J Real-Time Image Proc
DOI 10.1007/s11554-016-0619-6

ORIGINAL RESEARCH PAPER

Acceleration techniques and evaluation on multi-core CPU, GPU and FPGA for image processing and super-resolution

Georgios Georgis[1] · **George Lentaris**[1] · **Dionysios Reisis**[1]

Received: 24 December 2015 / Accepted: 5 July 2016

Abstract Super-resolution (SR) techniques constitute a key element in image applications, which need high-resolution reconstruction, while in the worst case, only a single low-resolution observation is available. SR techniques involve computationally demanding processes, and thus, researchers are currently focusing on SR performance acceleration. Aiming at improving the SR performance, the current paper builds up on the characteristics of the *L-SEABI* SR method to introduce parallelization techniques for GPUs and FPGAs. The proposed techniques accelerate GPU reconstruction of ultra-high definition content, by achieving three (3×) times faster than the real-time performance on mid-range and previous generation devices and at least nine times (9×) faster than the real-time performance on high-end GPUs. The FPGA design leads to a scalable architecture performing four (4×) times faster than the real-time on low-end Xilinx Virtex 5 devices and 69 times (69×) faster than the real-time on the Virtex 2000t. Moreover, we confirm the benefits of the proposed acceleration techniques by employing them on a different category of image processing algorithms: on window-based disparity functions, for which the proposed GPU technique shows an improvement over the CPU performance ranging from 14 times (14×) to 64 times (64×), while the proposed FPGA architecture provides 29× acceleration.

Keywords Real-time image processing · Graphics processing unit · Field-programmable gate array · Super-resolution · Comparative power-performance evaluation

✉ Dionysios Reisis
dreisis@phys.uoa.gr

[1] Electronics Laboratory Department of Physics, National and Kapodistrian University of Athens, Athens, Greece

1 Introduction

There is a wide range of image processing applications such as satellite and medical reconstruction/enhancement, high definition video broadcasting/processing, iris recognition and text images upgrade, for which the presence of high-resolution images is essential; meanwhile, hardware limitations and/or an increasing implementation cost prevent the integration of high-resolution sensors in the systems supporting the above applications. Researchers and engineers addressed this problem by modeling image degradations and by introducing signal processing techniques to post-process the acquired images. Such approaches exploit super-resolution (SR) techniques that construct high-resolution (HR) images from several low-resolution (LR) observations, and they thus trade off computational with hardware/implementation cost [1]. Consequently, current research focuses on improving the complexity and/or the running time performance of these processes [2–5].

Aiming at providing an efficient solution for these processes, the current manuscript proposes acceleration procedures of image processing techniques on multi-core, General Purpose Graphics Processing Units (GPUs) and Field-Programmable Gate Array (FPGA) platforms. It emphasizes the benefits of employing our *Low-complexity Statistical Edge-Adaptive Back-projected Interpolation* (*L-SEABI*) SR method [6] to speed up the reconstruction process by presenting efficient parallelization techniques for this method performing on GPUs and FPGAs. Our goal is to provide techniques and the corresponding implementations, which overcome the underlying platform's drawbacks and ultimately to present a comprehensive performance assessment. To our knowledge, this assessment contributes to the current literature by jointly examining four parameters: (1) absolute performance, (2) input

Published online: 21 July 2016

size, (3) power consumption and (4) cost of ownership on multiple GPU, FPGA and multi-core CPUs designating low to high-end ICs. Additionally, to confirm the benefits of our acceleration techniques, we apply the entire design/optimization/evaluation process on a computationally demanding window-based disparity algorithm. Finally, taking into account all the results on multiple platforms and devices presented by this work, we perform a comparative study at the platform level to evaluate the relative performance of FPGAs, GPUs and CPUs, in terms of speed and power consumption.

The proposed GPU parallelization/optimization techniques are effective for a large span of GPU architecture generations including the latest. We apply these techniques on multiple abstraction levels ranging from the design phase to the implementation API and exploit GPU architectural features to fuse increased throughput, instruction-level parallelism, with decreased latency and divergence [7]. These techniques allow the achievement of real-time (i.e., 30 frames/s) GPU reconstruction of Ultra-High Definition content. Furthermore, as it will be demonstrated in the remaining sections of the paper, we achieve 3× faster performance (frame rate) than the conventional real-time requirement on mid-range and previous generation devices and at least 9× faster performance on the currently available high-end GPUs. Moreover, we introduce a parameterizable and highly scalable hardware architecture for *L-SEABI* and we evaluate its performance under varying parallelization factors and FPGA devices. By exploiting pixel and task-level pipelining, the proposed architecture performs four (4×) times faster than the conventional real-time requirement on low-end Virtex 5 devices and at most 69 (69×) times faster than real-time on the Virtex 2000t device. When implementing the proposed architecture on a Virtex 7 485T device, we accomplish real-time processing of 182 Mpixel images, which is twice the resolution of upcoming 11K monitors. Furthermore, we comparatively evaluate the performance of *SIL-SEABI* among CPU, GPU and FPGA implementations, taking into account the power dissipation of each platform thereby estimating the achieved performance per Watt.

To enhance our evaluation and to strengthen the results reached from our SR study regarding the efficiency of our acceleration techniques, we consider a more computationally intensive scenario: the stereo correspondence problem and more specifically a disparity algorithm, which computes a depth map based on metric aggregations with non-separable filters, left-right consistency checks and sub-pixel accuracy estimations. As with *SIL-SEABI*, we provide (1) a GPU-accelerated implementation aiming at assessing bottlenecks through kernel profiling and (2) an FPGA architecture targeting the reduction of memory requirements. On a mid-range device, our GPU disparity implementation achieves an acceleration of over 14× against the fastest CPU examined for 1120 × 1120 input and 200 disparities, while the proposed FPGA architecture attains an acceleration of 29× over the same CPU.

The manuscript is organized with the following section presenting related work in the field of application acceleration through many-core architectures and FPGAs. Section 3 highlights the benefits of using *SIL-SEABI* through a quality evaluation of the SR algorithm both as stand-alone solution as well as an enhancement to more recent state-of-the-art SR. Section 4 presents a thorough profiling of the CPU and the coarse-grain GPU techniques leading to our optimized real-time GPU implementation. Section 5 delineates our scalable FPGA SR architecture and evaluates its performance. Section 6 extends our study onto the disparity algorithm of Stereo Correspondence and Sect. 7 displays our comparative study. Finally, Sect. 8 concludes the paper.

2 Related work

2.1 GPU acceleration

Throughout the literature, many-core architectures have been widely employed as an efficient means to accelerating image and/or video processing. In the field of super-resolution, Freedman et al. [8] applied their single-image SR method on a GTX 480 GPU to achieve near real-time performance when upsampling from 640 × 360 to 1920 × 1080. Using *dictionary-based* methods, the authors of [9] also provide a GPU implementation of their deformable image patch method. On a different approach, the authors of [3, 4] exploited the *cuda-convnet* library [10] to train their Convolutional Neural Network mapping between low and high-resolution images. More recently, Jung et al. employed the *cuFFT* [11] library in their deconvolution-based SR approach, achieving 53 ms for 2× upsampling to 1920 × 1080 on a GTX 580 GPU device. GPUs have also been common early on in accelerating stereo vision applications, as the authors of [12] displayed. Their box filter window-based matching method attains real-time performance on a 8800GTX in 1024 × 768 resolution for 50 disparities. Also, in [13], a hardware-efficient bilateral filter that features high accuracy and fast aggregation was proposed and implemented on a GTX 580 GPU. The authors of [14, 15] combine spatial with temporal-based processing to achieve real-time performance for 320 × 240 resolutions on more recent GPU microarchitectures.

2.2 FPGA acceleration

Whenever the application's computational characteristics necessitate a combination of high performance and low power consumption, mapping algorithms onto integrated circuits (ICs) seems like the most preferable solution. Field-Programmable Gate Arrays (FPGAs) consist of configurable logic blocks and embedded functions which altogether cater for such requirements with great flexibility, especially in problems pertinent to computer vision. For instance, Bowen et al. [16], combined a weighted-mean estimator with iterative refinement for multi-frame image upsampling to 720p at 61 frames per second (fps) on a Xilinx Virtex II FPGA. The authors of [17] implement a multi-frame Iterative Back-projection framework on a Virtex 4 FPGA to maintain a performance of 25 frames per second for 2 iterations on 1024 × 1024 output.

In more recent developments, Sanada et al. [18] proposed an edge enhancement single-image super resolution architecture relying on integer operations which achieved a performance of 60, 400 × 400 fps on an Altera ep2s130 Stratix II FPGA. Pérez et al. [19] designed a stream-processing FPGA architecture to super-resolve data from micro-lens arrays in light-field cameras. Their solution requires 105.9 ms to produce 589 × 589 images from 291 × 291 micro-lenses on low-power FPGA platforms. Extending the problem to High Definition resolutions, the authors of [20] present a real-time Altera Aria II SR implementation. For more computationally intensive computer vision problems, Greisen et al. [21] propose a window-based stereo matching FPGA implementation, which can compute 256 disparities of 1080p images in real-time. The latter is feasible due to the authors' hierarchical approach, which initially computes the disparity for 16 times smaller images and then refines the upsampled disparity map by searching at a local pixel area. In general, a common feature of real-time stereo implementations is their increased resource cost as [22, 23] display.

2.3 GPU-FPGA comparison

Throughout the literature, the comparative assessment of performance between GPUs and FPGAs is based on accelerating well-known algorithms or generic implementations of a specific algorithmic category. For instance, Che et al. [24] employ a 8800GTX GPU and a Virtex II FPGA to examine Gaussian Elimination used in linear algebra, the DES used in cryptography and the Needleman-Wunsch algorithm used in DNA sequence alignment. The work in [25] proposes fully customized Cholesky Decomposition implementations on the high-end GTX 480 and Virtex 6 xc6vsx475t. The authors of [26] assess a wide range of applications (i.e., random number generation, matrix multiplication, parallel reduction and N-body simulation) on a GTX 285 and the HC-1 HPC multi-FPGA platform. In the field of computer vision, Kalarot et al. [27] implement a generic disparity algorithm on the GTX 280 and the Altera Stratix III platforms. A complete blood vessel detection system from medical images [28] is implemented on a GTX 295 GPU and a Spartan-3 FPGA, while more recently, (a) Pietron et al. [29] compare a human skin classifier implementation on a Tesla m2090 and a Virtex 5 device and (b) the ceramic tile defect detection algorithm of [30] is evaluated on the 9800GT GPU and three different FPGAs. In a slightly different direction, the authors of [31] evaluate the performance of high-level synthesis of GPU to FPGA stereo matching code.

The above literature survey leads to the conclusion (to be revised by our current work) that GPUs are more suitable for SIMD computations with no inter-dependencies. They enable easier migration from a conventional software implementation, provide flexibility whenever the designer needs to implement changes and present a low total cost of ownership. On the other hand, they are less efficient in applications requiring a high amount of and/or irregular memory accesses, extensive synchronization, and finally, they consume more power. FPGAs, being more suitable for bit-level streaming operations, provide a higher level of control over the implementation's details. To their advantage is the fact that they combine high throughput, low power consumption, feature deterministic performance and resource cost. They are though more complex to program/customize, require thorough comprehension of the underlying algorithm and are not fully suitable for (a) applications of high dataflow complexity or (b) large input problems due to their restricted memory resources.

Focusing on improving solutions for image processing problems, this work presents parallelization techniques for the low-complexity *SIL-SEABI* SR algorithm [6]. As a preface to the *SIL-SEABI* parallelization, the following section stresses the advantages of using it as an enhancement to well-established and more recent state-of-the-art SR techniques.

3 The *L-SEABI* algorithm as a standalone SR solution and state-of-the-art enhancement

Our proposed *L-SEABI* super-resolution algorithm [6], achieves objective quality comparable to highly involved SR methods at significantly lower computational cost, i.e., 3 orders of magnitude less execution time. It consists of a construction phase which is executed once and a refinement phase which is iterative. The single-iteration case of *L-SEABI* [6], namely *SIL-SEABI*, further reduces the execution time by an order of magnitude at the expense of negligible quality degradation.

Comparison to the state-of-the-art shows [6] that *L-SEABI* is faster than (a) dictionary-based methods such as the Sparse Representation with over-complete dictionaries

[32], the Nonlocal Similarity Adaptive Regularization [33] (*ASDS-AR-NL*) and the Anchored Neighborhood Regression (*ANR*) [2], (b) *Bayesian* prior methods such as the Norm *l*1 [34], and better than (c) *IBP*-based methods such as the Nonlocal Iterative Back-Projection [35] (*NLIBP*). More recently, the authors in [36] proposed the concept of a nonlocal autoregressive model (*NARM*) i.e., the adaptive exploitation of nonlocal image redundancies in a sparse representation context to improve the performance of the standard sparse representation model. Authors in [9] also enhance the sparse-representation approach by proposing a regularized deformation field instead of fixed vector patches (i.e., deformable patch super-resolution, *DPSR*). Common ground in the aforementioned state-of-the-art methods is an initial reconstruction step which builds an initial estimation image through bicubic interpolation.

Motivated by the result that *L-SEABI* can increase the quality of common interpolation methods we use *L-SEABI* to improve the performance of state-of-the-art SR. Ergo, this section presents the application of (a) *L-SEABI*, (b) *SIL-SEABI* and (c) their construction phase only (noted as *L-SEAI*), in place of the initial reconstruction step of the [2, 9, 32, 33, 35, 36] methods to upsample each low-resolution image by a factor of two in each dimension. The output quality is measured by employing the full-reference MSSIM [37] and the no-reference BRISQUE [38] metrics.

Table 1 presents the average MSSIM and BRISQUE metrics for *L-SEABI*, *SIL-SEABI*, *L-SEAI*, versus the more recent *NARM* and *DPSR* methods of the dataset used in [6] (26 images in resolutions from 176 × 144 up to 3840 × 2160). The results display that appending adaptive iterations to *L-SEAI* marginally increases the output quality with respect to the MSSIM metric, though the difference is more pronounced with BRISQUE. All *L-SEABI* cases perform comparably to the *NARM* and *DPSR* methods and even achieve higher quality when considering BRISQUE. These results are consistent with those documented in [6] regarding dictionary-based methods processing a single image.

Table 2 displays the average per resolution quality difference between the results obtained by the methods in [2, 9, 32, 33, 35, 36] when using *L-SEABI*/*SIL-SEABI*/*L-SEAI* and the results obtained by the same methods when using the parameters proposed by their respective authors in place of their construction phase. Providing alternative initialization to the method in [35] results in a slight MSSIM degradation of 0.08 on average. ΔBRISQUE on the other hand signifies a quality increase, especially when using the *L-SEABI*/*SIL-SEABI* methods in 720p or higher resolutions. Note that the method in [35] as proposed by the authors performs 20 back-projection iterations which, when combined with the already back-projected result of *L-SEABI*/*SIL-SEABI* oversharpen the image. Thus, in *NLIBP*, using the *L-SEAI* produces more balanced results in all resolutions, while *L-SEABI* and *SIL-SEABI* can be used in high-definition content. Similar results were obtained when considering the more robust *ANR* method; we measure an average MSSIM degradation of 0.02 while *BRISQUE* favors the application of *L-SEABI*/*SIL-SEABI* by displaying a quality increase in almost all resolutions. When augmenting the method in [32], we measure similar MSSIM results to the *ANR*. *L-SEABI* and *SIL-SEABI* now increase quality under all resolutions when considering BRISQUE. Regarding *NARM* and *ASDS-AR-NL*, MSSIM displays a negligible increase of approximately 0.002, while BRISQUE shows that *SIL-SEABI* and *L-SEABI*, respectively, provide the highest quality in all resolutions. Our low-complexity methods also enhance the quality of *DPSR*, by 0.005 and 2.63 when evaluating MSSIM and BRISQUE, respectively. Therefore, the above results show that in most cases, the *SIL-SEABI* algorithm can be used to further improve state-of-the-art SR quality, as evaluated objectively by the *MSSIM* and *BRISQUE* metrics. Please see "Appendix" for Table 9 which includes the results for all resolutions.

To subjectively assess state-of-the-art SR enhancement based on our methods, in Fig. 1, we present the output of [32, 36] when processing the 256 × 256 *Cameraman* image. Using *SIL-SEABI* to complement *NARM*, (Fig. 1b) noticeably reduces the aliasing effects of the method. When we employ *SIL-SEABI* to initialize [32] (Fig. 1d), the results display a perceptible contrast enhancement. Similar results (i.e., aliasing reduction and contrast enhancement) can be observed when *SIL-SEABI* is applied before *DPSR* [9] and *ANR* [2], respectively. Please see Fig. 17 in "Appendix" which displays subjective results for 4 different images.

4 Acceleration of L-SEABI on GPU

The current section presents first, the strategy followed for parallelizing *L-SEABI* on GPU and second, the implementation, optimization issues, and performance evaluation.

