PC-2022/23 Parallel Adaptive Histogram Equalization

Panconi Christian
E-mail address
christian.panconi@stud.unifi.it
panconi.christian@gmail.com

Abstract

Image histogram equalization (HE) is a popular image processing technique employed to enhance the contrast of digital images by remapping the pixel values (or their luminance component). The mapping function depends on the image histogram, used as a discrete approximation of the pixel values probability density function.
Ordinary HE does not produce qualitatively good results on images which contains regions with different overall brightness because the method uses the same mapping function for all the pixels.
The adaptive variant (Adaptive Histogram Equalization, AHE) overcomes this limitation by using a different function for each pixel, which depends on the pixel neighborhood only, but is significantly more computationally expensive than ordinary HE.
This report examines some parallelization methods for AHE on CPUs using OpenMP and GPUs using CUDA.

1. HE and AHE

A single-channel, bi-dimensional image can be represented as a discrete function $I(x, y)$ where $x \in \{0, \dots, W-1\}$ and $y \in \{0, \dots, H-1\}$ are the spatial coordinates, and the function can assume L different discrete values $I(x, y) \in \{0, \dots, L-1\}$.
Multi-channel images can be represented by a tuple of functions, for example in the case of RGB images: $\big(I^R(x, y)\ ,\ I^G(x, y)\ ,\ I^B(x, y) \big)$.
The channel depth is the number of bits required to represent the values in $\{0, \dots, L-1\}$, usually we deal with images channels with depth 8 (bits) where $L = 256$.
The histogram of an image channel is a function

$$h : \{0, \dots, L-1\} \longrightarrow \mathbb{N}$$

which counts the number of occurrences of each value in the channel:

$$h(l) = \big|\{(x, y) : I(x, y) = l\}\big|\ ,\ l \in \{0, \dots, L-1\}$$

1.1. Brief Histogram Equalization derivation

A greyscale image whose pixels tend to assume the entire range of possible channel values, and additionally the values are distributed uniformly, will have an high contrast appearance and a large variety of gray tones.
Histogram equalization is a point operator which uses a transformation function for the pixel values to produce a channel where all the values are equiprobable.
Consider a continuous function of one variable $r \in [0, 1]$, where r is the normalized gray level, $r = 0$ represents the black and $r = 1$ the white.
The transformation function

$$s = T(r)$$

must satisfy:

(a) single valued and monotonically increasing in $[0, 1]$

(b) $0 \leq T \leq 1$ for $0 \leq r \leq 1$

The gray levels can be considered as random variables in $[0, 1]$, with $\mathbf{P}_r(r)$ and $\mathbf{P}_s(s)$ representing the PDFs of r and s. From a probability result we have that if $\mathbf{P}_r(r)$ and $T(r)$ are known and $T^{-1}(r)$ satisfies condition *(a)*, then the pdf of s can be obtained as:

$$\mathbf{P}_s(s) = \mathbf{P}_r(r)\Big|\frac{dr}{ds}\Big|$$

If we choose the cumulative distribution function of r as T:

$$s = T(r) = \int_0^r \mathbf{P}_r(w)\, dw$$

we have:

$$\frac{ds}{dr} = \frac{dT(r)}{dr} = \mathbf{P}_r(r)$$

thus:

$$\mathbf{P}_s(s) = \mathbf{P}_r(r)\Big|\frac{dr}{ds}\Big| = \mathbf{P}_r(r)\Big|\frac{1}{\mathbf{P}_r(r)}\Big| = 1$$

showing that independently of the form of $\mathbf{P}_r(r)$, $\mathbf{P}_s(s)$ is always uniform.

Digital images have discrete values, so the probability of occurrences is used and it can be obtained from the histogram. The transformation function is:

$$s_k = T(r_k) = \sum_{j=0}^{k} \mathbf{P}_r(r_k) = \sum_{j=0}^{k} \frac{h(r_k)}{n}$$

where r_k is the k-th gray level, $h(r_k)$ is the value of the histogram for r_k and $n = W \cdot H$ is the total number of pixels . s_k is the transformed value in $[0, 1]$ that must be re-scaled to a value in $\{0, \ \dots \ , L-1\}$

In the RGB case, the image is first converted to the YCbCr color space and the equalization is performed only on the Y (luminance) channel.

