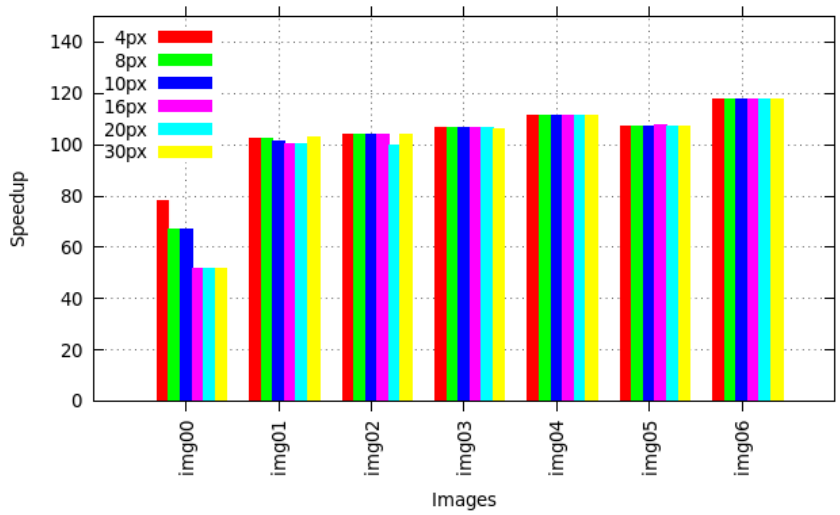



Figure 24: More pixels per thread

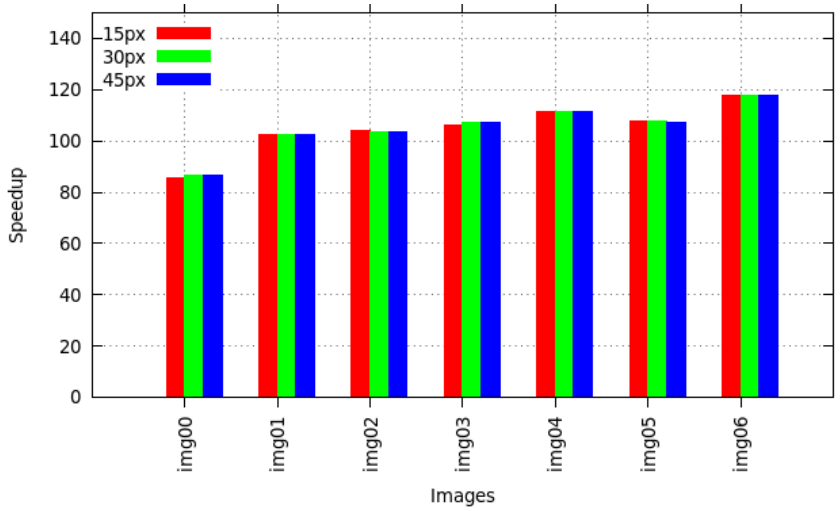



Figure 25: Fixed Height

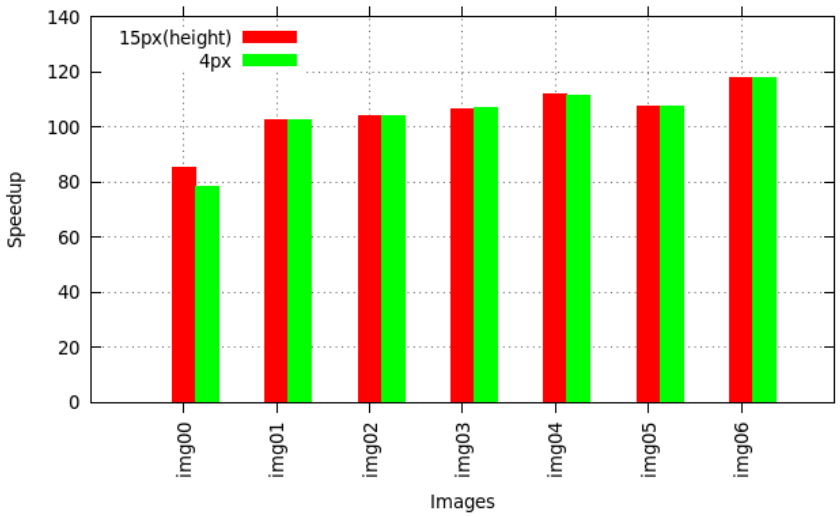



Figure 26: Solutions comparison

| | img00 | img01 | img02 | img03 | img04 | img05 | img06 |
|---|---|---|---|---|---|---|---|
| **Pixels above threshold** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 9: Differences between the Sequential and the Parallel output

# 9 Algorithms analysis with the Visual Profiler

Instruction-level parallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. The potential overlap among instructions is called instruction level parallelism.

