GPU-Accelerated Computer Vision on the Linux Platform

Richard Viney¹ and Richard Green²
Department of Computer Science
University of Canterbury, Christchurch 8140, New Zealand.
¹rjv20@student.canterbury.ac.nz, ²richard.green@canterbury.ac.nz

Abstract
The GPUCV library accelerates computer vision algorithms using the huge computational power of modern graphics processing units (GPUs). This paper presents a series of alterations to the GPUCV library that enable it to be used natively on the Linux platform and so bring the performance advantages provided by GPUs to computer vision applications running on Linux. The considerations and decisions involved in this process are presented and discussed, and the performance characteristics of the GPUCV library on this new platform are compared to the original Microsoft Windows platform. Finally, ideas for future work on increasing the performance and ease-of-use of GPU accelerated computer vision development are put forward.

Keywords: Computer Vision, GPU, GPUCV, OpenGL, GLSL, Linux

1 Introduction

Computer vision algorithms often demand large amounts of processing power, and this has resulted in efforts to allow these algorithms to take advantage of the often under-utilized processing power present in modern GPUs. The massively parallel stream processing nature of modern GPUs is an excellent match for accelerating many common computer vision tasks, and can be used to both increase the performance of existing algorithms and to enable the use of much higher resolution images.

Until now, access to this type of library on the Linux platform has been restricted due to the lack of platform and compiler compatibility in the source code of existing projects. This paper details the process and outcomes involved in bringing an existing GPU-enhanced computer vision library to this increasingly popular platform.

This report is structure as follows. Section 2 describes the architecture of GPUs and why they are a suitable hardware target for accelerating computer vision algorithms, section 3 introduces the GPUCV library, section 4 details the processes involved in bringing the GPUCV library to the Linux platform, section 5 presents and analyzes performance results of the GPUCV library on the Linux platform, section 6 contains the conclusions, and section 7 discusses possible future work.

1.1 Why GPUCV?

This paper focusses exclusively on the GPUCV [1] library, however there are other GPU computer vision acceleration libraries available such as NVIDIA [2]. GPUCV was chosen because it is reasonably complete and because it is being more actively developed than other similar projects.

1.2 Motivation

There were several motivating factors for bringing the GPUCV library to the Linux platform. GPUCV has close ties with the OpenCV library [3] [4] which is available on a wide variety of platforms, and so it is useful for GPUCV to also be cross-platform as one of its goals is for existing OpenCV applications to be able to take advantage of GPU acceleration simply by using the GPUCV image processing operators instead of the equivalent OpenCV ones. Bringing GPUCV to the platforms supported by OpenCV helps to realize this goal more fully.

Also, it is not desirable to tie computer vision researchers to a specific development platform in order to be able to take advantage of the functionality that GPUCV has to offer. Supporting a range of platforms helps to ease and encourage the use of the GPUCV library.
2 GPU Architecture

The two overarching trends in GPUs over the last decade or so have been a massive increase in speed and steadily increasing programmability. Consistently more and more pieces of the traditional graphics processing pipeline have been able to be controlled by user-defined programs, which has resulted in a wide array of outside parties seeking to utilize this computational power for non-graphics purposes.

The languages most commonly used to program GPUs are HLSL [5] (for DirectX), Cg [6] (for DirectX and OpenGL) and GLSL [7] (for OpenGL). This paper will focus primarily on the use of GLSL and OpenGL to program GPUs as this is the most straightforward combination of GPU programming language and graphics API that has the platform support required to run on both Microsoft Windows and Linux.

Each new generation of hardware has lifted previously imposed restrictions on programmability, so many limits of older hardware such as an upper bound on the length of programs that can be executed on the GPU do not apply to the current hardware generation. Future developments are likely to see GPUs converge more and more towards becoming massively parallel programmable stream processors, and indeed this has already begun to happen.

Figure 1 shows the high level data flows between the CPU and GPU as well as the internal data flows inside the GPU itself. The individual GPU components are described in more detail in the rest of this section.

2.1 Vertex Processing

Vertex processing is the first programmable stage in the GPU pipeline. In computer graphics this stage typically computes the final position of vertices and sets up per-vertex texture and lighting information. In computer vision this stage can be used to analyze incoming data and scatter it into an output buffer, which can achieve results such as histogram generation on the GPU [8].

2.2 Fragment Processing

Fragment processing is done following vertex processing, and involves the execution of a user-defined program that defines the final output color for each pixel being processed. Note that modern GPUs allow multiple outputs to be generated by this stage. The user-defined fragment programs can also be given access to texture objects which are often used as n-dimensional lookup tables where $0 \leq n \leq 3$.