4.1 Implementation strategy: CPU to GPU

This section describes the steps leading from a *SIL-SEABI* CPU implementation to our GPU implementation. Figure 2 provides an overview of the two phases and the distinct processes involved in the *L-SEABI* algorithm. All image operations are executed serially, beginning with the operation of the gradient computation and summation for each pixel position in the original image. Following the construction phase of *L-SEABI*, the initial reconstruction

Table 1 Objective comparison of the *L-SEABI* algorithm against its single-iteration case, its construction phase, the *NARM* SR and the super-resolution method using deformable patches (scaling factor $f = 2$)

Output size	Method				
	L-SEABI	SIL-SEABI	L-SEAI	NARM [36]	DPSR [9]
MSSIM					
QCIF	0.9114	0.9086	0.9013	0.9278	0.9106
CIF	0.8763	0.8743	0.8604	0.8817	0.8834
SD1	0.9702	0.9679	0.9652	0.9730	0.9751
720p	0.9403	0.9567	0.9537	0.9618	0.9636
1080p	0.9796	0.9784	0.9783	0.9808	0.9816
2160p	0.9880	0.9879	0.9874	0.9887	0.9883
Avg	0.9474	0.9456	0.9411	0.9523	0.9504
BRISQUE					
QCIF	39.9237	43.5880	42.5035	39.7551	30.1888
CIF	33.8815	36.1080	33.0046	45.2763	42.7670
SD1	42.7928	44.4549	47.5913	46.7715	38.9799
720p	41.6382	43.5580	46.1487	46.9208	41.8770
1080p	40.4554	42.0595	46.7294	47.1687	44.9506
2160p	56.8680	57.9444	58.4345	59.8403	55.7759
Avg	42.5948	44.6188	45.7353	47.6221	42.4232

All algorithms use the parameters proposed by their authors. Lower BRISQUE indicates higher quality

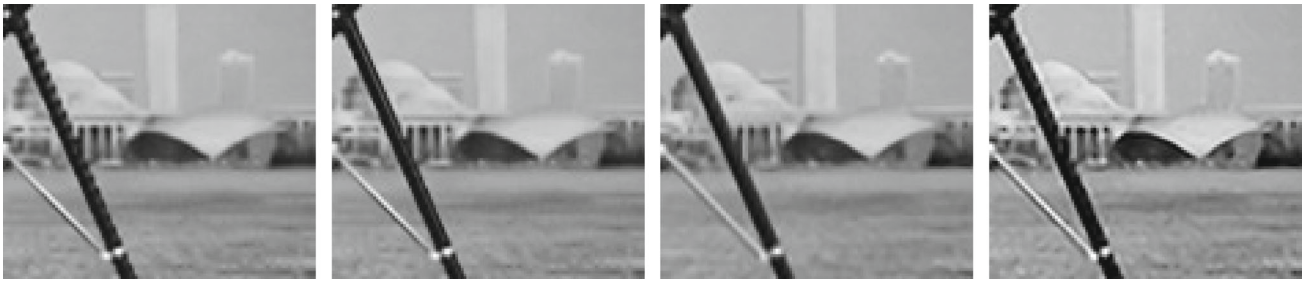
Table 2 Per resolution objective comparison of state-of-the-art SR algorithms (scaling factor $f = 2$) when using *L-SEABI* (a), *SIL-SEABI* (b) and *L-SEAI* (c) as their initial reconstruction phase against the parameters proposed by their authors

Metric	Initialization	Algorithm					
Output size		NLIBP [35]	ANR [2]	Yang et al. [32]	NARM [36]	DPSR [9]	ASDS-AR-NL [33]
ΔMSSIM							
720p	(a)	−0.07074	−0.01943	−0.01730	0.00116	0.00276	0.00171
	(b)	−0.07234	−0.02162	−0.01914	0.00113	0.00267	0.00169
	(c)	−0.01511	−0.01082	−0.01407	0.00114	0.00206	0.00164
1080p	(a)	−0.05187	−0.01130	−0.01427	0.00128	0.00087	0.00168
	(b)	−0.05229	−0.01223	−0.01473	0.00126	0.00083	0.00168
	(c)	−0.02102	−0.00564	−0.00973	0.00125	0.00064	0.00165
ΔBRISQUE							
720p	(a)	−8.77396	−2.94443	−6.10462	0.05638	−0.93106	−0.0634
	(b)	−5.48932	−2.42523	−4.26859	−0.38838	−0.83522	−0.07618
	(c)	−5.10736	−0.39096	−0.25632	0.09672	−0.79050	−0.03506
1080p	(a)	−7.86562	−7.99265	−10.8402	1.36240	−1.83855	−0.01110
	(b)	−4.88077	−7.44972	−9.51325	0.48452	−1.76727	0.00032
	(c)	−3.97212	−2.69070	−3.31287	0.70232	−1.56985	−0.00012

Lower ΔBRISQUE indicates higher quality

outcome is copied to a temporary buffer. Subsequently, the image is refined iteratively and adaptively, based on the error minimization criterion.

To determine the proposed parallelization strategy on the GPU platform, we use *oprofile* [39] to analyze the single-threaded execution of the *SIL-SEABI* 4.9.2 GCC executable. We evaluate our code on an Intel *Core i5-3470* CPU for 100 repetitions on at least 50 successive frames of each dataset sequence and average the results over all image sizes. As is plotted in Fig. 3, approximately 42 % of the total execution time is on average consumed on the address clamping (*clamp x,y*) function which restricts pixel coordinates according to the image boundaries, while clamping the luminance values to the range of [0, 255]



(a) normal SR: [36]
MSSIM: 0.865780
BRISQUE: 39.3129

(b) [36]+*SIL-SEABI*
MSSIM: 0.869826
BRISQUE: 38.4339

(c) normal SR: [32]
MSSIM: 0.863308
BRISQUE: 37.0668

(d) [32]+*SIL-SEABI*
MSSIM: 0.837666
BRISQUE: 33.3080

Fig. 1 Subjective comparison of [32, 36]: normal execution and enhanced with *SIL-SEABI* ($f = 2$)

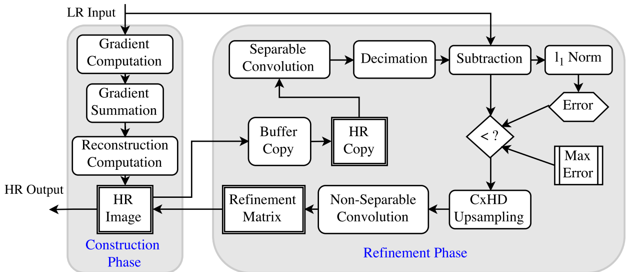


Fig. 2 *L-SEABI* algorithm: distinct processes

requires 3 % of the total upsampling time (*Y-clamp*). The non-separable, 5×5 back-projection filter (*bp-flt*) is accountable for 20 % of the total computation time, i.e., almost the same as the construction phase and the *CxHD* upsampling combined (12.04 and 4.27 %, respectively). Separable convolution on the other hand takes up 12 % of CPU time (*blr-flt*). Adding the error back to the HR image consumes approximately 1.36 % of the total time (*adderror* in Fig. 3), creating a copy of the constructed HR buffer for the algorithm's refinement phase (*copybuffer*) accounts for approximately 1.25 % CPU time and the remaining processes sum up to 4 %. Based on the obtained profiling results, we will first describe our generic *L-SEABI* GPU technique, then document the improvements toward the proposed *SIL-SEABI* implementation and finally evaluate the performance achieved.

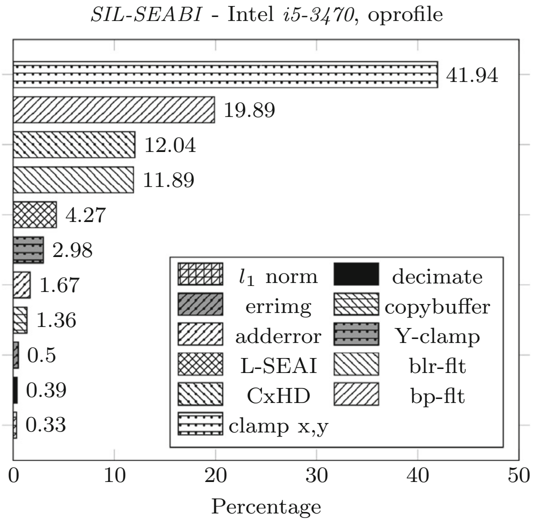


Fig. 3 Profiling the *SIL-SEABI* algorithm for single-threaded execution on the CPU

4.2 GPU implementation and optimizations

The CUDA programming model allows for unified employment of massively multi-threaded processors for both graphics and general purpose parallel computing. CUDA-capable devices consist of scalable arrays of streaming multiprocessors (SMs) to which the programmer distributes parallel threads through high-level GPU programs called kernels [7]. The CUDA API also defines an assembly-resembling intermediate language, the parallel thread execution virtual machine and instruction set architecture (PTX ISA) [40]. The nVidia compiler translates CUDA code to PTX, and the driver converts the PTX into GPU-executable code. For all evaluation purposes, we will test our GPU code on three devices designating successive major compute capability versions: the *GeForce GTX 550 Ti* (Fermi microarchitecture) with compute capability 2.1, the *GeForce GTX 670* (Kepler microarchitecture) with compute capability 3.0 and the *GeForce GTX 960* (Maxwell microarchitecture) with compute capability 5.2. The respective number of CUDA cores are 192, 1344 and 1024. In our evaluation, we first provide a coarse-grained profiling analysis for a single iteration of our generic *L-SEABI* GPU implementation. Using CUDA 7.0, we will compare against the results of Fig. 3 and then we will modify the GPU-accelerated version as required for *SIL-SEABI*. To implement *SIL-SEABI*, we will focus on improving the overall flow and the specifics of each kernel, using our *L-SEABI* implementation as the groundwork for additional optimizations. We will focus on creating a GPU implementation which does not rely on specific libraries and provides good performance regardless of the underlying architecture.

As reported in [41], the use of page-locked memory (stored in the Host's physical RAM) allows for an approximate twofold increase in bandwidth compared to pageable memory. Thus, on our Host code, we begin by allocating page-locked memory for the LR and HR image buffers using the *cudaHostAlloc* function. Besides, in cases where memory access patterns display a spatial locality—such as is our case throughout the *L-SEABI* algorithm—the use of texture memory is preferred, as it is also cached. Additionally, boundary addressing is handled automatically, and therefore, the increased cost related to the address clamping function of the CPU implementation will be mitigated. As a result of texture memory usage, we copy the page-locked buffers onto the device using the *cudaMemcpy2D* function with which we also handle the required texture memory alignment.

For performance reasons, it is recommended practice that the number of threads in each block are a multiple of the fixed warp size of 32 threads (i.e., the number of threads executed in a lockstep). According to Xu et al. [42] splitting the image buffer into 2D tiles of 32 × 4 samples provides better performance results on devices having compute capability of 1.3 or higher. Based on the above result and our experimentation, we employ the aforementioned thread partitioning scheme on the *GTX 670*, *GTX 960* GPUs and increase the block size to 32 × 8 threads on the *GTX 550Ti* by adapting the grid size according to the input image dimensions.

```
Input: LR Image, max_iterations, max_error
Output: HR Image
1:  gradientCompute$_{\mathcal{K}}$
2:  reduce$_{\mathcal{K}1}$ => rdc_res$_1$
3:  lseai$_{\mathcal{K}}$
for (int i=0; i<max_iterations; i++)
{
4:   cudaMemcpy2D(Device->Device)
5:   convCols$_{\mathcal{K}}$
6:   convRows$_{\mathcal{K}}$
7:   decimate$_{\mathcal{K}}$
8:   errimg$_{\mathcal{K}}$
9:   abs$_{\mathcal{K}}$
10:  reduce$_{\mathcal{K}2}$ => rdc_res$_2$
          if (rdc_res$_2$ < max_error)
          {
11:         CxHD$_{\mathcal{K}}$
12:         convNoSep$_{\mathcal{K}}$
13:         adderror$_{\mathcal{K}}$
          }
          else break;
}
```

Fig. 4 *L-SEABI* GPU implementation: pseudocode listing kernels

Figure 4 lists the pseudocode of the device processing flow in the proposed *L-SEABI* implementation; the GPU kernels are denoted with the $\mathcal{K}$ subscript. In the *gradientCompute*$_{\mathcal{K}}$ kernel every thread computes the gradient in both dimensions and in the process creates a $W \times H$ buffer storing the per-pixel sums of squared gradients. Next, to compute the total variation, we employ the nVidia designed parallel reduction kernel [43], which features unrolling of operations and summation of multiple values per each thread (*reduce*$_{\mathcal{K}}$ in Fig. 4). The *lseai*$_{\mathcal{K}}$ kernel involves computation of the initial reconstruction by employing the *rdc_res*$_1$ result of the *gradientCompute*$_{\mathcal{K}}$ and it features a balanced cost between arithmetic and memory transfer operations; each thread fetches 16 surrounding samples per-pixel from texture memory to registers, performs characterization based on the threshold result and computes the luminance outcome for the horizontal, vertical and diagonal positions of the upsampled grid. Note here that in order to avoid thread divergence when calculating the coordinates in the HR grid, we execute our upsampling kernels by using the LR grid partitioning scheme. Thus, the HR offsets are computed as follows:

$$
\begin{aligned}
I_{\text{offset}} &= [(ty * 2) * (W * 2)] + (tx * 2) \\
HH_{\text{offset}} &= [(ty * 2) * (W * 2)] + [(tx * 2) + 1] \\
HV_{\text{offset}} &= \{[(ty * 2) + 1] * (W * 2)\} + (tx * 2) \\
HD_{\text{offset}} &= \{(ty * 2) + 1] * (W * 2)\} + [(tx * 2) + 1]
\end{aligned}
\tag{1}
$$

where *tx*, *ty* are the translated 2D thread coordinates inside a 1D memory array and *W* is equal to the LR image width or the pitch size in bytes, in case of a non-texture or texture-bound output HR buffer, respectively. Furthermore, this approach effectively increases the instruction-level parallelism (ILP) to 4 because each thread computes and stores four pixel values in the upsampled grid.

The iterative refinement phase (lines 4–13) begins its first/subsequent iterations by copying the reconstructed/refined image to an intermediate, texture-bound buffer. This copy takes place inside the device and precedes the two separable filtering kernels (*convCols*$_{\mathcal{K}}$, *convRows*$_{\mathcal{K}}$), which also rely on texture memory and register storage: each thread fetches 5 pixels per input sample from the HR buffer into device registers, computes the convolution sum and copies the result from registers back into global memory. We note here that all the filtering operations in our kernels are based on unrolled operations.

The image is then subsampled by *decimate*$_{\mathcal{K}}$, which employs the LR thread partitioning scheme and it simply copies data from the samples in the HR buffer with coordinates $(tx * 2, ty * 2)$, to the samples in the LR buffer with coordinates (tx, ty). The *CxHD*$_{\mathcal{K}}$ and the *convNoSep*$_{\mathcal{K}}$ follow similar design choices to the *L-SEAI* kernel, i.e., they rely on texture fetches for input and per-thread registers to store intermediate results; threads of the *CxHD*$_{\mathcal{K}}$ read 12 pixels per input sample, while *convNoSep*$_{\mathcal{K}}$ threads read 25 pixels per input sample. The remaining kernels are functionally straightforward: *errimg*$_{\mathcal{K}}$ subtracts the decimated image from the LR input and then its result is conditionally forwarded to *CxHD*$_{\mathcal{K}}$ based on the current norm l_1 value (*rdc_res*$_2$, i.e., the outcome of *abs*$_{\mathcal{K}}$ and *reduce*$_{\mathcal{K}}$).

Figure 5 displays the execution analysis of the proposed *L-SEABI* implementation using nVidia's profiler on the devices tested averaged over all image resolutions. Bars outlined in black describe the corresponding cost on the *GTX 670*, and bars outlined in blue refer to the cost as measured on the *GTX 550 Ti*. As mentioned earlier, due to the use of texture-bound buffers, address clamping is handled with minimal overhead and therefore, the profiler does not report a discrete cost. Luminance clamping on the other hand requires an explicitly designed device function. The compiler fuses our luminance clamping code with the kernels it is employed in, namely *lseai*$_{\mathcal{K}}$ and *adderror*$_{\mathcal{K}}$ and thus, its computational cost is included in the corresponding cost reported in Fig. 5 for the *L-SEAI* and *adderror* processes. Note here that the *L-SEAI* process incorporates lines 1–3 of Fig. 4 and additionally a single-value D2H copy, which accounts for 0.24 % of the total time. In the case of the GPU implementation, more than 30 % of the computation time is consumed on filtering operations, namely the *convNoSep*$_{\mathcal{K}}$ displayed as *bp-flt* (back-projection filter) and *convCols*$_{\mathcal{K}}$, *convRows*$_{\mathcal{K}}$ shown as *blr-flt* (blur filter) in Fig. 5. There is also a noticeable discrepancy between the same filtering kernels on the two devices tested. Since the *convNoSep*$_{\mathcal{K}}$ requires 25 pixel reads per input sample, its performance is texture memory bandwidth bound and the kernel reaches the limit of 232.7 GB/s on the *GTX 550 Ti* (680.7 GB/s on the *GTX 960*). The utilization is more balanced in the cases of separable convolution kernels; in the case of *convRows*$_{\mathcal{K}}$ the performance is compute-bound because all the texture fetches refer to sequential addresses while the *convCols*$_{\mathcal{K}}$ displays a memory-bound performance. The memory transactions that do not belong to the kernels have a significant cost, i.e., they account for approximately 26 % of the total computation time. Specifically, the H2D and D2H transfers—related to transferring the LR image to and the HR result from the device—are obligatory and apart from the already adopted page-locked memory approach there cannot be further improved. There is also a negligible amount of time related to two single-value D2H transactions of the integer *reduce*$_{\mathcal{K}}$ results. Intra-device transfers though (denoted as D2D in Fig. 5) can be minimized as they pertain to copying data to texture-bound memory.