## 9.1 Application 1

After collecting the profile of the applications using **nvprof**, the output files were evaluated using the **Nvidia Visual Profiler**. The choosen configuration, uses unidimensional *Blocks* of 256 threads, 12 *Registers* and 0 *Shared Memory*. For all the images the profiler found no issues for *Divergent Execution* (threads that follow different if branches) and a *Warp Execution Efficiency* (ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor) of 100%. This last value comes from the fact that is used a block of 256 threads, that is a multiple of 32 (warp size), so no useless threads are launched and from the fact that there is no thread divergent execution, so the threads within a warp can execute in a SIMD way, avoiding inactive threads within the warp. Examples of occupancy results are shown in Figure, all the the images, obtained an occupancy over 91% except the first that obtained 82.5%. This is probably correlated to its small size, but in any case the occupancy is not limiting the performance of the application. For all the images exept the 5th, the Profiler found no issues related to Global Memory Access Pattern. The 5th image, is the only one of the given set that has an amount of pixels that is not a multiple of 128, the memory in the device is allocated with a 128-byte line granularity, so probably, the addresses fall within 2 cache lines, so 2 transaction insteads of one and so a decrease of the memory bandwidth utilization (Figure )

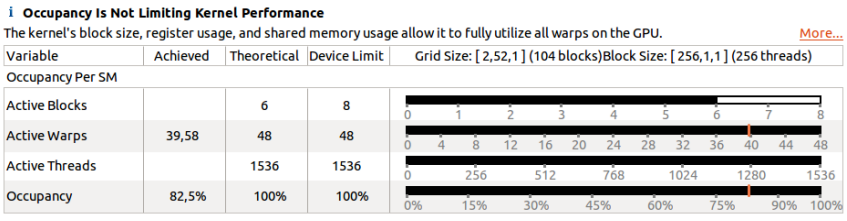



Figure 27: Img00

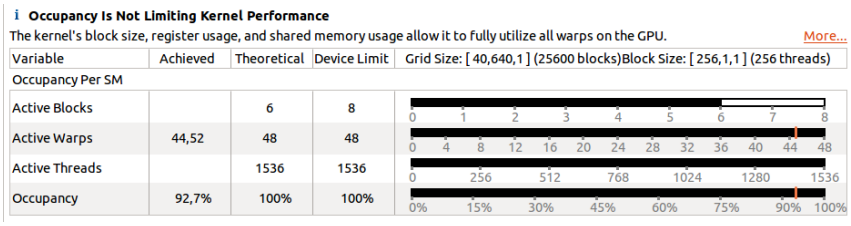



Figure 28: Img05

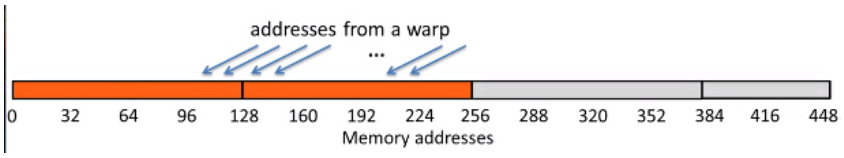



Figure 29: Img00

```
float g = static_cast< float >(inputImage[(width * height) + (y * width) + x]);
float b = static_cast< float >(inputImage[(2 * width * height) + (y * width) + x]);
```

Listing 5: Unaligned accesses

HISTO -> Atomic implementation in CUDA may vary by GPU architecture. On the GTX 480 (a Fermi-class GPU), __shared__ memory atomics are implemented not as a single machine instruction, but in fact by a sequence of machine (SASS) instructions that form a loop.

