

# Balancing High-Performance Parallelization and Accuracy in Canny Edge Detector

Valery Kritchallo<sup>1</sup>(✉), Billy Braithwaite<sup>2</sup>, Erik Vermij<sup>3</sup>, Koen Bertels<sup>1</sup>, and Zaid Al-Ars<sup>1</sup>

<sup>1</sup> Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands  
v.v.kritchallo@tudelft.nl

<sup>2</sup> University of Eastern Finland, Kuopio, Finland

<sup>3</sup> IBM Research, Dwingeloo, The Netherlands

**Abstract.** We present a novel approach to tradeoff accuracy against the degree of parallelization for the Canny edge detector, a well-known image-processing algorithm. At the heart of our method is a single top-level image-slicing loop incorporated into the sequential algorithm to process image segments concurrently, a parallelization technique allowing for breaks in the computational continuity in order to achieve high performance levels. By using the fidelity slider, a new approximate computing concept that we introduce, the user can exercise full control over the desired balance between accuracy of the output and parallel performance. The practical value and strong scalability of the presented method is demonstrated by extensive benchmarks performed on three evaluation platforms, showing speedups of up to 7x for an accuracy of 100 % and up to 19x for an accuracy of 99 % over the sequential version, as recorded on an Intel Xeon platform with 14 cores and 28 hardware threads.

## 1 Introduction

Three decades on since publication of the highly influential paper by John F. Canny [4], his edge detector algorithm (referred to as CED further on in the text) remains a standard and a basis of many efficient solutions in the fields of pattern recognition, computer vision, and a number of others. The algorithm is also known as computationally challenging due to its high latency that prevents a direct implementation from being employed in real-time applications [15]. And, while there exist numerous multi-core implementations of the algorithm, the issue of combining both high performance and quality of the edge detection in a single solution remains relevant as ever.

Among the previously published works on parallelization of the CED, [5] is, to our knowledge, the only one that could be meaningfully compared with our in terms of the recorded optimization levels. It reports a speedup of 11 times over sequential achieved on a 16-core CPU for a  $2048 \times 2048$  pixels test image, but lacks, in our view, sufficient scalability analysis of the presented solution. All other previous efforts in parallelization of the CED have been carried

out using a hardware acceleration platform of some sort, typically NVIDIA's GPU/CUDA [3, 5, 10, 14, 15]; furthermore, none of the works offered any user-defined scheme of tradeoff between the performance and quality of the traced output.

In contrast, the parallelization approach that we propose in this paper is not based on any particular hardware acceleration architecture, and thus can be implemented on a commodity multi-core platform equipped with an OpenMP-enabled GCC compiler. Similarly to the approach used in [5], in our parallelization of Canny edge we employ *domain decomposition* [11], a widely used data parallelization strategy that divides an image into equally-sized segments (with segments being in our case contiguous blocks of image pixel rows, or *slices*, as we call them), and processes them concurrently. However, unlike all previous approaches where the standard practice would dictate parallelizing the algorithm's sequential code on the laborious loop-by-loop basis, our method relies solely on incorporating a *single* image-slicing loop atop every other loop already existing in the code. This novel technique, besides being highly portable, requires minimal modification of the existing code and thus can be easily applied in a template manner to a wide range of image processing algorithms and applications.

Due to the enforced nature of the slicing loop-based parallelization, however, it is possible for our method to induce certain violation of the computational continuity of the sequential code, resulting in an output that is different from the sequential one. This issue is addressed in our method by introducing principles of approximate computing into the solution, such as the fully original *fidelity slider*, a mechanism that allows the user to maintain the desired level of edge detection quality in the parallel output by trading off a controlled amount of the achieved parallel performance.

Because of its universal character, our method of incorporating an image-slicing loop combined with the approximate computing-based tradeoff of accuracy against performance can be successfully applied to a wide range of image processing algorithms, particularly those not easily amenable to efficient parallelization due to their inherent constraints of computational continuity and internal data dependencies. As we estimate, such algorithms may vary from computer vision methods, such as Sobel's and Prewitt's edge detector method [7], anisotropic diffusion [13] and mean-shift filter [6], to more general data-processing ones, such as discrete wavelet transform [12], and many others.

To our knowledge, given the described above unique characteristics of our method, there are no existing analogues to compare it with.