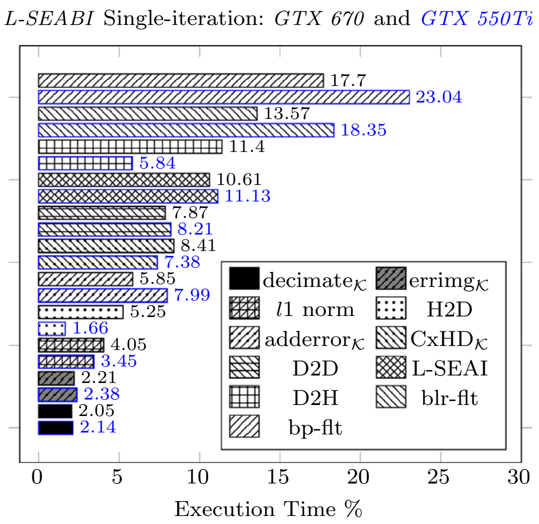


Fig. 5 *L-SEABI*: single-iteration GPU analysis

The following subsection will describe and classify the performed optimizations based on their scope, as *algorithmic-level* and *platform-dependent*. Moreover, it will evaluate these ameliorations by comparing the relative performance gains against the proposed baseline *L-SEABI* implementation. All modified kernels will be denoted by prepending "*m*" on their symbolic names.

4.2.1 *Algorithmic-level optimizations*

Optimizations at the algorithm's level are achievable because the proposed technique's goal is the single-iteration execution of *L-SEABI* assuming a priori that the adaptive back-projection of the error image occurs in all cases. Thus, in the proposed *SIL-SEABI* implementation, we can omit the execution of the *abs*$_{\mathcal{K}}$ in line 9 of Fig. 4, and the second reduction kernel in line 10 both of which account on average for 3.75 % of the total execution time (l_1 norm in Fig. 5) on the devices tested. Additionally, the *CxHD*$_{\mathcal{K}}$ kernel is modified to limit the neighboring checks from 8 pixels to only the 4 pixels surrounding each diagonal sample. This results in the modified *mCxHD*$_{\mathcal{K}}$ kernel, which applies the same averaging filter: the improvement is that it requires 2 less registers per thread, and 12 less diagonal edge-characterization comparison operations up from 14 in the original *CxHD* algorithm.

To optimize the separable convolution operations, we first exploit the fact that our 5 × 5 separable Gaussian kernel features symmetric coefficients. Therefore, we can reduce the number of multiplication operations to three per input sample by rewriting the convolution sum as $p^{\text{out}} = h_0 * p_0^{in} + h_1 * p_1^{in} + h_2 * p_2^{in} + h_3 * p_3^{in} + h_4 * p^{in} \quad 4 = h_0 * (p_0^{in} + p_4^{in}) + h_1 * (p_1^{in} + p_3^{in}) + h_2 * p_2^{in}$, where p^{out} denotes the luminance of the output sample, h_n and p_n^{in}, $n \in [0, 4]$ the filter coefficients and the input luminance values, respectively. Notice also that in Fig. 4 the *convRows*$_{\mathcal{K}}$ kernel is always followed by *decimate*$_{\mathcal{K}}$. The fusion of these two operations into the single *convRowsdec*$_{\mathcal{K}}$ kernel provides the following two benefits: (a) it circumvents the overhead of creating and executing a separate decimation kernel; and (b) it leads to the reduction of the number of computations and read operations related to *convRows*$_{\mathcal{K}}$ since due to the immediate decimation process only the even numbered samples are essentially needed. To avoid thread divergence we follow a similar approach to Eq. (1): we employ the LR thread partitioning scheme and fetch the even samples from the texture-bound buffer as follows:

$$p_n^{\text{in}} = \text{tex2D}(\text{texref},\, tx * 2 + n - 2,\, ty * 2), n \in [0, 4] \qquad (2)$$

where *tex2D* is the two-dimensional texture lookup CUDA function and *texref* is the texture-bound device array containing the column-filtered HR construction.

4.2.2 Platform-dependent and lower level optimizations

More specialized optimizations can be applied through the use of specific CUDA API code modifications by studying the proposed implementation both at a high (CUDA C code) and at a low level (PTX ISA code). The GPU execution analysis in this work (Fig. 5) showed that most of the processing time is consumed on the non-separable 5 × 5 filtering operation during the *L-SEABI* refinement phase. Since the overall performance is bounded by the memory bandwidth, *convNoSep*$_{\mathcal{K}}$ can be improved by using shared memory architecture. We employ the scheme that Eklund et al. proposed in [44] since this scheme is compatible with all of our devices: i.e., blocks of 32 × 32 threads transfer an image window of 96 × 64 samples to the shared memory prior to performing unrolled 2D convolution. Note that the *GTX 960* has 33 % larger shared memory size, a fact which allows increasing the window to 128 × 64 samples.

The detailed execution analysis of the proposed separable kernels revealed that their performance is compute-bound. As documented in [45], increasing the instruction-level parallelism (ILP) in CUDA can improve the performance even with fewer threads. In our case, we modify the *convRows*$_{\mathcal{K}}$ and *convCols*$_{\mathcal{K}}$, so that each thread fetches four 5-pixel sets from four consecutive blocks, performs four convolution sums and stores four results. This increases the number of required registers per thread from 12 to 28 on the *GTX 550Ti*, from 14 to 31 on the *GTX 670* and from 13 to 31 on the *GTX 960*. On the Fermi-based device, the modified kernels then reach the texture memory bandwidth limit, to further improve the performance, we turn our attention to the use of shared memory by modifying the nVidia separable convolution code sample [46]. Reworking the kernels for ILP = 4 causes the texture memory bandwidth to exceed 800 GB/s on the *GTX 670* without hindering performance, while the Maxwell-based device can reach 534 GB/s. In [6], the operations of *L-SEABI* were designed to rely on integer arithmetic excluding the separable filtering process. In particular, to enhance the output, quality convolution is applied by using the $\langle h_0, h_1, h_2 \rangle = \langle 0.00257, 0.165795, 0.664904 \rangle$ filter and converted the result back to integer.

Conveniently, the modern GPUs incorporate native floating-point units that provide higher multiplication performance than the use of integer operations; the respective throughput of 32-bit floating-point multiply-add operations (ops) per clock cycle (cc) and multiprocessor (MP) is 48, 192 and 128 on the *GTX 550Ti*, the *GTX 670* and the *GTX 960*, respectively. Note that we have to explicitly cast all coefficients and variables as floating-point in the convolution code; otherwise, the compiler will assume 64-bit precision, which has quite lesser throughput (4, 8 and 1 ops/cc/MP, respectively). To comply with the integer storage/operations of the proposed implementation though, additional type conversions are required. The review of the PTX ISA generated code reveals that apart from the computation instructions, the floating-point convolution requires 5 conversions from 32-bit to 16-bit unsigned numbers, 5 conversions from 16-bit to 32-bit floating-point and 1 conversion from floating-point to 32-bit unsigned. The generated instructions considering the integer and the floating-point separable convolution implementation are shown in Table 3 juxtaposed by their respective throughput. On the Fermi and Kepler devices, the throughput of 32-bit integer multiply-add operations/cc/MP is 16 and 32, respectively [47]. To compare against the floating-point filtering process, we compute integer Gaussian coefficients by multiplying the original coefficients with 2^{23}, resulting in the filter $\langle h_0, h_1, h_2 \rangle = \langle 21559, 1390789, 5577619 \rangle$. Note that the division operation is converted to an arithmetic shift by the compiler (*shr.u32*). Also, Table 3 shows that while the floating-point multiplication throughput is considerably higher compared to integer operations, the amount of conversions is non-trivial, and therefore, the

Table 3 Separable convolution PTX ISA instruction type, amount ran and associated throughput per arithmetic/device

blr-flt Arith. Instr. Type	Run	ops/cycle/MP	
int ■ float		550Ti	670
and.b32 ■ and.b16	5	48 ■ 48	160 ■ 160
add.s32 ■ add.ftz.f32	2	48 ■ 48	160 ■ 192
mul.lo.s32 ■ mul.ftz.f32	1	16 ■ 48	32 ■ 192
mad.lo.s32 ■ fma.rn.ftz.f32	2	16 ■ 48	32 ■ 192
shr.u32 ■ cvt.rzi.ftz.u32.f32	1	16 ■ 16	32 ■ 32
-■ cvt.u16.u32	5	- ■ 16	-■ 32
-■ cvt.rn.f32.u16	5	- ■ 16	-■ 128

```
$mCxHD_{\mathcal{K}}$ HH Computation:
Initial Boolean Conditions:
 1: $A>0$            // condition #1, vertical edge
 2: $A<0$            // condition #2, horizontal edge
 3: $B>0$            // condition #3, NE direction
 4: $B<0$            // condition #4, SW direction
Computation using nested if statements:
 5: if $A>0$:
 6:     if $B>0$:        $p_{HH}^{out} \leftarrow$ bicubic filtering #1
 7:     else if $B<0$:   $p_{HH}^{out} \leftarrow$ bicubic filtering #2
 8:     else:            $p_{HH}^{out} \leftarrow$ bilinear filtering
 9: else if $A<0$:       $p_{HH}^{out} \leftarrow$ bicubic filtering #3
10: else:                $p_{HH}^{out} \leftarrow$ bilinear filtering
Additional Boolean Conditions:
11: $A=0$            // condition #5, homogeneity
12: $B=0$            // condition #6, strictly vertical
Divergent-free computation:
13: $p_{HH}^{out} \leftarrow$
      (cnd#1 && cnd#3) * (bicubic #1) +
      (cnd#1 && cnd#4) * (bicubic #2) +
      (cnd#2) * (bicubic #3) +
      (cnd#6 | (cnd#1 && cnd#5)) * (bilinear)
```

Fig. 6 HH pixel code in $mCxHD_{\mathcal{K}}$: divergent-free computation

impact on the computation time requires evaluation. The *GTX 960* has in some cases lower throughput than the compute capability 3.0 *GTX 670*—this is mitigated by the Maxwell-based device's reduced instruction latency and stall cycles.

In this work, the upsampling optimizations mainly target the $\mathrm{lseai}_{\mathcal{K}}$ and $CxHD_{\mathcal{K}}$ and they are based on lower level PTX ISA observations. Notice that the original *L-SEABI* algorithm requires a square root to compute the edge-characterization threshold [6]. On CUDA, the floating-point square root is computed as a reciprocal square root followed by a reciprocal. In PTX ISA, this is translated as the *sqrt.approx.ftz.f32* instruction. Due to the integer nature of *L-SEABI* and the threshold being $T = \sqrt{\frac{\mathcal{TV}^{LR}}{WH}}$ where $\mathcal{TV}^{LR} = \mathrm{rdc\_res}_1$ (Fig. 4), we consider two alternatives: (a) set $T = \frac{\mathcal{TV}^{LR}}{WH}$ and then compute the square of the luminance differences in order to characterize edges, (b) compute the reciprocal square root of $\frac{WH}{\mathcal{TV}^{LR}}$. Besides these two options, the following subsection will evaluate the performance of a lower precision threshold computation, which employs integer division. Finally, based on the generated PTX ISA code, we rewrite all comparisons as Boolean statements and fine-tune all comparison operations in $mlseai_{\mathcal{K}}$ and $mCxHD_{\mathcal{K}}$ to eliminate nested *if* statements by applying branch fusion. As an example, Fig. 6 shows how we convert the computation of HH output samples $p_{\mathrm{HH}}^{\mathrm{out}}$ in the $mCxHD_{\mathcal{K}}$ kernel (lines 5–10) to a divergent-free optimized version (line 13). Notice that in this case, we have to explicitly define two additional Boolean conditions (lines 11–12) to procure the same result. Through this optimization, we seemingly convert the branch statement to a sum of integer products; actually though, only one of the products contributes to the $p_{\mathrm{HH}}^{\mathrm{out}}$ value for any given sample. Moreover, since every left operand of these multiplication operations evaluates to either 0 or 1 we can improve the performance by converting each Boolean multiplication to two bitwise and one integer addition operations as follows: $cnd * flt = [!(cnd) + 1] \,\&\, flt$, where $cnd \in \{0, 1\}$ is the Boolean condition, *flt* is the integer result of the filter, where ! is the bitwise *NOT* and & is the bitwise *AND* operator. Both of these operations have significantly higher throughput compared to the integer multiplication, especially on more recent devices [47].

Regarding the remaining kernels, the proposed optimizations target the computation of the absolute difference values in the $mgradientCompute_{\mathcal{K}}$, $mlseai_{\mathcal{K}}$ and $mCxHD_{\mathcal{K}}$ kernels. To that end, for the computation of the absolute difference $|dif|$ between pixel values p_0^{in} and p_1^{in}, we utilize the *_usad* integer intrinsic of CUDA as: $|dif| = |p_0^{in} - p_1^{in}| = \_usad(p_0^{in}, p_1^{in}, 0)$. We also compare the performance of *_usad* against four distinct implementations of the absolute function: (1) the *fabsf* function; (2) the *_vabsdiffu4* intrinsic, which computes per-byte unsigned absolute difference in an SIMD fashion; (3) an inline implementation using the ternary operator (denoted as ABS_t) and (4) an inline implementation using shift operations ($ABS_s = [dif + \{dif \gg 31\}] \wedge \{dif \gg 31\}$, where "$\gg$" and "$\wedge$" denote right shift and *XOR* operations, respectively). Finally, we focus on reducing the amount of D2D transfers by directly writing to texture-bound memory or by employing shared memory instead.

4.2.3 Performance evaluation

This section will first evaluate the relative performance, i.e., the induced speedup of the proposed optimizations against the baseline *L-SEABI* implementation. Second, it will assess the computation time required to upsample each LR input. To evaluate the performance of the proposed optimizations regarding *SIL-SEABI*, we perform the tests on the *GTX 550 Ti*, representing mid-range performance and on the *GTX 670*, *GTX 960* VGAs representing the high-end.

Figure 7 shows the achieved speedup per device and kernel depending on the type of the optimization. *Algorithmic-level* and *Platform-dependent* optimizations are highlighted in the green and blue areas, respectively. Regarding algorithmic-level optimizations, we observe an overall execution speedup of up to 1.29× by omitting the $abs_{\mathcal{K}}$ and the second reduction kernel (single-iter in Fig. 7). When we limit the number of comparisons during adaptive back-projection we notice a speed improvement in $CxHD_{\mathcal{K}}$ by up to 1.16× (cxavg in Fig. 7). Fusing decimation with separable row convolution by the proposed divergent-free solution (Eq. 2, “decfusion” in Fig. 7) produces a more pronounced speedup: *convRows*$_{\mathcal{K}}$ is accelerated by over 2.7×. Note though that rewriting the convolution computation to exploit the Gaussian kernel symmetry does not provide additional speedup as the compiler automatically generates identical PTX ISA code in both cases.

Platform-dependent optimizations provide the highest speedup as the remaining bar graphs show. Specifically, the use of shared memory in *mconvNoSep*$_{\mathcal{K}}$ (shown as *BP-flt: smem* in Fig. 7) is critical, increasing the performance of the original kernel by 4.65× on the *GTX 670* and more than 6× on the *GTX 550Ti* and *GTX 960*, which have less texture memory bandwidth. Note here that taking advantage of the increased shared memory of the *GTX 960*, i.e., modifying the method by Eklund et al. [44] to effectively process a 124 × 60 area and thereby increasing ILP to 8 compared to 6 slightly increases occupancy by 1.5 %; though it does not produce a significant performance advantage as a result of the small 5 × 5 filter size. When considering texture-based separable convolution, increasing the ILP to 4 for each thread in the *convCols*$_{\mathcal{K}}$ and *convRows*$_{\mathcal{K}}$ kernels results in over 1.6× speedup on the *GTX 550Ti* and 1.15× on the *GTX 670* (*blr-flt: ILP4,tex* in Fig. 7). The texture-based implementation can be further enhanced by employing shared memory combined with doubling the ILP (*blr-flt: ILP8,smem* in Fig. 7, [46]). Due to the small radius of our Gaussian convolution filter, the high register bandwidth of the GPUs and the spatial locality of the convolution process, the acceleration achieved is more prominent when modifying *convRows*$_{\mathcal{K}}$ to use shared memory (speedup of approximately 2.6×). This level of parallelism is the maximum that can be achieved for these kernels on the devices tested, as increasing the ILP to a greater extent results in arithmetic latency due to decreased warp occupancy. Evaluating the use of integer operations on the *SIL-SEABI* filtering process confirms the observations of Table 3: the absence of additional conversion instructions required for floating-point filtering slightly favor our integer implementation by 1.05× on average. The speedup related to Device to Device (D2D) memory transfers emphasizes the importance of memory latency in GPUs; using a single D2D memory transfer (i.e., by directly writing to texture-bound memory or using shared memory instead) reduces the time spent on D2D transfers by approximately 5× compared to our original GPU implementation.

