DETECTING REGIONS OF INTEREST USING A CELL PROCESSOR

B00236297 – LIVINGSTONE, JONATHAN
B00235610 – TAYLOR, WILLIAM
B00243868 – TOADER, CONSTANTIN

Game Console Development
COMP10037

Contents

1. Overview 2
2. Background work 2
3. Methodology 3
4. Algorithms and code 3
The PC version 4
4.1 Noise reduction 4
4.2 Edge detection 5
4.3 ROIs mask generation 6
4.4 Highlight the ROIs in the original image 7
5. Parallelisation and performance 8
5.1 Parallel Code 9
5.1.1 SPU Manager 9
5.1.2 SPU Program 9
5.1.3 PPU Program 9
5.2 Visual improvements 9
5.3 Performance improvements 11
6. Testing 12
7. Results 12
8. Conclusions 14
References 15

1. Overview

There is a requirement to develop an automatic *Region of Interest (ROI)* detector as part of a cancer cell analysis computer vision application. A ROI is a region detected within an image scene that bounds a target object of interest with an image. The same concept appears commonly in face-detection and gesture recognition software.

The solution proposed for this problem is to develop an application that should be optimised to run on PS3 Cell Broadband Engine hardware to exploit as much of the available processing power as possible.

The ROI detector that is to be developed, should be capable of detecting ROIs from a video sequence of rat prostate cancer cells. The premise is that a bounding box selects a likely cell from the background image from frame to frame.

The solution is inspired by Phung and Bouzerdoum (2007)[1] where a box with a certain edge density will be the likely target ROI. The edge density map for the image is searched until an appropriate candidate is found. This requires that the edge densities are averaged for many rectangular boxes in the search space.

Initially edge densities are to be found using simple kernel processing. The Sobel transform suggested by Juneja, Sandhu (2009)[2] will help generate a gradient map to find edges from each original image that forms the video sequence.

In the following phase the solution will be ported on the PS3 Cell Broadband Engine and all the compatibility issues will be solved, so our solution will natively run on a PS3.

Finally, the application will be adapted to use optimised SIMD intrinsics, multicore decomposition, SPU intercommunication and algorithmic optimisation where possible.

Output and timing will be taken and analysed at each step and our findings will be presented in the conclusion part of this paper.

2. Background work

When it comes to differentiating objects from non-objects, or identifying *Regions of Interest*, there are many academic resources available. This is probably why the progress in Computer Vision is so remarkable, allowing us to frequently use related technologies in: s*ecurity* (CCTV), *road and driving safety (*traffic signs or lane recognition embedded in the latest generation of cars), *image search* (Google Image), *health* (medical diagnosis based on patterns recognition), *retail* (UPC barcode), *games industry* (Microsoft Xbox Kinect), *augmented reality*, et cetera.

It is not very clear who started the first research on edge detection, but Szeliski (2010)[3] places the beginning of edge detection in the 1970s with research into better edge and contour detection continuing through the 1980s.

Canny (1986)[4] took an analytical and mathematical approach on edge detection, using numerical optimisation to find operators to be used with edge detection, trying to find a balance between detection quality and localisation. He created designs for detectors using operators of different values to cope with different signal-to-noise ratios in an image and came up with a general method which can be used for different images. This paper will use a variation of Canny's edge detection algorithm and will be presented in the next chapter.

Nalwa and Binfdord (1986)[5] define an edge in an image as being an intensity discontinuity in the scene. Edges are made of small linear segments named *edgels,* each represented by a position and an angle*;* basically, any curve can be represented as a sequence of edgels which further form an edge. They designed an approach where they detect edgels in one-dimensional surfaces. They highlight that their algorithm performs effective noise-reduction without blurring the edges too much and is implementable as a parallel process. The algorithm used in this paper will try to make use of parallelism and will be implemented to run on the CELL processor.

Xue, Wenxia and Guodong (2015)[6] used cellular neural networks (CNN) to prove their algorithm efficiency in edge detection. They compare their output with different operators like Sobel, Roberts, Prewitt, LOG, Canny, 1 CNN and 2 CNN. From their test results, it is visible that their method outputs more edges and seems to be more clear than others, but along with thicker and better contoured edges, some noise is also visible in the output. Comparing 1 CNN template against 2 CNN template, and considering that both output noise, the later does indeed return a sharper, more accurate output when it comes to edge detection. Unfortunately, as previously stated, with these 2 methods (CNN Neighbourhood radius equals 2 and CNN Neighbourhood radius equals 1) noise is also present in the result, reducing the quality of the algorithm.

To better identify ROIs, XIA, FENG et. al. (2009)[7] do not rely only on edge detection. When computing the salient map introduced by Itti (1998)[8], they consider another feature as well: the depth. They noticed differences between Itti's method and the human vision perception. Their paper highlights that after the salient map is formed, several ROIs would be grouped and identified as just one object instead of several individual ROIs. Using edge and depth information, the resulted ROIs are more like what humans would consider a ROI. This would be a good fix for the application proposed by this paper, as the same problem occurs here, but it is outside of the scope of this paper.

The proposed solution will try to accommodate some of the previous work done. The application will use Canny's method, will be implemented for parallel processes and will use hardware that will benefit from the use of intrinsic SIMD, such as the Cell Broadband Engine.

3. Methodology

Given a set of ten images, the ROIs will be identified for each individual image using the following methodology consisting of four tasks:

1. Reduce noise using Gaussian Blur.
2. Apply Sobel operator to identify the edges.
3. Create a mask for ROIs.
4. Apply mask on original image

For each image the output is saved for analysis.

To highlight the performance gain when using the CELL processor, the application will be initially developed for PC and the runtime will be measured. Then the application will be ported on the PS3 platform and again the timings will be recorded. Further improvements will be applied by using intrinsic Symmetric Instructions and Multiple Data available on the Cell Broadband Engine. Performance time will be again recorded and findings will be presented in the results chapter.

4. Algorithms and code

No matter what runtime environment will be targeted, the core algorithm should follow the above-mentioned methodology.

The PC version

The PC version is basically the root of the application. Considering that is difficult to program without a proper IDE on the Play Station, it felt natural that a working version should be developed for pc first. The images that need to be processed in this application are displayed in Figure 1. They represent 10 frames of a short clip showing cells in motion.