The two major contributions of this paper are:

1. a novel efficient data parallelization technique based on introduction of a single top-level *image-slicing loop* into the application;
2. the *fidelity slider* as a new approximate computing concept to balance performance and precision in the parallelized application, such as the CED.

The rest of the paper is organized as follows. Section 2 describes the method we used to parallelize the CED. The achieved experimental results are shown and discussed in Sect. 3, followed by Sect. 4 that concludes this paper.

## 2 Parallelization of Canny Edge

### 2.1 Canny Edge Detection Algorithm

The Canny edge detection algorithm, developed by John F. Canny [4], is a well-known image processing algorithm used in many fields ranging from pattern recognition to computer vision. Description of this algorithm can be found in many introductory texts on image processing. Here, we only list the main stages of the algorithm:

1. noise reduction by filtering with a Gaussian-smoothing filter;
2. computing the gradients of an image to highlight regions with high spatial derivatives;
3. relating the edge gradients to directions that can be traced;
4. tracing valid edges using hysteresis thresholding to eliminate breaking up of edge contours.

The baseline sequential version that we used for parallelization, was the CED implementation by Heath et al. [1,8]. The function and variable names used throughout the text of this section, as well as included in the Fig. 2 and Table 1, refer to the source code of that implementation.

```

/* iterate over image slices */
for (row_ix=0; row_ix<rows; row_ix+=rows_slice) {
#pragma omp task shared(edge_file) if (do_async_tasking)
{
    /* call the main filter function to process
    * the image slice as a concurrent task */
    canny_par(row_ix, rows_slice, cols, image, ...);
}
}

```

**Fig. 1.** The source code fragment implementing the main image-slicing loop (simplified).

**Table 1.** Runtime breakdown by functions in the CED, percents

gaussian_smooth (Gaussian smoothing)	non_max_supp (non-maximum suppression)	apply_hysteresis (hysteresis edge thresholding)	derrivative_x_y (Gaussian derivative x&y)	magnitude_x_y (magnitude x & y)	follow_edges (edge tracing)
76.7	8.2	4.4	4.4	3.8	2.3

### 2.2 Introduction of the Image-Slicing Loop

The domain decomposition-based strategy we employed to parallelize the sequential version enforces coarse-grained data parallelization onto the application through incorporating a top-level image-slicing loop into its code. In the case of the CED implementation, the slices are equally-sized, contiguous blocks of pixel rows that are processed concurrently by asynchronous tasks spawned by a dedicated OpenMP-driven loop (Fig. 1). The loop has been parallelized with a single

`#pragma omp task` OpenMP directive, which we chose over the more commonly used `#pragma omp for` due to its ability to parallelize non-canonical loops. To host the image-slicing loop, a separate function was added at the top of the program's logic; the (formerly) main function is called from within the loop to process a single slice, instead of the whole image, as before.

Figure 1 shows the (simplified) fragment of the source code implementing the image-slicing loop, with the `#pragma omp task` OpenMP directive launching concurrent slice-processing tasks.

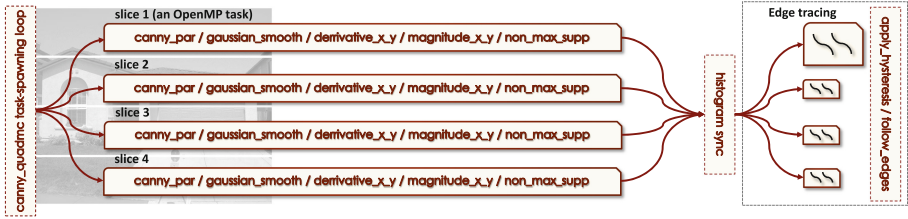
Right choice of the slice size, denoted as the `rows_slice` variable in the source code fragment, is crucial for reaching the best parallel performance in our solution. It is controlled via a parameter in the application's command line that defines number of pixel rows in each slice. If the parameter is not specified, the optimal slice size will be calculated automatically by the application as simply the quotient of the image's vertical dimension divided by the current number of active threads. When the active thread count is equal to the maximal number of hardware threads supported by the system, the slice size calculated this way is nearly guaranteed to lead to the highest parallel performance for the host platform, since it effectuates an ideal workload balance between slice-processing tasks, with OpenMP efficiently distributing the tasks among active threads.