Next, we evaluate the proposed optimizations pertinent to the *lseai*$_{\mathcal{K}}$ and *CxHD*$_{\mathcal{K}}$ kernels. Omitting a reciprocal computation provides a negligible advantage compared to the GPU floating-point square root implementation (*rsqrt/sqrt* in Fig. 7), which is justified as there is only a single threshold computation involved in the whole *lseai*$_{\mathcal{K}}$ kernel. Notice that avoiding the square root altogether substitutes a single reciprocal square root with 6 integer multiplication instructions required for comparison reasons and thus decreases the instruction throughput on GPUs (*rsqrt/nosqrt* in Fig. 7). Hence, the reciprocal square root provides the highest performance when computing threshold T in *lseai*$_{\mathcal{K}}$. We can also exchange accuracy for a marginal performance increase by using integer arithmetic on the $\frac{WH}{TV^{LR}}$ division as *lrrsqrt/rsqrt* in Fig. 7 suggests. Branching optimizations in difference comparisons are used throughout *lseai*$_{\mathcal{K}}$ and *CxHD*$_{\mathcal{K}}$ (averaged as “comparisons” in Fig. 7); particularly these depicted in Fig. 6, display speedup of up to 1.35× on average for the aforementioned kernels. Performance increase is more prominent on the

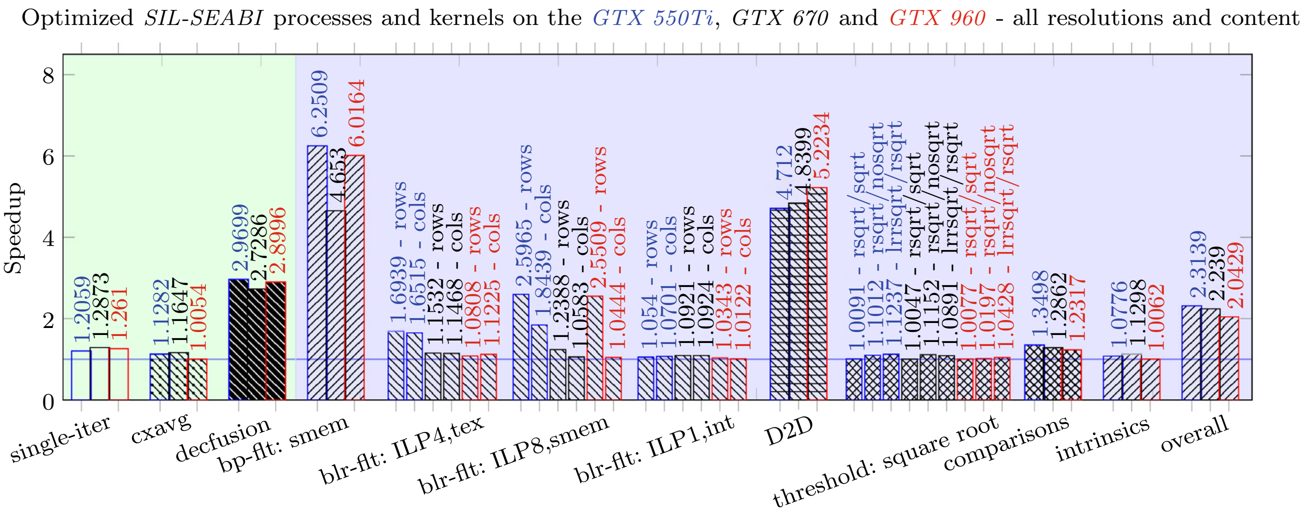


Fig. 7 Optimized GPU analysis for SIL-SEABI speedup per kernel involved versus the baseline implementation, averaged over all resolutions and image content

$CxHD_{\mathcal{K}}$ kernel compared to $lseai_{\mathcal{K}}$ (1.48× vs. 1.15× on average) because the compiler derives predicated instructions for the latter due to the absence of multiple/nested conditions in the original algorithm [6]. Also, the perceived gains are more pronounced on the lower compute capability *GTX 550Ti* GPU, as the *GTX 670* and *GTX 960* can schedule four warps and two independent instructions per warp per cycle [48]. To conclude our relative performance evaluation, we assess the five different implementations of the absolute difference as mentioned in Sect. 4.2.2. According to our evaluation regarding $lseai_{\mathcal{K}}$ and $CxHD_{\mathcal{K}}$ (averaged as "intrinsics" in Fig. 7), the *_usad* solution accomplishes the highest performance. The speedup we measured is consistent between devices and kernels, i.e., 1.0617× compared to the *fabsf* function, 1.1416× compared to the ternary operator (ABS_t), 1.0719× compared to the integer shifts implementation (ABS_s) and 1.1591× compared to the *_vabsdiffu4* intrinsic (supported on the Kepler and Maxwell devices). Notice that even though the latter SIMD operation has an instruction throughput of 160 ops/cc/MP on the *GTX 670*, when considering $lseai_{\mathcal{K}}$ the *_usad* solution translates to only 4 *sad.u32* instructions. These 4 *sad.u32* instructions have the same throughput as 4 *shl.b32* instructions required to employ the SIMD intrinsic. Studying the PTX ISA code though reveals that the *_vabsdiffu4* implementation requires 8 additional *and.b32*, 2 additional *shl.b32* and 6 additional *or.b32* instructions which rationalize the performance penalty.

Finally, to evaluate the absolute performance of the proposed *SIL-SEABI* implementation, we set apart H2D / D2H transfers, thus measuring the computation time consumed only by the upsampling kernels. To that end, we compare against our own implementations of separable 4-tap and 6-tap integer kernel interpolation, both of which represent two of the simplest upsampling solutions. We also compare against the GPU implementation of *DPSR*, representing the state-of-the-art. Evaluation results are plotted in Fig. 8 where computation time required to upsample each image by 2 in both dimensions is displayed as the average for every image resolution in the experimental dataset (Sect. 3). We note here that although the *GTX 670* has approximately 24 % more cores compared to the *GTX 960*, it achieves similar performance, i.e., less than 10 % discrepancy and it is 14.77 % faster when executing the *SIL-SEABI*. This is mainly attributed to the benefits of the internal design of the Maxwell architecture: each Streaming Multiprocessor has a power of two number of cores, functional units have a dedicated scheduler, there are 8 Load/Store units per 32 cores (compared to 16 per 96 cores for Kepler) and finally, more registers per core.

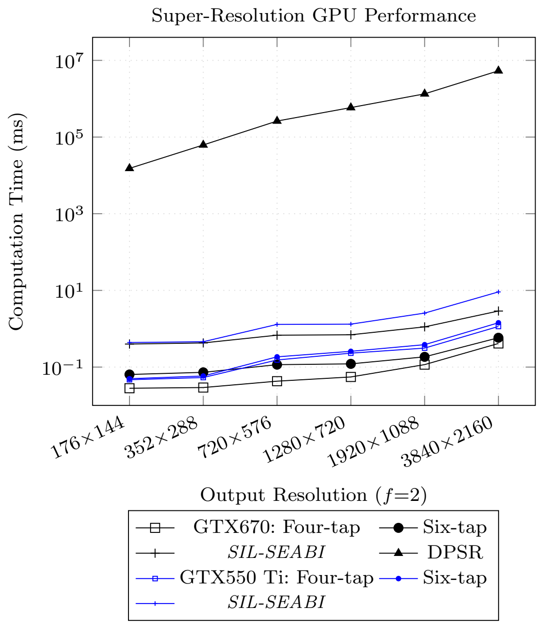


Fig. 8 SIL-SEABI on GPU: execution time ($f = 2$)

As expected, the simple interpolation kernels prove to be the fastest option; upsampling to 3840 × 2160 from 1920 × 1080 by employing Bicubic interpolation requires up to 0.4 ms on the *GTX 670* and up to 1.14 ms on the *GTX 550 Ti*. The 6-tap filter is 1.4× slower on average, a result which is consistent with the filtering kernel ratio. Thus, when considering 1920 × 1080 input, our simple interpolation kernels attain a performance of 698 frames per second (fps) on the mid-range GPU and up to 1718 fps on the high-end device. Also, it is anticipated that these methods perform approximately an order of magnitude faster than *SIL-SEABI* (5.1× and 7.6× considering 6-tap and 4-tap interpolation, respectively). Still, our *SIL-SEABI* implementation remains suitable for real-time 1080p processing as it achieves 109 fps on the *GTX 550 Ti* and 345 fps on the *GTX 670*. Contrastingly, super-resolution using deformable patches [9], consumes four orders of magnitude more time when upsampling to 176 × 144 and up to six orders of magnitude in the case of 2160p output. The memory footprint of *SIL-SEABI* is also relatively small and comparable to the 46 MB of the simple methods because it requires only 151 MB of device memory; hence, it can be executed on all modern discrete GPUs. *DPSR* execution on the other hand is significantly restricting because it consumes at least 466 MB (176 × 144) and up to 1452 MB of device memory for larger inputs.

5 Acceleration of L-SEABI on FPGA

The proposed hardware acceleration involves the parallel architectural design, the parametric VHDL development and the deployment of *L-SEABI* on a variety of FPGA devices. The purpose of implementing *L-SEABI* in a parametric fashion on various FPGA devices is to assess its performance and power dissipation with multiple parallelization factors and FPGA technologies. The following subsection describes the proposed architecture and the parallelization techniques devised to accelerate *L-SEABI*, whereas the second subsection presents our design exploration and the implementation results.

5.1 Parallel architecture design of SIL-SEABI

The super-resolution algorithm in [6], besides improved quality, targets parallelization amenability and low-cost implementation on hardware, especially on reconfigurable platforms with limited memory resources, such as the FPGA. Hence, compared to similar and more sophisticated super-resolution algorithms, *SIL-SEABI* avoids highly involved mathematical/statistical solutions, abundant dependencies among the calculations, or increased storage of side information (e.g., dictionaries). Instead, *SIL-SEABI* promotes locality and regularity.

The proposed architectural design relies on extensive pipelining of the computation, both on pixel basis and on task level. It is based on a technique involving the continuous processing of the image in a pixel-by-pixel basis by using a long pipeline, which lines up all arithmetic operations required to complete a given task of the algorithm for a single pixel. Note that the term *task* refers here to any elementary, conceptually distinct, transformation of the image, such as blurring, decimation, 2D convolution with a fixed kernel, calculation of image derivatives. Generally, the hardware cost of a pixel-based pipeline reduces to the cost of a single-pixel processing engine and it is independent of the image size. In specific cases, the pipeline must cache internally a group of pixels (to operate on a local area of the image, e.g., compute derivatives) the number of which could be proportional to the width of the image (e.g., when reading the pixels in raster-scan order). In the case of a pipeline designed optimally to achieve 100 % utilization, the throughput of the pixel-based pipeline increases to one pixel per cycle and allows the entire task to complete in $W \times H + L$ cycles, where W is the image width, H is the image height, and L is the latency of the pipeline. In other words, we design the pipeline so that it will complete one algorithmic task/iteration in a single burst read of the image. To achieve the aforementioned throughput, we parallelize the calculations required for each pixel, both in terms of arithmetic operations within a formula evaluation, as well as in terms of partial products within a task. That is, considering the latter, we compute in parallel the vertical and horizontal derivatives of the pixel, we compute in parallel the multiple FIRs considered by our adaptive upsampling mechanisms, we compute in parallel the two 1D convolutions of a separable kernel by using its 2D equivalent kernel, etc. The key idea in our parallelization approach is to operate concurrently on a local area of the image and complete a pixel transformation, virtually, in a single cycle. To facilitate such concurrency, we design and integrate in our pipeline a serial-to-parallel buffer, which inputs one sample per cycle (equal to the input rate of the pipeline) and outputs $N \times N$ samples per cycle. Figure 9 depicts the architecture of the proposed buffer interconnecting in series N FIFO memories with $N \times N$ registers. We set the depth of each FIFO to $W - N$ and we feed the buffer with pixels/samples in raster-scan order. As a result, the $N \times N$ output of the buffer will scan the image in a raster-scan order providing one distinct $N \times N$ window per cycle. The proposed design uses the minimum amount of connections (1 input and 1 output per component, excluding the unavoidable output ports of the buffer) and the minimum amount of flip-flops (equal to the number of parallel output ports). Subsequently, in a single cycle, the $N \times N$ samples are forwarded to the processing part of the pipeline, where we perform parallel calculations by using multiple arithmetic units interconnected according to the implemented arithmetic formula. Figure 10, on the left, depicts a processing part performing generic convolution; we use $M = N \times N$ independent constant multipliers, each developed as a fine-grained pipeline with 2–4 stages depending on the width of the x_i inputs, and a fine-grain pipelined adder tree of height $\log M$. This configurable engine can implement blurring, Laplacian, Derivatives of Gaussian, etc. Figure 10, on the right, depicts another processing part performing adaptive upsampling. We use multiple FIRs working in parallel to perform bilinear and

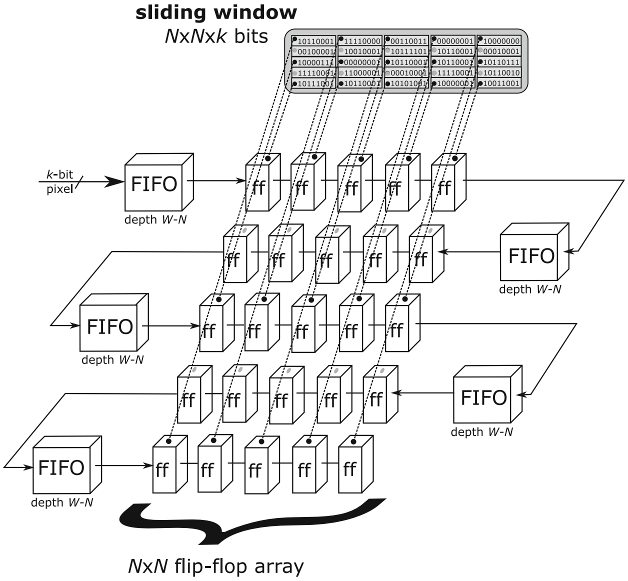


Fig. 9 Proposed architecture of a serial-to-parallel buffer for 2D raster-scanning of the image with a N × N sliding window

bicubic filtering in a fine-grain pipelined fashion. Finally, we select the highest quality output via a multiplexer controlled by the if-clauses detecting the dominant local orientation/edge (according to the rules of the construction phase of *L-SEABI*, or the CxScale technique). Altogether, a long pipeline consisting of the buffer and its processing part, it will input one datum per cycle and it will output one result per cycle in a continuous flow (with very small gaps at regular time intervals corresponding to the wrap-around step of our 2D scanner at the end of each image row). A substantial fraction of the latency is equal to the time required to initially fill the FIFOs of the buffer and is negligible when compared to the total execution time (e.g., 0.4 % when blurring a 1920 × 1080 image with a 5 × 5 kernel).

On top of pixel-based pipelining, we apply pipelining at task level by connecting in series the five distinct engines developed for the five tasks of *SIL-SEABI*. Figure 11 depicts the entire architecture designed in a "continuous flow transformation" approach. That is, every sample/pixel is forwarded directly to the next stage for further processing to complete the entire SIL-SEABI in a single pass of the long pipeline, without any intermediate caching/delays. The architecture features 100 % utilization, and at any time instant, it processes concurrently hundreds of samples throughout its length (within the fine-grain pipelined PUs of the five engines). At the first stage (Fig. 11), we perform an initial upsampling of the image according to *L-SEABI*'s construction phase; we employ a serial-to-parallel buffer (B) to forward 4 × 4 low-resolution pixels per cycle to two processing parts, i.e., to the parallel FIRs doing adaptive upsampling (PUs) and to a smaller pipe accumulating image derivatives and adjusting *L-SEABI*'s threshold according to the total variation of the image (we use the total variation of an image stripe to predict the total variation of the next stripe, on-the-fly). The threshold is returned to the main engine, which outputs a local region of 2 × 2 high-resolution pixels per cycle. The purpose of increasing the pipeline paths to 4 is to avoid multiple clock domains (we now operate in 1/4 clock frequency with enough slack for our critical paths) and also to exploit the parallelization capabilities of HW design. The four remaining stages perform *SIL-SEABI*'s refinement phase. More specifically, at stage 2, we perform low-pass filtering and decimation of the HR image to output one LR pixel per cycle. Notice that the quadruplet input to stage 2 is given to a serial-to-parallel buffer sliding quadrupled words and providing a 3 × 3 quadruplet output, which contains the 5 × 5 pixel region to be blurred; due to this technique, our window slides by two pixels per cycle facilitating the decimation of the image by 2 without decreasing the engine's output rate in half (we achieve 100 % utilization). The error of the reconstructed pixel is forwarded to stage 3, which performs adaptive upsampling similarly to stage 1, however, according to the CxScale rules. The HR results are forwarded to stage 4 in a 2 × 2 format via 4 parallel paths. Stage 4 employs a serial-to-buffer providing 3 × 3 quadruplets per cycle, i.e., a local region of 6 × 6 samples, which itself contains four regions of 5 × 5 samples. In parallel, the four regions are multiplied with a 5 × 5 Laplacian kernel, each, such that the engine will output 4 HR sharpened samples per cycle. At the final stage, we add the 4 HR samples to the 4 HR pixels delayed after stage 1 to refine the HR image and complete the *SIL-SEABI* algorithm. Notice that, depending on the application, the 2 × 2 pixel output of the architecture can be easily transformed to a single raster scan port (with 4× faster clock rate) by employing a RAM buffer to temporarily store two HR image rows. Also notice that the above described architecture can be easily modified to support more iterations of L-SEABI; we can employ an external memory to store the entire image ${}^{i}Y_{\mathrm{HR}}$ output from the pipeline after each iteration i, such that we can bypass stage 1 and feed ${}^{i}Y_{\mathrm{HR}}$ back to the pipeline, together with Y_{LR}.

