Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2015, Vol. 29(4) 461–472

DOI: 10.1177/1094342014567907
hpc.sagepub.com
SAGE

Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern

Marco Aldinucci[1], Guilherme Peretti Pezzi[1], Maurizio Drocco[1], Concetto Spampinato[2] and Massimo Torquati[3]

Abstract

In this paper, a highly effective parallel filter for visual data restoration is presented. The filter is designed following a skeletal approach, using a newly proposed *stencil-reduce*, and has been implemented by way of the FastFlow parallel programming library. As a result of its high-level design, it is possible to run the filter seamlessly on a multicore machine, on multi-GPGPUs, or on both. The design and implementation of the filter are discussed, and an experimental evaluation is presented.

Keywords

Impulsive noise, Gaussian noise, image restoration, image filtering, GPGPUs, parallel patterns, skeletons, structured parallel programming, iterative stencil, stencil-reduce, MapReduce

1. Introduction

In the last decade, the advances in camera technology and the reduction of costs of memory storage have led to a proliferation of visual data content in the form of images and videos, e.g. 72 hours of video are uploaded to YouTube every minute. This visual data deluge combined with the ever-increasing computing power available from off-the-shelf processors has opened new frontiers in the research on automatic video and image analysis with the goal to push such problems into consumer applications. This latter aspect demands for real-time analysis of digital images and videos, i.e. effective strategies for reducing the processing times and resources needed by the existing approaches from image pre-processing to visual content extraction and understanding.

Image restoration is a key module of any machine vision system and aims at removing noise and restoring the original visual content: noise often generated by image sensor failures. Variational methods are well known for their effectiveness in image restoration (Nikolova, 2004; Chan et al., 2005), but they cannot be employed for real-time video and image analysis because of their high computational cost, due to function minimization over all of the image pixels, and complexity of tuning. An efficient image restoration approach for images corrupted by Salt and Pepper noise based on variational methods is described in Aldinucci et al. (2005). In detail, a two-phase parallel image restoration schema running on both multicore and GPGPUs is described: (a) in the first step, an adaptive median filter is employed for detecting noisy pixels, while (b) in the second step, a regularization procedure is iterated until the noisy pixels are replaced with values able to preserve image edges and details. The restoration method is implemented according to a high-level pattern-based approach (also known as a skeletal approach (Cole, 1989)), and deployed as a sequence of detect and restoration stages that are defined according to the *map* parallel paradigm and the *map-reduce* parallel paradigm, respectively.

This paper extends the previous work in the following directions:

- a newly designed *stencil-reduce* pattern that allows offloading iterative and stencil-based computation

[1]Computer Science Department, University of Torino, Italy
[2]Department of Electrical, Electronics and Computer Engineering, University of Catania, Italy
[3]Computer Science Department, University of Pisa, Italy

Corresponding author:
Marco Aldinucci, Computer Science Department, University of Torino, Corso Svizzera 185, 10149 Torino, Italy.
Email: aldinuc@di.unito.it

to GPGPU devices is presented and tested on image restoration applications;
- the restoration schema, proposed previously, is extended in order to deal also with Gaussian noise and not only with impulse noise;
- the restoration schema is, finally, applied to video streams and is expressed using *pipeline* parallelism between the two stages.

A first prototypal implementation working on video streams has been proposed by Aldinucci et al. (2014b), which is based on hand-written OpenCL kernels for GPGPUs offloading, while in this paper a higher-level approach is proposed, based on the recently proposed *stencil-reduce* pattern from the FastFlow framework, where all of the memory management and offloading calls are hidden from the programmer.

The remainder of the paper is as follows. In Section 2, the background is presented, including the state of the art on variational based image restoration and an introduction about algorithmic skeleton solutions and the FastFlow framework. In Section 3, the new iterative *stencil-reduce* pattern is presented along with the design of the parallel restoration process. In Section 4, experimental evaluation of quality and performance of the proposed parallel restoration algorithm are presented. Finally, concluding remarks and future works are discussed in Section 5.

2. Background

In this section the related work on image restoration process will be first introduced. Then, an overview of state-of-the-art structured parallel programming frameworks is given, together with an introduction to the FastFlow framework.

2.1 Variational image restoration: related work

In the past 15 years, a large number of methods have been proposed for effective image filtering and restoration with a particular emphasis to impulse noise (Astola and Kuosmanen, 1997). Most of these methods employ order statistic filters that exploit the rank-order information of an appropriate set of noisy input pixels. The median filter is the most popular nonlinear filter for removing impulse noise, because of its restoration effectiveness (Bovik, 2000) and computational efficiency (Huang et al., 1979). However, the performance of median filter is unsatisfactory both in detail preserving and in suppressing signal-dependent noise with noise over 50% (Yang and Basir, 2003).

An effective impulse noise filtering should be able to remove as much noise as possible, while preserving high-frequency image components. This issue has generally been addressed by methods based on median filter (Yin et al., 1996; Lin, 2007). For example, the center-weighted median (CWM) filter weights more the central value of a kernel window in order to make it more robust against outliers, thus preserving more details than the median filter. This is obtained at the expense of less noise suppression. The adaptive center-weighted median (ACWM) (Lin, 2007), instead, exploits local features computed on image patches to calculate weights. Typically, median-based filters are used invariantly across the whole images to remove noise, thus smearing image details (Wang, 2001) since it also changes the values of uncorrupted pixels.

To achieve a good compromise between image-detail preservation and noise reduction, a noisy pixel detector is generally adopted to identify corrupted pixels, which are then iteratively restored. An example is the multi-state median filter (MSMF) (Chen and Wu, 2000), which consists of a two-part noisy pixel detection stage and a correction stage by applying the median filter. The detection is based on the concept of *homogeneity level*, which defines a pixel as noisy according to the intensity differences between that pixel and its neighbours (i.e. according to its homogeneity to the neighbourhood). The second part of the detection stage re-analyses the noisy pixels using a 5×5 observation window. The main shortcoming of these methods is that they introduce undesirable distortions into the details and texture of the input image during the noise removal process. Moreover, their performance is strongly influenced by several factors, such as noise density estimate, weighting factors, impulse detection thresholds, etc. that are often heuristically determined.

The problem of image restoration for edge preserving is an inverse problem and it has been tackled mainly with variational based approaches (Nikolova, 2004). This type of methods identify the restored image by minimizing an energy function based on a *data-fidelity* term, related to the input noise, and a *regularization* term where a priori knowledge about the solution is enforced. More specifically, this kind of approaches identifies the restored image u through an optimization problem in the form of

$$\min_{u \in N} F(u) = \alpha \int R(u) + \mu \int D(u, d) \qquad (1)$$

where d is the image corrupted by the noise, $D(u, d)$ is data fidelity, which depends on the kind of noise and provides a measure of the dissimilarity between d and the output image u, $R(u)$ is a regularization term where a priori knowledge about the solution is enforced, μ and α are the regularization parameters that balance the effects of the two terms. This optimization problem is restricted only to the noisy pixels N.