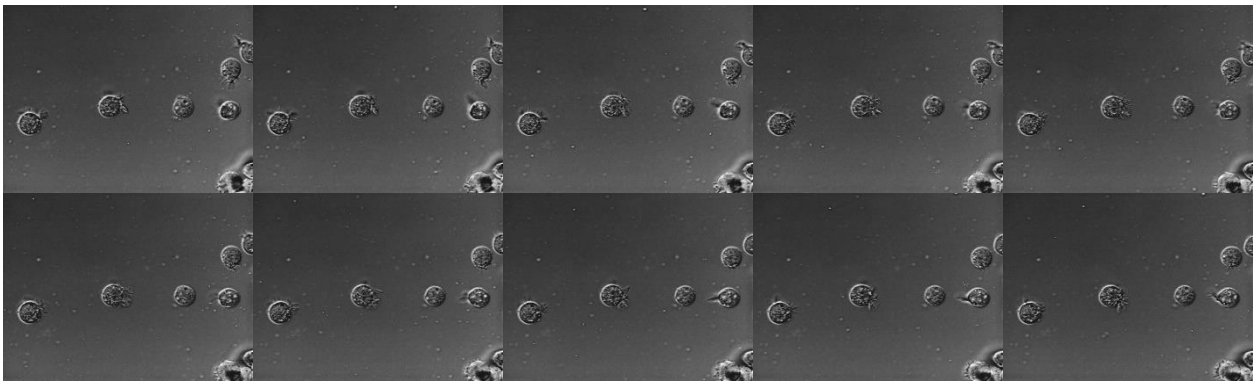

Figure 1. 10 frames of moving cells

4.1 Noise reduction

First step was to take the original image and reduce noise as much as possible to avoid false positives when detecting edges. This is done by applying Gaussian blur on the image[9]. Each pixel is replaced by the averaged of itself and the surrounding pixels to which the blur filter has been applied.

Let the pixel representation in Figure 2 be the pixel layout of a 10x10 picture. When processing pixel 25 for example, a 3x3 kernel will be generated containing its neighbour pixels.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

0.111	0.111	0.111
0.111	0.111	0.111
0.111	0.111	0.111

Figure 2. Pixels positions on a 10x10 image with kernel surrounding pixel 25, and a Gaussian blur filter box

The kernel will contain pixels 14,15,16, 24, 25, 26, 34, 35, 36. To apply Gaussian blur, the kernel matrix will have each element multiplied by its correspondent from the filter matrix and the weighted sum of all values will represent the final value of the blurred pixel 25.

A special case is when the pixels that needs processed are at the edge. In this case the pixel could be pushed into the surrounding, where the neighbours are missing, or the pixel can be wrapped around the other side. This will make tiled images seamless. For example, when processing pixel 0, the kernel values should be as in this kernel.

99	90	91
9	0	1
19	10	11

Algorithm:

```
for each row in input image:
	for each pixel in row:
		set sum to zero
		for each element in kernel row:
			multiply filter value corresponding to pixel value
			add result to accumulator
	set pixel value to sum
```

After applying blur to the input images, the result is visible in Figure 3.

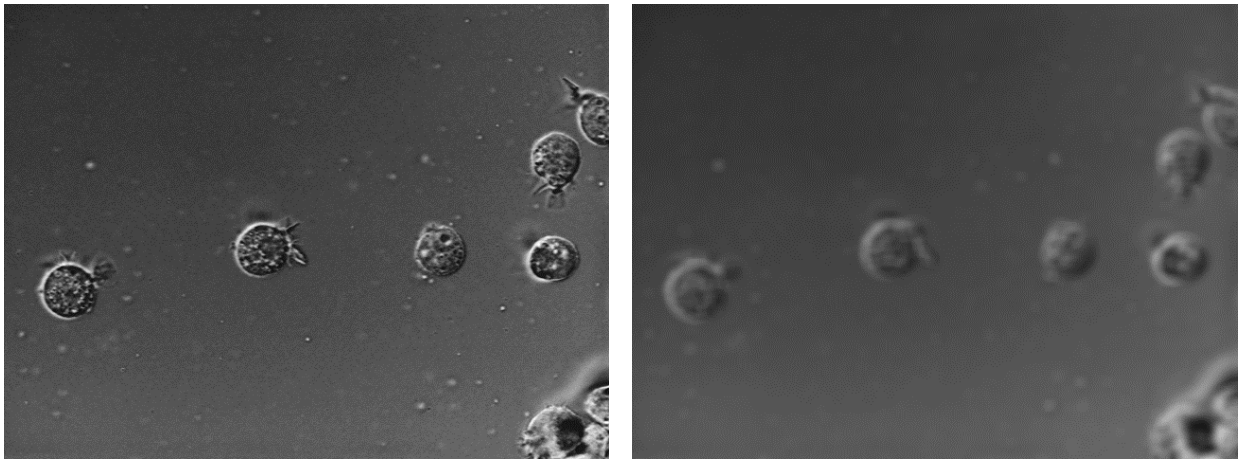

Figure 3. On the left side is the original image, on the right side is the image with noise reduced

4.2 Edge detection

The next step in highlighting the ROIs in the set of input image is to detect the ROIs edges. This will be done using the Sobel[10] operator. Basically, for each pixel in the image the magnitude of horizontal and vertical gradients is computed and the value is assigned to the pixel. If the intensity in change is bigger than a certain threshold, then that pixel is part of an edge, otherwise, the pixel should be ignored (made black).

Similar to the Gaussian blur, the Sobel operator is applied on a 3x3 kernel. The two gradients are obtained: one for x direction and one for y direction and then their magnitude is calculated and assigned to each pixel and if necessary the pixel value is then changed to black. Figure 4.

				4	5				
			13	14	15	16	17	18	19
		22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

horizontal

-1	0	1
-2	0	2
-1	0	1

vertical

-1	-2	-1
0	0	0
1	2	1

Figure 4. Pixels positions on a 10x10 image with kernel surrounding pixel 25, and Sobel matrices

The difference is that the input image is now the processed blurry image, which had noise reduced in the previous step.

Algorithm:

```
for each pixel in input image:
	calculate gradient magnitude
		for each pixel neighbour
			multiply neighbour value with horizontal gradient correspondent
			add result to pixel's horizontal gradient component
			multiply neighbour value with vertical gradient correspondent
			add result to pixel's vertical gradient component
		return magnitude of the gradient
	if gradient is smaller than threshold
		ignore pixel (make it black, invisible)
	else
		highlight the pixel (multiply value)
	endif
```

After applying Sobel to the blurred images, the result is visible in Figure 5.

