Parallel video denoising on heterogeneous platforms

M. Aldinucci, G. Peretti Pezzi, M. Drocco, F. Tordini
Computer Science Dept. - University of Torino, Italy
{aldinuc, peretti, tordini}@di.unito.it, maurizio.drocco@gmail.com

P. Kilpatrick
Computer Science Dept. - Queen's University Belfast, UK
p.kilpatrick@qub.ac.uk

M. Torquati
Computer Science Dept. - University of Pisa, Italy
torquati@di.unipi.it

Abstract

In this paper, a highly-effective parallel filter for video denoising is presented. The filter is designed using a skeletal approach, 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 multi-core machine, on GPGPU(s), or on both. The design and implementation of the filter are discussed, and an experimental evaluation is presented. Various mappings of the filtering stages are comparatively discussed.

Keywords skeletons, fastflow, parallel patterns, multi-core, OpenCL, GPGPUs, heterogeneous platforms

1. Introduction

The ever increasing computing power available from off-the-shelf processors has allowed researchers to extend the number of applications in image processing and machine vision. One important step in any machine vision system is the image restoration phase, which has attracted the attention of the image processing community, especially with the increasing importance of real-time analysis of digital images and videos for video surveillance, etc. Variational methods, which basically solve optimization problems, are well known for their effectiveness, but are rarely exploited in image restoration due to their high computational cost and complexity of tuning [12, 26] .

An efficient variational image restoration template based on *FastFlow* technology [4, 21] has been proposed in previous work [3]. Given a traditional noise reduction filter (e.g. based on adaptive median, adaptive center-weighted median) the template generates an efficient parallel variational filter running on both multi-core and GPUs. The resulting variational filter is edge-preserving and, compared with the original filter, exhibits a better Peak-Signal-to-Noise-Ratio (PSNR) and can restore images with higher levels of noise.

The proposed image restoration schema is organised in two successive stages: an early outline of an image is first *detected* and then it is *denoised*. The schema is implemented according to a high-level pattern-based approach (a.k.a. skeleton approach [15]), and deployed as a sequence of detect and denoise stages that are defined according to the *map* parallel paradigm and the *map-reduce* parallel paradigm, respectively.

This paper extends the previous work in two directions:

1. The restoration schema is applied to video streams as opposed to a single image, and is expressed using *pipeline* parallelism between the two stages.
2. The implementation, realised by way of the FastFlow programming framework, is tested on a heterogeneous platform composed of a multi-core equipped with two GPGPU devices.

Thus, this work incorporates the business code of the earlier work [3]: the difference from the earlier work lies in the nature of the skeleton framework in which the business code is placed. Here the sequential composition of stages is replaced by a two stage pipeline through which a stream of images is directed. In addition, the FastFlow framework itself has evolved since the earlier work: now it incorporates the facility to deploy applications onto heterogeneous platforms comprising both CPUs and GPGPUs. This has an interesting implication for the application: originally the second stage (denoising) was orders of magnitude slower than the first stage (noise detection) and so there was no point in parallelising the first stage. Now, as will be seen, with the denoise stage on a GPGPU, in some cases we expect it to become faster than the (sequential) noise detection stage and so it is worthwhile also to parallelise the noise detection stage.

For the sake of simplicity, in this paper impulsive noise (e.g. salt-and-pepper) is considered. The same approach has been shown to be effective for several types of noise (e.g. Gaussian). The analysis of restoration quality for different types of noise is beyond the scope of the present work.

The structure of the paper is as follows: in the next section the denoising filter is reviewed by presenting some related work and its mathematical foundations; also, algorithmic skeleton solutions are introduced and reviewed. Section 3 shows the performance of the sequential implementation of the described filter, while section 4 and section 5 present, respectively, the parallel implementation of the proposed algorithm over heterogeneous architecture and the experimental results. Finally, in the last section concluding remarks are provided.

2. Background

In this section the image restoration process will be introduced, by presenting its mathematical foundations and some related work.

HLPGPU '14, January 21, 2014, Vienna, Austria.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Then an overview of state-of-the-art structured parallel programming frameworks is given, together with an introduction to the FastFlow framework. Finally, the original FastFlow-based image restoration application, which is the forerunner of the video denoiser, is described.

2.1 Variational Image Restoration: Related Work

In the past fifteen years a large number of methods have been proposed to deal with salt and pepper noise (and, more generally, impulse noise) from digital images [5]. 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 non-linear filter for removing impulse noise, because of its good denoising power [8] and its computational efficiency [22], but it affects image detail while removing noise. This issue has generally been addressed by filtering techniques based on median filter modifications[25, 36]. However, the performance of median filtering based approaches is unsatisfactory in suppressing signal-dependent noise [35] when the noise percentage is high (more than 50%). To achieve a good compromise between the image-detail preservation and the noise reduction, an impulse detector must be used before filtering. Several types of impulse detectors exist: the most famous is the progressive switching median (PSM) [33]. Machine learning approaches have also been widely used in recent years, e.g. approaches relying on Bayesian networks [19], fuzzy logic [34] and neuro-fuzzy [37]. The filtering is then selectively applied to the noisy regions detected by the noise detector. To the best of our knowledge, one of the most effective algorithms for edge preservation in salt and pepper denoising has been proposed by Nikolova in [26]: it applies a variational method for image detail preservation that is based on a data-fidelity term related to the impulse noise. Based on this approach Chan *et al.* in [12] (called for simplicity Chan's method) proposed a powerful filter capable of removing salt and pepper noise as high as 90%. Similar approaches to Chan's method, aimed at improving the noisy detection step and at reducing the processing times, are those proposed in [9–11, 14, 19].