An example of variational approach for image restoration is the one proposed by Chan et al. (2005) which is capable of removing effectively impulse noise

as high as 90%. Similar approaches to Chan et al.'s method, aimed at improving the noisy detection step and at reducing the processing times, are those proposed by Faro et al. (2008), Cai et al. (2007, 2010), Chen and Yang (2007) and Cannavò et al. (2006).

Over the years, a large variety of variational and regularization methods have also been conceived (although less frequently than in the case of impulse noise) to cope with blurring effects in order to recover the loss of high-frequency information entailed by blurring. An example is the Mumford–Shah regularization term (Mumford and Shah, 1989), which, however, does not work well in the case of only Gaussian noise (Cai et al., 2008). Moreover, most of the existing methods though being able to effectively filter and deblur an image, are extremely time-consuming (e.g. the method in Chan et al. (2005) takes about 27 hours to restore a 320×240 image on a standard PC using a MATLAB implementation), thus making their application to real-world cases unfeasible and, therefore, demanding for effective parallelization strategies.

2.2 Algorithmic skeletons

Algorithmic skeletons have been around since the 1990s as an effective means of parallel application development. An algorithmic skeleton can be regarded as a general-purpose, parametric parallelism exploitation pattern (Cole, 1989). Application programmers may instantiate skeletons (or compositions of skeletons) to encapsulate and exploit the full parallel structure of their applications. Business code may be passed as a parameter to the generic skeleton, thus turning the generic skeleton into a part of a parallel application. Programming frameworks based on algorithmic skeletons have been recently introduced to alleviate the task of the application programmer when targeting data parallel computations on heterogeneous architectures as multi-GPGPU systems.

OpenCL is a parallel API provided for GPGPU programming, which allows the users to exploit GPUs for general-purpose tasks that can be parallelized (Khronos OpenCL Working Group, 2008). It is implemented by different hardware vendors such as Intel, AMD and NVIDIA, making it highly portable and allowing the code written in OpenCL to be run on different graphical accelerators. It has the capability to revert to the CPU for execution when there is no GPGPU in the system. Its portability makes it suitable for hybrid (CPU/GPGPU) or cloud-based environments. However OpenCL is quite low level, actually focusing on low-level feature management rather than high-level parallelism exploitation patterns. In Muesli (Ernsting and Kuchen, 2011) the programmer must explicitly indicate whether GPGPUs are to be used for data parallel skeletons. StarPU (Augonnet et al., 2011) is focused on handling accelerators such as GPGPUs. Graph tasks are scheduled by its run-time support on both the CPU and various accelerators provided the programmer has given a task implementation for different programming models and architectures. SkePU (Enmyren and Kessler, 2010) provides programmers with GPGPU implementations of map and reduce skeletons and relies on StarPU for the execution of stream parallel skeletons (pipe and farm). MCUDA (Stratton et al., 2008) is a framework to mix CPU and GPGPU programming. In MCUDA it is mandatory to define kernels for all available devices but the framework cannot make any assumptions about the relative performance of the supported devices.

Moving to frameworks not specifically designed for targeting GPGPU offloading, MPI is often considered as the mainstream solution for writing efficient parallel applications (Pacheco, 1996) for heterogeneous architectures. The low-level approach advocated by MPI falls short in supporting performance portability, especially when hundreds or thousands of concurrent activities are involved and hybrid solutions have to be adapted (i.e. MPI + OpenMP). Applications must often be re-designed or manually tuned for each new platform by an expert parallel programmer. *OpenMP* (Park et al., 2001) is a popular thread-based framework for multicore architectures mostly targeting data parallel programming (although it is currently being extended to incorporate stream processing). OpenMP supports, by way of language pragmas, the low-effort parallelization of sequential programs; however, these pragmas are mainly designed to exploit loop-level data parallelism (e.g. *do_independent*). OpenMP does not natively support either *farm* or *Divide&Conquer* patterns, even though they can be simulated with lower-level features. Intel *Threading Building Blocks* (TBB) Intel Corp. (2014) is a C++ template library, which eases the development of concurrent programs by exposing (simple) skeletons and parallel data structures used to define tasks of computations. TBB is designed as an application-level library for shared-memory programming only; furthermore it does not provide any formal definition of its own skeletons to support global optimizations of the code.

2.3 The FastFlow programming framework

The FastFlow parallel programming environment (Aldinucci et al., 2010; Aldinucci and Torquati, 2014) was originally designed to support efficient streaming on cache-coherent multicore platforms. It is realized as a C++ pattern-based parallel programming framework aimed at simplifying the development of applications for (shared-memory) multicore and GPGPUs platforms. The key vision of FastFlow is that ease-of-development and runtime efficiency can both be

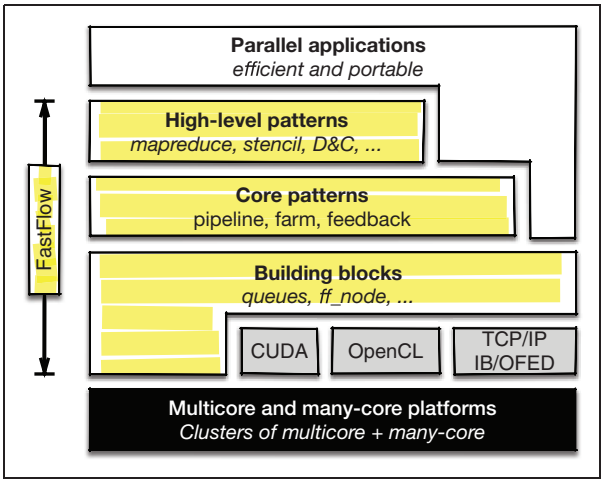

Figure 1. Architecture of FastFlow framework.

achieved by raising the abstraction level of the design phase. It provides developers with a set of parallel programming patterns (also know as *algorithmic skeletons*), in particular data-parallel patterns (such as *map, stencil, reduce* and their composition, and also a ParallelFor accelerator (Danelutto and Torquati, 2014)). High-level patterns are implemented on top of the Core patterns level, consisting of arbitrarily nested and composed basic stream-parallel patterns (*farm, pipeline* and *feedback*).

The latest extensions of the FastFlow framework, aimed at supporting heterogeneous platforms, make it possible to easily port the application to hybrid multicore/GPGPU systems by embedding CUDA/OpenCL business code. In particular, data-parallel patterns can be run both on multicore and offloaded onto GPGPUs. In the latter case, the business code can include GPGPU-specific statements (i.e. CUDA or OpenCL statements).

At the bottom level (i.e. the building blocks level) FastFlow CPU implementation of patterns are realized via non-blocking graphs of threads connected by way of lock-free channels (Aldinucci et al., 2014a), while the GPGPU implementation is realized by way of the CUDA/OpenCL bindings and offloading techniques (Aldinucci et al., 2011). The framework also takes care of memory transfers between CPU host and GPGPU device. In general, different patterns can be mapped onto different sets of cores or accelerators, thus, in principle, using the full available power of the heterogeneous platform. The architecture of FastFlow framework is reported in Figure 1.