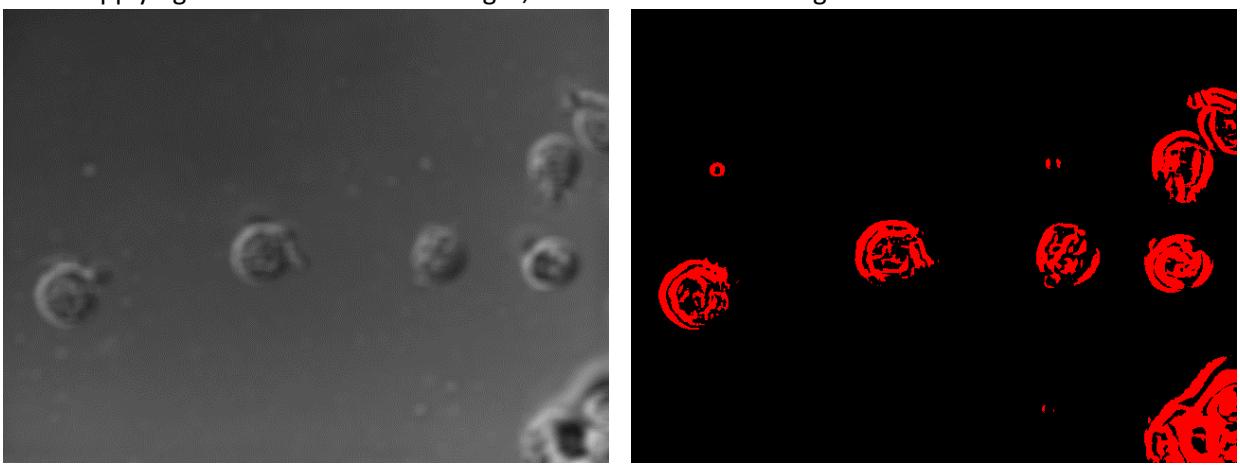

Figure 5. On the left side is the blurred image, on the right side is the image with edges detected

4.3 ROIs mask generation

The following step involves using the detected edges and creating a mask for regions of interest. In this step the redundant edges are eliminated.

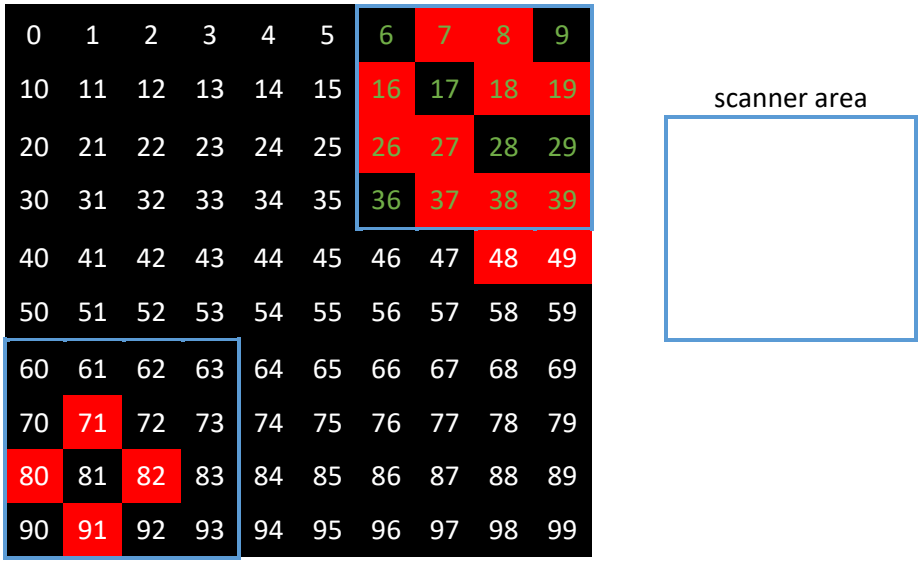

Figure 6. 10x10 image with edges detected surrounded by scanner. The top right area will be marked, the bottom left will be ignored

A custom area is moved across the image and its edge density is calculated. All the pixels in the scanning area is marked if it contains strong edges.

Algorithm

```
while origin of scanning area is not outside image boundaries
	calculate edge density for area
		set density to 0
		for each pixel in the scanning area
			add pixel value to density
		return area density related to picture
	if density shows a strong edge
		mark it
	endif
	scan to the right
	if origin is outside the width of the row
		move to next row
	endif
```

After creating the ROIs mask, the result is visible in Figure 7.

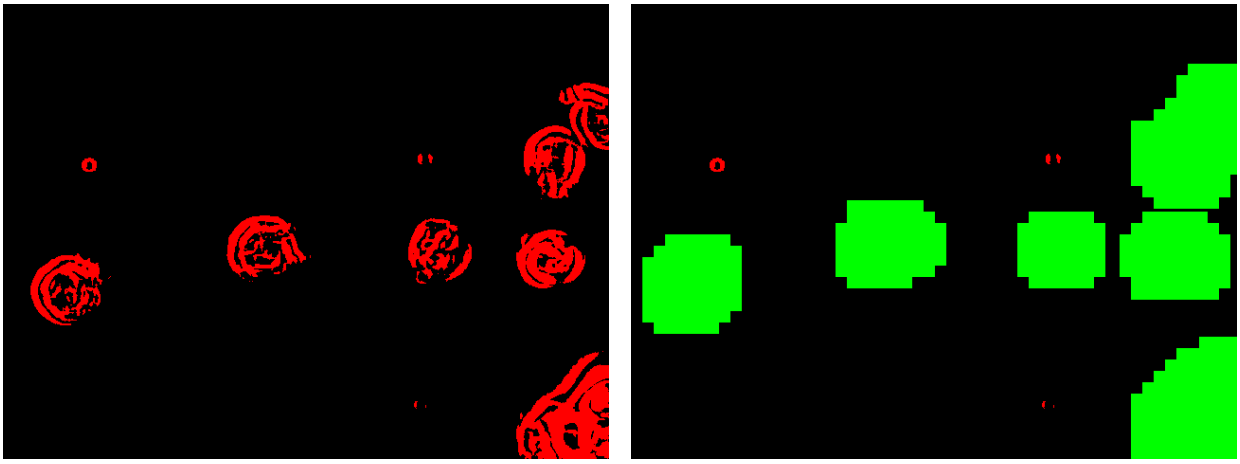

Figure 7. On the left side is the image with edges detected, on the right side is the image with regions of interest

4.4 Highlight the ROIs in the original image

In the last step, all the pixels from the original image are darkened if they are not on the same position as the pixels from the ROI mask. All the pixels of the original image are traversed keeping track of the current index, and from the ROIs map the correspondent value is checked if is a strong edge or not. If the current pixel is not part of a ROI in the ROIs map, then the current value is decreased to 40%, making the pixel darker.

Algorithm

```
for each pixel in input image:
	if the current pixel is green in the ROIs map
		output pixel is the current pixel
	else
		output pixel value is multiplied by 0.4
	endif
```

After applying the ROIs mask to the original image, the result is visible in Figure 8.