1.2. AHE

Ordinary histogram equalization uses the same transformation function for all the pixels, and this function depends on the histogram of the whole image. This causes the contrast to not be sufficiently enhanced in areas which are brighter or darker than the rest of the image.

The *Adaptive Histogram Equalization* improves the enhancement by using a different transformation function for each pixel, which is calculated from the histogram of a square region around the pixel, of size $s_w \times s_w$. The derivation is the same as the ordinary HE.

The pixels near the borders must be handled by providing additional pixels to fill the entire window, this is done by mirroring the rows and colums near the borders, producing an image of size $(W + 2\lfloor \frac{s_w}{2} \rfloor) \times (H + 2\lfloor \frac{s_w}{2} \rfloor)$.

2. Algorithms

The *Algorithm 1* shows the global histogram equalization. Its complexity is $O(W \cdot H)$ since it loops over all the pixels twice: one for the histogram calculation and one for assigning the equalized pixel values.

AHE can be obtained by using a window centered in each pixel to compute the new pixel value. AHE's complexity in its basic form is $O(W \cdot H \cdot s_w^2)$, which is undesirable, especially on large images or when using large windows.

To reduce the number of operations performed, AHE can be implemented using a window which slides over the image. The histogram for a given pixel can be calculated incrementally using the one from the previous iteration, by adding the histogram of the next row or column (depending on the sliding dimension) to be included in the window and subtracting the histogram of the row/column to be removed.

Algorithm 2 shows the Sliding Window AHE. The histogram on the entire window is computed only once per row/column, reducing the complexity to $O\big(W \cdot (s_w^2 + H \cdot s_w)\big)$.

Algorithm 1: Histogram Equalization

Data: I , n , L
Result: the equalized image/channel I^{eq}

$h =$ histogram of I;
$cdf\_val = 0$;
$lut = \{0\}^L$; /* Holds transformed values */
for $l = 0;\ l < L;\ l = l + 1$ **do**
  $cdf\_val = cdf\_val + h(l)$;
  $lut(l) = cdf\_val \cdot \frac{1}{n} \cdot (L-1)$;
end
for *each* (x, y) **do**
  $I^{eq}(x, y) = lut(I(x, y))$;
end

A further improvement can be obtained by sliding the window along both image dimensions, where the "primary" dimension alternates the direction at each iteration over the other dimension. For example, if y is the primary dimension, in the first iteration the window will slide down along the first column, calculating the histogram incrementally by adding the histogram of the last row of the current window and subtracting the histogram of the first row of the previous window. In the second iteration the window will slide up along the second image column, updating the histogram according to the direction change. When the window starts a new column, the histogram is updated by subtracting the histogram of the first column of the previous window and adding the histogram of the last column of the current region.

The complexity with this implementation, referred as "bi-directional", becomes $O\big((s_w^2 + H \cdot s_w) + (W-1) \cdot (s_w + H \cdot s_w)\big)$, which is only a minor improvement over the mono-directional one and depending on the practical implementation can result in no improvement at all, but it has showed some benefits for the parallelization on GPUs.

Algorithm 2: Sliding window AHE

Data: I , W , H , L , s_w
Result: the equalized image/channel I^{eq}

for $i = 0;\ i < W;\ i = i + 1$ **do**
  $h =$ histogram of the window centered in the first pixel of the column i;
  for $j = 0;\ j < H;\ j = j + 1$ **do**
    Incremental histogram update on h;
    Accumulate the cdf_val up to the bin $I(j, i)$ of h;
    $I^{eq}(j, i) = cdf\_val \cdot \frac{1}{s_w^2} \cdot (L-1)$;
  end
end

3. Parallelization and implementation details

The algorithms have been implemented in C++ using GCC (10.3.0) as compiler along with NVCC (CUDA version 11.7).