3. Visual data restoration: a skeleton-based approach

In this work, we consider the extension of the two-phase image restoration to a two-phase *video stream* restoration. We advocate the *high-level* extension of the application working for images presented in Aldinucci et al. (2005) to video, and to heterogeneous platforms. We emphasize the *high-level* approach. In the long term, writing parallel programs that are efficient, portable and correct must be no more onerous than writing sequential programs. To date, however, few parallel programming models have embraced much more than low-level libraries, which often require the architectural re-design of the application. This approach is unable to effectively scale to support the mainstream of software development where human productivity, total cost and time to solution are equally, if not more, important aspects.

3.1 Stencil-reduce pattern

Let **map** $f[a_0, a_1,\ldots] = [f(a_0), f(a_1),\ldots]$ and **reduce**$\oplus$ $[a_0, a_1,\ldots] = a_0 \oplus a_1 \oplus \ldots$, where f is the *elemental function*, $\oplus$ is the *combinator* (i.e. a binary associative operator), and $[a_0, a_1,\ldots]$ an array (e.g. the pixels of an image). These parallel paradigms have been proposed as patterns for multicore and distributed platforms, GPGPUs, and heterogeneous platforms (Enmyren and Kessler, 2010; González-Vélez and Leyton, 2010; Hammond et al., 2013). Let **stencil** be a map, except each instance of the elemental function accesses neighbours of its input (e.g. cross-shaped four-neighbourhood of a pixel), offset from its usual input (McCool et al., 2012). They are well-known examples of data-parallel patterns; since elemental function of a map/stencil can be applied to each input element a_i independently from each other as well as applications of the combinator to different pairs in the reduction tree of a reduce can be done independently, thus naturally inducing a parallel implementation. FastFlow framework provides constructors for these patterns, in the form of C++ macros that take as input the code of the elemental function (combinator) of the map/stencil (reduce) patterns, together with an eventual read-only structure (i.e. the *environment*) that can be regarded, under a pure functional semantics perspective, as a function parameter. The language for the *kernel* code implementing the elemental function, which is actually the business code of the application, has clearly to be platform-specific (i.e. CUDA/OpenCL for GPGPUs, C++/OpenCL for multicore CPUs), since FastFlow is a plain C++ header-only library.

In this work we use *stencil-reduce* pattern to realize iterative applications of nonlinear filters followed by residual reduction. In order to show the usage of the *stencil-reduce* pattern in FastFlow data-parallel programming model, here we present a simpler case study.

In order to use the proposed pattern, the programmer has to write three macros:

1. that defines the parameters and the elemental CUDA function that will be applied to each element using the desired stencil shape;

2. that defines the CUDA combinator function that will be applied after the map phase;
3. the *stencil-reduce* macro that takes as arguments the user-defined datatype that will encapsulate all needed data and macros as described in (1) and (2).

The user can additionally choose the convergence criteria that will decide when the iterative mechanism should stop, based, for example, on the result obtained from the combinator function or by simply defining an upper bound to the number of iteration.

In Figure 2 is sketched the implementation of a simple convolution filter followed by a trivial sum operator. In the stencil phase, a classic One-dimensional convolution filter (N-neighbours MeanFilter with N odd) is applied on input array A, thus producing output array A'. In the reduce phase, the residual array $|A' - A|$ is passed through a reduce operator (a standard Sum) which computes the sum of its elements. In the example, CUDA version of the stencil pattern is used, thus exploiting automatic GPGPU offloading.

We remark, under the ease-of-development perspective, that the user has to provide only the kernel code of the elemental function of the mean filter stencil-based operator and the combinator of the sum reduce-based operator. Moreover, under the performance portability perspective, note that the user is not required to care about the underlying platform, except provide platform-specific kernel code, since FastFlow manages platform heterogeneity during the application deployment.

3.2 Two-phase image restoration

The restoration process used in this work is a *two-step* algorithm and consists, first, of a noisy pixel detection phase (in the following referred as *Detect*) where pixels which are likely to be corrupted by noise are identified and, then, of a pixel restoration phase (referred as *Restore*), based on variational method (Nikolova, 2004), where the set of noisy pixels are restored in order to preserve image details.

3.2.1 Detect phase. The well-known adaptive median filter is first applied to the noisy image with the goal of identifying corrupted pixels. Let $\hat{y}$ be the map obtained by thresholding the difference between the output of the adaptive median filter and the noisy original image. This map has ones for candidate noisy pixels and zeros for the remaining pixels. Each pixel of the image, at this stage, is processed independently, provided that the processing element can access a read-only halo of the pixel. Hence, the set of noisy pixels can be defined as follows:

$$N = \left\{(i,j) \in A : \hat{y}_{i,j} = 1\right\}$$

The set of all uncorrupted pixels is $N^c = N \setminus A$, where A is the set of the all pixels and N is the set of the noisy pixels.

3.2.2 Restore phase. The restore phase is carried out by means of regularization and, among all of the variational functions identified by formula (1), we adopt the following, as it proved (Nikolova, 2004) to operate effectively for impulse noise:

$$\begin{aligned} F_d|_N(u) &= \textstyle\sum_{(i,j)\in N} \left[|u_{i,j} - d_{i,j}| + \frac{\beta}{2}(S_1 + S_2)\right] \\ S_1 &= \textstyle\sum_{(m,n)\in V_{i,j}\cap N^c} 2 \cdot \varphi(u_{i,j} - d_{m,n}) \\ S_2 &= \textstyle\sum_{(m,n)\in V_{i,j}\cap N} \varphi(u_{i,j} - u_{m,n}) \end{aligned}$$

where N represents the noisy pixels set and where $V_{i,j}$ is the set of the four closest neighbours of the pixel with coordinates (i, j) of image d and u is the restored image. Here $F_d|_N(u)$ is given by the sum of two terms: a *data-fidelity term* which represents the deviation of restored image u from the data image d, which is marred by noise, and a *regularization term* that incorporates a function that penalizes oscillations and irregularities, although it does not remove high-level discontinuities.

```
1  CUDA_STENCIL_KERNEL ( /* elemental function */
2      /* kernel parameters */
3      MF_kernel /* kernel id */,
4      double /* basic argument type */,
5      input /* argument name - i.e. neighbours of a pixel
           */,
6      N /* neighbourhood width */,
7      NULL /* readonly environment */,

9      /* begin CUDA kernel code */
10     double sum = 0;
11     for(int i = 0; i < N; ++i)
12         sum += input[i];
13     return sum / N;
14 )

16 CUDA_Stencil<MF_kernel> MeanFilter;

19 CPP_REDUCE_KERNEL ( /* combinator */
20     /* kernel parameters */
21     SUM_kernel /* kernel id */,
22     double /* basic argument type */,
23     a /* first argument name - left arg of the combinator
           */,
24     b /* second argument name - right arg of the
           combinator */,
25     NULL /* readonly environment*/,

27     /* begin CPP kernel code */
28     return (a + b);
29 )

31 CPP_Reduce<SUM_kernel> Sum;
```

Figure 2. One-dimensional convolution stencil-reduce example.