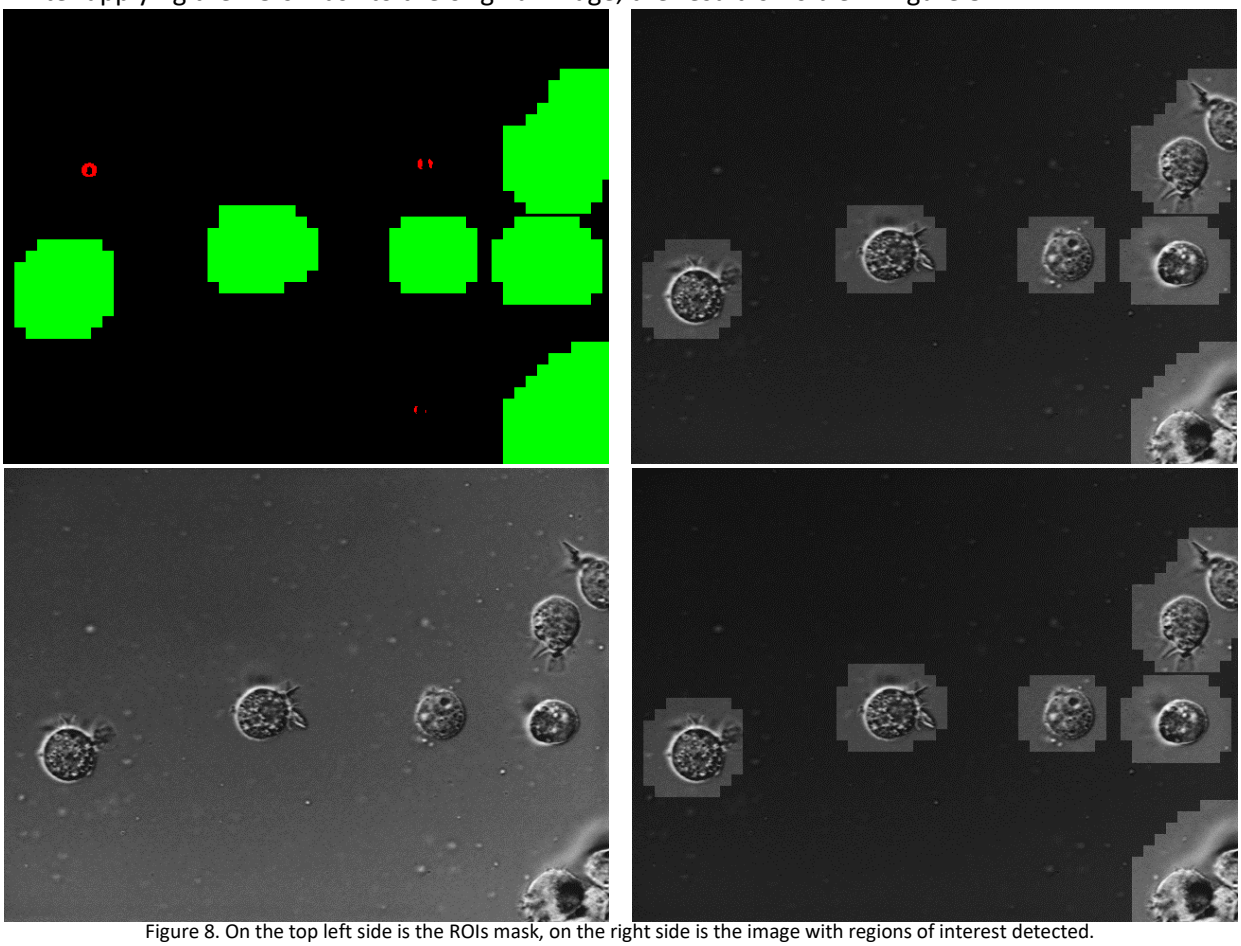

Figure 8. On the top left side is the ROIs mask, on the right side is the image with regions of interest detected. On the bottom left side is the original input image, on the right side is the output image

5. Parallelisation and performance

Currently the time needed to process **one** image is 45 milliseconds on an i7-1790 CPU @ 3.6 GHz.

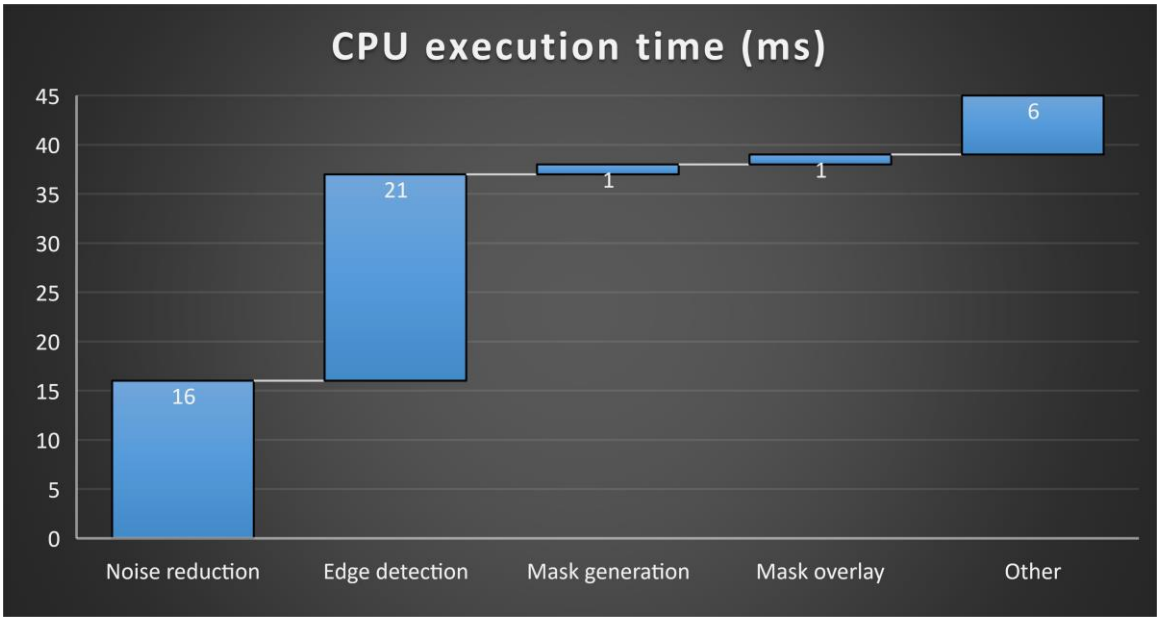

Figure 9. Image Processing time breakdown in milliseconds

As show in Figure 9 the most intensive tasks seem to be the noise reduction and edge detection.