3.1. Parallelization on CPU

The computations for each pixel in the *Algorithm 2* are independent from each other, the resulting value for a pixel depends only on the original values in the $s_w \times s_w$ neighborhood around it.
In terms of loop vectorization this algorithm suffers from an access pattern problem during the histogram computation, because for each neighboring pixel in the window, the histogram bin to increment depends on the pixel value, which results in an almost random access pattern.
The only vectorizable part is the accumulation of the CDF at line 5 of the *Algorithm 2*, which consists in a summation reduction, but this provides little to no improvement.
A parallel version of this algorithm has been obtained by parallelizing the outer for loop using *OpenMP*.
Each thread processes a subset of adjacent columns of the image, for each pixel they read the window from the original image buffer and write the equalized value to the output buffer into non-overlapping chunks. Thus the output buffer can be kept shared by the thread pool and the access to it doesn't need any synchronization. The buffers for the windows histograms, and CDF values are instead private to each thread.

3.2. Parallelization on GPU

A CUDA implementation for the AHE algorithm cannot be obtained easily as the OpenMP one due to the different execution model.
As specified in section 4 the GPU used for the evaluation has compute capability 6.1, the hardware limits used in this section to describe the GPU implementation will mostly refer to such compute capability version.
Pseudocode in algorithm 3 summarize the kernel operations, without claim of completeness, for the Bi-directional SWAHE implementation.
The "Collaboratively" keyword in the algorithm 3 statements has to be intended as the usual block-level collaboration between threads (based on threads ids and block ids).
Further details on the CUDA implementation are outlined in the following subsections (3.2.1 - 3.2.4).

3.2.1 Blocks organization

The thread blocks size cannot match the window size, as it may be tempting to do at first, because even for $s_w = 63$ the resulting block will be composed by $63 \cdot 63 = 3969$ threads and the physical limit for block size is 1024[1], making this approach not suitable to be applied for large windows.
The block size is fixed to a value which maximizes the multiprocessors occupancy for the target compute capabilities. For example, for compute capability 6.1 we have a maximum of 2048 threads and 32 blocks per SM, using blocks of $2048/32 = 64$ threads maximizes the SMs occupancy.
Every thread block operates on a different region of the image using a sliding window to compute the histograms.
Unlike the OpenMP implementation, where each thread has a private histogram buffer, the buffer is shared at block level and different threads within a block can potentially update the same histogram bin concurrently, thus the histogram updates (increments and decrements) must be performed atomically.

Algorithm 3: Kernel for Sliding window AHE

Data: I , W , H , L , s_w
Result: the equalized image/channel.

```
1  Compute the histogram on the whole window
     centered in the first pixel of the region of I
     assigned to this block (leave out last column);
2  for each column i in the assigned region of I do
3      Collaboratively add the histogram of the next
         window's last column;
4      for each row j in the assigned region of I do
5          Collaboratively add the histogram of the
             next window's last row;
6          Perform parallel scan to accumulate the
             CDF on the histogram;
7          Only first thread of the block computes the
             equalized value for the pixel (j,i);
8          Collaboratively subtract the histogram of the
             current window's first row;
9      end
10     Collaboratively subtract the histogram of the
         current window's first column;
11 end
```

3.2.2 Shared memory

The need for atomic updates introduces the problem of latency in the atomic increment and decrement operations. If multiple (possibly several) threads within a block updates the same grey-level bin, the latency of the atomic updates will be high, because the threads will have to wait for the memory to be available.
To improve the latency, the histograms buffers are kept in

[1] This limit has remained unchanged up to the last compute capability version as the time of this writing, which is 8.7

