Sobel Filter Report by Federico Beuter

Introduction

Sobel filters have been used in image processing for a long time, ever since its inception back in 1968. The basic idea behind it consists in performing two mathematical convolutions, one for each axis at every point of the original image, minus the edges.

The following implementations are discussed in this report:

- Reference `C#` implementation
- Fast `C#` implementation using unmanaged memory
- Parallel version of the the unmanaged implementation
- `CUDA` implementation
- `OpenCL` implementation

As requested, all the different implementations are integrated into a Visual Studio 2017 solution, it requiers the `CUDA` toolkit to compile the GPU versions of the algorithm.

Implementations

Reference implementation: This was the first version done, it makes use of the `Image` library, with its associated `SetPixel` and `GetPixel` methods. It's by far the worst performing implementation of the algorithm, as the overhead of using the pixel manipulation methods is too great.

Unmanaged memory implementation: Following the reference implementation, this was the next version to be considered. It consists in making use of `LockBits` and manipulating the bytes of the original image directly, without having to rely on `SetPixel` and `GetPixel`. In terms of performace, this version was much faster than the reference implementation.

Parallel unmanaged memory implementation: One of the key aspects of the Sobel filter, is that we can calculate the convolutions for each pixel independently. This is ideal for parallelization, and it's the core idea behind not only this version, but the GPU versions as well. In this case, this version runs on the main CPU, using as many tasks as there are processors available on the system.

`CUDA` and `OpenCL` versions: Both versions were implemented directly using official toolkits and headers, then compiled to `DLL` files, and linked from the main `C#` program. Sadly, there isn't any official support for GPU libraries from within the base `C#` environment, so additional steps need to be taken. While there are multiple projects that aim to integrate GPGPU functionality to `C#`, I decided not to use them. There are two main reasons behind this decision, first and foremost were licensing concerns, as most of them were commercial solutions that need to be licensed before they can be used commercially. The second reason was the complexity of the filter itself, because it's a rather simple filter, I didn't feel it justified having to make use of a third-party solution to simplify its integration to `C#` as it can alredy be easily integrated using an external `DLL` and unmanaged memory.

Comparison

	Reference	Fast Memory	Fast Memory multithreaded	CUDA	OpenCL
1920x1080	15374	685.36	272.53	390.93	272.27
3840x2160	63786	2707.76	796.18	658.14	429.79

Table 1: Time and resolution comparison (time expressed in ms)

The tests were done with an Intel Core-i7-4700MQ and a NVIDIA GeForce 740M, the time was measured using `C# System.Diagnostics.StopWatch` and the final time was obtained averaging 30 time measurements.

Conclusion

As we can see in the comparison, the reference implementation is clearly the worst, and just by using unmanaged memory we can see a gigantic improvement in terms of performance. However, the line becomes blurrier when we start using a multithreaded approach. While all 3 different versions have similar performance, further tests need to be done on different hardware, as the GPU in my computer is a low-end model. This means that there is a very high chance that the same code could perform much better under more powerful hardware.

References

- Feature Detectors - Sobel Edge Detector:
  http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm