Conference on
Information Technology and
Data Science

Compute Shader in Image Processing Development*

Robert Tornai, Péter Fürjes-Benke

University of Debrecen, Faculty of Informatics
`tornai.robert@inf.unideb.hu`
`furjes.peter99@gmail.com`

Proceedings of the 1st Conference on Information Technology and Data Science
Debrecen, Hungary, November 6–8, 2020
published at http://ceur-ws.org

Abstract

This paper will present the OpenGL compute shader implementation of the BlackRoom software. BlackRoom is a platform-independent image processing program, which supports multiple execution branches like Vulkan fragment shader, OpenGL fragment shader, and CPU-based rendering. In order to support a wider range of devices with different amounts of memory, users can utilize tile rendering, and the program can be run in browsers thanks to the WebAssembly format.

Thanks to our program's built-in benchmark system, the performance differences between the implemented CPU- and GPU-based executing branches can be easily determined. We made a comprehensive comparison between the rendering performance of our CPU, OpenGL compute shader, fragment shader and Vulkan fragment shader branches. Latter is under development, which induces a relatively higher runtime presently.

A further aim is to optimize our algorithms, which are using Vulkan API. Besides that, the program will be capable of rendering multiple effects at once with Vulkan fragment shader. Furthermore, the available GPU rendering and multi-threading features are planned to be enabled for WebAssembly platform yielded by the Qt framework [2].

Keywords: Image processing, benchmark, CPU, shaders, Vulkan, OpenGL

AMS Subject Classification: 65D18, 68U10, 97R60

*This work was supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002. The project was supported by the European Union, co-financed by the European Social Fund.

1. Introduction

BlackRoom is an image processing application developed in Qt 5.15 version [6]. The goal was to use the most modern techniques, so we implemented the algorithms using compute shader also beyond fragment shader of OpenGL and Vulkan fragment shader. This paper will cover the results of these implementations.

The structure of BlackRoom is based on the standard skeleton, which is introduced in GPU Gems [4]. As a source operator, we have a load operator for processed and raw formats. Our system contains image filters. Some filters, as the Harris shutter, have additional load operators for the color channels, thus extending the simple demo structure mentioned in GPU Gems. Consequently, not just linear processing paths can be accomplished. We have implemented both image view and save operator as sink operators. Similar to the framework of Seiller et al., each processing path is implemented in a separate class to follow the basic principles of object-oriented programming [7].

During the research we have studied the existing accelerated image processing libraries. Although the progressive GPUCV library was found to be one of the best, it has not been developed since 2010, and it was never made available for web applications [3]. However, Allusse et al. enhanced the system with CUDA support, which is a proprietary technique, and it is not web-enabled either [1]. Our program supports Linux, macOS and Windows platforms, and most of its features are also available for the WebAssembly. The optimization of our image processing software yielded better and customizable memory management regarding the memory usage of image modification algorithms. Our other solution for achieving better performance was implementing our algorithms for Vulkan [8]. Vulkan API will get significance in adding Android support for our program in the near future.

2. Utilized Technologies

Nowadays, several application programming interfaces are available for image processing. Our goal was to implement the most common platform independent APIs and to collect statistics about their performances in different use cases. Therefore, the BlackRoom software provides multiple execution branches based on OpenMP, OpenGL and Vulkan APIs. In this section, these technologies will be presented in order to give a short summary of their characteristics.

2.1. OpenMP

The Open Multi-Processing is an API which supports shared-memory multiprocessing on CPU. It was released in 1997 for Fortran and since 2000 it has supported C and C++ programming languages, as well. This API is quite wide-spread solution for implementing parallel execution in applications. Its usage is straightforward in C++ programs since the programmers only need to use #pragma-s before the specific program code parts. In terms of performance, the difference between the sequential and parallel processing highly depends on the given use case and the number of utilized threads, but with complex calculations the latter usually provides significantly better results.