The application processes the images in sequence needing a total of 450 milliseconds on average on a PC. While it is not expected that the CELL processor would be faster than the i7 processor, some performance in using the PS3 processor can still be measured. To improve this, the program will be adapted to run some of the tasks in parallel so the performance and efficiency can both increase.

5.1 Parallel Code

To get the code working in the most efficient manner on the PS3 all 6 available synthetic processing units found in any PS3 would be used. The idea is simple, each task in the x86 version is broken down into 6 parts that can run in parallel across all SPUs. To make interaction with all SPUs easy, a spu_manager class was developed.

5.1.1 SPU Manager

The SPU manager deals with loading a SPU program to run on each SPU independently. Upon the manager's construction, it reads system info to find out how many usable SPUs are available. Once it is known how many SPUs should be used, the location of an SPU executable that should be loaded across the SPUs is passed to the manager. Then a struct is passed to the SPU manager, which is made available to the SPU program when it starts. The struct contains information on the work the SPU should do.

```
// Location & size of the struct to be passed to SPU program
void spe_arg(void* address, int size);
// The exe location for the SPU program
void spe_program(const std::string& filename);
// Runs the spu program, count specifies how many SPUs it should run on
void spe_run(int count);
```

Because running an SPU program via spe_context_run is a blocking operation, the SPU manager starts all SPU program instances on a separate thread to ensure they all launch at the same time (roughly). A join is performed when all SPU programs have finished executing. In the application, these methods are called for each task that is needed to execute: for blur, for sobel, for detection and for overlay.

5.1.2 SPU Program

From the struct passed to the program on start-up, a subset of the bitmap is taken to work with before putting it back so it can be written to a file or used in another SPU program. To achieve this the Memory Flow Controller is used, reading the data chunk by chunk in the largest chunks possible until all data required to perform the task is acquired. Where available Symmetric Instructions and Multiple Data are executed on the SPU to make sure that multiple operations can be performed simultaneously. The SIMDs are mostly used in the blur SPU program and the sobel SPU program as there are matrix multiplication in these functions and intrinsic operations can be executed.

5.1.3 PPU Program

Even though the PPU program acts more as a controller for SPUs on the system we do take advantage of SIMD instructions to take our grayscale bitmap and save it as a 24-bit bitmap.

5.2 Visual improvements

Before improving the performance, some values had to be tweaked to arrive at an acceptable visual result. There are a few pre-processor directives defined in the visual_tunning.h file which can dramatically change the final output of the application.

```
#define ACCEPTED_VALUE_FOR_EDGE 50
```

This value helps identify an edge. A too high value and some edges will not be found when applying the Sobel operator. A too low value and the filter gets too sensitive and captures more noise as shown in Figure 10. Needless to mention that this further affects both the ROIs mask and final output.

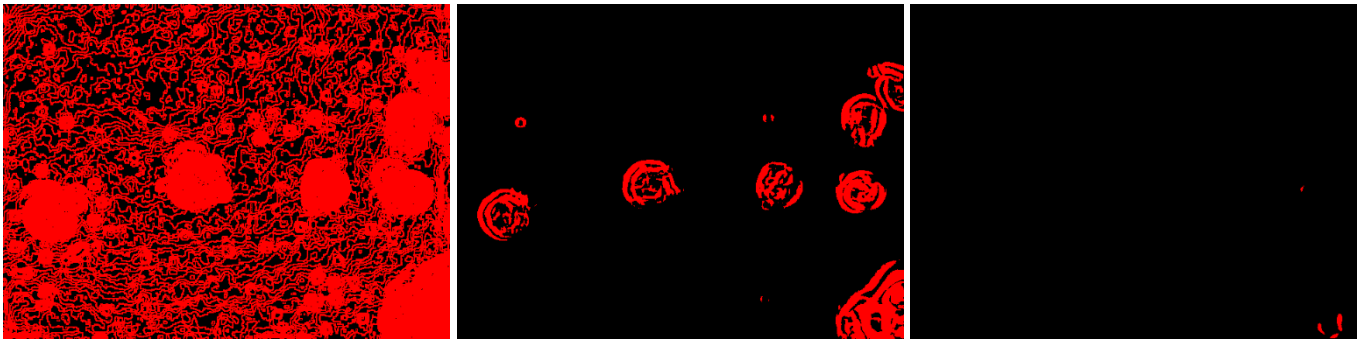

Figure 10. Different outputs for ACCEPTED_VALUE_FOR_EDGE: left 1, middle 50, right 200

```
#define IGNORABLE_EDGE_DENSITY 0.3f
```

The IGNORABLE_EDGE_DENSITY filters out the non-significant edges found in the previous step. This affects how the ROIs mask is being built. The optimal value seems to be 0.3f. The density is calculated in relation to the area that is being scanned and analysed and the number of pixels per step. This area is defined by other two values: `#define REGION_HEIGHT 45` `#define REGION_WIDTH 45` or `#define REGION_LENGTH 45` if the region to be scanned is always a square.

Further it will be detailed how each of these values affect the ROIs mask.

A lower value would detect false positives in the ROIs mask and would mark an area to be filled, while a high value would ignore the detected cells as shown in Figure 11. All these outputs are obtained without changing the ACCEPTED_VALUE_FOR_EDGE.

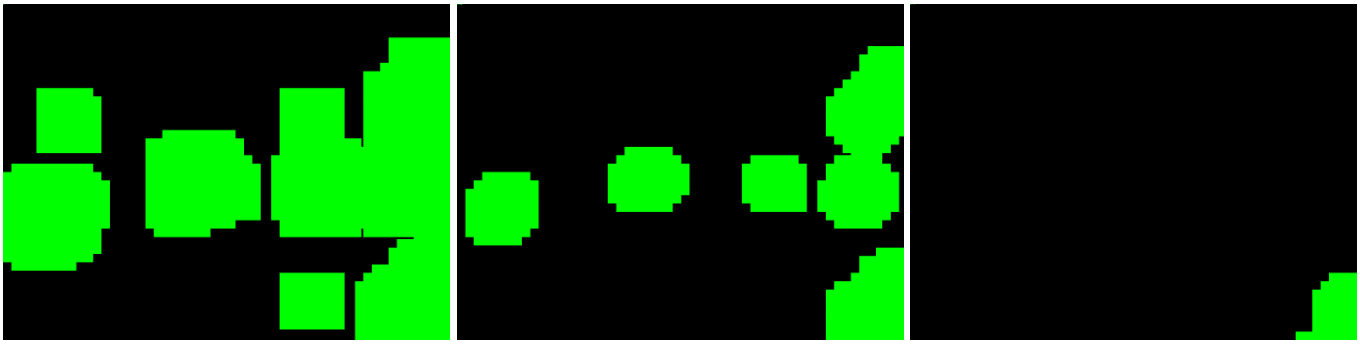

Figure 11. Different outputs for IGNORABLE_EDGE_DENSITY: left 0.01, middle 0.3, right 0.8

```
#define REGION_HEIGHT 45
#define REGION_WIDTH 45
#define REGION_LENGTH 45
```

All these values define the area of pixels which should be filled on x and y whenever a significant edge is detected. If the area is a square (most commonly), the REGION_LENGTH should be use for code consistency. Edge density is impacted directly by the area defined by these parameters. Smaller areas would pixelate the cells and would make the edge density incorrect returning false positives, while bigger areas would capture a lot of area surrounding the cells as displayed in Figure 12.