2.1.1 Detection of Outliers for Salt-and-Pepper Noise

Let $\hat{y}$ be the map of noisy pixels (obtained by applying the adaptive median filter classifier to the noisy image) which has a 1 in the corresponding position for a noisy pixel, and a 0 for an uncorrupted pixel. Hence the set of noisy pixels N (to which the restoration algorithm has to be applied) consists of the overall pixels of the original image y whose values in the $\hat{y}$ map are equal to 1, i.e.: $N = \{(i,j) \in \mathrm{A} : \hat{y}_{i,j} = 1\}$

The set of all uncorrupted pixels is $N^c = A \setminus N$, where A is the set of all pixels. Note that, since pixels with colour different from 0 and from 255 are uncorrupted, the AMF filter [23] can be modified to exclude them with significant benefits in term of performance and false positive rate.

2.1.2 Variational Denoising

The problem of image restoration for edge preserving is an inverse problem solved by using regularization, where the restored image u is obtained by solving the following optimization problem restricted to the set of the noisy pixels N.

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

where d is the image corrupted by the noise; $D(u,d)$ is the data-fidelity term which is related to the kind of noise and provides a measure of the deviation between d and the output image u; and $R(u)$ is a regularization term that uses a-priori knowledge for enforcing the solution and should be represented by a function that penalizes/removes only irregularities due to the affecting noise, thus ignoring high-level discontinuities (edges). β and α are the regularization parameters that balance the effects of both mentioned terms. Among the functionals $F(u)$ (see [13]) for edge preserving proposed during the last fifteen years, we have selected the one proposed in [12] which has been shown to be very effective for Salt-and-Pepper noise. The same performance might not be guaranteed for a different type of noise. However, it is crucial to note that substituting $F(u)$ by a different functional does not require rewriting of the code, since the whole parallel denoising process is a higher-order function where $F(u)$ is a parameter. In the present case:

$$F_d|_N(u) = \sum_{(i,j) \in N} [|u_{i,j} - d_{i,j}| + \frac{\beta}{2}(S_1 + S_2)] \tag{2}$$

where

$$S_1 = \sum_{(m,n) \in V_{i,j} \cap N} 2 \cdot \varphi(u_{i,j} - d_{m,n}) \tag{3}$$

$$S_2 = \sum_{(m,n) \in V_{i,j} \cap N^c} \varphi(u_{i,j} - u_{m,n}) \tag{4}$$

where N represents the noisy pixels set, N^c the set of uncorrupted pixels, and $V_{i,j}$ is the set of the four closest neighbours of the pixel with coordinates (i,j) and d is the corrupted image. As in [12], we have used the following φ function that provides the best trade-off between edge preserving and denoising: $\varphi(t) = |t|^\alpha$ $\varphi(t) = |t|^\alpha$ with $1 < \alpha \leq 2$. The values of α and β were, respectively, set to 1.3 and 4 in order to guarantee the trade-off between noise removal and edge preservation provided by the function φ.

A video stream is restored by independently filtering each of its frames. The restoration of a single frame follows an iterative process. At each iteration, for all outliers, its value is updated with the value u which minimizes the functional (1). The iterative process is stopped in accordance with a quasi-Newton method [7] using no PSNR variation across successive iterations as fix-point. The presentation of algorithmic details, which is beyond the scope of the present work, can be found in [3].

2.2 Algorithmic Skeletons

Algorithmic skeletons have been around since the '90s as an effective means of parallel application development. An algorithmic skeleton is a general-purpose, parametric parallelism-exploitation pattern [15]. 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.

MPI is often considered as a solution for writing efficient parallel applications [27]. 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* [28] is a popular thread-based framework for multi-core 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 parallelisation of sequential programs; however, these pragmas are mainly designed to