Figure 1: Visualization of the data flow between the CPU and the GPU, and also internally inside the GPU between the major pipeline stages.

This stage is highly parallel, large numbers of fragments are processed simultaneously, and it is well suited to accelerating computer vision algorithms that satisfy the constraint that the processing on each pixel does not rely on the results of processing of other pixels in the image, as the massively parallel nature of GPUs does not allow access to this information. Such algorithms include image differencing and image thresholding.

2.3 Geometry Processing

The GeForce 8000 series from NVIDIA that was released in the second half of 2006 introduced an entirely new programmable stage of the graphics pipeline that allows incoming graphics primitives such as points, lines and triangles to be both created and destroyed in hardware by user-defined programs called geometry shaders.

This level of dynamic control over the global mesh structure was not possible to achieve on earlier programmable hardware, and opens up a new avenue for graphic algorithm research as well as new ways to accelerate computer vision algorithms on this generation of graphics hardware.

3 The GPUCV Library

GPUCV is "an open library meant for easy development of GPU accelerated image processing and computer vision applications" [1]. It is written in C++ and uses the OpenGL graphics API and its native GLSL shading language to implement the GPU algorithms.

A flexible design allows it to provide access to all stages of the graphics pipeline, and creating new image operators is a straightforward task provided the developer has an understanding of C++ and GLSL. In under one hour and with no prior exposure to the GPUCV library one of the authors
of this paper was able to implement a GPU accelerated version of the OpenCV \texttt{cvAddWeighted()} operator that adds two weighted input images together with a constant term using the formula

\[ r = \text{img1} \times \alpha + \text{img2} \times \beta + \gamma \]

The GPUCV programming interface is purposefully similar to that of OpenCV. This is to make it simple to take existing computer vision applications that use OpenCV and accelerate them using the routines provided by GPUCV. In the case of the OpenCV \texttt{cvAddWeighted()} operator this involves changing the source code to instead use the \texttt{cvAddWeighted()} function provided by GPUCV and then rebuilding the application.

### 4 GPUCV on Linux

While the GPUCV library was intended from the start to be cross-platform, as of September 2007 it had not been used on any non-Windows platform which presented a number of technical challenges when bringing it to an alternative platform and compiler. Support for the Linux platform and GCC compiler was added in three stages, and the final result is a fully functional native version of the GPUCV library running on Linux. The three stages of the conversion were as follows.

#### 4.1 Stage One

The first stage was to define a cross-platform build solution for GPUCV. There was no existing build system that could function in the required cross-platform manner and so a new set of build scripts was created using the SCons build tool [9]. This resulted in a few small Python scripts that could be used to build GPUCV on multiple platforms. The build scripts were tested on the Windows platform to ensure that they created valid and correct builds of the GPUCV source code.

#### 4.2 Stage Two

The second stage was to take the new cross-platform build solution for the GPUCV library and use it to try and build GPUCV on the Linux platform using the GCC compiler. The Linux distribution used was Ubuntu Linux 7.04 with GCC 4.1.2. This resulted in the detection of a very large number of places in the source code for the GPUCV library that were not compatible with either the new platform or the new compiler. Through a series of rigorous debugging sessions each of these issues was addressed and appropriate changes made to the GPUCV source code to fix compilation and platform compatibility problems. This resulted in a lot of small edits to the GPUCV source code to bring it to a point where it would compile safely and without any warnings on both the Linux platform and the Windows platform. Finally, Windows-specific parts of the code were re-implemented in a cross-platform manner.

Encountering large numbers of platform and compiler incompatibilities when bringing a code-base of this size to a new platform is not uncommon, and it could be argued that this process is a very positive one for the project as it results in making sure that the source code is robust enough to be compiled on different compilers and run on different platforms. The extra attention to detail and adherence to good coding standards that this requires helps greatly in ensuring secure and stable code.

#### 4.3 Stage Three

The third and final stage was to test the native Linux version of the GPUCV library functioned correctly and equivalently to the Windows version. A number of GPU-accelerated image processing operations were run on both platforms and no differences were encountered in the output. This initial result suggests that the native Linux version is a reliable and accurate port of the GPUCV library, however the only real way to verify this is to put the Linux version into public circulation so that it can be tested as widely as the Windows version in a variety of different applications.