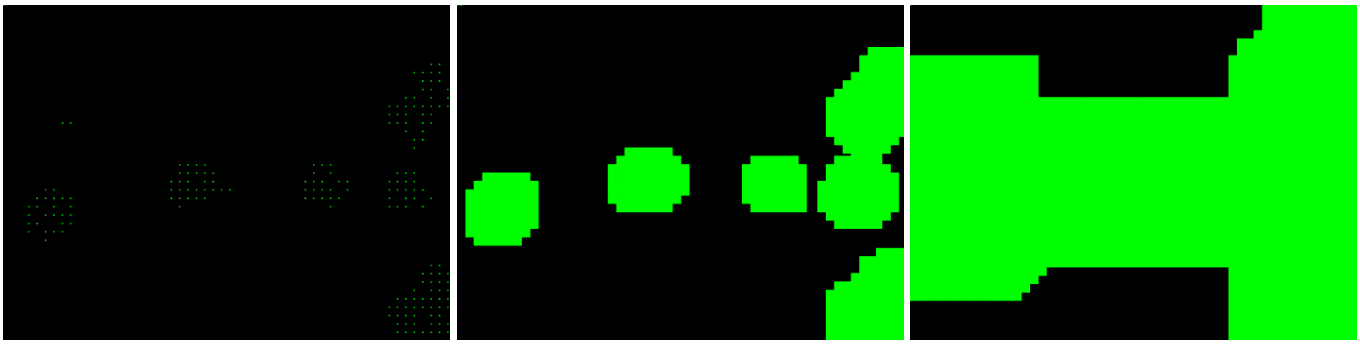

Figure 12. Different outputs for REGION_LENGTH: left 2, middle 45, right 100

```
#define PIXELS_PER_STEP 12
```

Finally, PIXELS_PER_STEP defines how many pixels should the scanning area move to the right on each row when filling the regions of interest areas. This has a direct impact on the performance as well. A small value, means that more scanning will occur, false positives can be encountered, but the detected regions will be more smooth, while a bigger value, would output the shape of the regions and would increase inaccuracy of localisation.

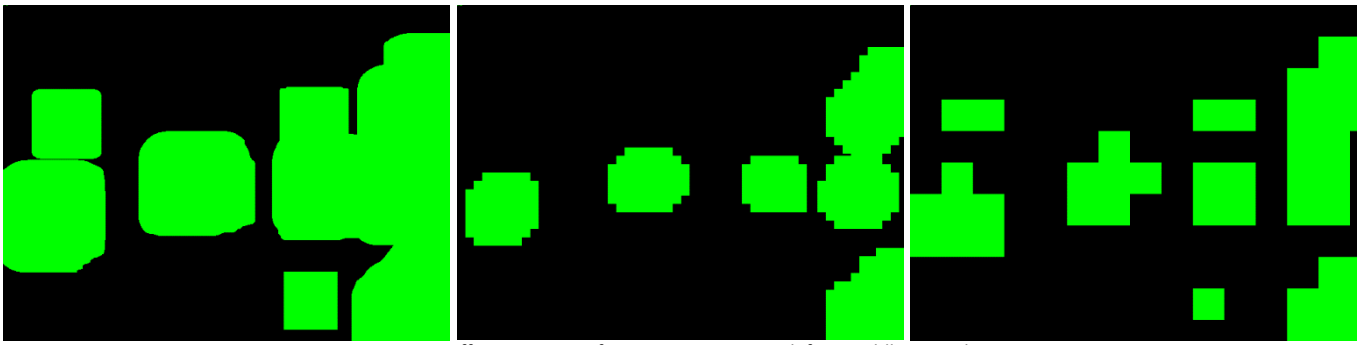

Figure 13. Different outputs for PIXELS_PER_STEP: left 2, middle 12, right 45

The higher value is set for PIXELS_PER_STEP, the bigger the distance the scanning area will travel. This means that for values over 45, some edges might be skipped. For values over 80, means that for an image with a width of 640 pixels, the scanning area will be moved 8 times, and 35 pixels will be skipped (as the scanning area is only 45) rendering the ROI detection task completely inaccurate.

5.3 Performance improvements

Comparing to the PC version, the PS3 version should be in theory more efficient because of the task parallelisation. Also, other minor improvements were done:

- wherever was possible functions were replaced with inline functions
- wherever was possible pre-processor directives were used instead of functions (macros), to reduce function overhead
- reduced the amount of required memory by using the right type (float instead of double, etc.)
- where the type was not known at compile time unions were favoured to structs
- some operations were replaced with intrinsic SIMDs.

```
__vector float vec_x_weight = spu_mul(vec_w, vec_x);
__vector float vec_y_weight = spu_mul(vec_w, vec_y);
```

- some vectors were replaced with stack-arrays
- enabled the maltivec in the build script

```
cmd = 'ppu-g++ -maltivec -o app ../common/*.cpp main.cpp -lspe2 '
```

When compiling, the -0n argument is used, not only to "*remove the debug information, but also to make the compiler rearrange statements to improve performance and reduce code size*"[11]

The following 3 options are available for optimisation:

- -0 or -01: merge identical constants, attempt to remove branches, optimize jumping
- -02: align loops, functions and variables, remove null-pointer checks, reorder instructions
- -03: inline simple functions, rename registers, parse all source before compiling code

To benefit from this, the third level of optimisation was used when building. When optimisation was turned on, the **-03** argument was added to the cmd in the build script.

6. Testing

After all improvements were completed, the application was tested on the Cell Broadband Engine. No more changes were made to the visual_tunning.h to avoid biasing the results. The application was executed 4 times for each version using only one image for input: for pc, for the cell without optimisation and for the cell with optimisation enabled. The best times were recorded for each of the task.

Then all 10 images were sent for processing using the same methodology (best time out of 4 execution attempts) and again the results were recorded.