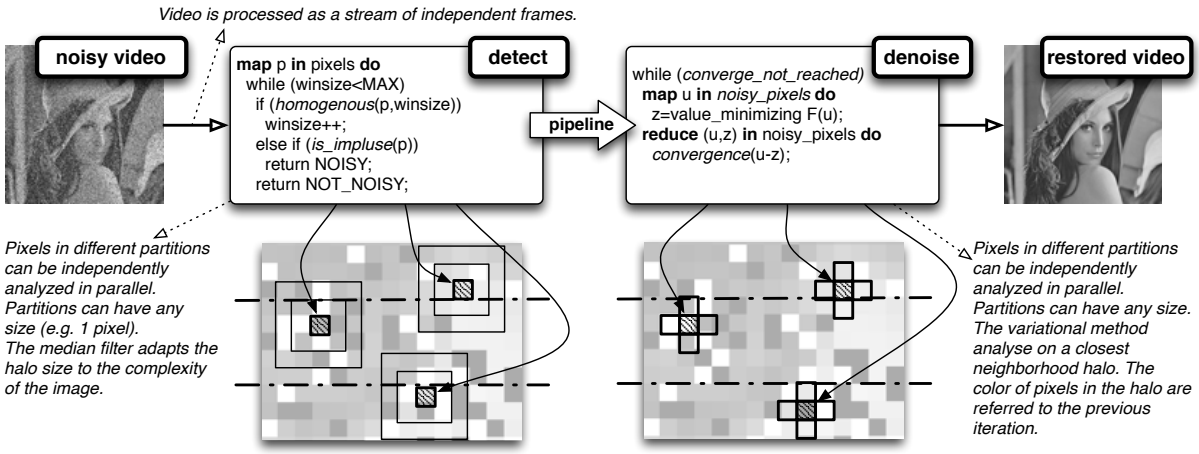

Figure 1. Two-phase denoiser.

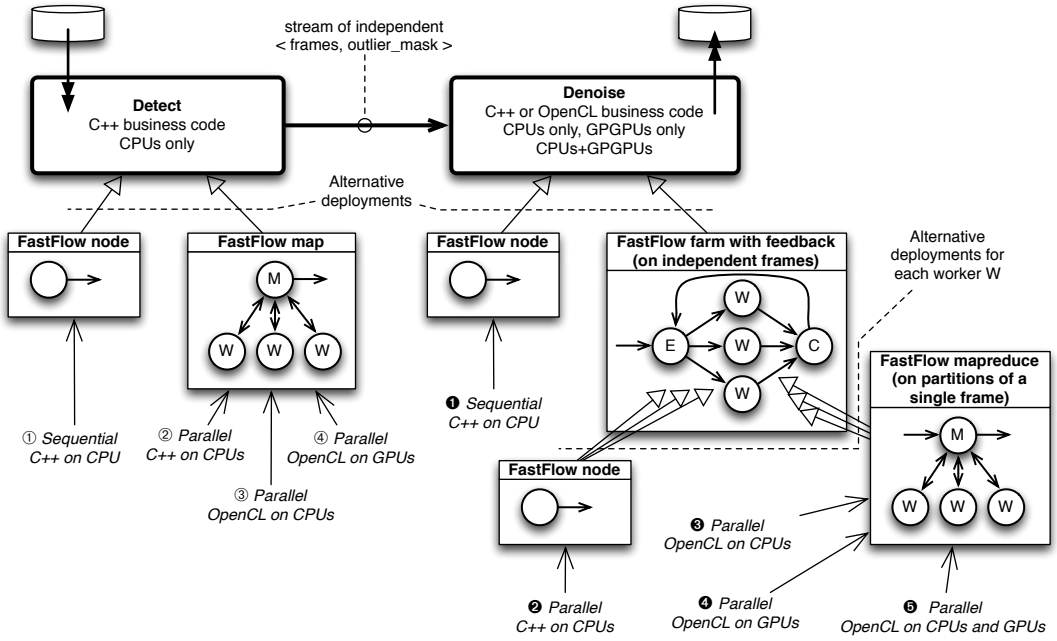

Figure 2. Variational denoiser: architecture and refinement lattice.

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) [32] is a C++ template library which provides easy 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 optimisations of the code.

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. OpenCL is a parallel API provided for GPGPU programming, which allows the users to exploit GPUs for general purpose tasks that can be parallelised [24]. 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. OpenCL is quite low level, focusing on low-level feature management rather than high-level parallelism exploitation patterns. It has the capability to revert to the CPU for execution when there is no GPU in the system. Its portability makes it suitable for Hybrid(CPU/GPU) or cloud based environments. In Muesli [18] the programmer must explicitly indicate whether GPUs are to be used for data parallel skeletons. StarPU [6] is focused on handling accelerators such as GPUs. 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 each architecture. SkePU [17] provides programmers with GPU implementations of map and reduce skeletons

and relies on StarPU for the execution of stream parallel skeletons (pipe and farm). MCUDA [31] is a framework to mix CPU and GPU programming. In MCUDA it is mandatory to define kernels for all available devices but the framework can not make any assumptions about the relative performance of the supported devices.

2.3 The FastFlow Programming Framework

The FastFlow parallel programming environment was originally designed to support efficient streaming on cache-coherent multi-core platforms. It is realised as a C++ pattern-based parallel programming framework aimed at simplifying the development of applications for (shared-memory) multi-core and GPGPUs platforms. The key vision of FastFlow is that ease-of-development and run-time efficiency can both be achieved by raising the abstraction level of the design phase. It provides developers with a set of parallel programming patterns (aka *algorithmic skeletons*) such as *farm*, *divide&conquer*, *pipeline*, *map*, *reduce*, and their arbitrary nesting and composition is supported [1, 4]. Map and reduce patterns can be run both on multi-cores and offloaded onto GPGPUs. In the latter case, the business code can include GPGPU-specific statements (i.e. CUDA or OpenCL statements).

The latest extensions of the FastFlow framework, aimed at supporting GPGPUs (via OpenCL) and CPU+GPGPU platforms, make it possible to easily port the application to heterogeneous platforms. OpenCL is certainly a suitable solution for programming hybrid architectures, as its specifications have been implemented by several hardware vendors, thus making the same OpenCL code runnable either on different GPGPU devices or on CPUs [30]. More details on the FastFlow OpenCL run-time can be found in [20].