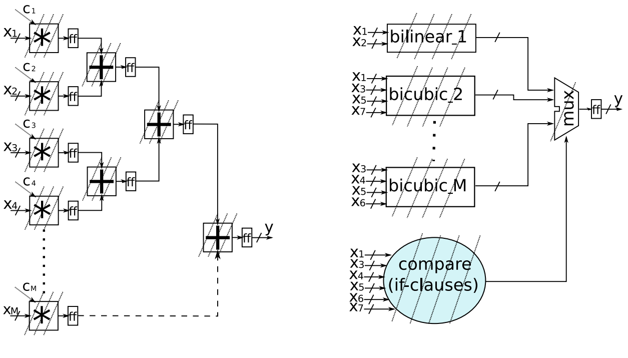


Fig. 10 Architecture of two fine-grain pipelined processing parts which follow a serial-to-parallel buffer to perform 2D convolution (*left*) or adaptive FIR upsampling (*right*)

5.2 Design exploration and implementation results

The proposed architecture was developed in parametric VHDL to allow the straightforward deployment of *L-SEABI* on various diverse FPGA devices and adaptation to distinct application requirements. The parameters include the datapath widths, image size, kernel sizes, and moreover, the parallelization factor P of *L-SEABI* on image level. Parallelization on image level is yet another capability of HW design, besides those described in the previous subsection, which refers to the number of images or image stripes being processed concurrently. That is, we deploy P individual *L-SEABI* pipelines on the FPGA and we feed them with distinct horizontal stripes of the image.

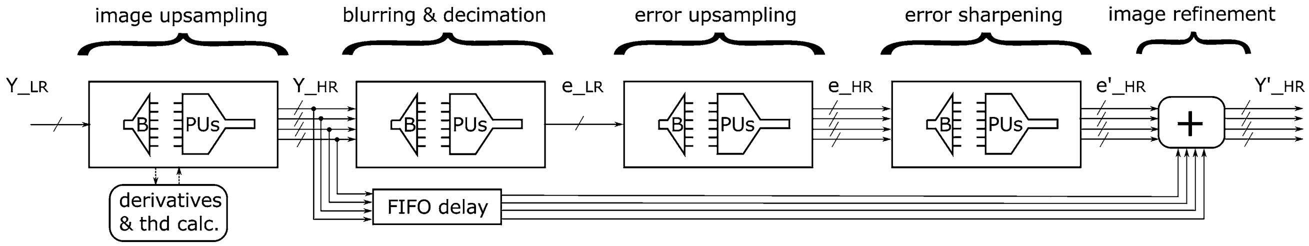


Fig. 11 Architecture of an L-SEABI super-resolution engine with 5 fine-grained pipelines including serial-to-parallel buffers (B) and processing parts with parallel units (PUs). The rate varies from 1 to 4 samples/cycle depending on spatial resolution

Each stripe has height $H/P + O$ pixel rows, where H is the input image height and O is the number of rows used to overlap successive stripes (in order to make the parallel algorithm functionally equivalent to the original execution, we must extend the borders of each stripe to slide the convolution kernels seamlessly between stripes). In the current paper, we assume 8-bit pixels, input image resolution 1920 × 1080, overlapping $O = 9$ due to the size of *L-SEABI*'s kernels, and also, we abstract away the I/O functions of the FPGA to focus on the cost of the actual algorithm in terms of resource utilization (we assume that I/O is handled externally, depending on the application). We show how multiple pipelines can fit in various FPGA devices (with some slack in the utilization, e.g., 100 RAMBs, for other hypothetical components of the application, e.g., the I/O controller). For this purpose, we implement the architecture on 10 FPGA devices representing four technology generations (Xilinx Virtex 5, Virtex 6, Artix and Virtex 7, as well as Ultrascale) and we explore the performance of *L-SEABI* by varying the parallelization factor P from 1 to 29 depending on the size of the underlying FPGA device.

Implementation results are summarized in Table 4, which includes the number of LUTs and RAMB18s as a metric for evaluating the FPGA resources, as well as the total execution time and power required to upsample a 1920 × 1080 image to 3840 × 2160 pixels with the maximum achievable clock frequency. In the upper part of Table 4, we report absolute numbers for a single *L-SEABI* pipeline implemented on low-range low-price devices representing 4 FPGA families/generations (Virtex 5, 6, 7, and Ultrascale). These implementations achieve 120–200 fps while dissipating only 0.5–2.3 W (estimated with the XPower Analyzer tool of Xilinx). The power dissipation for each device depends on its technology node and die size (static power), as well as on the operating frequency and utilization ratio (dynamic power). The maximum frequency for the single-pipeline implementation ranges in 250–420 MHz among devices and decreases to 185–300MHz when placing/routing multiple pipelines in each device (most often decreases to 200–220 MHz). The lower part of Table 4 shows how multiple pipelines can fit in each device. Column 2 reports the maximum pipelines fitted in the device, while columns 3 and 4 report the utilization of the device's LUTs and RAMBs. We note that the bottleneck is usually in the on-chip RAM (except for the large devices, which run out of IOBs).

Table 4 FPGA resources versus execution time (a 1920 × 1080 image is input to various devices with various parallelization factors P and $f = 2$)

Device	P	LUT	RAMB18	Power (W)	Time (ms)
xc5vlx30t	1	3753	64	0.9	8.3
xc6vlx75t	1	3493	64	2.3	6.8
xc7a100t	1	3501	64	0.5	8.2
xcku035	1	3637	64	1	5
xc5vlx110t	4	22 %	89 %	3	2.7
xc5vlx330t	8	15 %	79 %	7.5	1.5
xc6vlx75t	3	22 %	61 %	3	3
xc6vlx240t	11	26 %	85 %	7.8	1
xc6vlx550t	18	18 %	91 %	11	0.71
xc7a100t	3	16 %	71 %	1	3.5
xc7vx485t	16	19 %	49 %	5.5	0.70
xc7v2000t	29	9 %	71 %	11	0.48
xcku035	12	24 %	68 %	6.2	0.65

According to the above results, when increasing the parallelization factor P in each FPGA family, we achieve an almost linear acceleration of *L-SEABI*; we pay a time penalty due to the lower achievable frequency (due to the high utilization ratio of the device) and due to the overhead of overlapping the image stripes to achieve functionally equivalent super-resolution (the neighboring pipelines process partially common inputs). Roughly, for small to large factors P, the acceleration ranges from $\frac{3}{4}P$ to $\frac{1}{2}P$. In the most expensive FPGA device, Virtex7 2000T, the architecture achieves 2083 fps for $P = 29$ (the time overhead of the image level parallelization increases at around 25 % due to the extensive overlapping of the 29 stripes). In these multi-pipeline implementations, the power dissipation ranges from 1 to 11 W, while the energy per image decreases roughly in half when using newer generation FPGAs (e.g., Virtex7 vs. Virtex5/Virtex6). Overall, by using a mid-price xc6vlx240t FPGA, we can achieve 1000 fps with less than 8 W, whereas a low-price xc7a100t FPGA will provide 285 fps for only 1 W. Hence, the proposed acceleration can support a plethora of applications, single- or multi-view, high- or ultra-high definition, achieving real-time and low-power processing.

Compared to other super-resolution implementations on FPGA in the literature, the proposed single-pipeline implementation on xc6vlx240t requires almost 5× fewer logic resources than the similar module in [18]. The proposed 3-pipeline implementation requires similar resources with [19] to achieve one order of magnitude faster execution on the same xc6vlx75t FPGA device. The proposed 11-pipeline implementation on xc6vlx240t consumes a similar number of LUTs with that of the 4-core UHDTV case of [20] to achieve one order of magnitude faster execution than [20] running on Altera Aria II FPGA. Finally, when comparing to more demanding motion-estimation-based SR implementations, the proposed FPGA design proves to be significantly faster and considerably cheaper: for QCIF image upsampling, a single-pipeline on Virtex5 provides up to 400× more fps with around 3× less LUTs than [49] (note however that, in general, motion-estimation based SR provides higher quality results).

6 Application of the proposed acceleration techniques to stereo correspondence algorithms

To improve our evaluation study and support the conclusions drawn from the analysis performed on L-SEABI regarding the efficiency of our acceleration techniques and the comparison of the computational platforms, we extend our work to consider a distinct type of image processing scenario: the stereo correspondence problem which, in general, requires more intensive computations compared to super-resolution. Most often, the dense stereo correspondence algorithms base their execution on intensive block-wise comparisons between two images by following an iterative full-search approach. This distinctive feature will allow us to evaluate our parallelization techniques and HW platforms in a algorithmic case that requires increased data caching and data reuse, while it offers fewer opportunities to do calculations on-the-fly due to limited spatial localities. In the current section, we consider the most representative and well-known algorithm of its category [50], the Disparity.

The Disparity algorithm considered here, has input one stereo image of $2 \times W \times H$ 8-bit pixels, in total, to provide as output a $W \times H$ disparity map with quarter pixel accuracy [51]. Internally, the module produces two $W \times H$ disparity maps, one after the other (the first based on the left image and the second based on the right image of the stereo pair), which are then cross-checked to retain only those disparities that are consistent in the two maps. To produce the left-based map, the algorithm assumes rectified images to operate in a dynamic programming and iterative fashion; at each iteration, the right image is shifted horizontally by one pixel and subtracted from the left image to store $W \times H$ differences in a distinct layer of the *Disparity Space Image* (DSI) structure. After D iterations, where D denotes the amount of the examined disparities/depths, the algorithm completes the DSI construction having stored $D \times W \times H$ values. Subsequently, these values are aggregated within each of the D layers by sliding a 7×7 window over the entire layer and computing a Gauss-weighted sum of every underlying 7×7 group of differences. Each sum represents a similarity metric between a left image region and a right image region, i.e., a means to deduce the apparent displacement of a left image pixel on the right image plane. More specifically, for each pixel of the left image, the algorithm selects the minimum of the D metrics calculated across the depth ray (across the epipolar line of the pixel, or more practically, across the D layers of the DSI) to deduce its disparity result. Finally, to achieve subpixel accuracy, the algorithm refines each disparity result by performing a 1D parabola fitting on the 3 values surrounding the detected minimum (across the depth ray). The right-based map is produced analogously by shifting the left image instead of the right.

6.1 GPU implementation

We design the disparity implementation of CUDA employing distinct kernels to facilitate profiling and following the optimizations described in Sect. 4. Figure 12 shows the kernel execution scheme on our GPU implementation. Initially, we copy the stereo image pair to the Device from page-locked Host memory. Using the kernel $buildRightCube_{\mathcal{K}}$ we compute the difference between the right and left images ($R - L$), creating a cube of D, $W \times H$ slices in the process (D is the number of disparities). Next, the kernel $aggrCube_{\mathcal{K}}$ performs per-slice 3D filtering on the cube by employing a 13×13 non-separable filter. Due to the relatively large filter size, we choose the shared memory scheme of Eklund et. al [44] for this process, and copy the result onto a second cube. In the case of the *GTX 960* we modify the above method to compute 116×52 valid responses with each thread processing 8 values. Afterward, the kernel $cubeMin_{\mathcal{K}}$ searches on the z-axis of the filtered cube to locate the minimum disparity per element in each $W \times H$ slice, i.e., also the distance between two consecutive elements on the z-axis when using a serialized buffer. The output (minimum index) is stored

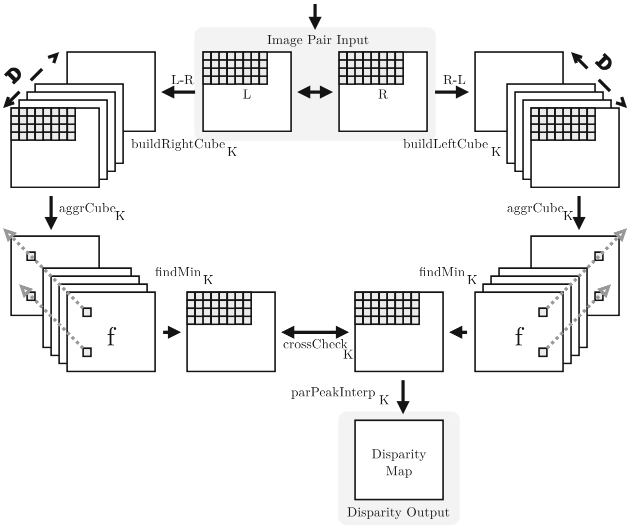


Fig. 12 *Disparity* on CUDA: GPU flow and kernels

into a $W \times H$ buffer. Note that since D is expected to be at least two orders of magnitude smaller than the number of elements in each slice, finding the minimum index does not require a parallel reduction kernel. The same process (*Difference*, *Expansion*, *Convolution* and *Minimum Search*) is repeated for the cube containing the left and right pair $(L - R)$ difference. The kernel *crossCheck*$_\mathcal{K}$, similarly to Fig. 6, compares the indices of minimum disparity as obtained by both the *cubeMin*$_\mathcal{K}$ kernels. The Boolean condition result is then multiplied by the minimum disparity on the left cube so that disparities in non-equal indices are nullified. Finally, through kernel *parPeakInterp*$_\mathcal{K}$ we compute the parabolic peak interpolation of disparities. Overall, the aforementioned implementation employs 2, $W \times H \times D$-byte cubes as only the $L - R$ difference values affect the interpolation result.

We also consider an alternative disparity implementation, which allocates only a single cube. In such a version, we store the convolution result of each cube slice into a separate $W \times H$ buffer and copy the result back to the corresponding memory position inside the cube. This version saves $W \times H \times D \times 4$ bytes of device memory at the expense of D, $W \times H \times 4$-byte D2D transfers. According to our experimentation, all two-dimensional kernels are based on 32×4 threads for block partitioning on the *GTX 670*, *GTX 960* and the 32×8 scheme on the *GTX 550Ti*. The kernels that involve 3D processing use one thread on the z-axis and a number of blocks equal to D. Filtering kernels rely on blocks of 32×32 threads.

6.1.1 *Performance evaluation*

To evaluate the performance of our CUDA disparity implementations, we measure the execution time from *buildRightCube*$_\mathcal{K}$ to *parPeakInterp*$_\mathcal{K}$ excluding D2H/H2D transfers for $10 \leq D \leq 200$ on the aforementioned GPUs. We use 12 images from the Middlebury stereo dataset [52–54], their resolution ranging from 348×288 to 1800×1500 pixels. Specifically, we employ the *books, computer, cones, dolls, drumsticks, dwarves, laundry, moebius, reindeer, teddy, tsukuba* and *vintage* images. Computation time results involving the single $W \times H \times D$ buffer are depicted in Fig. 13. At the lowest resolution of 384×288 pixels, the proposed implementation on the *GTX 670* achieves 3.4 ms for $D = 10$, 8.756 ms for $D = 32$ and 53.07 ms for $D = 200$. In mid-range resolutions such as 1120×1120 computation time ranges from 30.6 ms $(D = 10)$ to 81.28 ms $(D = 32)$ and up to 498.4 ms $(D = 200)$. Computation time displays a linear dependency to both the input resolution as well as the number of disparities; in 1800×1500 images, the same GPU requires 67.16 $(D = 10)$ up to 1119.9 ms $(D = 200)$. The *GTX 550Ti* performs approximately 1.66×–2× slower in all cases. Note that the increased data caching requirements of the algorithm favor the *GTX 960* by 1.4× on average compared to the *GTX 670* as the former features a larger 24KB unified cache, and of course a larger shared memory per Streaming Multiprocessor. Compared to the displayed results, our dual cube model attains a speedup of approximately 1.09× in all GPUs.

Figure 13 also displays the computation time of [12], a method similar to the method proposed by this paper as it computes both depth maps, performs window-based matching using a separable box filter and achieves subpixel accuracy. For a fair comparison, we compiled [12] on the GPUs tested, modifying the window to aggregate 13×13 samples. Our results show that [12] is up to 2.9× faster on 1120×1120 or larger input and up to 3.9× on 384×288 input—an expected result associated with the separable properties of the authors' employed filter and the simple running sum instead of our integer multiplication.

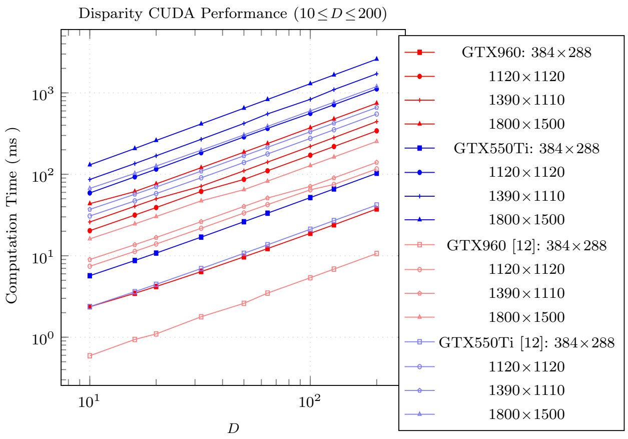


Fig. 13 Execution time of disparity on CUDA

The proposed implementation achieves a performance of 38.48 fps considering inputs of 1390×1110 ($D = 10$) while it surpasses 423 fps for 384×288 images at the same level of disparity.