7. Results

Overall when running the application for all the images the expected result was obtained: the i7 processor is the fastest of all even if the application is running the task in sequence, not in parallel.

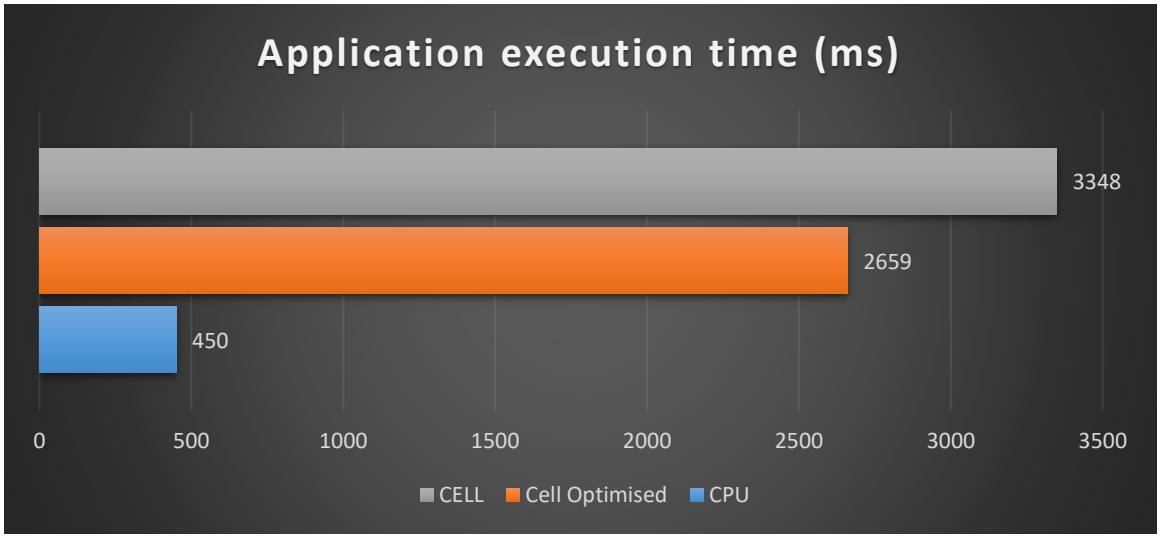

Figure 14. Application execution time in milliseconds (lower is better)

What is interesting to see though is that using the optimisations and the intrinsic SIMD, a 21% performance gain is obtained.

Also, a very interesting aspect is the fact that at task level, as we can see in Figure 15, for one image, the optimised CELL application managed to be almost on par with the i7. There were other factors that slowed down the application, such as file loading, file saving, memory access which delayed each image processing by about 210 milliseconds, which for 10 frames would accumulate into 2160 milliseconds latency. So, in theory, the difference in Figure 14 between the sequential application run on i7 and the optimised application run on the CELL is only because of other factors, as those previously mentioned.

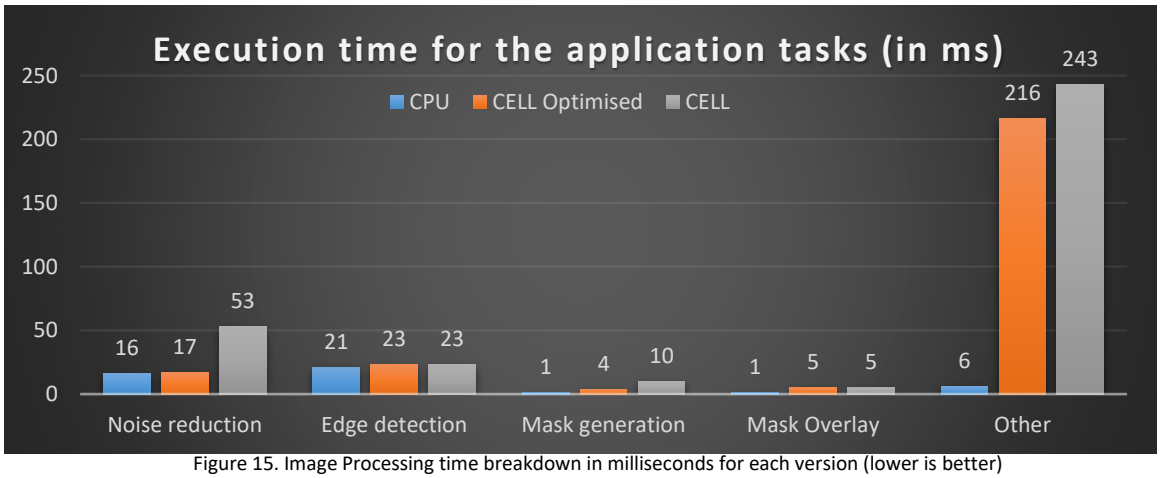

Figure 15. Image Processing time breakdown in milliseconds for each version (lower is better)

Next, in Figure 16 all images will be displayed side by side to highlight the results of the ROIs detection.