FastFlow CPU implementation of patterns are realized via non-blocking graphs of threads connected by way of lock-free channels [4], while the GPU implementation is realized by way of the OpenCL bindings and offloading techniques [2]. Also, 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 business code running on a GPU can be further hand-tuned by exploiting OpenCL specific features, This kind of fine-tuning might bring significant performance improvements but, in general, is difficult to automate at the FastFlow level, since it is very related to the business code and requires extensive low-level programming effort.

Note that FastFlow does not provide any automatic facility to convert C++ code into OpenCL code. Thus any parallel activity (e.g. a map body) that can be possibly mapped onto a GPGPU should be coded with OpenCL. FastFlow, however, helps this task with a number of features including:

- Integration of the same pattern-based parallel programming model for both CPUs and GPGPUs. Parallel activities running on CPUs can be either coded in C++ or OpenCL.
- Setup of the OpenCL environment.
- Simplified data feeding to both software accelerators and hardware accelerators (with asynchronous Host-to-Device and Device-to-Host data movements) [2].

In this work, we consider the extension of the two-phase image denoiser to a two-phase *video* denoiser.

2.4 A High-Level Pattern-Based Approach for Image Restoration

As we already stated above, the image restoration problem can be tackled by applying the variational filter independently to each frame of the image. In order to optimally parallelise this task, we propose a structured approach using high-level algorithmic skeletons.

2.4.1 Two-phase Edge-Preserving Image Restoration

Variational filtering is performed using a two-stage approach:

1. *Detect*. Accurate detection of the location of noise (the outlier candidates) using a parallel implementation of the original filter.
2. *Denoise*. Edge-preserving restoration of outlier pixels by way of an iterated application of a variational filter up to a global convergence criterion (e.g. no PSNR increment in the last step). The full knowledge of the outlier mask at this stage helps in weighting the optimization function on neighbour pixels.

The detect stage can be implemented by plugging the code of a standard sequential filter into a FastFlow map template (i.e. a higher-order parallel pattern implemented as a header-only C++ template), which provides all the additional code to partition the image; dispatch partitions to a (dynamically) malleable set of workers; and join partial results. The outlier mask is produced as the difference between the restored and the noisy images. FastFlow lock-less run-time support ensures minimal coherence traffic and a close-to-optimal speedup on cache-coherent multi-core.

The denoise stage, computationally more demanding, is implemented as a map-reduce pattern. As we shall see, this stage is implemented in OpenCL in such a way that it can be run on both CPUs, GPGPUs, and CPUs+GPGPUs. A C++ version of the same code is used as a baseline for comparison purposes. Also, due to the fact that the restoring code is almost independent of noise type, the kernel does not need to be recoded for different filters (possibly exhibiting different or dynamic convolution kernel stencils).

The two stages can be pipelined to guarantee high or real-time throughput on video applications and the full utilization of the coupled multi-core+GPU platform. Typically, the detect stage is fast, orders of magnitude faster than the variational denoise stage. The GPGPU's higher computational power naturally balances higher computational requirements on the second stage. If needed, the FastFlow run-time is able to dispatch different frames to different GPU devices dynamically (autonomically) acquiring new devices during the application execution.

The design of the proposed schema is summarised in Fig. 1.

2.4.2 Effectiveness of Two-Phase Denoiser on Images

The denoising quality of the two-phase image restoration was tested on standard images of different size. To compare results with the literature, the results reported in this section are in relation to two 512x512 8-bit grayscale images: Lena (Fig. 3) featuring homogeneous regions and Baboon (Fig. 4) characterised by high activity. These images have been corrupted by 70% and 90% salt-and-pepper noise.

In [3] has already been demonstrated the good performance of the restoration algorithm for one single image (for both the Lena and Baboon images of size 512x512 with 90% of noise a performance improvement of more than 25x on a Intel 32-core platform has been achieved using FastFlow).

With respect to the quality of restoration, the low rate of false positives in the detection of noisy pixels achieved by the adaptive median filter brings an overall improvement of the quality of restoration as non-noisy pixels are not changed by the denoise phase. We noticed an improvement of both PSNR (Peak Signal-to-Noise Ratio) and MAE (Mean Absolute Error) with respect to state-of-the-art results reported in the literature, e.g. Chan's method

Figure 3. Lena: 70% & 90% of noise and restored version.

[12], Cai's method [10, 19] , Chen's method [14] (e.g. 75s for Lena 256x256 with 90% of noise, PSNR=24.42, MAE=8.81). The detailed analysis of the quality of our approach on different test images of different sizes can be found in [3].