Table 5 displays the achieved performance of the proposed disparity method as executed on the *GTX 960* compared to GPU implementations of similar algorithms in the literature. All methods involve window-based aggregation, while [15, 55] combine spatial with temporal-based processing. Also, all methods use input images of 320×240 pixels with $D = 32$. For sake of comparison, we also report a relative measure of the performance of the four distinct GPUs in Table 5. The relative GPU performance was measured with a common benchmark using the median values of the *TV-L1 Optical Flow* results reported online in [56] and the GTX 960 as baseline. (We acknowledge here that this ratio does not suggest an absolute indication of the relative GPU performance, as the listed GPUs represent multiple architectural generations).

6.2 FPGA implementation

To accelerate the Disparity algorithm on FPGA, we develop an architecture with pipelining on pixel basis, parallel arithmetic calculations and, most importantly, twofold partitioning of the DSI cube to achieve on-chip memory minimization. Specifically, we exploit the fact that detecting the minimum metric value over the D candidates is accomplished via a winner-takes-all procedure and that the images are rectified. The former allows us to proceed iteratively in computing each of the D layers of the DSI cube and updating only those metric values that are better than the currently detected minimum. Thus, we only store the $W \times H$ global minima instead of the entire DSI cube (effectively, one layer instead of D). Rectification allows us to divide the image in s horizontal stripes of size $W \times H/s$ and process them independently, one after the other. Notice that successive stripes must overlap by $M/2 - 1 = 3$ rows to allow the $M \times M$ aggregation mask to slide seamlessly between stripes. The above twofold partitioning of the DSI cube leads to storing only $W \times H/s$ values instead of $D \times W \times H$, i.e., to memory optimization by 4 orders of magnitude.

Table 5 Frame rate of disparity on GPU

GPU	Relative GPU power[a]	Method	fps
GTX 960	1	This work	225
GTX 960	1	[12] (Box filter)	807
GTX 680	0.966	[15]	90
GTX 580	0.789	[14]	62
GTX 480	0.693	[55]	24

[a] Estimated according to the ratings in http://compubench.com/ based on the *TV-L1 Optical Flow* benchmark performance

Following the design approach described in the previous section, we develop a pipeline that reads one pixel per cycle from each image in a raster-scan order to ultimately produce one similarity metric per cycle and construct any i-th layer of the DSI cube, $0 < i < D$ (also in raster scan order). The pipeline proceeds in i successive iterations, one for each DSI layer. At each iteration, we start reading the pixels of the left image at position $x = 0$ and the pixels of the right image at $x = -i$ to create a virtual displacement by i pixels. As described for the convolution components in the previous section, we sustain a throughput of 1 metric/cycle by fully parallelizing the 7×7 Gauss mask multiplication and developing a 7×7-to-1 pipelined adder tree. This parallel arithmetic circuitry is preceded by a 1-to-7×7 serial-to-parallel buffer, which inputs sequentially the differences of the left and right image pixels. The metric values are forwarded from the adder tree to a *map updating* component, which stores only the minimum metric value per pixel and compares it to the newly computed value for any possible update. Additionally, to support the 3-tap interpolation, the *updating* component stores the previous of the minimum value (temporarily stored during the previous iteration) and performs on-the-fly interpolation when the third tap/value arrives (during the next iteration). Therefore, we double the storage requirements (effectively, we store two layers of the DSI) to perform on-the-fly interpolation. The left-based disparity results are stored on-chip, and the pipeline resources are reused to compute the right-based map via D new iterations. At a final step, we perform a left-right consistency check of the two maps to output the final results. Having completed the disparity map of a $W \times H/s$ stripe, we continue with the next stripe by reusing the HW resources until the entire image is processed. More details regarding the proposed architecture can be found in [51].

The aforementioned disparity module was realized using parametric VHDL on a Xilinx Virtex xc6vlx240t-2 FPGA device. For image size $1120 \times 1120, D = 200$ disparity levels, and stripe size 1120×28, we get a HW cost of 998 slices (3 %), 2978 LUTs (2 %), 3116 registers (1 %), 0 DSPs, and 101 RAMB36 (24 %). If we enlarge the aggregation mask from 7×7 to 13×13, the HW cost will increase to 2690 slices (8492 LUTs and 8640 registers) showing that the most expensive component is the parallel arithmetic circuit performing aggregation (depending on mask size, aggregation consumes 60–85 % of the utilized logic resources). Achieving up to 344 MHz clock frequency, a single disparity module will process the entire 1120×1120 image pair in 1.87 s (time is almost independent of the mask size due to the applied full mask parallelization). If we further parallelize the architecture at image level, i.e., employ multiple disparity modules to process multiple stripes concurrently (as already described in the previous section for L-SEABI), we can decrease the execution time down to 0.54 s, with approximately 8.8 W, for $f = 4$ modules on xc6vlx240t-2. Compared to other similar designs in the literature, the proposed single-module architecture proves to be cost-efficient, i.e., it consumes 20× fewer LUTs and 0 DSPs and 1/3 RAMBs compared to [22] (which however examines only 64 disparity levels in only 4 ms), as well as 24× fewer LUTs and 0 DSPs instead of 625 DSPS compared to [23] (which however can process 1080p images at 30 fps).

7 CPU, GPU and FPGA comparison

In this section, we will comparatively evaluate the performance of the proposed acceleration implementations, taking into account the output quality, the power envelope and the cost of the underlying platform. To that end, we will compare performance on the aforementioned GPU and FPGA devices to a common reference point: the performance as measured on general purpose multi-core x86_64 CPUs. In all of our experiments, we upsample each LR image by a factor $f = 2$ in both dimensions in order to obtain output images, which have the same size as the ground truth. We will first compare the output image quality and then the acceleration performance.

7.1 L-SEABI quality results

As presented in Sects. 4.2.1, 4.2.2 and 5.1, the accelerated implementations of *SIL-SEABI* include both arithmetic (i.e., Gaussian filtering coefficients) and algorithmic approximations, such as the coarse-grain averaging of HD output samples (GPU implementation), or the stripe-based total variation computation (FPGA implementation). To assess the impact of these approximations, we compare the image output quality of the accelerated implementations to that of the CPU-based implementation used as reference (same as in Table 1). Results (Table 6) reveal an MSSIM and BRISQUE quality that is similar to that of the reference for both the GPU-based and the FPGA-based implementations. As expected, increasing the number of stripes in the FPGA implementation does incur a slight quality degradation, which though remains negligible even when processing QCIF resolution images using 29 horizontal stripes.

7.2 L-SEABI acceleration results: CPU analysis

To assess the proposed multi-core CPU execution, we employ a fully vectorized *SIL-SEABI* model which exploits the implicit multi-threading of MATLAB functions. Moreover, we consider an implementation based on single program multiple data (*spmd*) statements in which we explicitly define the number of workers to be equal to the CPU cores. In this model, we partition the image based on the number of workers and employ the *labSend*, *labReceive* functions of MATLAB for message passing during computations. We implement *SIL-SEABI* on the *Atom 330* dual-core, the *Core i5-3470* quad-core and the *FX-8120* octa-core processors. In the *spmd* cases, we measure computation time up to and including the image reassembling step.

To provide a relative graphical overview among platforms, computation time results per output resolution are plotted in Fig. 14 for all the CPU, GPU and FPGA devices tested. Additionally, to summarize the measurements in a more accurate manner, Table 7 shows the achieved speedup per platform against the fastest CPU-based implementation.

As expected, the lowest performance—approximately 10 s for upsampling to UHD—is achieved by the *Atom 330*, which features 1 MB of L2 cache and it does not support out-of-order or speculative execution. The quad-core *i5-3470* achieves sub-second performance in almost all cases and hence, it is used as the baseline for speedup comparisons. When outputting to SD1 or higher resolutions, the *i5-3470* computes the result earlier than the octa-core *FX-8120* mainly due to the latter's increased memory latency. Notice that on a different level of comparison, the

Table 6 Quality performance of the *SIL-SEABI* implementations on CPU, GPU and FPGA platforms

SIL-SEABI implementations quality					
Platform	CPU	GPU	FPGA		
Output size	Ref.	Opt.	8 str.	16 str.	29 str.
MSSIM					
QCIF	0.9086	0.9032	0.9085	0.9084	0.9074
CIF	0.8743	0.8701	0.8740	0.8744	0.8740
SD1	0.9679	0.9681	0.9703	0.9702	0.9704
720p	0.9567	0.9568	0.9593	0.9592	0.9592
1080p	0.9784	0.9784	0.9798	0.9797	0.9797
2160p	0.9879	0.9871	0.9879	0.9879	0.9879
Avg	0.9456	0.9440	0.9467	0.9467	0.9464
BRISQUE					
QCIF	43.5880	41.5805	42.5290	40.9607	40.4722
CIF	36.1080	35.9498	36.4587	36.4763	36.8599
SD1	44.4549	41.8728	44.0211	44.2512	44.0111
720p	43.5580	41.7760	43.3747	43.4144	43.4126
1080p	42.0595	41.6913	42.0178	42.0280	42.0207
2160p	57.9444	55.9248	56.8053	56.9504	57.0885
Avg	44.6188	43.1269	44.2011	44.0135	43.9775

spmd message-passing modification increases MATLAB's implicit multi-threading performance by 1.09× and up to 1.28× when upsampling to 720p or higher resolutions on the *i5-3470* (and by up to 1.38× for 2160p output on the other processors); in lower resolutions, the communication overhead is much higher than the actual time spent for arithmetic computations and it would be thus preferable to rely on MATLAB's implicit multithreading in such cases. When we apply upsampling to input images of 6750 × 6750 pixels, computation time increases almost linearly, i.e., the *spmd FX-8120* model completes in 21.39 s.

7.3 L-SEABI acceleration results: GPU analysis

When evaluating GPU acceleration against the fastest CPU results (i.e., the *i5-3470* baseline in Table 7), we measure a minimum speedup of 69.7× on the mid-end *GTX 550Ti* for 720 × 576 output. The high-end *GTX 670* attains a speedup of up to 285.8× as it requires 2.89 ms on average when upsampling to 2160p. Due to memory restrictions, a further increase of the input size will limit the number of GPU devices, which are capable of such intensive processing: for instance, the *GTX 960* requires 347.55 ms to upsample a 6750 × 6750 image by 2.

7.4 L-SEABI acceleration results: FPGA analysis

To facilitate our discussion of the results of the Table 7 and Fig. 14, we denote a single engine fitted on the xc5vlx30t-1 FPGA as V5-30-1 (1), 11 engines on the xc6vlx240t-2 as V6-240-2(11), 16 engines on the xc7vx485t-2 as V7-485-2(16), 12 engines on the xcku035-2 as U-035-2(12) and so on. As shown in Table 7 and Fig. 14, the achieved speedup on a FPGA is significantly higher than the aforementioned devices, especially for 88 × 72 input where the ratio of memory to arithmetic operations is high: even a single engine on the xc5vlx30t-1 can upsample a 88 × 72 image to 176 × 144 in 0.026 ms, i.e., a speedup of 1283× (Table 7). When upsampling to 1080p or larger images, the low-end Virtex 5 single-engine *SIL-SEABI* implementation performs close to the *GTX 550Ti*, as they also do the 3-engine xc7a100t-1, xc6vlx75t-1 and the 4-engine xc5vlx110t-1 FPGAs against the higher-end GPUs i.e., when upsampling to 2160p. The implementation of 12 parallel engines on the xcku035-2 provides the highest performance for 176 × 144 output while the 29 engines of the xc7v2000t-2 achieve the highest speedup when upsampling to 2160p from 1080p, i.e., 6160× and 1716×, respectively, as Table 7 displays. Note that a significant increase in the input size to 6750 × 6750 pixels restricts both the range of suitable devices and the degree of FPGA parallelism, as a single engine now requires 252 RAMBs. Moreover, the performance gap between GPUs and FPGAs narrows—even though the latter still maintain their advantage; the xc6vlx240t-2, now supporting up to 3 engines, produces a 13,500 × 13,500 output in up to 80.35 ms. Higher-end devices can reach real-time performance such as the xc7vx485t-2, which, when configured with 8 engines can achieve 33 fps (28.9 ms using 2016 RAMB modules). Notice that this performance translates to real-time processing of 182 Mpixel images, i.e., over 2 times the resolution of upcoming 11K (11,520 × 6480) monitors.

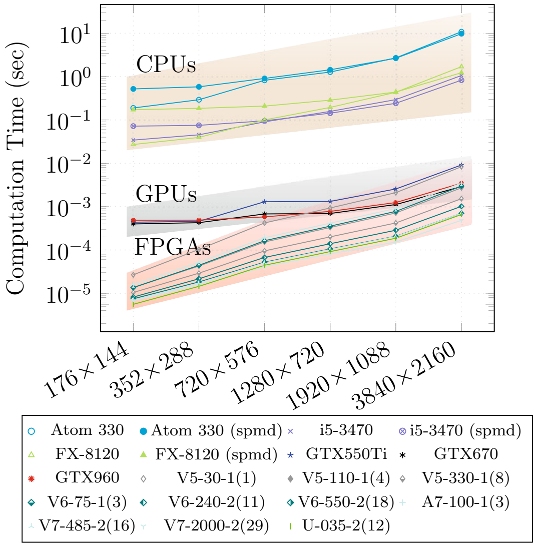


Fig. 14 Overview of *SIL-SEABI*'s performance per platform and output resolution: computation Time required to upsample each input image by $f = 2$

Overall, the general purpose processors reach an absolute upsampling performance which ranges between 2.72 µs/pixel on the dual-core *Atom 330* and 0.4 µs/pixel on the higher-end CPUs. GPUs provide two orders of magnitude higher performance (0.0048 and up to 0.0038 µs/pixel), while on FPGAs, the modular design of the architecture combined with a high degree of parallelism can accelerate the execution even further than GPUs (i.e, by another order of magnitude at 0.12 ns/pixel on the 7-2000-2).

7.5 Disparity results: validation of the proposed acceleration techniques

Regarding the acceleration of the presented here disparity algorithm, the minimum GPU speedup against CPU-based execution (*GTX 960* against the *i5-3470*) for $D = 200$ can rise up to 50.33× in 1800 × 1500 images and is 45.5× in 1120 × 1120 input for which the xc6vlx240t-2 can accelerate CPUs by 28.8× when employing 4 parallel modules.

Table 7 Acceleration performance of the *SIL-SEABI* implementations on CPU, GPU and FPGA platforms

SIL-SEABI implementations speedup							
Platform	Output resolution ($f = 2$)						
	QCIF	CIF	SD1	720p	1080p	2160p	Avg
i5-3470	1	1	1	1	1	1	1
FX-8120	1.26	1.15	0.90	0.73	0.55	0.49	0.85
GTX550Ti	78.54	98.95	69.70	108.79	94.68	90.66	90.22
GTX670	86.04	106.5	133.2	206.3	216.4	285.8	172.4
GTX960	70.58	93.01	156.1	184.6	194.9	238.2	156.2
V5-30-1(1)	1283	440.8	215.1	153.9	115.1	99.5	384.6
V5-110-1(4)	2517	1065	591.2	435.2	338.6	304.8	875.3
V5-330-1(8)	3281	1558	948.7	714.9	576.2	542.7	1270
V6-75-1(3)	2530	1027	553.3	404.5	311.5	277.5	850.6
V6-240-2(11)	4191	2101	1347	1029	848.1	814.6	1722
V6-550-2(18)	4589	2485	1733	1358	1167	1181	2086
A7-100-1(3)	2161	877.0	472.6	345.5	266.1	237.0	726.5
V7-485-2(16)	4905	2610	1783	1389	1181	1179	2174
V7-2K-2(29)	5039	2905	2191	1764	1588	1717	2534
U-035-2(12)	6160	3132	2037	1564	1297	1256	2574

7.6 Platform comparison: power versus performance

To enhance the thoroughness of our evaluation, we also perform a joint power–performance assessment by measuring the power consumed by each platform when executing the *SIL-SEABI* algorithm. Regarding CPUs and GPUs, we measured the power consumption for iterative execution (i.e., maximum 10^6 loops, until fluctuation became negligible) by using a power meter at an ambient temperature of 25 degrees Celsius and averaged results over all image content. Regarding FPGAs, power consumption was estimated at the same ambient temperature using the *Xilinx Power Estimator* (XPE) tool. The results are plotted in Fig. 15 juxtaposed against the time required to upsample the input from 1080p to 2160p. Figure 15 also presents a rough clustering of the measured performances, which clearly shows the gap between CPUs, GPUs and FPGAs using shaded areas (brown for CPUs, gray for GPUs and light red for FPGAs).

Among the examined CPUs, the *Atom 330* has a power envelope similar to that of the FPGAs: it consumes approximately 3 W when idle and its total consumption ranges between 5.1 W (QCIF) up to 6.5 W (UHD). On the other hand, it requires at least three orders of magnitude more computation time, thus resulting in a performance of 0.091 Msamples/W. Between the other two processors, the octa-core *FX-8120* consumes almost two times the power of the quad-core *i5-3470*: the former requires up to 97.5 W in 1080p to UHD upsampling, compared to the latter which peaks at 50.78 W. Their respective performance is estimated at 0.042 and 0.112 Msamples/W.

GPUs feature a similar power envelope, ranging from 67 W (*GTX 960*) up to 76 W (*GTX 670*), yet at two orders of magnitude less time. At 21.66 Msamples/W, the *GTX 960* is the most power efficient followed by the *GTX 670* at 20.69 and the *GTX 550Ti* at 8.32 Msamples/W.

As expected, FPGAs prove to be the most power efficient among all the devices tested. Notice that even though their computation time is directly comparable to that of GPUs, they consume at least an order of magnitude less power. Notice also (left side of Fig. 14) that there is an increased performance gap between the two platforms for small images, which can be attributed to the low GPU occupancy resulting in idle cores. Additionally, the plots tend to converge (right side of Fig. 14 and Table 7) as the significantly increased memory requirements of large image inputs push the limits of FPGA Block RAM resources, while GPUs on the other hand can have ample memory space (e.g., 4GB on the *GTX 960*). Regarding the joint power-performance assessment, low-end FPGAs are the most prominent: the 3-engine xc7a100t-1 achieves the highest performance of 2060.4 Msamples/W, followed by the 7-485-2(16) at 1460.37 Msamples/W and the xcku-035-2(12) at 1453.67 Msamples/W. Our disparity assessment verifies the above results as presented on the inset plot of Fig. 15 for 1120 × 1120 input; the *GTX 960* consumes approximately 80 W thus reaching a performance of 9.17 MDEs/W (millions of disparity estimations per Watt) and while the xc6vlx240t-2 is approximately 1.6× slower, it

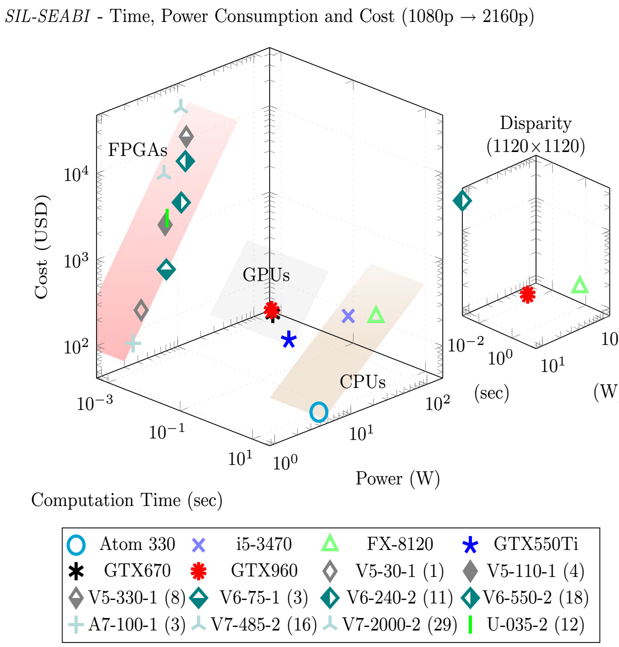


Fig. 15 Time versus power while performing super-resolution (*SIL-SEABI*) and stereo correspondence (disparity). The figure includes various devices with their respective cost of ownership (USD) and distinctive aliases (notation explained in text)

only consumes 8.8 W which translates to a performance of 51.8 MDEs/W. Contrastingly, the corresponding performance of the *FX-8120* is 0.11 MDEs/W.

From the energy perspective, aggregating the power and performance results in Table 8 illustrates the distinction between platforms in an unambiguous manner. When considering the Joules required to generate each output sample, there is a two orders of magnitude difference between CPUs and GPUs and between GPUs and FPGAs. Notice also that fluctuation recedes as the image size increases, resulting in a more deterministic behavior and thus less wasted energy. Moreover, energy efficiency tends to increase in the newer generation devices (i.e., *i5-3470, GTX 960* and the 7-series FPGAs).

7.7 Platform comparison: development cost

Figure 15 also plots (z-axis) the average cost in USD of each integrated circuit (IC) for 1000-unit quantities at the time of writing of this manuscript.[1] Notice that FPGAs generally have the highest cost of ownership: though for the low-power A7-100-1, it can be less than \$140; in mid-level ICs such as the V5-110-1 and V6-240-2, it rises to approximately \$2300, while the high-end V6-550-2 and V7-2000-2 cost \$6206 and \$23,838, respectively. Taking into account, the ownership cost allows us to obtain a more comprehensive view. For instance, compared to the *GTX 670*, the 7-485-2(16) demonstrates an increased power efficiency and an increased ownership cost by approximately the same factor, which leads to an estimation of a similar combined performance-power-cost index, i.e., 0.2 Msamples/(W USD). The highest combined performance-power-cost index is achieved by low-power FPGA implementations such as the 7-100-1(3) and 5-30-1(1), at 14.7 and 2.7 Msamples/(W USD), respectively.

Gathering all the results in the joint assessment plot of Fig. 16 allows us to observe that all platforms tend to increase their combined index as transistor size shrinks. This performance increase can be attributed to the availability of more transistors at the same power envelope and at a lower production cost. GPU ICs feature a combined index which is more than two orders of magnitude higher than that of general purpose processors. FPGAs are able to maintain their performance advantage over all platforms even when the cost of ownership is considered. On the other hand, high-end FPGA solutions require a much steeper premium than the additional performance they're offering compared to their mid-end counterparts. Notice that GPUs tend to offer the same performance with the FPGAs ranging from low to mid-end and to approach the combined index of mid-end FPGA devices. At the same time, low-power FPGAs increase both their available resources and performance while lowering their cost of ownership (Table 4; Fig. 16).

The total design and development time is also distinct for each platform; assuming development starts from square one, the GPU implementations per algorithm were completed in approximately one person-month including optimizations, while FPGA implementations require approximately a triple effort. Based on our assessment of the examined algorithms and their implementation, high-end FPGAs can be an order of magnitude faster than GPUs. When jointly evaluating performance and power consumption, small-scale FPGA devices can be up to two orders of magnitude more power-efficient than current high-end GPUs. The same analogy holds between the latter and multi-core general purpose CPUs. Finally, when we additionally factor the IC cost into our assessment, the results disclose that small-scale FPGAs and GPUs constitute the better choice.

8 Conclusion

The current paper presented accelerating techniques for super-resolution and image processes. Aiming at the improvement of their performance with respect to the

[1] As recovered on-line mainly using the http://octopart.com search engine (April 2016).

Table 8 Energy evaluation of the *SIL-SEABI* implementations on CPU, GPU and FPGA platforms

SIL-SEABI implementations energy (*n*J/sample)							
Platform	Output resolution ($f = 2$)						
	QCIF	CIF	SD1	720p	1080p	2160p	Avg
Atom 330	38,065	16,679	11,503	8021	8410	7660	15,056
i5-3470	59,899	20,104	10,013	7393	5696	5069	18,029
FX-8120	97,206	35,224	22,074	19,490	19,680	14,456	34,689
GTX550Ti	784.1	206.6	167.6	80.11	78.06	80.88	232.9
GTX670	865.1	245.2	104.2	53.12	38.18	37.94	223.9
GTX960	652.5	167.9	51.71	32.88	26.17	28.92	160.03
V5-30-1(1)	0.927	0.899	0.890	0.888	0.885	0.882	0.895
V5-110-1(4)	1.624	1.279	1.113	1.079	1.033	0.988	1.186
V5-330-1(8)	3.097	2.174	1.724	1.633	1.509	1.389	1.921
V6-75-1(3)	1.601	1.315	1.179	1.151	1.113	1.076	1.239
V6-240-2(11)	2.530	1.683	1.268	1.184	1.070	0.959	1.449
V6-550-2(18)	3.119	1.920	1.330	1.211	1.050	0.893	1.587
A7-100-1(3)	0.639	0.525	0.471	0.459	0.444	0.429	0.495
V7-485-2(16)	1.530	0.958	0.677	0.621	0.544	0.469	0.799
V7-2K-2(29)	2.957	1.709	1.095	0.970	0.803	0.640	1.362
U-035-2(12)	1.379	0.904	0.671	0.624	0.560	0.498	0.773

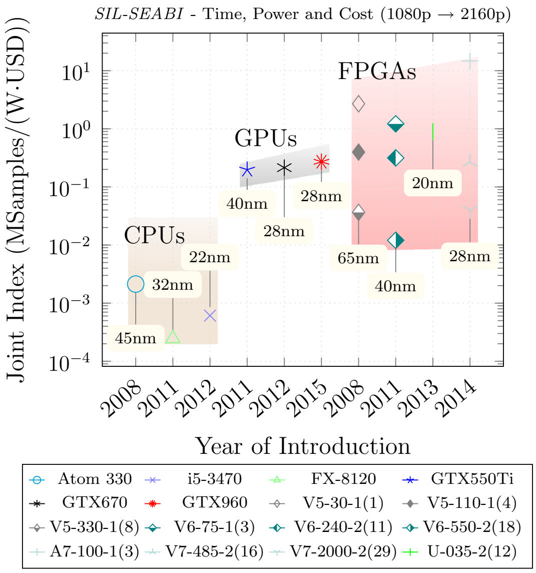


Fig. 16 Combining *SIL-SEABI*'s performance and cost on various platforms

execution time, this work has based on the low-complexity results of the *L-SEABI* method and it introduced parallelization techniques and their consequent optimizations for application on GPUs and FPGAs.

The proposed GPU acceleration techniques proved to constitute a powerful methodology for a wide range of GPU architecture generations, including the latest. Applied on multiple abstraction levels, spanning from the design phase to the implementation API, they were designed to exploit the GPU architectural features and they are able to combine increased throughput, instruction-level parallelism, with decreased latency and divergence. Compared to the conventional real-time performance of 30 frames/s, the proposed GPU techniques accelerate the reconstruction of Ultra-High Definition content to 109 fps on mid-range and early generation devices and 345 fps on the currently available higher-end GPUs.

The proposed parameterizable and highly scalable *L-SEABI* FPGA architecture was evaluated for a variety of parallelization factors as well as FPGA devices. Optimizing of the pipelining at both pixel and task-level led the proposed architecture to perform four (4×) times faster than the conventional real time requirement on earlier generation and low-end Virtex 5 devices and at most 69 (69×) times faster than real-time on the powerful Virtex 2000t.

Furthermore, this work presented the results of the comparative evaluation of the performance of *SIL-SEABI* among CPU, GPU and FPGA implementations with respect to the power dissipation of each platform. Therefore, it provided a graphical representation of the achieved performance per Watt showing an overview of the relative power efficiency per platform.

Finally, the current paper consolidated the results of the SR study by applying the proposed acceleration strategy to a disparity algorithm for computing a depth map based on metric aggregations with non-separable filters, left-right consistency checks and sub-pixel accuracy estimations. For this problem, it introduced a GPU implementation aiming at assessing bottlenecks through kernel profiling and an FPGA architecture targeting memory requirements reduction. The proposed GPU disparity implementation achieved an acceleration of at least 14× over the fastest CPU (*i5-3470*) on the mid-range *GTX 550 Ti* for 1120 × 1120 input and 200 disparities, while the proposed FPGA architecture showed an acceleration of 29× over the same CPU. The power-performance results of Disparity verified the comparative evaluation of the platforms based on the *L-SEABI* algorithm.

Appendix

In this section, we provide the results of our entire super-resolution enhancement evaluation, in tabular (Table 9) and image form (Fig. 17). As Table 9 shows, when we

Table 9 Per resolution objective comparison of state-of-the-art SR algorithms (scaling factor $f = 2$) when using *L-SEABI* (a), *SIL-SEABI* (b) and *L-SEAI* (c) as their initial reconstruction phase against the parameters proposed by their authors

Metric	Initialization	Algorithm					
Output size		NLIBP [35]	ANR [2]	Yang et al. [32]	NARM [36]	DPSR [9]	ASDS-AR-NL[33]
$\Delta\text{MSSIM}_{(\text{enhanced}-\text{original})}$							
QCIF	(a)	−0.13793	−0.02854	−0.02495	0.00337	0.01224	0.00242
	(b)	−0.14526	−0.03719	−0.03268	0.00321	0.01208	0.00223
	(c)	−0.04064	−0.00560	−0.00984	0.00469	0.00918	0.00178
CIF	(a)	−0.13655	−0.02631	−0.01849	0.00220	0.01481	−0.00021
	(b)	−0.13367	−0.03043	−0.02266	0.00208	0.01444	−0.00011
	(c)	−0.03763	−0.00423	−0.01087	0.00235	0.01121	0.00138
SD1	(a)	−0.06806	−0.01726	−0.01665	0.00116	0.00225	0.00160
	(b)	−0.06947	−0.01901	−0.01864	0.00119	0.00214	0.00155
	(c)	−0.02102	−0.00902	−0.01039	0.00113	0.00164	0.00150
720p	(a)	−0.07074	−0.01943	−0.01730	0.00116	0.00276	0.00171
	(b)	−0.07234	−0.02162	−0.01914	0.00113	0.00267	0.00169
	(c)	−0.01511	−0.01082	−0.01407	0.00114	0.00206	0.00164
1080p	(a)	−0.05187	−0.01130	−0.01427	0.00128	0.00087	0.00168
	(b)	−0.05229	−0.01223	−0.01473	0.00126	0.00083	0.00168
	(c)	−0.02102	−0.00564	−0.00973	0.00125	0.00064	0.00165
2160p	(a)	−0.02498	−0.05470	−0.01093	−0.00005	0.00037	–
	(b)	−0.02561	−0.00592	−0.01102	−0.00004	0.00035	–
	(c)	−0.00753	−0.00282	−0.00866	−0.00004	0.00029	–
Avg	(a)	−0.08169	−0.01805	−0.01710	0.00152	0.00555	–
	(b)	−0.08311	−0.02106	−0.01981	0.00147	0.00542	–
	(c)	−0.02379	−0.00636	−0.00106	0.00175	0.00417	–
$\Delta\text{BRISQUE}_{(\text{enhanced}-\text{original})}$							
QCIF	(a)	−7.41320	−3.68780	−3.47380	−0.64420	−7.93090	−0.61830
	(b)	−2.62820	−0.57520	−3.28160	−2.81310	−7.48240	−0.55620
	(c)	−6.34600	0.55840	2.81730	−2.56050	−7.89660	−0.12090
CIF	(a)	7.80928	−2.73474	−5.99424	−0.42284	−3.12196	−0.17576
	(b)	9.06760	−2.09434	−5.41352	−0.15354	−2.88236	−0.03572
	(c)	0.66116	−10.2587	−4.09130	−0.36940	−3.10202	0.08386
SD1	(a)	−11.7947	−4.39134	−4.0984	−0.15325	−0.068348	−0.02612
	(b)	−6.3031	−2.77888	−2.26036	−0.27947	−0.73072	−0.02005
	(c)	−6.7559	−0.06337	1.44172	−0.16850	−0.86090	−0.03902
720p	(a)	−8.77396	−2.94443	−6.10462	0.05638	−0.93106	−0.0634
	(b)	−5.48932	−2.42523	−4.26859	−0.38838	−0.83522	−0.07618
	(c)	−5.10736	−0.39096	−0.25632	0.09672	−0.79050	−0.03506
1080p	(a)	−7.86562	−7.99265	−10.8402	1.36240	−1.83855	−0.01110
	(b)	−4.88077	−7.44972	−9.51325	0.48452	−1.76727	0.00032