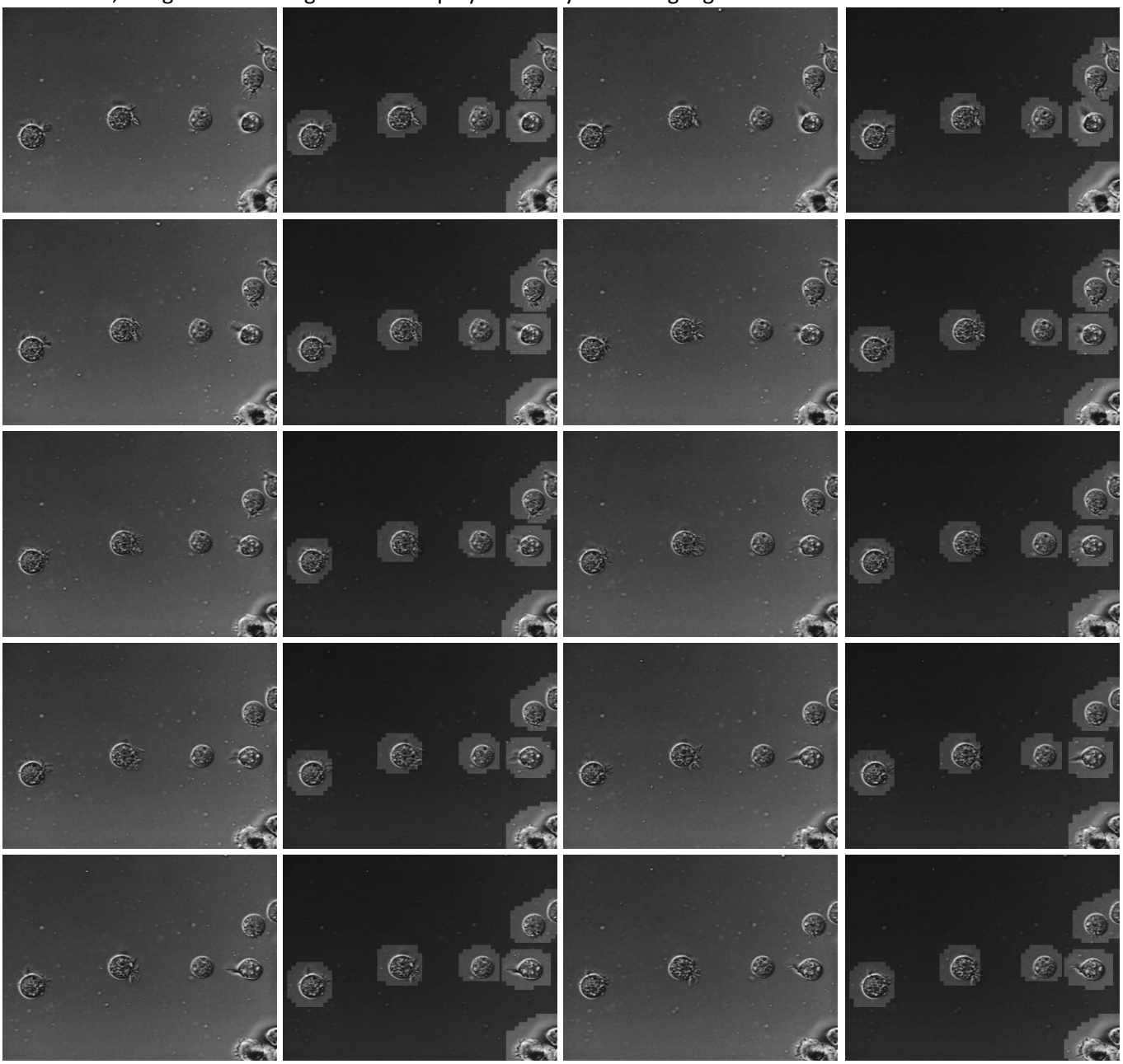

Figure 16. All the input frames and output results with ROIs detected

8. Conclusions

We have attempted to identify regions of interest (containing rat prostate cancer cells) on a sequence of a clip. An application was developed for pc where we applied blur, sobel and an overlay mask over the original image to highlight the cells. The Gaussian blur and the edge detection algorithms were the most intensive tasks. While the application was fast on an i7 CPU, we ported the application on a PS3 to improve performance using parallelisation and intrinsic Simple Instruction Multiple Data available on the Cell Broadband Engine. We presented the visual and performance improvements that were made and we could highlight the fact that without optimisation the code would run 21% slower on the ps3. At task level the ps3 was almost on par with the sequential CPU code if we ignore the time needed for loading and saving images or accessing data from memory. This demonstrates the benefits of using all the cores of a processor and shows how powerful the CELL is when intrinsic instructions are performed for multiple data. In the end, we have displayed the input image and output image in parallel for each frame to display how the application identified the regions of interest.

Work breakdown:

Overall we all feel that we have contributed evenly, and we would appreciate if there would be no differences in marking our work. (The authors)

Livingstone Jonathan (worked on pc and ps3): code optimisation, simd optimisation, benchmarks;

I have learned a lot over the course of this project. During it, I have become much more comfortable with how intrinsic functions work, and how SIMD is used to achieve high-performance code. I also learned about some of the difficulties faced when programming on the SPUs for the cell processor and writing thread-safe code. For my part, I worked on the optimization towards the second half of the project, mostly the SIMD stuff. Due to how my team was structured, I also learned a lot about Agile programming and feel comfortable knowing this will help me in my future.

Taylor William (worked on pc and ps3): working pc version, ported application to ps3;

During this project, I did a lot. I wrote the initial PC code and then ported the code to the PS3 to run on multiple SPU's. I also built the basic build system using a basic python script. I also wrote much of the common classes used to deal with files, logging and high resolutions timers. I've been equipped with an enhanced knowledge of the PS3 but this project has also made me better at programming for unique hardware offerings. Also, the low-level nature of the project has also helped me brush up on my low-level programming skills. These skills will certainly help me in future.

Toader Constantin (worked on pc): research, report, video and performed visual tuning;

After completing this module, I can better understand the power of the CELL processor and the benefits of running tasks in parallel. I see how difficult it is to program for a console, without using IDEs and using low-level APIs. I'm happy that I could deploy with my team the same procedures for working in an Agile team that I used during my internship. Also, I am glad I acquired researching and writing skills, which will benefit me for my honours project and I like the fact that I got introduced to Computer Vision algorithms.

References

[1] Phung, S. L., & Bouzerdoum, A. (2007, April). Detecting people in images: An edge density approach. In *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07* (Vol. 1, pp. I-1229). IEEE.

[2] Juneja, M., & Sandhu, P. S. (2009). Performance evaluation of edge detection techniques for images in spatial domain. *international journal of computer theory and Engineering*, *1*(5), 614.

[3] Szeliski, R., 2010. *Computer vision: algorithms and applications*. Springer Science & Business Media., pp.10-16.

[4] Canny, J., 1986. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6), pp.679-698.

[5] Nalwa, V.S. and Binford, T.O., 1986. On detecting edges. *IEEE transactions on pattern analysis and machine intelligence*, (6), pp. 699-714.

[6] Xue, W., Wenxia, X. and Guodong, L., 2015, January. Image Edge Detection Algorithm Research Based on the CNN's Neighborhood Radius Equals 2. In *Proceedings of the 2015 Sixth International Conference on Digital Manufacturing and Automation* (pp. 150-153). IEEE Computer Society.

[7] XIA, Z. Q., FENG, X. Y., & PENG, J. Y. (2009). Detecting the Regions of Interest with Edge and Depth Information [J]. *Computer Simulation*, *7*, 062.

[8] Itti, L., Koch, C. and Niebur, E., 1998. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on pattern analysis and machine intelligence*, *20*(11), pp.1254-1259.

[9] Goz (2016). *How do I gaussian blur an image without using any in-built gaussian functions?*. [online] Stackoverflow.com. Available at: http://stackoverflow.com/questions/1696113/how-do-i-gaussian-blur-an-image-without-using-any-in-built-gaussian-functions [Accessed 11 Nov. 2016].

[10] Sobel, I., 2014. History and definition of the sobel operator. *Retrieved from the World Wide Web*.

[11] Scarpino, M. (2009). *Programming the cell processor*. 1st ed. Upper Saddle River, NJ: Prentice Hall, p.38.