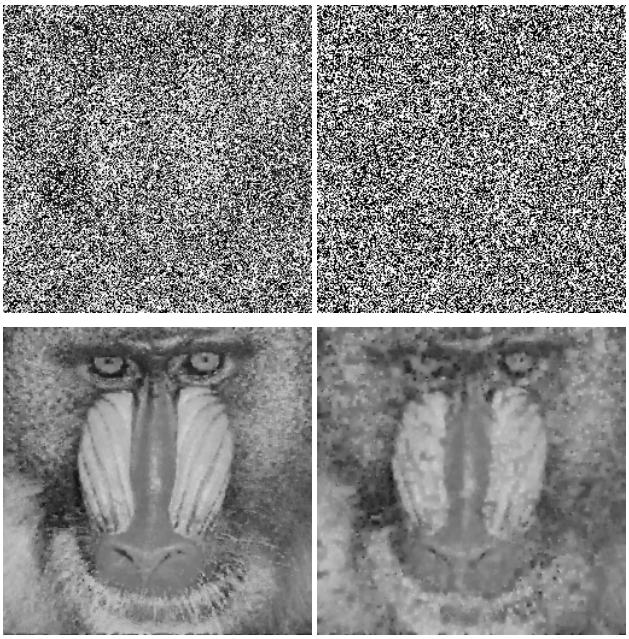

Figure 4. Baboon: 70% & 90% of noise and their restored versions.

3. Video Denoising: a Skeleton-based Approach

We advocate the *simple* extension of the application working for images presented in [3] to video, and to heterogeneous platforms. We emphasise the word "simple". 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 Two-Phase Video Denoiser

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 Fig. 1. The two phases can operate in a pipeline in the case where the two-phase denoiser is used on a stream of images (e.g. in a video application). In addition, both filters can be parallelised in a data-parallel fashion. 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 $\oplus$ is a binary associative operator, and $[a_0, a_1, \ldots]$ an array of pixels (e.g. the pixels of an image).

Detect is the first phase of the algorithm, where the Adaptive Median Filter is applied to the noisy image. Each pixel of the image is processed independently, provided the processing element can access a read-only halo of the pixel. The process is expressed sequentially over the elements of an array: each pixel of the image is analysed. A set of (candidate) noisy pixels is produced. The parallel processing here is exploited by way of the FastFlow map template, which can be easily added to the code using a ParallelFor high-level pattern provided by the FastFlow framework to accelerate for loops [16]. The ParallelFor pattern is used on the outer loop traversing the pixel matrix, which partitions the work on matrix strips. However, for small frame sizes, as those considered in the experimental section of this work, the detection phase has a negligible execution time w.r.t. the denoising phase, so the parallelisation of the Detect phase is not really necessary.

Denoise is the second phase of the algorithm. The variational method can be computed independently for each pixel (with the previous iteration halo). This step is iterated until a convergence criterion is reached. The convergence criterion is a global propriety of the image, and so it should be computed via an associative operator by reducing the difference between the last two iterations according to a given convergence criterion. The parallel processing is described as a loop of a sequence (i.e. functional composition) of map and reduce patterns. The map works with a fixed stencil of eight neighbours for each outlier pixel (except for borders). The reduce pattern computes the global convergence criterion, to be checked in the loop exit condition. Using the OpenCL back-end, it is possible to specify the way in which the underlying heterogeneous architecture has to be exploited for the computation.

In the sequential version, the detect stage exhibits linear execution time with respect to the total number of pixels. However, the detect phase, typically a simple convolution, is two–three orders of magnitude faster than a single iteration of the second step (denoise). The second phase shows a computational complexity of $\mathcal{O}(n\_noisy\_pixels \cdot n\_iterations)$, where n_noisy_pixels is the number of noisy pixels identified in the first step, and $n\_iterations$ is the number of iterations required to reach one of the convergence criteria. On a fully CPU deployment, a sequential detect stage is able to sustain the throughput of the parallel denoise stage even

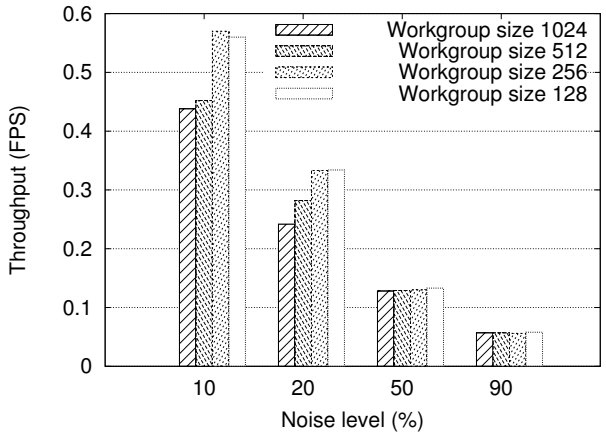

Figure 5. Throughput (in Frame Per Second) of the OpenCL denoise stage on CPUs: performance against working group size.

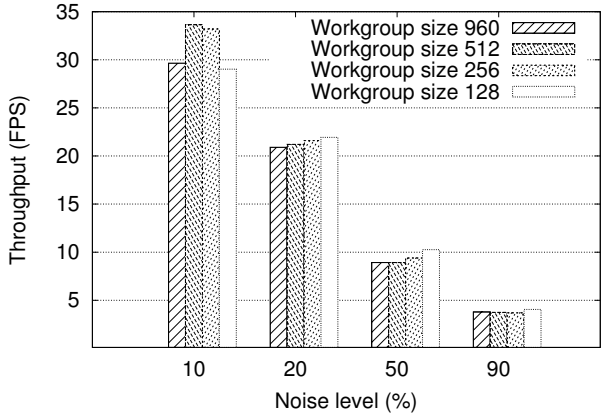

Figure 6. Throughput (in Frame Per Second) of the OpenCL denoise stage on GPGPUs: performance against working group size.

in a large multicore platform. This is no longer true if the denoise stage is run on one or more GPGPUs.