With the exception of the Gaussian-smoothing loops that we parallelized mostly for various testing and illustration reasons (see the *Earth, Gaussian smoothing loops only* curve in Fig. 3 showing modest overall speedup of 2.4 times over sequential achieved from a standalone parallelization of this loop), no other native loops in the application were explicitly parallelized. As the application profile data, as well as our analysis of the source code indicated, optimization benefits of such effort would have been minimal (see Table 1, which shows runtime breakdown by functions in the CED, as recorded during profiling of the sequential version).

### 2.3 Image-Slicing Challenges and Solutions

Due to the breaks in the computational continuity of the CED algorithm caused by the introduced image-slicing loop, the following issues have been observed in the edge-traced images rendered by the parallel version:

1. horizontal visual breaks appearing in the image, i.e. blank single-pixel rows between slices, as a result of broken continuity in the Gaussian-smoothing stage of the algorithm due to the smoothing filter now operating on separate image slices, instead of the whole image, as it was done by the sequential algorithm;
2. areas in the image rendered differently from the reference output due to the image histogram array no longer computed globally for the entire image, but computed piece-wise within each slice;
3. differently traced edges as a result of violating the logic of the recursive edge-tracing procedure used by the original code that allowed, in principle, for indefinitely long, contiguous edges traced from one arbitrary pixel in the image to another arbitrary one.



**Fig. 2.** OpenMP-based parallelization scheme, drawn as a simplified case of four asynchronous slice-processing tasks entering histogram synchronization before the edge-tracing stage.

To address the above-mentioned issues, we implemented the following additions to our solution.

The parallel code was adjusted such that the image slices included extra overlapping pixel rows blending into the neighbors, in order to mask the visual breaks that resulted from image-slicing. The thereby introduced vertical overlap size (expressed in pixel rows) is an integer parameter varying from 1 to  $\tau_\sigma$ , where  $\tau_\sigma$  is derived from  $\sigma$ , the input parameter of the Canny edge algorithm that defines the standard deviation of the Gaussian-smoothing filter. The vertical overlap size corresponds to the integer *window\_size* variable found in the sequential code of the filter, and is computed as follows:

$$\tau_\sigma = 2 * \lceil 2.5 * \sigma \rceil \quad (1)$$

For most typical use-cases where  $\sigma$  doesn't exceed 2.0, the value of 10 pixels calculated for  $\tau_\sigma$  in accordance with Eq. 1 results in a Gaussian-smoothed output identical to that of the sequential version of the algorithm, while the default of 4 pixels provides adequate masking effect. This adjustment fully fixed issue (1), and partially addressed issue (3), with a performance penalty depending on the overlap size and number of slices.

To produce the precise – in respect to the sequential version that is – output, a modification in the parallel code was necessary that would allow using a single histogram of the entire image for all concurrent slice tasks. To implement this, a *histogram synchronization* scheme was developed, where slice tasks performed most of the work independently, and only synchronized with each other briefly (see “histogram sync” block in Fig. 2), to compute and share among themselves the single global histogram, before proceeding with edge-tracing within their individual image slices, again concurrently. The scheme (Fig. 2) allowed to retain most of the parallel performance, and partially solved issue (2) of divergence with the reference output. The complete solution for issue (2) and (3) could only be found as a part of the fidelity slider design (Sect. 2.5).

## 2.4 The Fidelity Slider: Balancing Performance and Accuracy

In order to fully address issue (2) and (3), a new feature which we call *fidelity slider* has been introduced into the solution. By means of adding elements of approxi-

mate computing in a controlled fashion, it allowed to address the principal challenges of the CED parallelization in a more fundamental way.

The first step in implementing the slider is to introduce measurement of the actual binary difference between the parallel version's output and that of the sequential one. For this purpose, we implemented a simple metric expressed as a total sum of average pixel difference between two images (Eq. 2), which proved a suitable divergence measure for the purposes of our application, reflecting well the degree of the observable visual difference, as well as more subtle deviations in rendered edges (see Fig. 6, explained in more detail in Sect. 3.4.)