The method proposed by Vogel and Oman (1998) is used for functional minimization. The regularization term is given by the function φ and different forms of φ have been employed according to the type of noise affecting the image. In this paper we employed

$$\varphi_S(t) = |t|^{\alpha}, \quad 1 < \alpha \leq 2 \tag{2}$$

$$\varphi_G(t) = \sqrt{(t^2 + \alpha)}, \quad \alpha \ll 1 \tag{3}$$

for dealing with Salt and Pepper (a specific case of impulse noise) and Gaussian noise, respectively. As mentioned previously, especially for dealing with impulse noise, many other regularization functions have been devised in the last 10 years, e.g. $\varphi_1(t) = 1 + |t|/\alpha - \log(1 + |t|/\alpha)$ and $\varphi_2(t) = \log(\cosh(t/\alpha))$ with $\alpha > 0$.

The variational method based on minimization is iteratively applied until a convergence is reached, thus resulting into a fixed-point procedure. Let $u^{(k)}$ be the output of the k-th iteration of the procedure and $\Delta^{(k)} = |u^{(k)} - u^{(k-1)}|$ the pixel-wise residual of two successive iterations of the procedure. We used the following convergence criterion:

$$\frac{\sum_{i,j} \left| \Delta_{i,j}^{(k)} - \Delta_{i,j}^{(k-1)} \right|}{N} \tag{4}$$

Note that the computation of the convergence criterion involves three different (successive) outcomes of the procedure, since the naïve version of the criterion, based on the comparison of the two most recent outcomes, results in a strongly oscillatory convergence curve.

In this work, we only consider a two-copy approach for the *Restore* stage, in which the procedure at the k-th iteration takes as input the output of the (k-1)-th iteration in a read-only manner. Although, other restoration procedures are based on a intra-iteration approach, in which the output of the algorithm over a pixel uses as input the output of the algorithm at the same iteration for the already visited neighbours. This should guarantee a faster convergence of the iterative procedure. Note that in a sequential implementation this induces a semantics, which is dependent on the order in which pixels are visited, thus this aspect must be taken into account when designing a parallel implementation of a intra-iteration procedure. We experimented different parallelization schemes in such a scenario, from clustering pixels into non-overlapping blocks, thus respecting the sequential semantics, to a purely non-deterministic approach, in which pixels are read regardless the fact they have already been visited by the current iteration. Anyway we did not get any appreciable gain in terms of convergence speed for the proposed functional.

3.2.3 Parallel implementation. The parallel processing in the *Detect* phase could in theory be exploited in some data-parallel fashion (e.g. *stencil-reduce* pattern). However, for images of standard resolutions (up to HD quality), as those considered in the experimental section of this work, the *Detect* phase has a negligible execution time with respect to the *Restore* stage, so the parallelization of the *Detect* phase is not taken into account.

In the *Restore* phase, the parallel processing is described as an iterative *stencil-reduce* pattern. Within a single iteration, for each couple of outlier coordinates (i, j), corresponding to an outlier pixel (i.e. a pixel in the noisy set N, which is the outcome of the *Detect* phase), the stencil accesses the eight neighbour pixels (with replication padding for borders). For each outlier (i, j), the procedure computes a new value as the one minimizing the value of the functional $F_d|_N(u)$ in formula (1). We remark that in this scenario the user has to provide only the code of the computation over a single pixel, which is actually the business code of the *Restore* iteration, and the framework takes care of applying the computation over all of the (noisy) pixels and iterating the procedure until the convergence condition is reached. In the reduce pattern, the global convergence criterion is computed as a simple average of the three-iteration residuals, as discussed above. In this case the user has to provide only the code of the binary combinator, which is turned into a tree-shaped parallel computation by the framework.

From the considerations above, it is clear that the *Restore* phase is particularly well suited for being implemented via the *stencil-reduce* pattern since the computation of the functional on each pixel requires accessing to a neighbourhood of the input pixel. Alternatively, the stencil phase could be implemented via other high-level patterns, but some workaround would be required in order to express the convolution-shaped computation. For example, to compute a convolution with a map pattern, the input matrix (frame) should include also the index of each value (pixel) in order to access the neighbours from the code of the elemental function.

Finally, the two-phase filter methodology naturally induces a high-level structure for the algorithm, which can be described as the successive application of two filters as described in Figure 3. The two phases can operate in a pipeline in the case where the two-phase restoration process is used on a stream of images (e.g. in a video application). In addition, both filters can be parallelized in a data-parallel fashion.

3.2.4 Performance considerations. In a sequential setting, the *Detect* stage exhibits linear execution time with respect to the total number of pixels. The *Restore* phase shows a computational complexity of $\mathcal{O}(|N| \cdot n\_iterations)$, where $|N|$ is the number of noisy pixels identified in the *Detect* phase, and *n_iterations* is the number of

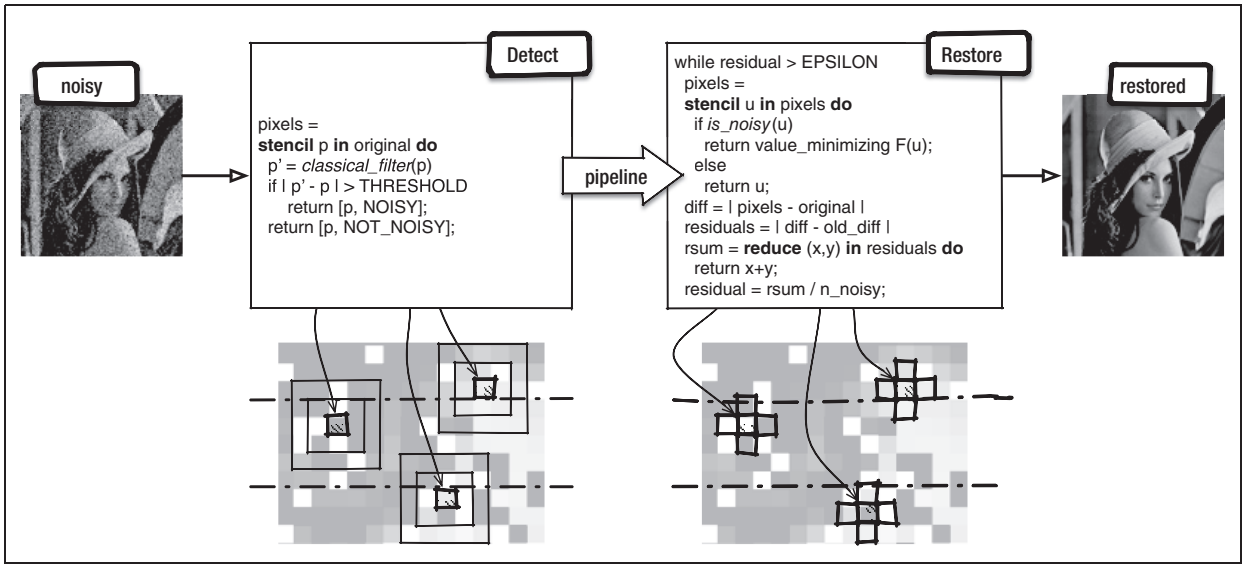

Figure 3. Two-phase restoration for visual data streams.