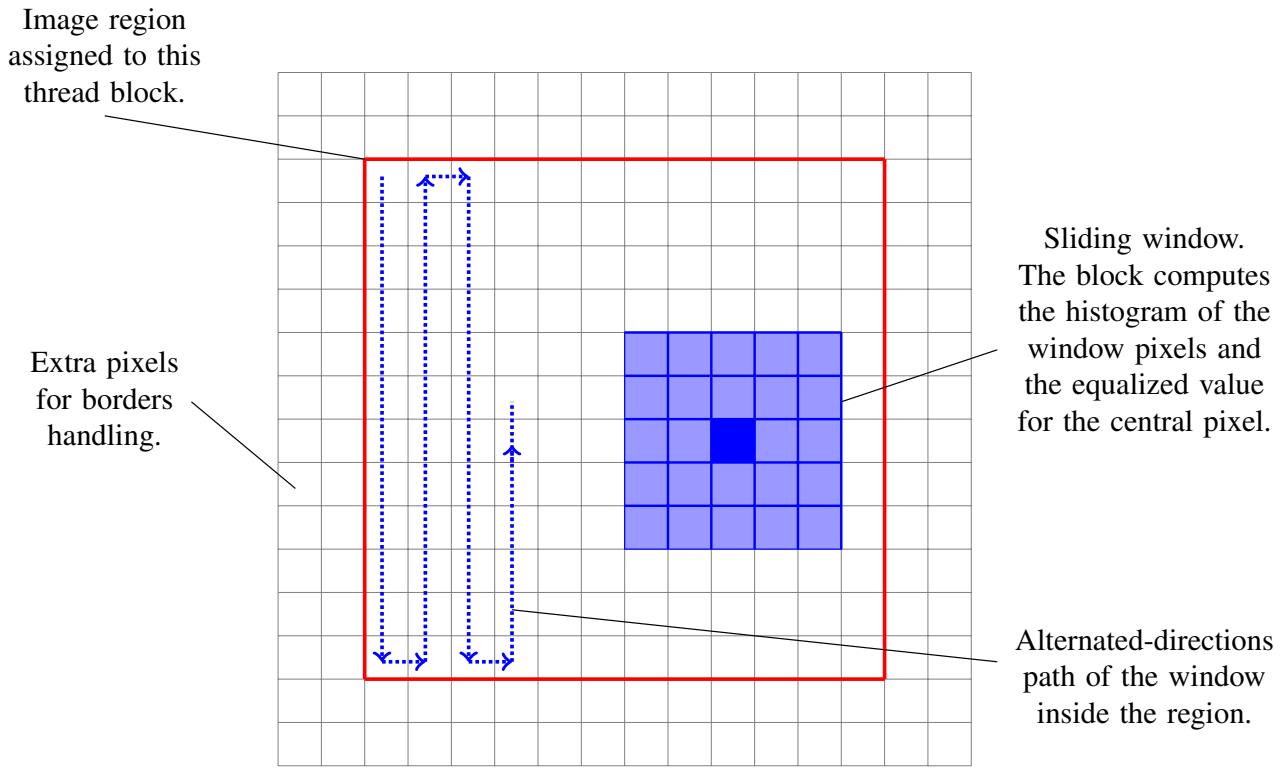

Figure 1: Figure illustrating the bi-directional kernel operations. Each thread block processes a different region of the image (delimited from the red border). At each iteration the thread blocks compute the equalized value for the central pixel in the window and slide the window following the dotted blue path.

shared memory instead of global memory[2].
For images with 256 grey levels (8-bit channels), using a 4-byte unsigned integer for every level occurrences count, the histogram buffer requires 1 Kb of shared memory.
CDF accumulation (algorithm 3 line 6) is implemented with a parallel scan in order to avoid divergence, this requires a double buffer, thus an additional Kb of shared memory. Furthermore the histogram of the row to subtract after the CDF accumulation (algorithm 3 line 8) is also kept in shared memory to speed up the access to it.
The total shared memory required by the kernel is 3 Kb, which is a reasonable amount according to the HW limits for recent compute capabilities. For compute capability 6.1 the maximum shared memory per SM is 96 Kb, which results in a maximum of $\frac{96\text{ Kb}}{3\text{ Kb}} = 32$ blocks per SM.

3.2.3 Mono-directional vs Bi-directional

As mentioned in section 2, a bi-directional implementation has advantages over the mono-directional one in this case, because it reduces the bank conflicts in the shared memory accesses.
Specifically, in the Bi-directional implementation, the histogram on the whole window is computed only once and, even when the thread block starts to compute the equalized values of an adjacent image column, the number of updates on the histogram is always $2s_w$. The Mono-directional version performs s_w^2 operations when the block starts from the topmost pixel of the next image column, potentially generating more bank conflicts during the accesses to the histogram buffer.

3.2.4 Registers

Combining all the mentioned techniques for the parallelization on GPUs leads to an implementation which utilizes a considerable number of registers.
An NVVP profiling run showed that the Bi-directional implementation uses 55 registers/thread which results in $55 \cdot 64 = 3520$ registers/block. Referring to compute capabilities 6.1, the maximum number of registers/SM is 65536, limiting the number of blocks/SM to $65536/3520 \approx 18$ (only 56% occupancy).
In order to achieve the maximum possible occupancy the number of registers/thread has been limited using

[2]Native shared memory atomics had been introduced with Maxwell and Pascal architectures on 32-bit integers. Previous architectures implements shared memory atomics in software using locks.