The rendering error  $RE$  used to calculate the accuracy of the parallel output is computed as the image distance metric described above, that is the average pixel difference between the parallel-produced image  $p$  and the sequential image  $s$  (both grayscale, the only type of images the CED works with), and is defined as

$$RE(p, s) = \frac{\sum_{i=1}^N \sum_{j=1}^M \frac{|LP_{ij} - LS_{ij}|}{255.0}}{N * M} \quad (2)$$

where  $LP_{ij}$  and  $LS_{ij}$  are pixel intensity values, and  $N$  and  $M$  are the images vertical and horizontal dimensions in pixels. The corresponding accuracy percentage value is calculated as

$$AC = 100 * (1.0 - RE)$$

## 2.5 The Design of the Fidelity Slider

The fidelity slider is constructed as a composite parameter driving the strength of the following three factors, each moderating its own component of the aggregate divergence from the reference image induced by the computational discontinuity issues (1), (2) and (3):

1. the vertical slice overlap size in pixel rows, from 1 to  $\tau_\sigma$ . This is the factor introduced to inhibit issue (1). It moderates the related component of the rendering error  $RE$  in a manner such that adding a pixel row to the slice overlap along incrementing the factor results in a steadily decreasing component of the rendering error;
2. the degree of the cross-slice histogram synchronization, expressed as the number of slice tasks synchronized before the algorithm's edge-tracing phase takes place, progressing from two to all slices synchronized. This is the factor introduced to inhibit issue (2). Advancing the slider value from 1 % to 100 % results in more and more slices using the globally synchronized histogram (as opposed to their locally computed one) and, correspondingly, decreasing component of the rendering error;
3. number of slices rendered in non-concurrent fashion during the last edge-tracing stage, progressing from two to all slices. This is the factor introduced to inhibit issue (3). This last-stage rendering is performed by the leading slice task, which is reflected in Fig. 2 by the slice on the top (drawn visibly larger than the rest). Advancing the slider value from 1 % to 100 % results in more and more slices jointly edge-traced as a single contiguous fragment of the input image and, correspondingly, decreasing component of the rendering error.

Further algorithmic details of the three introduced factors and the related to them moderation process can be found in [9].

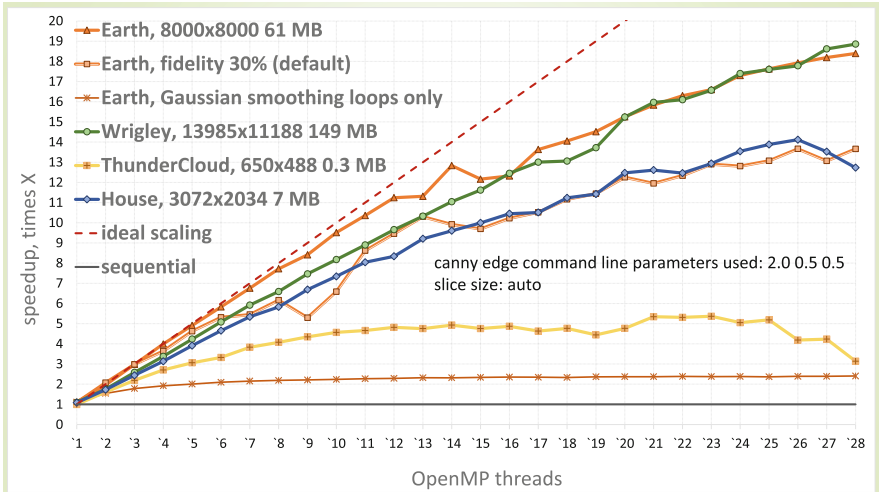
### 3 Results

#### 3.1 The Test and Development Platforms

In the course of benchmarking our CED implementation, we used the following three platforms. First, an IBM server equipped with two 4.2 GHz POWER8 CPUs both featuring six cores each supporting up to eight threads per core, running Ubuntu kernel v3.16, hereafter called “POWER8”. Second, an HPC server with two 2.6 GHz Xeon E5-2697 CPUs, each having 14 cores supporting two threads per core, running Ubuntu kernel v3.13, hereafter called “Xeon E5-2697”. And third, a Dell desktop with a 3.2 GHz Xeon E5-1650 CPU, having six cores supporting two threads per core, running Windows 7, hereafter called “Xeon E5-1650”. In all our experiments on the dual-socket machines, only a single socket was used. The development platform on the three systems was GCC compiler v4.8.2, v4.9.1, and v4.8.2, respectively, with OpenMP v4.0 as the parallelization environment.