iterations required to reach one the convergence criterion. Since the set N of the noisy pixels is a fraction of the whole set of all of the pixels in the image, the complexity of the two phases are actually the same. However, the *Detect* stage – typically a simple convolution – is two–three orders of magnitude faster than a single iteration of the second stage (*Restore*). Moving to parallel implementations, in particular on a fully CPU deployment, a sequential *Detect* stage is able to sustain the throughput of the parallel *Restore* stage, even in a large multicore platform. Interestingly, this is in theory no longer the case if the *Restore* stage is run on a large multi-GPGPU setting. In this latter case, the massive computation power provided by GPGPUs in pure data-parallel computations could reduce the execution time of the *Restore* phase enough to make the execution times of the two phases actually comparable. Despite the fact that we could not observe directly the described phenomenon, we can argue that a reasonably large multi-GPGPU deployment could benefit from a parallel implementation (e.g. based on *stencil-reduce* pattern) of the *Detect* phase.

4. Experimental evaluation

This section first shows the image restoration performance of the proposed method for Salt and Pepper and Gaussian noise on still images; second, the performance of our parallelization strategy is discussed when employing the proposed restoration process on sample video sequences (i.e. visual data streams).

4.1 Image restoration quality

Among the commonly tested 256 × 256 8-bit greyscale images, Lena, Baboon and Cameraman were selected for our simulations. In the case of Salt and Pepper noise, images were corrupted by "salt" (with value 255) and "pepper" (with value 0) noise with equal probability and noise levels varying from 10% to 90% with increments of 20% were tested. For Gaussian noise, we tested the restoration process performance as the variance of the noise increases from 1 to 100. The values of α in the regularization function (see the previous section) were set to 1.2 and 10^{-4} for, respectively, Salt and Pepper and Gaussian noise.

Restoration performance was measured quantitatively by the peak signal-to-noise ratio (PSNR) and the mean absolute error (MAE) defined as follows:

$$\mathrm{PSNR} = 10 \cdot \log_{10} \frac{255^2}{\frac{1}{MN}\sum_{i,j}(u_{i,j} - d_{i,j})^2},$$

$$\mathrm{MAE} = \frac{1}{MN}\sum_{i,j}(u_{i,j} - d_{i,j})^2$$

with M and N being the image length and width and $u_{i,j}$ and $d_{i,j}$ the pixel values of the restored image and the original noisy image, respectively. Table 1 shows the obtained results.

To better estimate the achieved restoration quality, Figure 4 shows the restored Baboon images as the percentage of Salt and Pepper noise increases. Figure 5 reports, instead, the restored Cameraman images with different levels (different values of variance) of Gaussian noise. Although the performance of the proposed method is superior in the case of Salt and Pepper noise, it has to be noted that the proposed approach also sensibly outperforms the common-adopted baseline filter for Gaussian noise removal, i.e. the Wiener filter, as shown in Figure 6. We decided to use the Wiener filter as a baseline since it is able to deal with

Table 1. PSNR and MAE obtained at varying of the noise affecting Lena, Baboon and Cameraman images (added Gaussian noise has mean 0 in both cases).

	Salt and Pepper										Gaussian									
	Lena – Noise level					Baboon – Noise level					Lena – Noise level (var.)					Cameraman – Noise level (var.)				
	10%	30%	50%	70%	90%	10%	30%	50%	70%	90%	1	10	30	70	100	1	10	30	70	100
PSNR	42.43	36.38	32.76	29.27	21.55	34.94	28.70	25.37	22.76	19.47	38.70	30.08	26.67	19.90	17.36	41.25	29.87	26.69	21.11	18.87
MAE	0.40	1.38	2.68	4.82	15.07	0.92	3.30	6.43	10.66	19.78	1.64	5.25	8.17	17.32	24.33	0.90	4.49	8.11	15.53	20.16

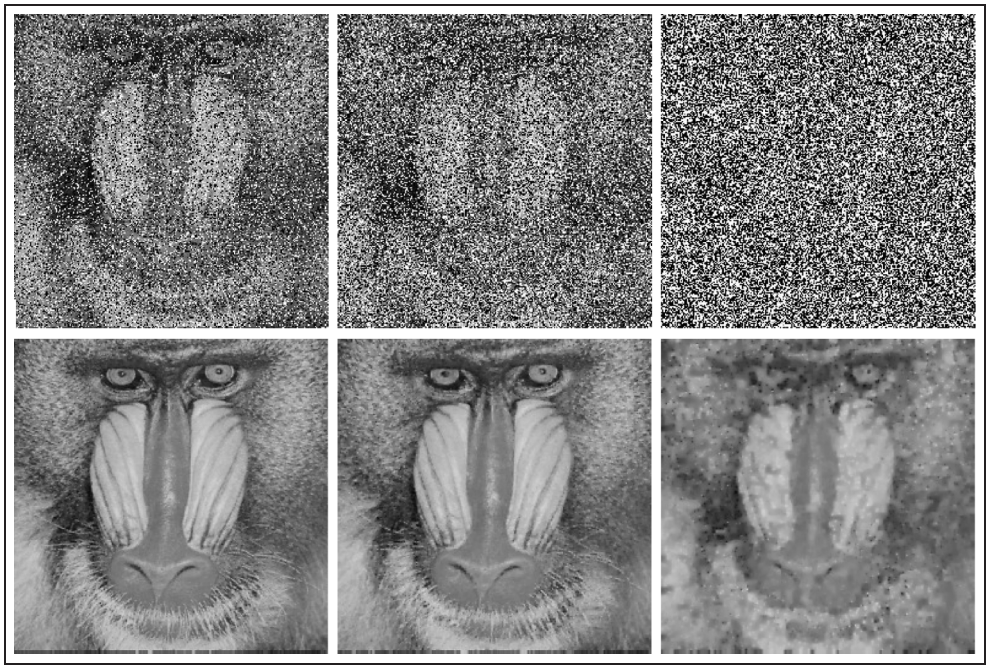

Figure 4. First row: Baboon image affected by 30%, 50% and 90% of Salt and Pepper noise. Second row: Restoration results.

pure Gaussian noise, while other existing methods (see Section 2) operate with combinations of noises (Cai et al., 2008).