4. Experimental Evaluation

All experiments reported in this section were conducted on an Intel workstation with 2 eight-core double-context (2-way hyperthreading) Xeon E5-2660 @2.2GHz, 20MB L3 shared cache, 256K L2, and 64 GBytes of main memory (also equipped with two NVidia Tesla M2090 GPGPU) with Linux x86_64.

Three main families of experiments are reported:

1. Performance for the denoise stage in the OpenCL version on both CPUs (Fig. 5) and GPU (Fig. 6) on different work group sizes (i.e. different parallelism degrees).
2. Elapsed execution time for the C++ version on CPUs varying the number of threads used for the entire application (Fig. 7).
3. Maximum performance on the given platform for different versions of the code (C++ against OpenCL) and different noise levels.

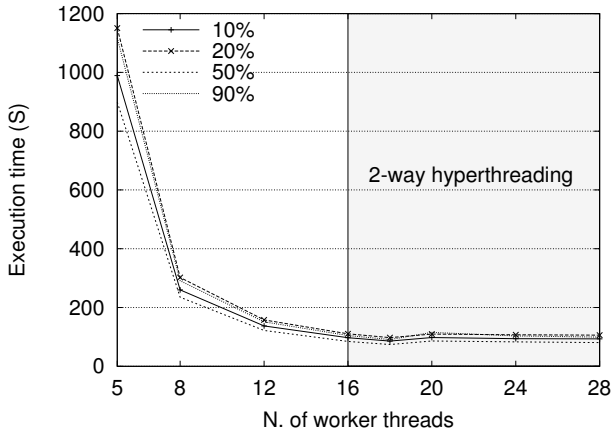

Figure 7. Execution time (in seconds) for computing k frames of the C++ video denoiser on CPUs where k is 800, 400, 60 and 30 for the different noise level, respectively.

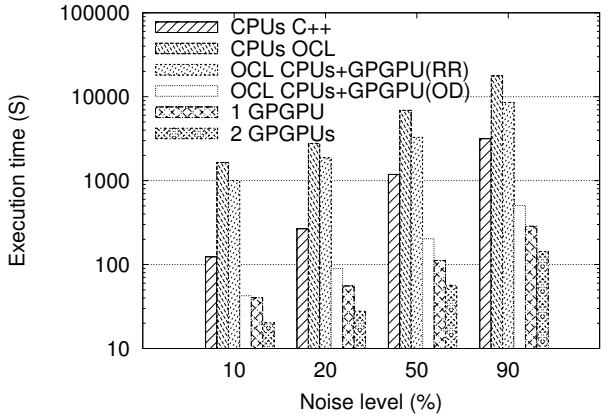

Figure 8. Minimum execution time (in seconds) for denoising 1160 frames (480x360 pixels) using different application deployments and different data sets.

A brief description of the different cases tested using only CPUs and both CPUs and GPGPUs is reported in Table 1.

In Fig. 5 and Fig. 6 the performance of the denoise stage is analysed for different sizes of OpenCL working groups. Here, the farm implementing the denoise stage is set to a single worker, i.e. no parallelism across different frames is exploited. In this case, the FastFlow Emitter and Collector (see Fig. 2) have been removed. It can be noticed that the working group size affects actual performance only for low levels of noise, i.e. for small problems. In the case of deployment on CPUs this leads to very fine grained parallelism and thus a significant synchronisation overhead.

Figure 7 reports the execution time obtained running the FastFlow C++ version deployed onto CPUs again in the case where the farm implementing the denoise stage is set to a single worker, i.e. no parallelism across different frames is employed. The exploited parallelism in the denoise stage comes entirely from the map pattern. The maximum speedup obtained with respect to the sequential CPU time ranges from 10.8 for 10% noise to 11.4 for 90% noise.

Experiment name	Deployment (Fig. 2)	Brief description
CPUs C++	①+❷	Detect C++/1CPU, Denoise C++/14CPUs
CPUs OCL	①+❸	Detect C++/1CPU, Denoise OpenCL/14CPUs
OCL CPUs+GPGPU(RR)	①+❺	Detect C++/1CPU, Denoise OpenCL 14CPUs+1GPGPU, Round-Robin scheduling
OCL CPUs+GPGPU(OD)	①+❺	Detect C++/1CPU, Denoise OpenCL 14CPUs+1GPGPU, On-Demand scheduling
1 GPGPU	①+❹	Detect C++/1CPU, Denoise OpenCL/1GPGPU
2 GPGPUs	①+❹	Detect C++/1CPU, Denoise OpenCL/2GPGPUs

Table 1. Description of the deployments tested for the two pipeline stages of the video denoiser application on the considered platform. In this particular case, being the Detect stage much faster than Denoise stage, not all deployments reported in Fig. 2 have been tested (i.e. ones including ②, ③, ④, and ❶).