#### 3.2 The Test Image Set and Benchmarking Routine

During the benchmarking stage, we used a wide variety of test images, ranging from 0.3 MB,  $650 \times 488$  pixels size (*ThunderCloud* in Fig. 3), to 149 MB,  $13985 \times 11188$  pixels (*Wrigley*), the size that, to all our knowledge, significantly



**Fig. 3.** Benchmark results for parallelized Canny edge, as registered for four test images at the fidelity level of 1% (unless otherwise noted) with file output disabled. Test platform: Xeon E5-2697 with 14 cores and 28 hardware threads.

exceeds any previously reported one for an edge-traced image. The visual contents of the test images varied greatly as well, from natural (*Earth*, *ThunderCloud*) to mostly geometric (*House*, *Wrigley*) and entirely synthetic/computer-generated.

For benchmarking of the presented here CED solution, we used a custom benchmarking sub-system incorporated in the application's code. Based on principles of statistic sampling, this sub-system can execute long batches of test runs, objectively measuring the resulting speedup, and performance in general, against a user-selected range of threads, image-slicing-related parameters and ranges of application-specific numeric options in arbitrary combinations. Each test run, in turn, is typically comprised of 6 or 12 individual application executions, to produce statistically correct performance average. In the course of benchmarking our CED solution, we have performed some 900+ test batches to edge-trace a set of twenty images in total, of which a selected, most representative subset of four images was chosen for the illustration purposes of this paper, rendered as speedup and accuracy curves in Figs. 3 and 5.

The source images and the full compilation of produced benchmarks, as well as the code and other supplemental material used in this paper are available on the OSF site [2].

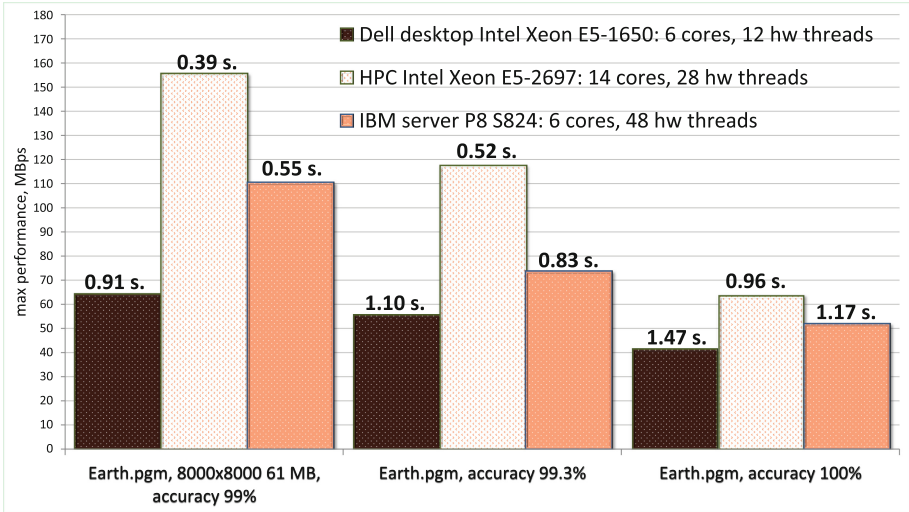
### 3.3 Recorded Speedups

Figure 3, displaying the range of speedups recorded when edge-tracing the representative sub-set of four test images on the Xeon E5-2697, demonstrates that our parallelization method performs the best when processing large-sized images (*Earth*, *Wrigley*), as compared to medium- and small-sized ones (*House*, *ThunderCloud*). Extensive benchmarking performed on the full test image set has confirmed that the performance of our method is generally proportional to the image's dimensions, although with a varying from platform to platform degree. On the other hand – as it was expected – the performance and method's scalability is affected negatively by the increased strength of the imposed by the fidelity slider accuracy factors, mainly by the histogram synchronization at 100%. This is illustrated by the curve marked *Earth, fidelity 30 % (default)* in this Figure (the fidelity level that enforces maximum synchronization of the histogram slices), with the maximum speedup of 13.67 times over sequential, compared to 18.39 times for the same image at fidelity 1%, which imposes no synchronization.

The maximal parallel execution speedup recorded for this application was 18.66 times over the sequential version, when processing the biggest test image on the Xeon E5-2697, with file output disabled and fidelity set at 1% (*Wrigley* in Fig. 3). On the other two evaluation platforms, benchmarking has recorded the highest speedups of 13.33 times over sequential, for the POWER8 (test image: seven MB *House.pgm*,  $3072 \times 2034$  pixels), and 7.74 times, for the Xeon E5-1650 (test image: 61 MB *Earth.pgm*,  $8000 \times 8000$  pixels).