	Avg ms/Mp			
Window size	Sequential	OpenMP	CUDA Mono	CUDA Bi
31	138.4	**13.5**	25.0	24.9
63	259.4	**21.5**	26.3	25.9
127	460.7	40.1	30.5	**27.0**
255	982.1	79.2	45.8	**30.6**
511	1937.7	158.9	103.9	**41.2**

Table 1: Average ms/Mp varying window size. These values are the average over the 5 sample sizes: 1Mp, 2Mp, 4Mp, 8Mp, 16Mp for each window size.

the `__launch_bounds__` NVCC directive to produce an executable which uses only 32 registers/thread, thus $32 \cdot 64 = 2048$ registers/block, which allows to schedule $65536/2048 = 32$ blocks/SM.
This is possible because the CUDA computational model provides a dedicated memory type (called *local memory* in the CUDA terminology) to offload the registers and temporarily save their content. This reduces number of registers used per threads with the drawback of increasing the instructions latency because data need to be fetched from (or transferred to) the local memory. This mechanism is known as *register spilling* and allows to trade between occupancy and latency when the performances are limited from the number of registers.
In this particular case, allowing some register spilling has the benefit of raising the SMs occupancy without introducing too much latency. Profiling the version with a limited number of register has shown a 30% memory traffic increase caused by the spilling, but SMs occupancy raises over 90%.

4. Experiments

The implementations have been evaluated using samples of different size obtained from large images.
In order to generate these samples without varying too much the image content, the test images have been produced by cropping the central 4000x4000 (16Mp) region of the original images and applying a downscale, without interpolation, to obtain the smaller samples with size: 8Mp, 4Mp, 2Mp, 1Mp.
The window size parameter s_w has been varied using five different values: 31, 63, 127, 255 and 511.
The following hardware has been used for the evaluations:

- AMD® Ryzen 7 1700 (8 core/16 threads, Q1 2017)
- NVidia® GeForce GTX 1060 (compute capability 6.1, Pascal arch., Q3 2016)

The OpenMP implementation has been evaluated using 16 threads.
Every benchmark is composed by ten runs and has been repeated ten times, for a total of 100 runs. The reported execution time is the mean wall clock time of all the runs.

Table 1 reports the average execution time per mega-pixel (ms/Mp) of the three implementations, using the five different window sizes. The CPU-based implementations show a relation between the s_w parameter and the processing time required per mega-pixel which is almost linear (doubling the window size roughly doubles the time to process a Mp).
The CUDA implementations start to perform similarly to the OpenMP for s_w around 63, suggesting that for very small windows the overhead of a GPU-based implementation dominates the time required to perform the

	Window size: 31		Window size: 63		Window size: 127		Window size: 255		Window size: 511	
Img size	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
1 Mp	**11.0**	5.3	**12.2**	8.6	10.6	**14.6**	10.9	**26.8**	11.2	**42.1**
2 Mp	**10.5**	5.0	**12.7**	9.8	12.4	**16.3**	13.8	**34.0**	13.4	**45.1**
4 Mp	**11.1**	5.2	**13.2**	10.1	12.5	**16.8**	13.4	**32.0**	12.2	**44.7**
8 Mp	**9.1**	6.0	**11.3**	10.9	10.7	**18.5**	12.0	**34.1**	13.0	**54.7**
16 Mp	**9.9**	6.4	**11.2**	10.9	11.4	**19.2**	11.9	**33.8**	12.3	**53.6**

Table 2: Speedup with respect to the sequential implementation of both OpenMP and CUDA (bi-directional) implementations.

