

Comparing a Serialised and a Parallelised Implementation of an Image Blurring Algorithm_OJORDAN_P144558

by Oliver Jordan

FILE	2911_OLIVER_JORDAN_COMPARING_A_SERIALISED_AND_A_PARALLELI SED_IMPLEMENTATION_OF_AN_IMAGE_BLURRING_ALGORITHM_OJORD AN_P14_1247040865.DOCX (310.59K)		
TIME SUBMITTED	07-MAY-2019 10:09PM (UTC+0100)	WORD COUNT	2285
SUBMISSION ID	106588249	CHARACTER COUNT	11455

**Comparing a Serialised and a Parallelised Implementation of an Image
Blurring Algorithm**

Oliver Jordan

(P144558)

High Performance Computing for Games and Visualisation

1. Abstract

This report reviews some of the common methods of image blurring. A CPU implementation of one of these methods, the Quincunx filter, is described, along with a parallelised GPU implementation. The timings of each implementation is provided, showing the GPU approach to be significantly faster.

Contents

1. Abstract.....	2
2. Introduction.....	3
3. Review	3
3.1. Common image blurring methods	3
3.1.1. Box Blur.....	3
3.1.2. Gaussian Blur	4
3.1.3. Quincunx Blur.....	4
3.1.4. Simplified Quincunx Blur.....	5
3.2. Separable Filters.....	5
3.2.1. Applying Separable Filters to Common Techniques	6
3.3. Application to the GPU.....	7
4. Implementation	7
4.1. Features Applicable to both Approaches.....	7
4.2. CPU Implementation	7
4.3. GPU Implementation.....	8
4.4. Profiling results.....	9
5. Conclusion.....	9
6. References	10

2. Introduction

Modern GPUs are now capable of running many tasks on their own processors, taking strain off of the CPU in a computer build. Additionally, many GPUs are capable of running multiple calculations simultaneously, allowing many problems that require identical operations on large data sets to be solved much faster than a CPU could.

This report presents research into a problem that would be suitable for a parallelised approach and documents the development and results of such an application.

3. Review

In this section, a review of image blurring methods is presented. Following this review, a justification for parallelising a solution is presented.

3.1. Common image blurring methods

Image blurring is a type of image convolution. Image convolution algorithms can be categorised as either linear or non-linear [1] and image blurring algorithms fall under the linear category – that is to say, image blurring algorithms follow a defined rule and produce intuitive results [1].

Many image blurring algorithms make use of a convolution filter – a grid that can be overlaid on an image, containing weightings defining how a blurred pixel gets its new value from its neighbours [1] [2].

3.1.1. Box Blur

One of the simplest convolution filters is a box blur. This filter is basically a simple grid of a defined size, with hand picked weightings for how each neighbour affects the new pixel value [1] [2]. These weightings may be identical, resulting in the new pixel value simply being the mean of itself and its neighbours, or may be some arbitrary weighting decided by the programmer.

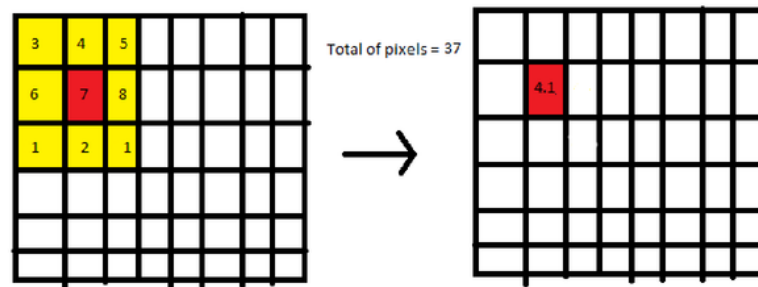


fig 1. An example of a box blur filter

This filter is incredibly simple to implement, requiring a simple mean calculation and can use as few pixels in its calculation as desired. However, a box blur is not ever guaranteed to give a smooth blur and can produce some coarse results [1].

3.1.2. Gaussian Blur

Gaussian blur is a convolution filter that uses values on a Gaussian curve – hence the name – to define the weightings used in the filter [1]. This filter can provide a more “perfect” looking blur. This method is popular and calculators that can create these filters can be found online [3].

0	0	0	5	0	0	0
0	5	18	32	18	5	0
0	18	64	100	64	18	0
5	32	100	100	100	32	5
0	18	64	100	64	18	0
0	5	18	32	18	5	0
0	0	0	5	0	0	0

fig 2. An example of a Gaussian convolution filter

The Gaussian blur is considered to be important for computer vision and these filters are used to help blur images not just for viewing, but also to reduce noise for other image processing algorithms [2].