4.2 Performance on images and video sequences

This section presents the obtained performance of the new visual data restoration process, implemented using the FastFlow *stencil-reduce* pattern. The performance has been evaluated on two different environments and compared with the multicore version (also implemented using FastFlow).

1. *The 16-core-HT + 2 × NVidia-M2090 platform*. Intel workstation with two eight-core double-context Xeon E5-2660 @2.2 GHz, 20 MB L3 shared cache, 256K L2, 64 GB of main memory and 2 NVidia Tesla M2090 GPGPU, Linux x86_64 and NVidia CUDA SDK 6.0.
2. *The 4-core-HT + NVidia-K40 platform*. Intel workstation with quad-core double-context Xeon E3-1245 v3 @3.4 GHz, 8 MB L3 shared cache, 256K L2, 16 GB of main memory and a NVidia K40 GPGPU, Linux x86_64 and NVidia CUDA SDK 6.0.

Table 2 presents the obtained execution times when restoring images corrupted with Salt and Pepper noise using the new FastFlow *stencil-reduce* GPGPU pattern, and the respective speedup comparing with the single-thread restoration on CPU. The results present consistent speed improvements with all noise levels, but the GPGPU version is clearly faster when dealing with highly corrupted images.

Figure 7 presents the multicore (CPU-only) execution times of the video restoration on *16-core-HT + 2 × NVidia-M2090 platform*. On the left (Salt and Pepper noise), the curves show that the amount of

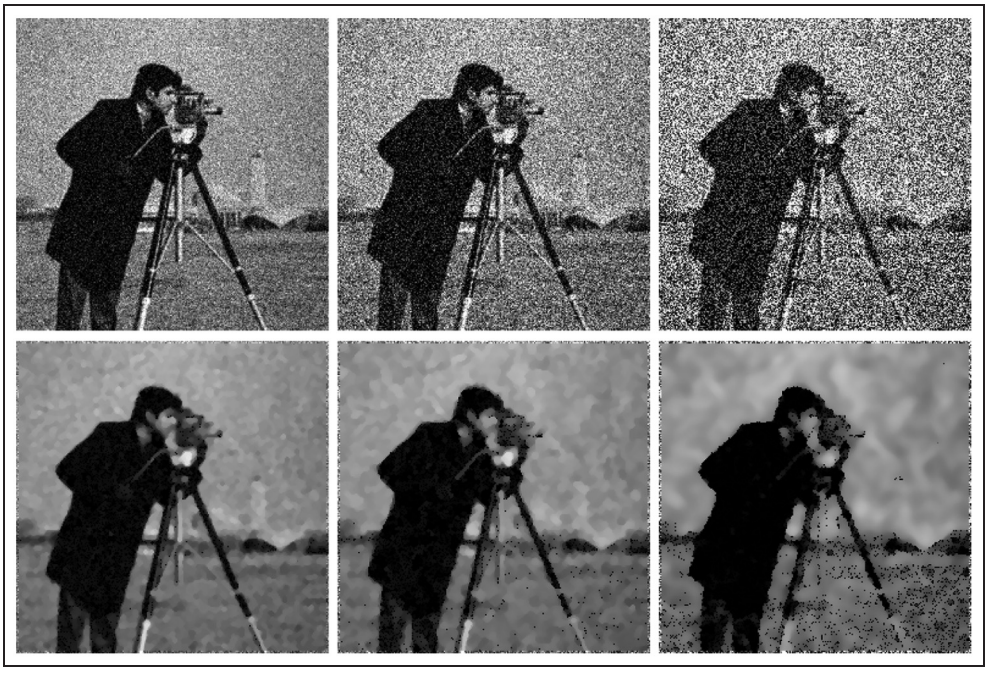

Figure 5. First row: Cameraman image affected by Gaussian noise with variances 30, 50 and 100. Second row: Restoration results.

Figure 6. Cameraman image affected by Gaussian noise with variance 50 (left image), restored with our filter (middle image) and restored with the Wiener filter using 5 × 5 neighbours (right image). The values of PSNR and MAE of our method are, respectively, 18.87 and 20.16, while those achieved by the Wiener filter are, respectively, 12.05 and 63.65.

noise directly affects the execution time. On the right, however, the curves follow the same pattern, which is explained by the fact that the added Gaussian noise affects the entire image and the detection stage uses a more sensitive threshold and, thus, the number pixels selected as noisy are not strongly related to the variance in these three cases. The obtained speedup, using 16 physical HT cores of the Salt and Pepper Restore varies from 13.2× to 13.8×, while the Gaussian Restore reaches a close-to-linear speedup of 15.8×.

Table 3 presents an overview of the obtained execution times and speedup on the two chosen execution environments. *Speedup vs multicore* (1CPU + 2GPGPUs vs 16CPUs and 1CPU + 1GPGPU vs 4CPUs) represents the performance gain of the GPGPU version over the multicore version using all of the available cores, whereas *Speedup vs sequential* (1CPU + 2GPGPUs vs 1CPU and 1CPU + 1GPGPU vs 1CPU) represents the gain over a version that uses only one thread for *Restore* stage. Since it is much longer compared with the other stages on the pipeline, this version can be considered as an upper bound for the sequential performance.

The filter has been also tested with a full length movie: a 1-hour video containing 107,760 frames of 400 × 304 pixels (30 fps), corrupted with 10%, 50% and 90% of impulse noise. This video has been restored using a NVidia K40 GPGPU and the obtained execution times are 1446, 2622 and 6424 s, respectively (yielding an average throughput of 75.52, 41.08 and 16.77 fps).

Table 2 Performance results for restoring two sample images corrupted with Salt and Pepper noise: Space (4096 × 4096 pixels) and Baboon (2048 × 2048 pixels), using the multicore version (with one Restore thread) and the version built upon the new *stencil-reduce* pattern. *"1CPU"* stands for the execution time using only one Restore thread and the other pipeline stages (input/output operations and noise detection) are performed in parallel.