### 5 Performance Analysis

This section looks at the performance of the native Linux version of the GPUCV library compared to the Windows version. The hardware and software setup used to conduct the performance tests was as follows: AMD Athlon 3500+ CPU, 1GB

<table>
<thead>
<tr>
<th>Operation</th>
<th>Windows: OpenCV</th>
<th>Windows: GPUCV</th>
<th>Linux: OpenCV</th>
<th>Linux: GPUCV</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x3 erosion</td>
<td>39.3 ms</td>
<td>9.9 ms</td>
<td>67.2 ms</td>
<td>10.2 ms</td>
</tr>
<tr>
<td>5x5 erosion</td>
<td>61.5 ms</td>
<td>15.3 ms</td>
<td>95.8 ms</td>
<td>15.4 ms</td>
</tr>
<tr>
<td>RGB to HSV</td>
<td>27.0 ms</td>
<td>7.4 ms</td>
<td>37.7 ms</td>
<td>7.6 ms</td>
</tr>
<tr>
<td>Binary threshold</td>
<td>2.6 ms</td>
<td>2.3 ms</td>
<td>3.7 ms</td>
<td>7.5 ms</td>
</tr>
<tr>
<td>Total</td>
<td>130.4 ms</td>
<td>34.9 ms</td>
<td>204.4 ms</td>
<td>40.7 ms</td>
</tr>
</tbody>
</table>

Table 1: GPUCV performance results on Linux and Windows.
The performance tests done were modeled on those presented in the original paper on GPUCV [1] and were carried out on this hardware setup for each platform. The systems were stripped of all non-essential processes and measurements were taken. Tests were done on standard RGB images with 1024 pixels along each dimension. Each image operation was run 100 times and the average running time of one operation used as the final performance measurement. The results are presented in table 1.

The results contain two notable trends. The first is that the GPU accelerated versions of the computer vision image processing algorithms are in general significantly faster than the CPU-based OpenCV equivalents, sometimes by a very significant factor of between 4 to 5 times. The only exception to this was the binary threshold operation running on Linux where the GPU accelerated version was around half the speed of the original OpenCV implementation. This may be due to the graphics subsystem on Linux not being optimized for the type of shader operations needed to execute thresholding on the GPU.

The second noticeable trend is that the Linux platform generally has lower performance than Windows, although the gap narrows considerably when using the GPU. In standard OpenCV calculations Linux is on the order of 50% slower than Windows, the reason for which is unclear, but for GPUCV-based calculations the gap practically disappears and only a few percentage points separate the two systems. This means that for Linux computer vision applications GPUCV provides a much more significant speedup than that enjoyed by Windows applications.

It is not uncommon for software running on a new platform to lack some of the performance of the original platform that it was designed and developed on and for, so it is likely that the native Linux version of the GPUCV library will be able to improve its performance through code optimizations specific to the Linux platform. Another possible reason for the slightly lower performance of GPUCV on Linux is that the Linux graphics driver supplied by NVIDIA is not as heavily optimised as its Windows counterpart, which results in lower overall performance.

However, the results definitely show conclusively that the performance advances made available to computer vision applications by GPUCV are now also available on the Linux platform, and computer vision researchers can now take advantage of the GPUCV library on that platform.

6 Conclusion

This report has looked at the role and importance of GPUs in accelerating computer vision algorithms and then focussed on bringing these advantages to the Linux platform by way of the GPUCV library. A native Linux version of this library was created and its performance was benchmarked and compared to the original Windows version. While performance of the GPUCV library under Linux was somewhat lower than on Windows the performance advantages present in GPUCV still represent a significant improvement in the performance of computer vision algorithms on the Linux platform than can be achieved using conventional CPU-based algorithms.

7 Future Work

Possibilities for future work include adding support for other platforms such as Mac OS X to the GPUCV library.

One of the limiting factors in the performance of the GPUCV library is in the transport of image data to and from the GPU. This process tends to be time consuming, sometimes to the point where it takes as long or longer than the actual computation performed on that image data. As described earlier, the current design of the GPUCV API supplies drop-in GPU-accelerated replacements for existing OpenCV functions, and because OpenCV applications assume they always have direct access to the image data through a data pointer it is difficult to avoid expensive back and forth copying to the GPU on each GPUCV call without breaking valid usage patterns of the OpenCV API. If there were a synchronization or locking mechanism added to OpenCV that was mandatory to use before accessing image data directly using the CPU then GPUCV could potentially cache intermediate results in video memory and avoid a lot of unnecessary copying. In this system the final image result would only be read back from the GPU when it was actually needed by the CPU. The potential advantages and complications of such an approach are left for future work.

8 Acknowledgements

The excellent work of the original authors of GPUCV is much appreciated: Jean-Philippe Farrugia, Patrick Horain, Erwan Guehenneux and Yannick Alluse.
References