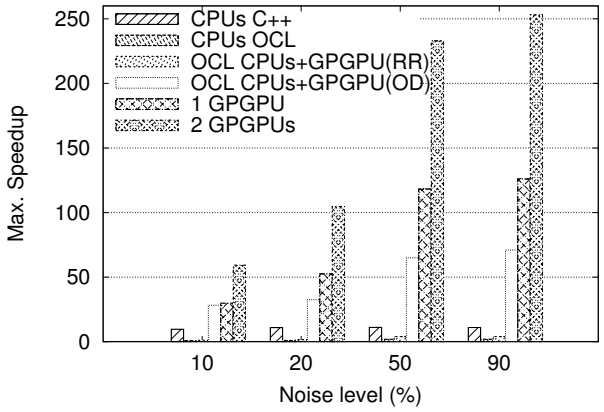

Figure 9. Maximum speedup (compared to sequential CPU version) for different application deployments and different data sets.

It can be noticed that the executions reported in Fig. 5 and Fig. 7 exploit exactly the same parallelism patterns on the same platform. They are actually the OpenCL and C++ versions of the same FastFlow application (using OpenCL and C++ as guest languages). They can therefore be safely compared in terms of absolute performance in the same graph (the CPUs C++ and CPUs OCL bars in Fig. 8) where the minimum overall execution time for the entire video denoising is reported. However, the maximum performance reached by the OpenCL version with sufficient working groups to saturate all cores is very low (it is comparable with the plain C++ version with parallelism degree 2 for the denoise stage). Such low performance results were in part expected since the OpenCL code was optimized for GPGPUs and not for CPUs. As noticed in some recent research work [29], performance of OpenCL code on CPUs may be improved tuning several aspects such as: host-to-device and device-to-host memory copies, increasing the computation granularity and avoid explicit caching of data through local memory which introduces unnecessary overhead on multi-core. This tuning is partly due to currently available OpenCL compiler technology, which is not yet fully mature.

Figures 8 and 9 compare all versions for maximum performance in terms of overall execution time of the whole restoration pipeline. As expected, mapping the denoise stage onto GPGPUs greatly accelerates the application. Also, multiple GPGPUs can be used with almost linear speedup. It is worth noticing that in the OpenCL case it is possible to deploy the denoise stage on both CPUs and GPGPUs, actually mapping different workers of the farm on different devices, thus fully using the aggregate computational power of CPUs and GPGPUs for the denoise stage. In order to illustrate this case, two different scheduling policies are presented in Fig. 8: an order-preserving strict round-robin (RR) policy and the auto-scheduling policy implemented using the on-demand (OD) FastFlow scheduling pattern. While the order-preserving policy can perform well on homogeneous environments, in this case it bounds the performance to the slowest device. On-demand policy manages to overcome this problem but at the price of some additional overhead due to frame buffering and reordering. The speed edge between CPUs and GPGPUs forces either suspension of the dispatching of frames to GPUs or storage of out-of-order frames in a buffer, inducing a high dynamic allocation pressure on the system and a jerky output stream.

It can be argued that, in the case of an ordered stream as the one produced by streaming frames of a video, the extra tuning effort needed for a proper deployment of a mixed CPUs and GPGPUs architecture for the denoising phase is not worth the performance gain (if any).

5. Concluding Remarks

In this work we have built upon earlier work on single images to produce a parallel video denoiser for heterogeneous platforms. The key result here is the ease with which the earlier system could be extended: in essence we were able simply to lift noise detection and denoising code from the earlier system and insert it in a new skeleton framework to turn image processing into vision processing. The ease with which this could be achieved is testament to the abstraction powers of skeleton programming.

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

From a machine vision perspective, the resulting application has succeeded to demonstrate the usability of variational denoising, considered very effective but slow, also for real-time video streams. The experiments conducted have also highlighted several issues in development of a unified parallel programming framework for heterogeneous platforms that are worth further investigation, and thus can steer further research in the area of parallel programming models for heterogeneous platforms.

Acknowledgement

This work has been supported by the EU FP7 grant IST-2011-288570 “ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems” and Compagnia di San Paolo project id. ORTO11TPXK “IMPACT: Innovative Methods for Particle Colliders at the Terascale”.

References

[1] M. Aldinucci and M. Torquati. *FastFlow website*, 2013 (last accessed). `http://mc-fastflow.sourceforge.net/`.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating code on multi-cores with fastflow. In E. Jeannot, R. Namyst, and J. Roman, editors, *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, pages 170–181, Bordeaux, France, Aug. 2011. Springer. .

[3] M. Aldinucci, C. Spampinato, M. Drocco, M. Torquati, and S. Palazzo. A parallel edge preserving algorithm for salt and pepper image denoising. In K. Djemal, M. Deriche, W. Puech, and O. N. Ucan, editors, *Proc of 2nd Intl. Conference on Image Processing Theory Tools and Applications (IPTA)*, pages 97–102, Istambul, Turkey, 2012. IEEE. .

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, Mar. 2014.

[5] J. Astola and P. Kuosmanen. *Fundamentals of Nonlinear Digital Filtering*. Boca Raton, CRC, 1997.

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

[7] J. F. Bonnans, J. C. Gilbert, C. Lemaréchal, and C. A. Sagastizábal. *Numerical Optimization: Theoretical and Practical Aspects (Universitext)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[8] A. Bovik. *Handbook of Image and Video Processing*. Academic Press, 2000.