3.1.3. Quincunx Blur

Quincunx blur is a technique developed to produce image blurring results comparable to the effects of 4x super sampling anti-aliasing techniques [4]. This method creates a second buffer of pixels, identical to the image, and shifts this buffer up and left by half a pixel. Each pixel in the original buffer is now surrounded by four pixels from the second buffer. Using these five values and a specific weighting, the new pixel value can be determined.

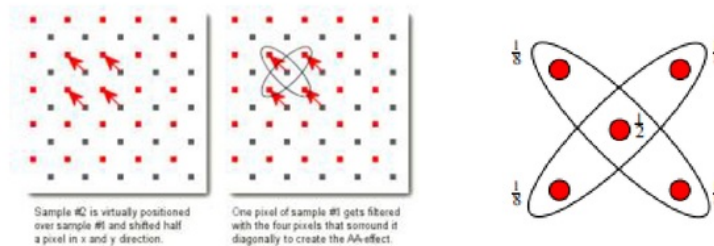


fig 3. (left) An illustration of how the buffers are positioned in a Quincunx filter implementation.

fig 4. (right) The weightings used to apply the image blur.

3.1.4. Simplified Quincunx Blur

While the original Quincunx implementation uses two buffers, Cant proposes that this can be further simplified [4]. Since the top left pixel of the second data set is identical to the centre pixel, these weightings can be combined, giving a new 2x2 filter. The remaining three pixels used are the neighbouring pixels to the right, below and diagonally below and right. The resulting 2x2 pixel can be aligned with the pixel being blurred in the top left, giving the same result in a much simpler fashion [4].

$$\begin{array}{cc} \frac{1}{8} & \frac{1}{8} \\ & \frac{1}{2} \\ \frac{1}{8} & \frac{1}{8} \end{array} = \begin{array}{cc} \frac{5}{8} & \frac{1}{8} \\ \frac{1}{8} & \frac{1}{8} \end{array}$$

fig 5. The simplified Quincunx filter.

3.2. Separable Filters

An interesting property of certain convolution filters is their ability to be defined as two separate filters. While not applicable to every filter, splitting a filter up into two smaller filters has the potential to reduce the number of operations required to apply the whole filter [5].

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

fig 6. A 3x3 convolution filter

$$\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \text{ followed by } \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

fig 7. The same filter described as two smaller filters

With the filter split, the two smaller filters are then applied to each pixel – using one filter on the whole image in one pass, then the other filter in a second pass [5].

Applying a filter in this fashion reduces the number of operations. Any filter has a number of operations $n*m$ where n is the filter width, and m is the filter height. Using a separable version of the filter reduces the number of operations down to $n+m$ instead [5], allowing a faster implementation of the same filter.

3.2.1. Applying Separable Filters to Common Techniques

Of the techniques presented above, consider only a Gaussian filter and a Quincunx filter.

Gaussian filters are mathematically defined and as such will always be a separable filter. If the filter presented in *fig 2*. were to be split, the same blurring could be applied to an image requiring only 14 operations per pixel, as opposed to the full filters 49. This makes the “perfect” blur of a gaussian filter much cheaper to achieve.

Quincunx on the other hand, is not able to be separated, and will always require exactly 4 operations per pixel. Separable filters are useful for reducing the cost of visually accurate blurring; however, the Quincunx filter is the fastest blurring algorithm of the two.

This falls in line with research from Nvidia, in their report titled “Image Convolution with CUDA”, where a discussion of separable filters takes place [5]. IN this report a graph comparing the speed of separable filters and their full filter counterparts is presented – shown here in *fig 8*. The blue line is the speed of the larger filter.

Performance

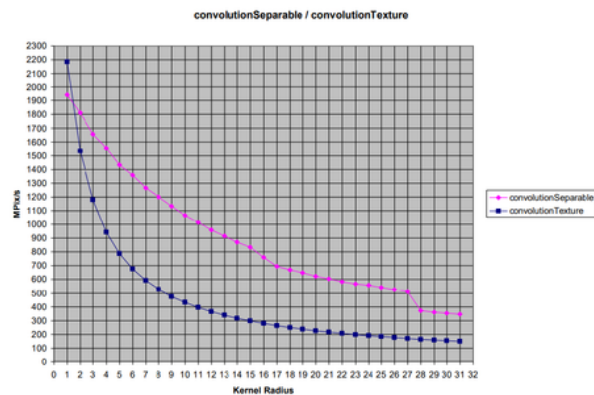


fig 8. Nvidia's comparison of separable filters [5]

When referring to kernel radius, this is the number of grid cells from the center of a convolution filter to the edge of the filter – the Gaussian filter in *fig 2*. would have a kernel radius of 3, for example.