		Space (4096 × 4096 pixels)			Baboon (2048 × 2048 pixels)		
		10%	50%	90%	10%	50%	90%
4-core-HT+NVidia-K40 platform	Time (s) – 1CPU	623.60	3084.36	5556.83	154.73	770.50	1385.86
	Time (s) – 1CPU+1GPGPU	4.55	17.03	28.73	1.22	4.33	7.26
	Speedup vs sequential – 1CPU+ 1GPGPU vs 1CPU	137.05	181.15	193.41	126.73	178.02	190.90
	PSNR	30.30	30.27	19.54	51.62	36.97	22.10
	MAE	0.32	2.34	20.49	0.13	1.63	14.37

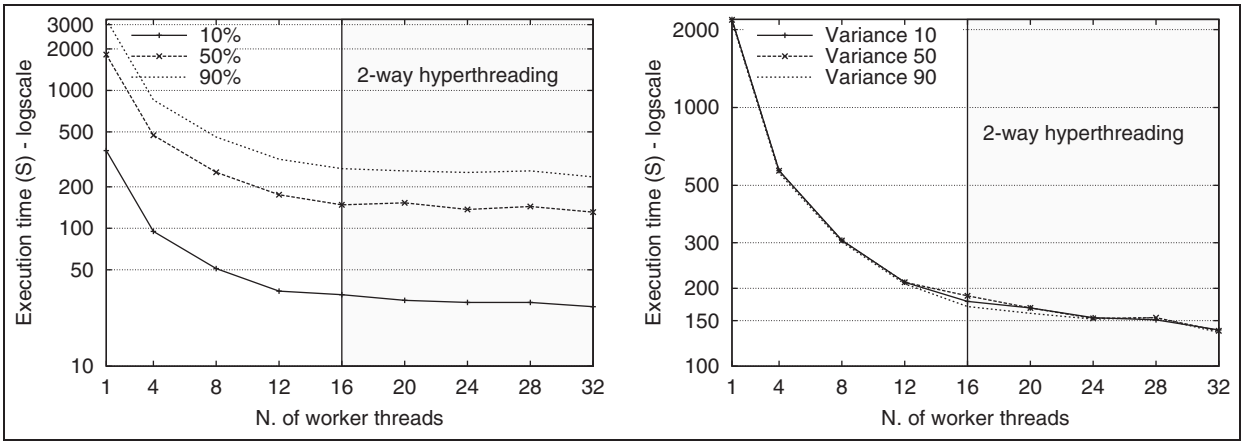

Figure 7. Overall restoration execution time (multicore) of a video containing 91 frames (768 × 512 pixels): on the left 10%, 50% and 90% of Salt and Pepper noise, on the right Gaussian noise with variance 10, 50 and 90 (and mean 0).

Table 3 Performance results for restoring a video containing 91 frames (768 × 512 pixels) on two environments, using the multicore version and the version built upon the new *stencil-reduce* pattern. *"1CPU"* stands for the execution time using only one thread for *Restore* stage and the other pipeline stages (input/output operations and noise detection) are performed in parallel.

		Salt and Pepper noise			Gaussian noise (mean 0)		
		10%	50%	90%	Var. 10	Var. 50	Var. 90
16-core-HT+2 × NVidia-M2090 platform	Time (s) – 16CPUs	27.66	131.87	236.42	138.99	137.21	135.83
	Time (s) – 1CPU + 2GPGPUs	3.94	16.86	30.43	10.33	10.30	10.14
	Speedup vs multicore – 1CPU+ 2GPGPUs vs 16CPUs	7.02	7.81	7.76	13.45	13.30	13.39
	Speedup vs sequential – 1CPU+ 2GPGPUs vs 1CPU	92.94	107.93	107.55	211.98	211.65	212.43
4-core-HT+ NVidia-K40 platform	Time (s) – 4CPUs	62.51	309.71	552.50	237.27	236.44	236.48
	Time (s) – 1CPU+1GPGPU	1.75	5.13	8.78	3.49	3.47	3.42
	Speedup vs multicore – 1CPU+ 1GPGPU vs 4CPUs	35.65	60.28	62.92	67.90	68.11	69.09
	Speedup vs sequential – 1CPU+ 1GPGPU vs 1CPU	120.85	204.70	211.87	383.76	382.07	388.22

5. Concluding remarks

In this work we presented a parallel visual data restoration tool built upon a high-level *stencil-reduce* pattern that allows to easily and efficiently program iterative applications of nonlinear filters on GPGPUs. The key results here are (1) the ease by which the user can write CUDA-enabled kernels, as the framework is in charge of memory management and offloading; and (2) the obtained high performance and GPGPU occupancy (including multi-GPGPU environments). Previous

work had already presented excellent performance results when restoring images affected by Salt and Pepper noise, while this work introduces a new functional for restoring Gaussian noise. Considering the good speedup and scalability preservation, we can conclude that the overhead generated by this mechanism is very low for this kind of (iterative) applications (Aldinucci et al., 2014b).

Moreover, the extension of the skeleton framework FastFlow to GPGPUs allowed us to seamlessly migrate the vision processing system to a heterogeneous setting and easily experiment with various assignments of application components to CPU/GPGPU devices.

From a machine vision perspective, the resulting application has succeeded to demonstrate the usability of variational restoration process, considered very effective but slow, also for real-time video streams. The experiments conducted have also highlighted several issues in developing a unified parallel programming framework for heterogeneous platforms that are worth further investigation (such as the ability of offloading OpenCL kernels using the new skeleton). Thus, they can steer further research in the area of parallel programming models for heterogeneous platforms.

6. Funding

This work has been supported by the EU FP7 Paraphrase project (no. 288570), EU FP7 REPARA project (no. 609666), Compagnia di San Paolo IMPACT project (no. ORTO11TPXK), and by the NVidia CUDA Research Center (programme 2013-2015).

References

Aldinucci M, Danelutto M, Kilpatrick P, Meneghin M and Torquati M (2011) Accelerating code on multi-cores with Fastflow. In *Proceedings of 17th international Euro-Par 2011 parallel processing*, Bordeaux, France, August 2011 (ed. Jeannot E, Namyst R and Roman J) (*Lecture Notes in Computer Science*, vol. 6853), pp. 170–181. New York: Springer.

Aldinucci M, Danelutto M, Kilpatrick P and Torquati M (2014a) Fastflow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems* (ed. Pllana S and Xhafa F) (*Parallel and Distributed Computing*, vol. 13). New York: Wiley.

Aldinucci M, Meneghin M and Torquati M (2010) Efficient Smith–Waterman on multi-core with fastflow. In *Proceedings of international Euromicro PDP 2010: parallel distributed and network-based processing*, Pisa, Italy, February 2010 (ed. Danelutto M, Gross T and Bourgeois J), pp. 195–199. IEEE.

Aldinucci M, Peretti Pezzi G, Drocco M, Tordini F, Kilpatrick P and Torquati M (2014b) Parallel video denoising on heterogeneous platforms. In *Proceedings of international workshop on high-level programming for heterogeneous and hierarchical parallel systems (HLPGPU)*.

Aldinucci M, Spampinato C, Drocco M, Torquati M and Palazzo S (2005) A parallel edge preserving algorithm for salt and pepper image denoising. In *Proceedings of 2nd international conference on image processing theory tools and applications (IPTA)*, Istambul, Turkey (ed. Djemal K, Deriche M, Puech W and Ucan ON), pp. 97–102. IEEE.

Aldinucci M and Torquati M (2014) *FastFlow website*. http://mc-fastflow.sourceforge.net/.

Astola J and Kuosmanen P (1997) *Fundamentals of Nonlinear Digital Filtering*. Boca Raton, FL: CRC.

Augonnet C, Thibault S, Namyst R and Wacrenier P-A (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23(2): 187–198.

Bovik A (2000) *Handbook of Image and Video Processing*. New York: Academic Press.

Cai J-F, Chan RH and Fiore C (2007) Minimization of a detail-preserving regularization functional for impulse noise removal. *Journal of Mathematical Imaging and Vision* 29(1): 79–91.

Cai J-F, Chan RH and Nikolova M (2008) Two-phase approach for deblurring images corrupted by impulse plus Gaussian noise. *American Institute of Mathematical Sciences* 2(2): 187–204.

Cai J-F, Chan R and Nikolova M (2010) Fast two-phase image deblurring under impulse noise. *Journal of Mathematical Imaging and Vision* 36(1): 46–53.

Cannavò F, Giordano D, Nunnari G and Spampinato C (2006) Variational method for image denoising by distributed genetic algorithms on grid environment. In *Proceedings of the 15th IEEE international workshops on enabling technologies: infrastructures for collaborative enterprises (WETICE-2006)*.

Chan R, Ho C and Nikolova M (2005) Salt-and-pepper noise removal by median-type noise detectors and detail-preserving regularization. *IEEE Transactions on Image Processing* 14: 1479–1485.

Chen S and Yang X (2007) A variational method with a noise detector for impulse noise removal. In *Scale Space and Variational Methods in Computer Vision (Lecture Notes in Computer Science*, vol. 4485). New York: Springer.

Chen T and Wu HR (2000) Impulse noise removal by multi-state median filtering. In *Proceedings 2000 IEEE international conference on acoustics, speech, and signal processing, 2000 (ICASSP'00)*, vol. 6, pp. 2183–2186.

Cole M (1989) *Algorithmic Skeletons: Structured Management of Parallel Computations (Research Monographs in Parallel and Distributed Computing)*. New York: Pitman.

Danelutto M and Torquati M (2014) Loop parallelism: a new skeleton perspective on data parallel patterns. In *Proceedings of international Euromicro PDP 2014: parallel distributed and network-based processing*, Torino, Italy, 2014. IEEE.

Enmyren J and Kessler CW (2010) SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on high-level parallel programming and applications (HLPP'10)*, New York, NY, pp. 5–14. New York: ACM Press.

Ernsting S and Kuchen H (2011) Data parallel skeletons for gpu clusters and multi-GPU systems. In *Proceedings of PARCO* 2011. IOS Press.

Faro A, Giordano D, Scarciofalo G and Spampinato C (2008) Bayesian networks for edge preserving salt and pepper image denoising. In *First workshops on image processing theory, tools and applications, 2008 (IPTA 2008)*, November 2008.

González-Vélez H and Leyton M (2010) A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software: Practice and Experience* 40(12): 1135–1160.

Hammond K, Aldinucci M, Brown C, et al. (2013) The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In *Formal methods for components and objects: international symposium (FMCO 2011)*, Torino, Italy, 3–5 October 2011, Revised Invited Lectures (ed. Beckert B, Damiani F, de Boer FS and Bonsangue MM) (*Lecture Notes in Computer Science*, vol. 7542), pp. 218–236. New York: Springer.

Huang TA, Yang GJ and Tang GY (1979) A fast two-dimensional median filtering algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 27(1): 13–18.

Intel Corp. (2014) *Threading Building Blocks*. http://www.threadingbuildingblocks.org/.

Khronos OpenCL Working Group (2008) *The OpenCL Specification, version 1.0.29*, 8 December 2008.

Lin T-C (2007) A new adaptive center weighted median filter for suppressing impulsive noise in images. *Information Sciences* 177(4): 1073–1087.

McCool M, Reinders J and Robison A (2012) *Structured Parallel Programming: Patterns for Efficient Computation* (1st edn). San Francisco, CA: Morgan Kaufmann Publishers Inc.

Mumford D and Shah J (1989) Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on Pure and Applied Mathematics* 42(5): 577–685.

Nikolova M (2004) A variational approach to remove outliers and impulse noise. *Journal of Mathematical Imaging and Vision* 20(1): 99–120.

Pacheco PS (1996) *Parallel Programming with MPI*. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Park I, Voss MJ, Kim SW and Eigenmann R (2001) Parallel programming environment for OpenMP. *Scientific Programming* 9: 143–161.

Stratton JA, Stone SS, Mei W and Hwu W (2008) MCUDA: An efficient implementation of CUDA kernels for multicore CPUs. In *Languages and Compilers for Parallel Computing, 21th International Workshop (LCPC)* (ed. Amaral JN) (*Lecture Notes in Computer Science*, vol. 5335), pp. 16–30. New York: Springer.

Vogel CR and Oman ME (1998) Fast, robust total variation-based reconstruction of noisy, blurred images. *IEEE Transactions on Image Processing* 7(6): 813–824.

Wang P (2001) *Computing with Words*. New York: Wiley.

Yang P and Basir O (2003) Adaptive weighted median filter using local entropy for ultrasonic image denoising. In *Proceedings of the 3rd international symposium on image and signal processing and analysis, 2003 (ISPA 2003)*, vol. 2, September 2003.

Yin L, Yang R, Gabbouj M and Neuvo Y (1996) Weighted median filters: a tutorial. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 43: 157–192.

Author biographies

Marco Aldinucci is an associate professor and lead the Parallel Computing research group at the CS Department of University of Torino, Italy. He is the P.I. of the NVidia CUDA research center at University of Torino. He received his PhD from University of Pisa in 2003, and has been researcher at the ISTI-CNR institute of the National Research Council of Italy (2003–2006). His research interests include programming models, languages and tools for parallel computing.

Guilherme Peretti Pezzi is a research associate at CS Department of the University of Torino, Italy. He obtained his PhD degree in 2011 at INRIA Sophia Antipolis, France. From 2007 to 2008 he has been a fellow at EIA-FR (Switzerland) and INRIA Sophia Antipolis. His current activities include the participation on the IMPACT project (Innovative Methods for Particle Colliders at the Terascale) and integration of FastFlow patterns with GPGPU architecture.

Maurizio Drocco is a PhD student and a member of the parallel computing Alpha research team at the Computer Science Department of the University of Torino. He is research associate at University of Torino since 2009 and he has been research intern at IBM Dublin Research Lab in 2013, within the HPC team. His research is focused on high-performance computing, in particular parallel programming and optimization on multicore and many-cores platforms.

Concetto Spampinato is assistant professor at the Department of Electrical, Electronics and Computer Engineering of the University of Catania (Italy) where he also received his PhD. His research interests lie in the areas of computer vision, pattern recognition and machine learning. He has been involved in several international projects (e.g. Fish4Knowlege, mEducator, etc.) and is part of the editorial boards and organizing committees of several international journals and conferences in the computer vision and pattern recognition areas.

Massimo Torquati is an assistant professor at the Computer Science Department of the University of Pisa, Italy. His main research interests include languages, tools and models for parallel and distributed computing. Over the years, he has contributed to the design and the development of a number of tools for parallel processing and he is one of the main designers and maintainers of the FastFlow parallel programming framework.