[9] J.-F. Cai, R. H. Chan, and C. Fiore. Minimization of a detail-preserving regularization functional for impulse noise removal. *J. Math. Imaging Vis.*, 29(1), 2007. ISSN 0924-9907.

[10] J.-F. Cai, R. Chan, and M. Nikolova. Fast two-phase image deblurring under impulse noise. *Journal of Mathematical Imaging and Vision*, 36, 2010. ISSN 0924-9907.

[11] F. Cannavò, D. Giordano, G. Nunnari, and C. Spampinato. Variational method for image denoising by distributed genetic algorithms on grid environment. In *Proc. of the 15th IEEE Intl. Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2006)*, 2006.

[12] R. Chan, C. Ho, and M. Nikolova. Salt-and-pepper noise removal by median-type noise detectors and detail-preserving regularization. *IEEE Trans. on Image Processing*, 14(10), October 2005.

[13] R. Chartrand and V. Staneva. Total variation regularisation of images corrupted by non-gaussian noise using a quasi-newton method. *Image Processing, IET*, 2(6):295 –303, Dec. 2008. .

[14] S. Chen and X. Yang. A variational method with a noise detector for impulse noise removal. In *Scale Space and Variational Methods in Computer Vision*, volume 4485 of *LNCS*. Springer, 2007.

[15] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

[16] M. Danelutto and M. Torquati. Loop parallelism: a new skeleton perspective on data parallel patterns. In *Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*, Torino, Italy, 2014. IEEE.

[17] J. Enmyren and C. W. Kessler. SkePU: a multi-backend skeleton programming library for multi-gpu systems. In *Proc. of the fourth Intl. workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[18] S. Ernsting and H. Kuchen. Data parallel skeletons for gpu clusters and multi-gpu systems. In *Proceedings of PARCO 2011*. IOS Press, 2011.

[19] A. Faro, D. Giordano, G. Scarciofalo, and C. Spampinato. Bayesian networks for edge preserving salt and pepper image denoising. In *Image Processing Theory, Tools and Applications, 2008. IPTA 2008. First Workshops on*, Nov. 2008.

[20] M. Goli and H. Gonzalez-Velez. Heterogeneous algorithmic skeletons for fast flow with seamless coordination over hybrid architectures. *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, 0:148–156, 2013. .

[21] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. González-Vélez, P. Kilpatrick, R. Keller, M. Rossbory, and G. Shainer. The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In B. Beckert, F. Damiani, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects: Intl. Symposium, FMCO 2011, Torino, Italy, October 3–5, 2011, Revised Invited Lectures*, volume 7542 of *LNCS*, pages 218–236. Springer, 2013.

[22] T. S. Huang, G. J. Yang, and G. Y. Tang. A fast two-dimensional median filtering algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 27(1), 1979.

[23] H. Hwang and R. A. Haddad. Adaptive median filters: new algorithms and results. *IEEE Trans Image Process*, 4, 1995.

[24] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008. URL `http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf`.

[25] T.-C. Lin. A new adaptive center weighted median filter for suppressing impulsive noise in images. *Inf. Sci.*, 177(4), 2007. ISSN 0020-0255. .

[26] M. Nikolova. A variational approach to remove outliers and impulse noise. *Journal of Mathematical Imaging and Vision*, 20(1), 2004.

[27] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1-55860-339-5.

[28] I. Park, M. J. Voss, S. W. Kim, and R. Eigenmann. Parallel programming environment for OpenMP. *Scientific Programming*, 9:143–161, 2001.

[29] J. Shen, J. Fang, H. J. Sips, and A. L. Varbanescu. Performance gaps between OpenMP and OpenCL for multi-core CPUs. In *ICPP Workshops*, pages 116–125. IEEE Computer Society, 2012.

[30] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12 (3):66–73, May 2010. .

[31] J. A. Stratton, S. S. Stone, and W. mei W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In J. N. Amaral, editor, *Languages and Compilers for Parallel Computing, 21th Intl. Workshop (LCPC)*, volume 5335 of *LNCS*, pages 16–30. Springer, 2008.

[32] TBB. Intel Threading Building Blocks, project site, 2013. `http://threadingbuildingblocks.org`.

[33] Z. Wang and D. Zhang. Progressive switching median filter for the removal of impulse noise from highly corrupted images. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 46(1), Jan 1999. .

[34] H. Xu, G. Zhu, H. Peng, and D. Wang. Adaptive fuzzy switching filter for images corrupted by impulse noise. *Pattern Recognition Letters*, 25(15), 2004. .

[35] P. Yang and O. Basir. Adaptive weighted median filter using local entropy for ultrasonic image denoising. In *Image and Signal Processing and Analysis, 2003. ISPA 2003. Proc. of the 3rd Intl. Symposium on*, volume 2, Sept. 2003. .

[36] L. Yin, R. Yang, M. Gabbouj, and Y. Neuvo. Weighted median filters: a tutorial. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 43(3), Mar 1996. .

[37] M. Yuksel. A hybrid neuro-fuzzy filter for edge preserving restoration of images corrupted by impulse noise. *IEEE Transactions on Image Processing*, 15(4), Apr. 2006.