The Quincunx filter could be considered to have a kernel radius less than 1, as it does not have a center cell. Following the trend of the curves in the graph past anything before a kernel radius of 1 suggests that non separable filters will be quicker than a separated separable filter,

confirming that the simplified Quincunx filter would be faster than the separated Gaussian filter. Even the smallest possible Gaussian filter, at a size of 3x3, would require 6 operations when separated.

3.3. Application to the GPU

This project uses the simplified Quincunx approach, as it provides the cheapest way to blur an image, in an attempt to discover just how powerful a parallelised approach can be even over a sufficiently fast CPU implementation. The following chapter details the implementation of both of these approaches and discusses the results.

4. Implementation

Implementation of the project will be looked at in three parts – the CPU approach, the GPU approach, and aspects of the program required for both approaches.

4.1. Features Applicable to both Approaches

In order for the image data to be blurred, an image must be read into a usable format. For this purpose, an open source bitmap file reader was added to the project [6]. This loads in the colour data of each pixel of an image, along with the meta data such as the image's height and width. It also provides suitable data structures to store this data in. The key parts of this library used for this project are the "bmpread_t" data structure, and a provided function "readBitmap" which returns the data of a bitmap image into that data structure's format.

The "bmpread_t" data structure stores pixel data in an unsigned char array. The data for each colour channel in a pixel is placed into a new element of the array, making the array's size equal to the image height * image width * 3. In order to access the red channel of any pixel at coordinate (x,y), the array must be accessed using the position $(x * width) + y$. The blue and green channels can be accessed by adding 1 or 2 to the position respectively.

Additionally, OpenGL 2.0 is used to bind bitmap images as textures and draw them to the screen, in order to showcase visually the blurring effects each approach is creating. This was used as opposed to writing the bitmap data back out to a new image due to its ease of implementation and viewing – unblurred and blurred images are presented side by side for better comparisons.

Quincunx blurring applies a small amount of blur to reduce the amount of aliasing in a scene. As such, the blur applied can be very hard to see. To circumvent this, both approaches apply the blur a total of 25 times to produce an easily visible blur.

4.2. CPU Implementation

For the CPU implementation, the image data is passed to a function, "cpuBlur". The function then blurs every colour channel of a every pixel.

For each pixel being blurred, the value 3 elements along the array is accessed, representing the same colour channel of the pixel to the right. To access the pixel below, the element $x * width$ elements along is accessed. The pixel to the right and below is another 3 elements along from that. These values are added to a total using the weightings defined for a simplified Quincunx filter and the new value is then placed in a second "bmpRead_t" structure, ready to be returned once all pixels have been calculated.

One problem to consider when blurring an image is how to address cells of a filter that fall outside of the image. If the pixel being blurring lies on the right edge of the image, the values used for the pixels on the right side of the filter are duplicated from the pixel values used on the left side. In similar fashion, pixels that fall on the bottom edge of the image duplicate the pixel values used in the top half of the filter to be used for the bottom half. The pixel in the bottom right corner of the image combines these two methods and results in all four parts of the filter using the same pixel value – the top left pixel. This sufficiently solved this problem.

The CPU approach was successful, and the results can be seen in *fig 9*. To see the results more clearly, it is recommended to view the program's resulting images directly, by running the code submitted with this project.

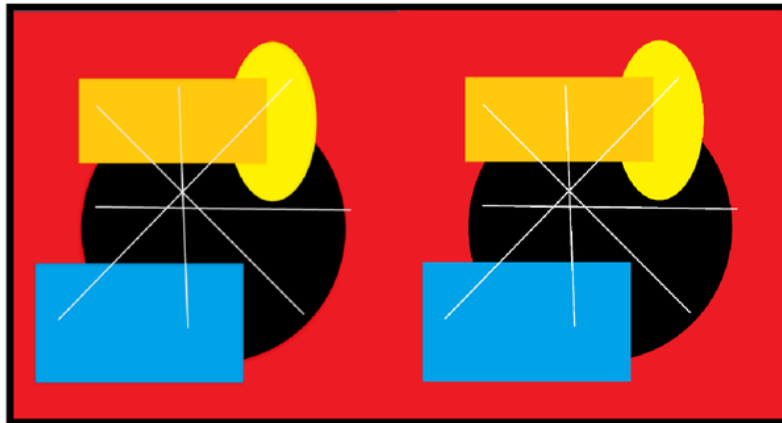


fig 9. The result of the CPU implementation

4.3. GPU Implementation

The GPU implementation uses CUDA to parallelise the approach onto the GPU. This approach works in much the same way, aside from each pixel being able to be blurred simultaneously.

The major problem with this approach was controlling access to pixel data. To ensure no thread on the GPU was accessing data that was already being used, and in order to ensure all data is done being updated before being accessed again, a four-pass system was implemented. In this system, every other pixel along the X axis, and every other pixel along the Y axis is blurred, allowing ample room for pixel data to be accessed without encroaching on the data another

thread is using. On each subsequent pass, a new offset is used, until all pixels have been blurred.

The results of this approach can be seen in *fig 10*. While there are some unwanted artefacts produced by this implementation, it is largely successful.

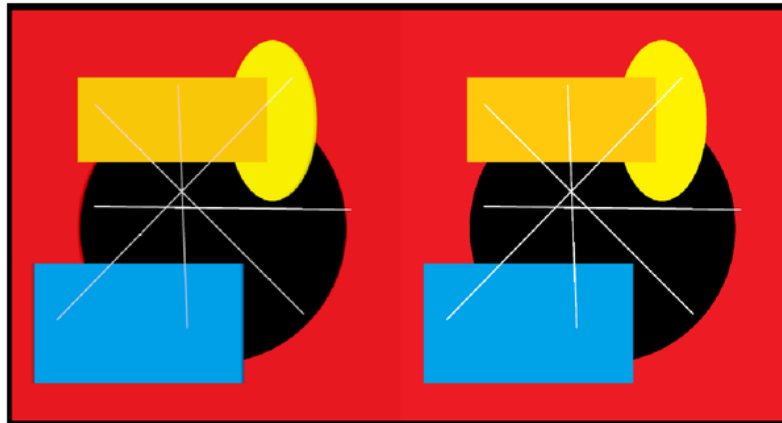


fig 10. The results of the GPU implementation

4.4. Profiling results

The time taken for the CPU approach to run all passes of its blurring function was 7451590 μ s. The time taken for the GPU approach was 1597181 μ s. The GPU approach was significantly faster, showing the power of parallelization on the GPU.

5. Conclusion

This report presented some common image convolution methods. An implementation of a Quincunx filter was described for both the CPU and the GPU. Profiling of each of these implementations shows the GPU approach to be faster, showcasing the usability of parallelisation for image convolution methods.

6. References

- [1] R. Chandel and G. Gupta, "Image Filtering Algorithms and Techniques: A Review," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 10, pp. 198-202, 2013.
- [2] U. Sinha, "Convolutions: Image Convolutions Examples - AI Shack," [Online]. Available: http://aishack.in/tutorials/image-convolution-examples/?fbclid=IwAR2MeoemYZnqhENQwma44Qqze-ccpmgIXZRuDxn7Vg_r22FAI3Mz-i4WLLw. [Accessed 20 April 2019].
- [3] theo, "Gaussian Kernel Calculator | The Devil in The Details," 30 January 2014. [Online]. Available: <http://dev.theomader.com/gaussian-kernel-calculator/>. [Accessed 20 April 2019].
- [4] R. Cant, N. Chia and D. Al-Dabass, "New Anti-Aliasing and Depth of Field Techniques for Games".
- [5] V. Podlozhnyuk, "Image Convolution with CUDA," Nvidia, 2013.
- [6] C. Lindsay, "GitHub - chazomaticus/libbmpread: tiny, fast bitmap (.bmp) file loader," 23 December 2018. [Online]. Available: <https://github.com/chazomaticus/libbmpread>. [Accessed 24 January 2019].

Comparing a Serialised and a Parallelised Implementation of an Image Blurring Algorithm_OJORDAN_P144558

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

84/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

GRADING FORM: SMU GENERIC_GJ

OLIVER JORDAN

83.50

RESEARCH-INFORMED LITERATURE



Refs - a little light possibly. However key ref identified and used. Abstract included too.

6.5

KNOWLEDGE AND UNDERSTANDING



Good review of related literature. Algorithms discussed and use of methods used in implementation covered here.

12

ANALYSIS



Good. CPU/GPU covered. Relevant methods identified for implementation. Research related to this covered and efficiencies noted including separable filters.

7.5

PRACTICAL APPLICATION AND DEP.



Both CPU and GPU code evaluations provided. Very Good. Timings need to be more comprehensive - try varying buffer sizes.

54

SKILLS FOR PROFESSIONAL PRAC.



Images - left/right images need to be clearly labelled - which is blurred? Refs included - some incomplete. Only 6. Report structured. Graphs helpful.

3.5