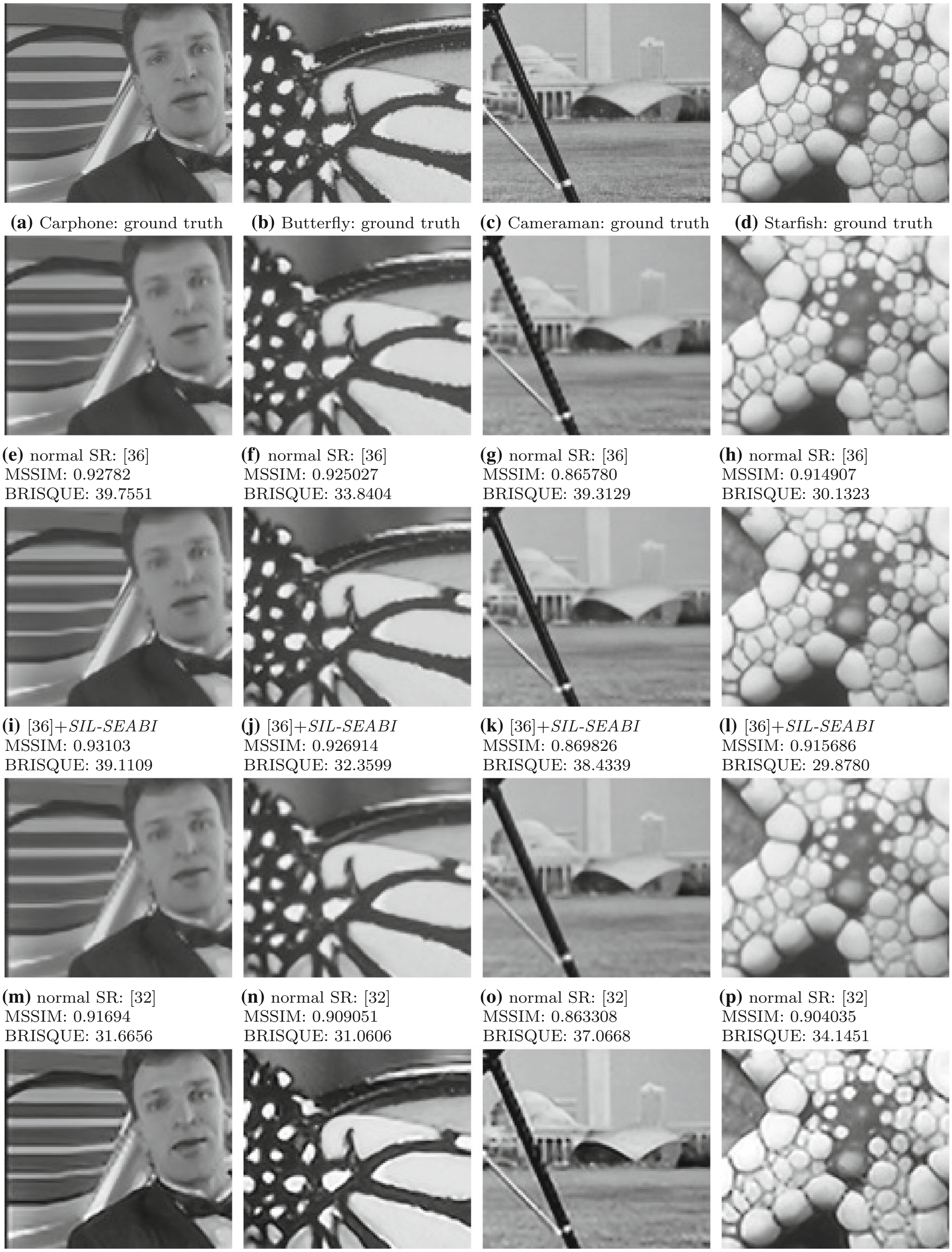
Table 9 continued

Metric	Initialization	Algorithm					
Output size		NLIBP [35]	ANR [2]	Yang et al. [32]	NARM [36]	DPSR [9]	ASDS-AR-NL[33]
	(c)	−3.97212	−2.69070	−3.31287	0.70232	−1.56985	−0.00012
2160p	(a)	−14.1632	0.58664	−5.16501	−0.40221	−0.40991	–
	(b)	−11.7626	1.43108	−3.87191	−0.48665	−0.33370	–
	(c)	−7.66023	1.50892	−1.82637	−0.17654	−0.28128	–
Avg	(a)	−7.03358	−3.52738	−5.94605	−0.03395	−2.48597	–
	(b)	−3.66607	−2.31538	−4.76820	−0.60610	−2.33862	–
	(c)	−4.86342	−1.88940	−0.87130	−0.41265	−2.41686	–

Lower ΔBRISQUE indicates higher quality

employ our algorithms prior to the technique presented in [35]. the MSSIM metric recedes, particularly on 352 × 288 resolutions. For this particular resolution, the *BRISQUE* results show that *L-SEAI* can have superior *ANR* enhancing performance than both *L-SEABI* and *SIL-SEABI*.

Apart from the *Cameraman* image, Fig. 17 subjectively assesses the output of [32, 36] when processing the 176 × 144 *Carphone* and 256 × 256 *Butterfly* and *Starfish* images. According to the results, the aliasing reduction effects of *SIL-SEABI* when it is applied before *NARM* are also apparent in the *Carphone* and *Butterfly* images (Fig. 17i, j). Finally, notice that *SIL-SEABI* improves the contrast of all images upsampled by [32] (Fig. 17q–t).



(a) Carphone: ground truth

(b) Butterfly: ground truth

(c) Cameraman: ground truth

(d) Starfish: ground truth

(e) normal SR: [36]
MSSIM: 0.92782
BRISQUE: 39.7551

(f) normal SR: [36]
MSSIM: 0.925027
BRISQUE: 33.8404

(g) normal SR: [36]
MSSIM: 0.865780
BRISQUE: 39.3129

(h) normal SR: [36]
MSSIM: 0.914907
BRISQUE: 30.1323

(i) [36]+*SIL-SEABI*
MSSIM: 0.93103
BRISQUE: 39.1109

(j) [36]+*SIL-SEABI*
MSSIM: 0.926914
BRISQUE: 32.3599

(k) [36]+*SIL-SEABI*
MSSIM: 0.869826
BRISQUE: 38.4339

(l) [36]+*SIL-SEABI*
MSSIM: 0.915686
BRISQUE: 29.8780

(m) normal SR: [32]
MSSIM: 0.91694
BRISQUE: 31.6656

(n) normal SR: [32]
MSSIM: 0.909051
BRISQUE: 31.0606

(o) normal SR: [32]
MSSIM: 0.863308
BRISQUE: 37.0668

(p) normal SR: [32]
MSSIM: 0.904035
BRISQUE: 34.1451

(q) [32]+*SIL-SEABI*
MSSIM: 0.88426
BRISQUE: 28.3840

(r) [32]+*SIL-SEABI*
MSSIM: 0.860352
BRISQUE: 26.8966

(s) [32]+*SIL-SEABI*
MSSIM: 0.837666
BRISQUE: 33.3080

(t) [32]+*SIL-SEABI*
MSSIM: 0.879176
BRISQUE: 29.1027

Fig. 17 Subjective comparison of [32, 36]: normal execution and enhanced with *SIL-SEABI* ($f = 2$)

References

1. Yang, J., Huang, T.: Digital Imaging and Computer Vision. CRC Press, Boca Raton (2010)
2. Timofte, R., De Smet, V., Van Gool, L.: Anchored neighborhood regression for fast example-based super-resolution. In: International Conference on Computer Vision (ICCV 2013) (2013)
3. Dong, C., Loy, C., He, K., Tang, X.: Learning a deep convolutional network for image super-resolution. In: Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T. (eds.) Computer Vision ECCV 2014, Volume 8692 of Lecture Notes in Computer Science, pp. 184–199. Springer, Berlin (2014)
4. Dong, C., Loy, C., He, K., Tang, X.: Image super-resolution using deep convolutional networks. IEEE Trans. Pattern Anal. Mach. Intell. **38**(2), 295–307 (2016)
5. Timofte, R., De Smet, V., Van Gool, L.: A+: adjusted anchored neighborhood regression for fast super-resolution. In: Cremers, D., Reid, I., Saito, H., Yang, M.-H. (eds.) Computer Vision—ACCV 2014, volume 9006 of Lecture Notes in Computer Science, pp. 111–126. Springer, Berlin (2015)
6. Georgis, G., Lentaris, G., Reisis, D.: Reduced complexity superresolution for low-bitrate video compression. IEEE Trans. Circuits Syst. Video Technol. **26**(2), 332–345 (2016)
7. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. ACM Queue Mag. **6**(2), 40–53 (2008)
8. Freedman, G., Fattal, R.: Image and video upscaling from local self-examples. ACM Trans. Graph. **30**(2), 12:1–12:11 (2011)
9. Zhu, Y., Zhang, Y., Yuille, A.L.: Single image super-resolution using deformable patches. In: 2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2917–2924 (2014)
10. Alex, K.: CUDA Convolutional Neural Networks (2015)
11. nVidia: NVIDIA CUDA Fast Fourier Transform library (cuFFT) (2015)
12. Gallup, D., Frahm, J.-M. Stam, J.: Cuda stereo. In: nVidia GPU Technology Conference 2009 (2009)
13. Yang, Q.: Hardware-efficient bilateral filtering for stereo matching. IEEE Trans. Pattern Anal. Mach. Intell. **36**(5), 1026–1032 (2014)
14. Kowalczuk, J., Psota, E.T., Perez, L.C.: Real-time stereo matching on cuda using an iterative refinement method for adaptive support-weight correspondences. IEEE Trans. Circuits Syst. Video Technol. **23**(1), 94–104 (2013)
15. Kowalczuk, J., Psota, E.T., Perez, L.C.: Real-time temporal stereo matching using iterative adaptive support weights. In: 2013 IEEE International Conference on Electro/Information Technology (EIT), pp. 1–6 (2013)
16. Bowen, O., Bouganis, C.: Real-time image super resolution using an fpga. In: International Conference on Field Programmable Logic and Applications, 2008 (FPL 2008), pp. 89–94 (2008)
17. Angelopoulou, M.E., Bouganis, C.-S., Cheung, P.Y.K., Constantinides, G.A.: Robust real-time super-resolution on FPGA and an application to video enhancement. ACM Trans. Reconfig. Technol. Syst. **2**(4), 22–29 (2009)
18. Sanada, Y., Ohira, T., Chikuda, S., Igarashi, M., Ikebe, M., Asai, T., Motomura, M.: FPGA implementation of single-image superresolution based on frame-bufferless box filtering. J. Signal Process. **17**(4), 111–114 (2013)
19. Pérez, J., Magdaleno, E., Pérez, F., Rodríguez, M., Hernández, D., Corrales, J.: Super-resolution in plenoptic cameras using fpgas. Sensors **14**(5), 8669–8685 (2014)
20. Okuhata, H., Imai, R., Ise, M., Omaki, R.Y., Nakamura, H., Hara, S., Shirakawa, I.: Implementation of dynamic-range enhancement and super-resolution algorithms for medical image processing. In: 2014 IEEE International Conference on Consumer Electronics (ICCE), pp. 181–184. IEEE (2014)
21. Greisen, P., Heinzle, S., Gross, M., Burg, A.P.: An FPGA-based processing pipeline for high-definition stereo video. EURASIP J. Image Video Process. **1**, 2011 (2011)
22. Jin, S., Cho, J., Pham, X.D., Lee, K.M., Park, S.-K., Kim, M., Jeon, J.W.: FPGA design and implementation of a real-time stereo vision system. IEEE Trans. Circuits Syst. Video Technol. **20**(1), 15–26 (2010)
23. Werner, M., Stabernack, B., Riechert, C.: Hardware implementation of a full hd real-time disparity estimation algorithm. IEEE Trans. Consum. Electron. **60**(1), 66–73 (2014)
24. Che, S., Li, J., Sheaffer, J.W., Skadron, K., Lach, J.: Accelerating compute-intensive applications with GPUs and FPGAs. In: Symposium on Application Specific Processors, 2008 (SASP 2008), pp. 101–107 (2008)
25. Yang, D., Sun, J., Lee, J., Liang, G., Jenkins, D.D., Peterson, G.D., Li, H.: Performance comparison of cholesky decomposition on GPUs and FPGAs. In: Symposium on Application Accelerators in High Performance Computing (2010)
26. Jones, D.H., Powell, A., Bouganis, C., Cheung, P.Y.K.: GPU versus FPGA for high productivity computing. In: 2010 International Conference on Field Programmable Logic and Applications (FPL), pp. 119–124 (2010)
27. Kalarot, R., Morris, J.: Comparison of FPGA and GPU implementations of real-time stereo vision. In: 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pp. 9–15 (2010)
28. Savarimuthu, T.R., Kjr-Nielsen, A., Srensen, A.S.: Real-time medical video processing, enabled by hardware accelerated correlations. J. Real-Time Image Process. **6**(3), 187–197 (2011)
29. Pietron, M., Wielgosz, M., Zurek, D., Jamro, E., Wiatr, K.: Comparison of GPU and FPGA implementation of SVM algorithm for fast image segmentation. In: Architecture of Computing Systems ARCS 2013, Volume 7767 of Lecture Notes in Computer Science, pp. 292–302. Springer, Berlin (2013)
30. Tomislav, M., Ivan, A., Željko, H.: CPU, GPU and FPGA implementations of mald: Ceramic tile surface defects detection algorithm. Automatika **55**(1), 1920–1927 (2014)
31. Gurumani, S.T., Cholakkal, H., Liang, Yun., Rupnow, K., Chen, D.: High-level synthesis of multiple dependent cuda kernels on FPGA. In: 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 305–312 (2013)
32. Jianchao, Y., Wright, J., Huang, T.S., Ma, Y.: Image super-resolution via sparse representation. IEEE Trans. Image Process. **19**(11), 2861–2873 (2010)
33. Dong, W., Zhang, D., Shi, G., Wu, X.: Image deblurring and super-resolution by adaptive sparse domain selection and adaptive regularization. IEEE Trans. Image Process. **20**(7), 1838–1857 (2011)
34. Villena, S., Vega, M., Molina, R., Katsaggelos, A.K.: Bayesian super-resolution image reconstruction using an $l1$ prior. In: Proceedings of 6th International Symposium on Image and Signal Processing and Analysis, 2009 (ISPA 2009), pp. 152–157 (2009)
35. Dong, W., Zhang, D., Shi, G., Wu, X.: Nonlocal back-projection for adaptive image enlargement. In: 2009 16th IEEE International Conference on Image Processing (ICIP), pp. 349–352 (2009)
36. Dong, W., Zhang, L., Lukac, R., Shi, G.: Sparse representation based image interpolation with nonlocal autoregressive modeling. IEEE Trans. Image Process. **22**(4), 1382–1394 (2013)
37. Zhou, W., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P.: Image quality assessment: from error visibility to structural similarity. IEEE Trans. Image Process. **13**(4), 600–612 (2004)
38. Mittal, A., Moorthy, A.K., Bovik, A.C.: No-reference image quality assessment in the spatial domain. IEEE Trans. Image Process. **21**(12), 4695–4708 (2012)

39. Levon, J.: Oprofile 1.0, a Statistical Profiler for Linux Systems (2015)
40. nVidia: Parallel Thread Execution ISA (2015)
41. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming, 1st edn. Addison-Wesley, Reading (2010)
42. Xu, C., Kirk, S.R., Jenkins, S.: Tiling for performance tuning on different models of GPUs. In: 2009 Second International Symposium on Information Science and Engineering (ISISE), pp. 500–504 (2009)
43. Harris, M.: Optimizing Parallel Reduction in CUDA (2007)
44. Eklund, A., Dufort, P.: GPU-Pro 5: Advanced Rendering Techniques—Non-separable 2D, 3D and 4D Filtering with CUDA, Chapter 5, 1st edn. CRC Press, Boca Raton (2014)
45. Volkov, V.: Better Performance at Lower Occupancy (2010)
46. Podlozhnyuk, V.: Image Convolution with CUDA (2012)
47. nVidia: CUDA C Programming Guide (2015)
48. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 (2012)
49. Szydzik, T., Callico, G.M., Nunez, A.: Efficient FPGA implementation of a high-quality super-resolution algorithm with real-time performance. IEEE Trans. Consum. Electron. **57**(2), 664–672 (2011)
50. Szeliski, R.: Computer Vision: Algorithms and Applications. Springer, Berlin (2010). ISBN: 978-1-84882-935-0
51. Lentaris, G., Diamantopoulos, D., Siozios, K., Soudris, D., Rodrigálvarez, A.M.: Hardware implementation of stereo correspondence algorithm for the exomars mission. In: 2012 22nd International Conference on Field Programmable Logic and Applications (FPL), pp. 667–670. IEEE (2012)
52. Scharstein, D., Pal, C.: Learning conditional random fields for stereo. In: IEEE Conference on Computer Vision and Pattern Recognition, 2007 (CVPR'07), pp. 1–8 (2007)
53. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. Int. J. Comput. Vis. **47**(1–3), 7–42 (2002)
54. Scharstein, D., Szeliski, R.: High-accuracy stereo depth maps using structured light. In: Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003, Volume 1, pp. I-195–I-202 (2003)
55. Hosni, A., Rhemann, C., Bleyer, M., Gelautz, M.: Temporally consistent disparity and optical flow via efficient spatio-temporal filtering. In: Ho, Y.-S. (ed.) Advances in Image and Video Technology, volume 7087 of Lecture Notes in Computer Science, pp. 165–177. Springer, Berlin (2012)
56. Kishonti Ltd.: Compubench, a Professional OpenCL and Renderscript Benchmark (2015)

Georgios Georgis received the B.Sc. in Physics from the Aristotle University of Thessaloniki (AUTH) in 2003, the M.Sc. degree in "Electronic Automation" from the National and Kapodistrian University of Athens (NKUA), Greece in 2009 and is currently pursuing the Ph.D. degree from the National and Kapodistrian University of Athens, Greece. He is a research associate at the Electronics Laboratory of the Department of Physics of the National and Kapodistrian University of Athens. His research interests include digital signal filtering, image/video processing and the design of relevant parallel algorithms and architectures, channel coding, data compression, machine learning and artificial intelligence.

George Lentaris holds a Ph.D. in Computing from the National and Kapodistrian University of Athens (NKUA), Greece, as well as two M.Sc. degrees in "Logic, Algorithms, and Computation" and in "Electronic Automation", with a B.Sc. in Physics. His Ph.D. thesis (2011) is entitled "Parallel Architectures and Algorithms for Digital Signal and Image Processing" and contributes in organizing parallel memories for graphics applications and designing motion-estimation architectures for video compression. His research interests also include digital circuit design, H.264/AVC and HEVC encoding, digital signal filtering, and computer vision algorithms. He is currently a research associate at the National Technical University of Athens (NTUA), Greece.

Dionysios Reisis has received his Ptychion in Electrical Engineering from the University of Patras, Greece, in 1983, and his M.Sc. and Ph.D. degrees in Computer Engineering from the Department of Electrical and Computer Engineering of the University of Southern California, USA, in 1989. In 1990, he started cooperation with the Telecommunications laboratory of the Division of Computer Science of NTUA as a research associate. In 1991, he became Lecturer, and currently, he is an Associate Professor of the Electronics Laboratory of the Department of Physics of the University of Athens (NKUA). His interests include parallel architectures and algorithms for image and graph signal processing with applications in VLSI environment, as well as real time hardware design and efficient algorithms design for telecommunication systems support.