For the majority of the images we tested, rendering at the 1% fidelity resulted in the accuracy of 98%–99% of the edge-traced output. The chart in Fig. 4 displays the highest absolute parallel performance in MBs per second registered for each





**Fig. 4.** Maximal parallel performance in MBs per second for the three test platforms at three accuracy levels. Test image: Earth, 8000 × 8000 pixels, 61 MB. The figures a top of every bar are the shortest runtime in seconds registered for the platform at each of the three accuracy levels.

of the three test platforms at three key accuracy levels (99 %, 99.3 % and 100 %), when rendering the main test image, with the shortest runtime in seconds atop of every bar. Although the Xeon E5-2697 appears a clear winner in the picture, the POWER8 comes out substantially better in the performance-per-core metric.

### 3.4 Fidelity Slider Benchmarks

In Fig. 5, the benchmark produced results for the four test images are displayed against the vertical speedup axis, with the accuracy value progressing from 99 % to 100 % along the horizontal one. The speedup is the highest (18.01 times over sequential for our main test image, 61 MB Earth.pgm of 8000 × 8000 pixel size) at the leftmost position of 99 percent accuracy.

To help in getting an idea of the degree of the edge detection divergence that can be expected at the lowest fidelity value of 1 %, Fig. 6 presents a fragment from another our test image, detail-rich Wrigley, with the original picture shown on the left and the output of the sequential version in the middle. The right side of this Figure (produced by an image-processing program in the layer difference mode) visualizes the edges traced spuriously by the parallel version; most of them, as can be observed, are concentrated along a horizontal row at about 1/3 height of the picture where a slice break happened. When comparing the two rendered outputs visually (without the help of such image-processing program that is), the difference of this scale is rather difficult to discover, unless one is hinted where to look first. The measured accuracy for this rendering was 98.01 %.

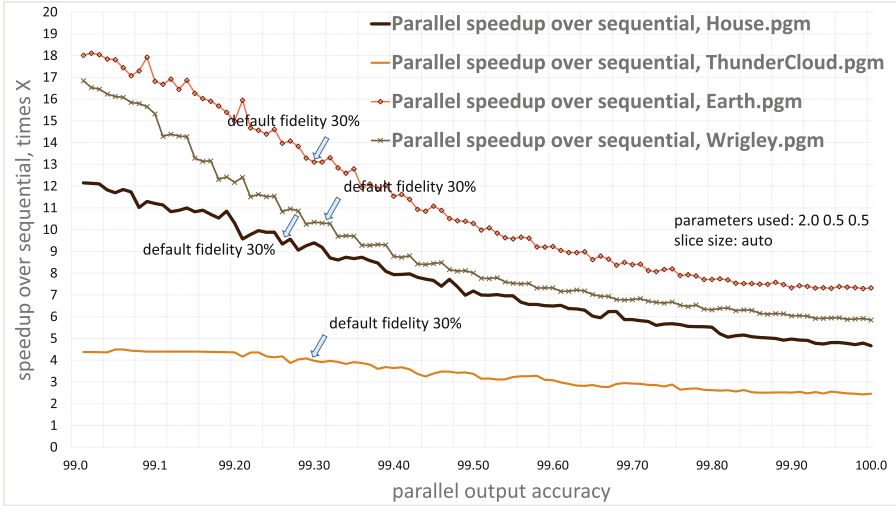


Fig. 5. Impact of the output accuracy on the parallel speedup, as recorded on the Xeon E5-2697 for four test images.



Fig. 6. A fragment from the Wrigley image (original picture on the left) with the sequential program’s traced edge output (middle) and spurious edges rendered by the parallelized version at the lowest fidelity value of 1% (right).

At the rightmost slider position of 100%, where the performance is the lowest (7.32 times over sequential for Earth), the image difference is zero. In Fig. 3, the range of speedups recorded for the default slider value 30% – chosen in our CED implementation for the best combination of speed and quality – can be seen, represented by the curve marked *Earth*, fidelity 30% (default).

## 4 Conclusions

The relevance and continued viability of the Canny edge detector as a robust computer vision algorithm remain undisputed, and there is no shortage of implementations of this venerable algorithm. What seems to be lacking among these implementations, however, is a consistent approach that would address the

performance and edge detection quality in an equal and flexible manner, and that would not require using a proprietary acceleration platform to achieve its performance levels.

The two major contributions of this paper were:

1. a novel efficient data parallelization technique based on introduction of a single top-level *image-slicing loop* into the application;
2. the *fidelity slider* as a new approximate computing concept to balance performance and precision in the parallelized application, such as the CED.

In this paper, we demonstrated that a successful application of coarse-grained parallelization and innovative principles of approximate computing through incorporating an image-slicing loop into the CED algorithm allows to achieve highly scalable optimization without using any dedicated hardware acceleration equipment. On all three test platforms we used, benchmarking has registered strong multi-core performance, with highest speedups varying from 7.74 times over sequential, as recorded on the lowest platform (a 6-core, 12 hardware threads Intel Dell desktop), to 18.66 times, on the highest (a 14-core, 28-hardware thread Intel HPC Xeon server). The desired balance between the performance and quality of the output is maintained via the specially-introduced fidelity slider, yielding speedups varying from 18.66x at the accuracy level of 99 percent, down to 7.32x at the accuracy level of 100 percent, as recorded for the fastest benchmark.

## References

1. Edge Detector Comparison (1996–2015). [http://marathon.csee.usf.edu/edge/edge\\_detection.html](http://marathon.csee.usf.edu/edge/edge_detection.html)
2. The Canny edge/QuadMC parallelization project (2015). <https://osf.io/i725h/>
3. Brethorst, A.Z., Desai, N., Enright, D.P., Scrofano, R.: Performance evaluation of canny edge detection on a tiled multicore architecture. In: Electronic Imaging, pp. 78720F–78720F. International Society for Optics and Photonics (2011)
4. Canny, J.: A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.* **8**(6), 679–698 (1986)
5. Cheikh, L.B.T., Beltrame, G., Nicolescu, G., Cheriet, F., Tahar, S.: Parallelization strategies of the canny edge detector for multi-core CPUs and many-core GPUs. In: IEEE 10th International Conference on New Circuits and Systems, pp. 49–52. IEEE (2012)
6. Comaniciu, D., Meer, P.: Mean shift: a robust approach toward feature space analysis. *IEEE Trans. Pattern Anal. Mach. Intell.* **24**, 603–619 (2002)
7. Gonzales, R.C., Woods, R.E.: *Digital Image Processing*. Prentice Hall, Upper Saddle River (2005)
8. Heath, M., Sarkar, S., Sanocki, T., Bowyer, K.: Comparison of edge detectors: a methodology and initial study. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 143–148. IEEE (1996)
9. Kritchallo, V., Vermij, E., Bertels, K., Al-Ars, Z.: Fidelity slider: a user-defined method to trade off accuracy for performance in canny edge detector. In: 2nd Workshop On Approximate Computing, HiPEAC (2016)

10. Lourenço, L.H., Perelman, D., Todt, E.: Efficient implementation of canny edge detection filter for ITK using CUDA. In: 13th Symposium on Computer Systems, pp. 33–40. IEEE (2012)
11. Papadrakakis, M., Stavroulakis, G., Karatarakis, A.: A new era in scientific computing: domain decomposition methods in hybrid CPU-GPU architectures. *Comput. Meth. Appl. Mech. Eng.* **200**, 1490–1508 (2011)
12. Park, I.K., Singhal, N., Lee, M.H., Cho, S., Kim, C.W.: Design and performance evaluation of image processing algorithms on GPUs. *IEEE Trans. Parallel Distrib. Syst.* **22**(1), 91–104 (2011)
13. Perona, P., Malik, J.: Scale-space and edge detection using anisotropic diffusion. *IEEE Trans. Pattern Anal. Mach. Intell.* **12**, 629–639 (1990)
14. Roodt, Y., Visser, W., Clarke, W.: Image processing on the GPU: implementing the canny edge detection algorithm. In: International Symposium of the Pattern Recognition Association of South Africa (2007)
15. Xu, Q., Varadarajan, S., Chakrabarti, C., Karam, L.J.: A distributed canny edge detector: algorithm and FPGA implementation. In: IEEE Transactions on Image Processing, pp. 2944–2960. IEEE (2014)