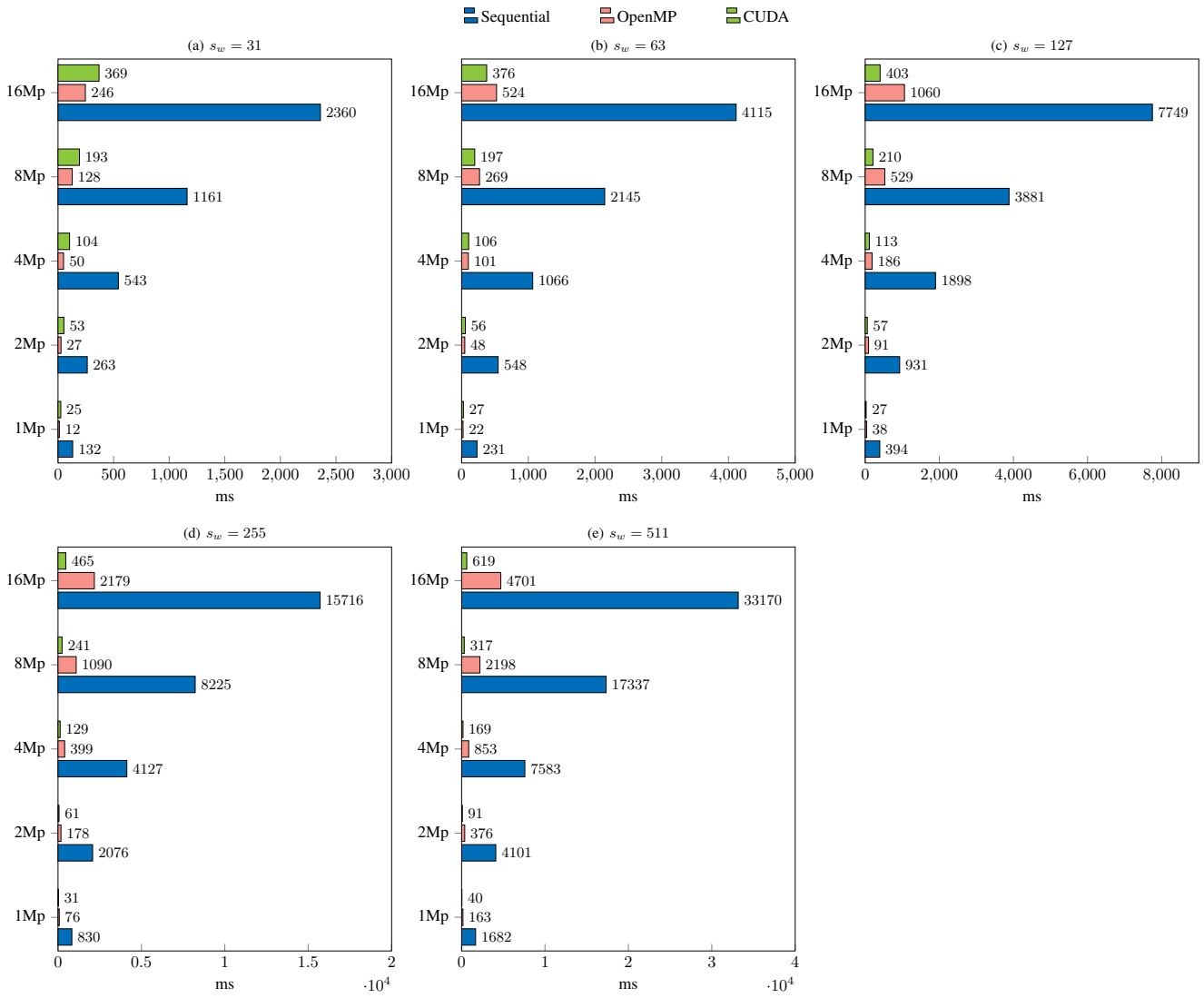

Figure 2: Mean execution time comparison of the three implementations: sequential, OpenMP parallel, CUDA(bi-directional) parallel.

equalization. On the other hand, for the bigger values of s_w, the GPU implementation is dramatically faster.

The average execution times for all the image and window sizes are reported in *Figure* 2, while *Table* 2 compares the speedups obtained by the OpenMP and CUDA (bi-directional) implementations with respect to the sequential version.
The speedup gained by the OpenMP implementation ranges between 10 and $\approx$ 14 and, overall, it does not seem related to the image or window size. Speedup relative to the GPU-based version shows a different distribution as it increases as the window or image size increases. In particular, the higher speedups have been obtained when $s_w = 511$ and, in general, the GPU implementation seems more sensitive to a variation of the window size than a change in the image size.
Figure 3 compares the execution times for the mono-directional and bi-directional CUDA implementations.
Up to $s_w = 127$ the two versions have almost the same average execution time with all image samples, with the bidirectional version being slightly faster on the bigger images, indicating that the effect buffer contention and bank conflicts does not impact too much when the window is not too big. For windows of 255 and 511 pixels, the mono-directional version shows a performance decrease with respect to the bi-directional version which is from 1.5 to 2.5 times faster.