2.2. OpenGL

The Open Graphics Library is a platform independent 2D and 3D graphics API which was released in 1991. It is quite a robust high-level API, and thanks to that it is widely adopted in the industry. With the release of OpenGL ES, it can be used on mobile devices and in web applications thanks to WebGL. Since OpenGL is used for hardware-accelerated rendering on GPU, it is really efficient in image processing and what is more, thanks to the compute shader support, it can be used for general calculations, as well.

2.3. Vulkan

Vulkan is the newest platform independent 2D and 3D graphics API which is based on the Mantle API developed by AMD. It was released in 2016, and its main goal was to provide higher performance and balanced CPU/GPU usage. Similarly to OpenGL, it is available for multiple platforms and hardware, like Windows, Linux, Android and macOS through MoltenVK. Compared to OpenGL, the Vulkan API can be 100% faster but it really depends on the application and the implementation. Latter is one of the key point of this new API, since the developer has almost full control over the graphics processing unit and because of that, for beginner programmers it is really difficult to implement the interface efficiently.

3. Benefits of OpenGL Compute Shader

Previously BlackRoom used only OpenGL fragment shader for computing the effects on GPU. Meanwhile, this way of the computing has its benefits for our needs, and this approaches brought in some challenges. First of all, the software contains multiple context-sensitive algorithms where the neighboring pixels are used for the final result. With OpenGL fragment shader, the access of the neighborhood was not really effective, although it has improved by introducing rectangle textures that enabled the usage of integer indices instead of float values. Secondly, the histogram generation may be even slower than computing it on the CPU [5]. Finally, the implementation of the OpenGL fragment shader is a little bit more complex compared to our needs.

So, we started to implement our algorithms in OpenGL compute shader because of the above reasons. For this executing branch, the program uses OpenGL fragment shader only for the onscreen rendering, and the effect chain calculation is done completely by OpenGL compute shaders. In terms of compute performance, the difference between the two approaches is not significant since both use the same hardware. However, the source code is more straightforward and simpler. The code for histogram generation is more elegant than by OpenGL fragment shader. From OpenGL 4.3 the atomic counters give a huge boost to histogram calculations.

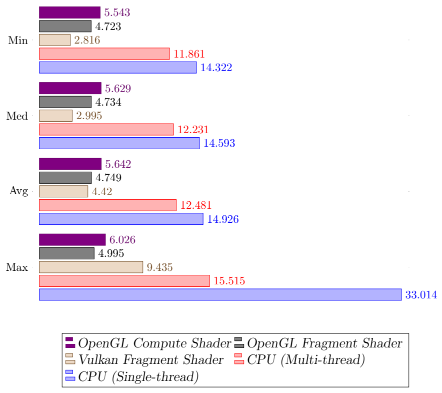
4. Performance Comparisons

Our test system contains an AMD Ryzen 5 3600 processor @ 4.35 GHz and an Nvidia GTX 960 graphics card with 4 GB memory. The following effects were used in order to compare the performance of the different executing branches: basic modifications, edge detection, Gauss filter, infrared and grayscale effects. As for basic modifications, we are talking about exposure value and brightness. To obtain the execution times we used the `QElapsedTimer` class, which measures the elapsed time in nanoseconds. Because the magnitude of the running time of our algorithms is millisecond, after readout, the timer variable is divided by 1 000 000 in order to yield values of milliseconds. The measured times in the figures represent only the runtimes of the effect executions without the bus transfer between the main memory and the video card. The effects were tested with multiple images and according to our observation the size of the image and the execution time is in linear relationship. The results below were measured with a 4608×3072 PNG image (see Figure 1). The color depth and the number of color channels do not affect the results since the program converts every image to single-precision floating-point format.

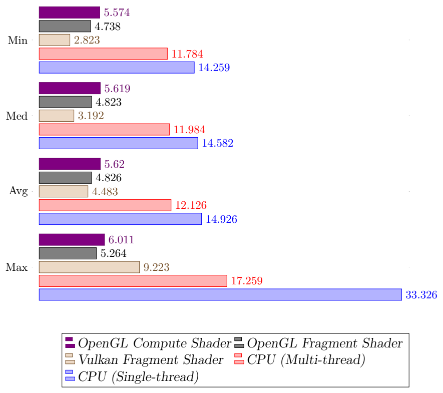
Figure 1. Tested image.