This loop is essentially contending for a lock. When the lock is acquired by a particular thread, that thread will then complete the requested memory operation atomically on the identified shared memory cell, and then release the lock.

The process of looping to acquire the lock necessarily involves branch divergence. The possibility for branch divergence in this case is not evident from the C/C++ source code, but will be evident if you inspect the SASS code.

Global atomics are generally implemented as a single (ATOM or RED) SASS instruction. However global atomics may still involve serialization of access if executed by multiple threads in the warp.

For this reason profiler outlined branch divergence

can't avoid banc conflics because the accesses are random for us, they depend on the image

On image number 5 => no coalesced access for the 2 instructions.. because image pixels non a multiple of 128, so non aligned (threadID+pizelsOfImage) but not effect the computation, compiling with -Xptxas -dlcm=cg worst results, Changing the configuration (cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);) 2)

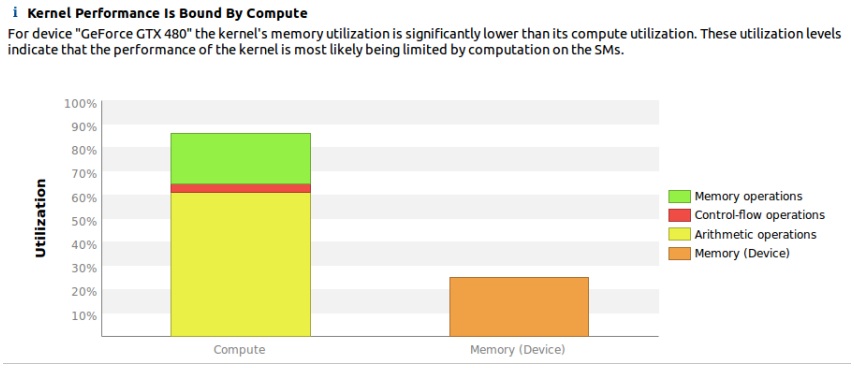



Figure 30: Img05

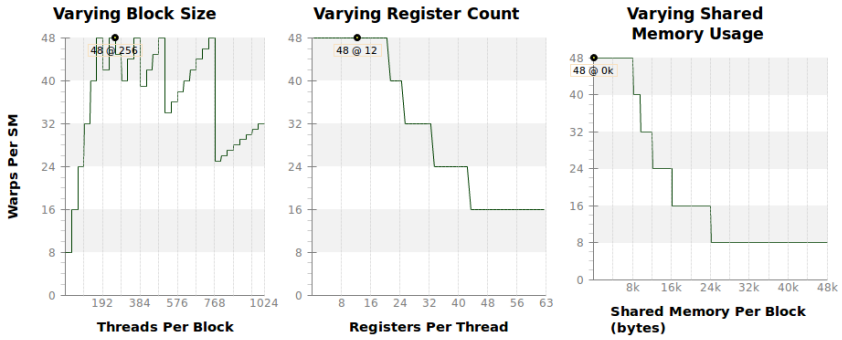



Figure 31: Img05

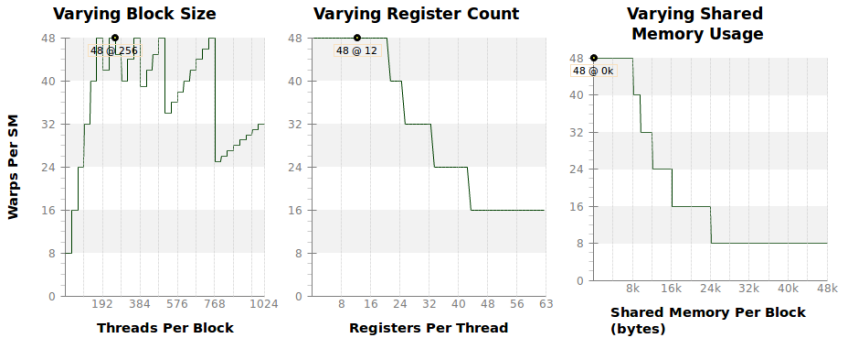



Figure 32: Img05

https://developer.nvidia.com/cuda-zone https://devblogs.nvidia.com/parallelforall/ http://www.gputechconf.com/