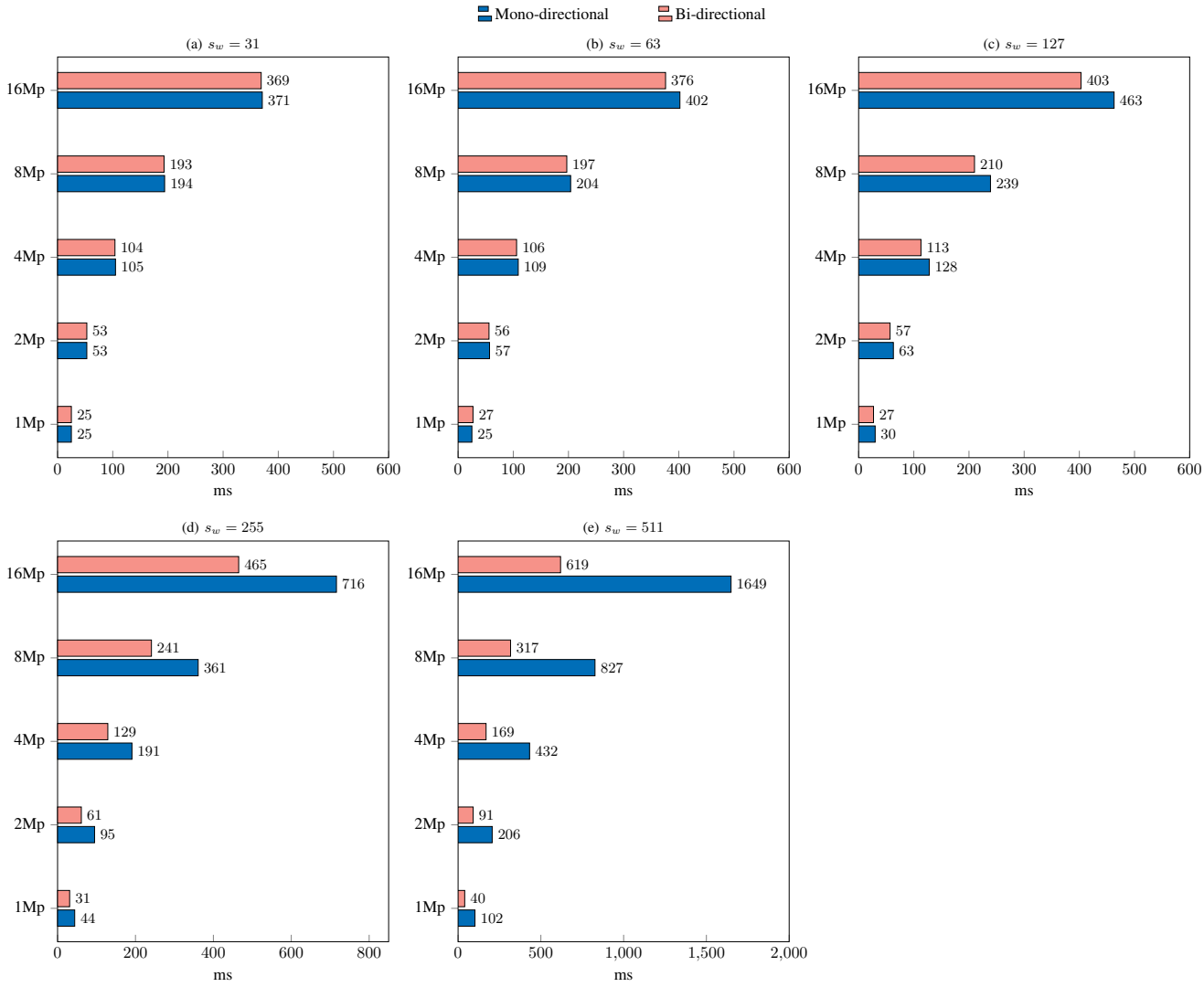

Figure 3: Average execution times comparison between mono-directional and bi-directional CUDA implementations.