4.1. Context-free Algorithms

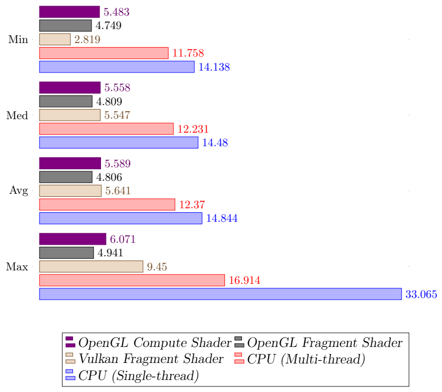
The processing times of the basic—exposure value and brightness—modifications, the infrared and the grayscale effects were measured. According to the results (see Figure 2), rendering by GPU is approximately twice as fast as rendering by CPU on a single core.



(a) Basic Effect



(b) Infrared Effect



(c) Grayscale Effect

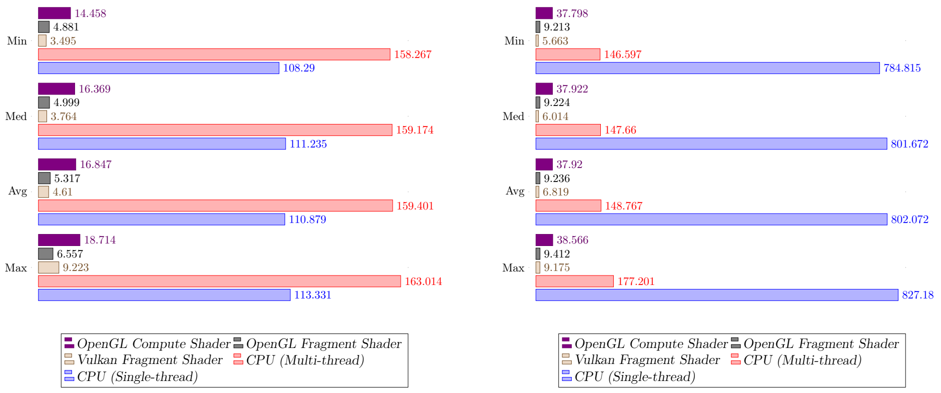
Figure 2. Time results of context-free algorithms in milliseconds.

Utilizing multi-threading and SIMD decreases the gap but raises another problem. The memory bandwidth is limiting the all core performance of the CPU. Furthermore, thread management also increases the execution time. Nevertheless, taking into consideration the basic, infrared, and grayscale effects we can see an almost 20% decrease in execution time compared to the single-thread performance. It can be seen that the multi-core performance is more consistent because of the smaller gap between the extreme values.

Looking at the comparison of different execution branches of the GPU rendering, we can see a little performance advantage in favor of OpenGL fragment shader with grayscale effect. The Vulkan fragment shader execution branch is under development at present, but even now, it has decent performance. Talking about basic and infrared effects' execution time, the Vulkan fragment shader is the best. Inconsistency is its worst drawback since the difference between the extreme values is here the biggest among the execution branches. The OpenGL compute shader is behind the two other GPU rendering branches in terms of execution time. Meanwhile, it provides really consistent performance.

4.2. Context-sensitive Algorithms

The context-sensitive algorithms were represented by edge detection and Gauss filter effects during the benchmarks. These effects calculate each pixel based on its neighbors. The benchmark tests show that there is quite a big difference between these two effects in terms of performance (see Figure 3). As an example, rendering the Gauss filter can profit from the extra threads of the CPU. Its execution time is almost eight times faster on multiple threads than on a single thread. Meanwhile, the edge detection's runtime is slower by utilizing multi-threading. The reason for this is the simplicity of the edge detection compared to the Gauss filter. In this case, supposedly, the limited memory bandwidth and the thread management increase the execution time.



(a) Edge Detection **(b)** Gauss Filter

Figure 3. Time results of context-sensitive algorithms in milliseconds.

The variance between the three GPU based execution branches is greater compared to the results with context-free algorithms. The OpenGL compute shader falls behind both OpenGL fragment shader and Vulkan fragment shader. According to our experiments, the overhead of compute shader causes a huge difference. Accessing the neighboring pixels in the same work group is really efficient but getting the pixel colors from other work groups takes too much time. OpenGL fragment shader provides the second best overall performance in edge detection and in Gauss filtering. The Vulkan execution branch – which is still under development – is a little bit faster. However, there is a huge room for improvement, especially in terms of consistency.

5. Results

We implemented OpenGL compute shaders, and most of our effects are now available for this execution path. Its performance increment is significant compared to the CPU computation. Meanwhile, the execution times of the context-sensitive algorithms are big due to the pixel access from other work groups. Our Vulkan implementation is mature enough for competing with our OpenGL compute shader and OpenGL fragment shader implementations. Furthermore, its Android support makes it useful for the BlackRoom software.

Unfortunately, the theoretical speedup of the parallelization of algorithms on either CPU or GPU cannot be achieved because the memory bandwidth is heavily limiting the multi-core performance of both of them.

6. Future Work

Since our Vulkan implementation can only handle just one effect at a time now, we are planning to develop it further for calculating a whole effect chain at once. Besides that, WebAssembly also remains in our scope, and we will provide a wider range of functionality of our program on this platform. Thanks to the compute shader implementations, the creation of histogram becomes easier, so we will add a panel to the user interface where the user can see the histogram change in real-time.

References

[1] Y. Allusse, P. Horain, A. Agarwal, C. Saipriyadarshan: *GpuCV: A GPU-Accelerated Framework for Image Processing and Computer Vision*, in: Advances in Visual Computing. ISVC 2008. Lecture Notes in Computer Science, Las Vegas, December, 2008, pp. 430–439, doi: `http://dx.doi.org/10.1007/978-3-540-89646-3_42`.

[2] L. Z. Eng: *Qt5 C++ GUI Programming Cookbook: Practical recipes for building cross-platform GUI applications, widgets, and animations with Qt 5*, 2nd, Birmingham, England: Packt Publishing Ltd., March 27, 2019.

[3] J.-P. Farrugia, P. Horain, E. Guehenneux, Y. Alusse: *GPUCV: A framework for image processing acceleration with graphics processors*, in: IEEE International Conference on Multimedia and Expo, Toronto, July, 2006, pp. 585–588, doi: `http://dx.doi.org/10.1109/ICME.2006.262476`.

[4] F. Jargstorff: *A Framework for Image Processing*, in: GPU Gems, ed. by R. Fernando, 1st ed., Boston: Addison-Wesley Professional, April, 2004, chap. 27, pp. 445–467.

[5] A. Kubias, F. Deinzer, M. Kreiser, D. Paulus: *Efficient computation of histograms on the GPU*, in: SCCG '07: Proceedings of the 23rd Spring Conference on Computer Graphics, April 2007, pp. 207–212, doi: `http://dx.doi.org/10.1145/2614348.2614377`.

[6] G. Lazar, R. Penea: *Mastering Qt 5: Create stunning cross-platform applications*, Birmingham, England: Packt Publishing Ltd., December 15, 2016.

[7] N. Seiller, N. Singhal, I. K. Park: *Object oriented framework for real-time image processing on GPU*, in: Proceedings of 2010 IEEE 17th International Conference on Image Processing, Hong Kong, September, 2010, pp. 4477–4480,
doi: `http://dx.doi.org/10.1109/ICIP.2010.5651682`.

[8] R. Tornai, P. Fürjes-Benke, L. File, D. M. Nyitrai: *WebAssembly and Vulkan API in Image Processing Development*, in: Proceedings of the 11th International Conference on Applied Informatics (ICAI) (Eger, Hungary, Jan. 29–31, 2020), ed. by I. Fazekas, G. Kovásznai, T. Tómács, CEUR Workshop Proceedings 2650, Aachen, 2020, pp. 382–391,
url: `http://ceur-ws.org/Vol-2650/#paper39`.