Serial and Parallel Sobel Filtering for multimedia applications

Gunay Abdullayeva
Institute of Computer Science
University of Tartu
Email: gunay@ut.ee

***Abstract*—GSteamer contains various plugins to apply filtering to media components and Sobel-Feldman operator is one of the popular filters which is used at the edge detection in computer vision processing. This article is about the serial and parallel implementation of Sobel filter.**

I. Introduction to GStreamer

GStreamer[1] is a pipeline-based multimedia framework to construct graphs of media-handling components. For example, GStreamer can be used to create a system that files can be read in one format, processed and exported in another format.There are lots of supports of GStreamer such as handling media components, audio and video playback,recording, streamimg and editing. The pipeline design is one of main parts in GStreamer to create multimedia applications such as video editors, transcoders, streaming media broadcasters and media players. GStreamer is designed to work with most operating systems such as Linux, Android, Windows, Max OS X, iOS, as well as most BSDs, commercial Unixes, Solaris, and Symbian.Additionally, It provides binary builds for Android, IOS, OSX, Windows.Theres a GStreamer plug-in called gst-plugins-cl to utilize OpenCL within this popular Linux video framework so that an OpenCL kernel can be applied against a video stream. gst-plugins-cl[7] can be used for many things like color conversion, filtering, transforming and etc. Everything is GPU based and can be used on various gst formats. Future work can be creation of pipelines on the GPU. This is (at a first glance) not more than providing elements for pushing/fetching buffers onto/from the gpu and an element to send kernels to the gpu.

GStreamer is written in the C programming language based on GObject and the GLib 2.0 object model. A library written in one programming language can be used in another language if bindings are written. GStreamer provides a range of bindings for different languages such as Python, Vala, C++, Perl, GNU Guile, C and Ruby.

GStreamer[2] handles media by connecting a number of processing elements into a *pipeline*. Elements are provided by a *plug-in* and communicate by means of *pads*. A source pad on one element can be connected to a sink pad on another. Data buffers flow from the source pad to the sink pad if the pipeline is in the playing state.

II. Sobel filter

Sobel filter[3] is utilized in computer vision andimage processing, specifically within edge detection algorithms. Irwin Sobel and Gary Feldman, who created this filter and was owner of the idea of an " Isotropic 3*3 Image Gradient Operator" at a talk at SAIL in 1968. TThe main approach is to get a discrete differentiation operator, calculating an approximation of the gradient of the gradient of the image intensity function. At each point of the image, the Sobel-Feldman operator is equal to either the appropriate gradient vector or the norm of the vector. This operator creates small, separable and integer-valued filter in the horizontal and vertical directions.

The operator uses 3*3 kernels which are to compute horizontal and vertical derivative approximations. This approximations are called $\mathbf{G_x}$ and $\mathbf{G_y}$ which are 2 separate images containing horizontal and vertical approximations respectively. If we define A as the source image, the calculation are as follows:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

where * describes the 2-dimensional signal processing convolution operation which is not matrix multiplications, despite the sign.

The products of averaging and a differentiation kernel can define Sobel kernels. For instance, $\mathbf{G_x}$ can be written as

$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} [+1 \quad 0 \quad -1]$$

The x-coordinate is determined as increasing in the right direction and y-coordinate is determined as increasing in the down direction. The result by horizontal and vertical approximations can be combined to introduce the gradient magnitude as follows:

$$G = \sqrt{G_x^2 + G_y^2}$$

The direction of gradients can be calculated by the following formula.

$$\Theta = \left(\frac{G_y}{G_x}\right)$$

For instance, if we get θ is 0 for a vertical edge, it means that this edge is lighter on the right side.

The Sobel-Feldman operator creates a 2-dimensional map of the gradient at each point. The result is processed as an image, which contains areas with high gradient visible as white lines. The following three images describe how the original image is converted to the simple image which is processed by the Sobel-Feldman operator. The original image is in pgm format which represents a grayscale graphic image.

Figure 1. The original image

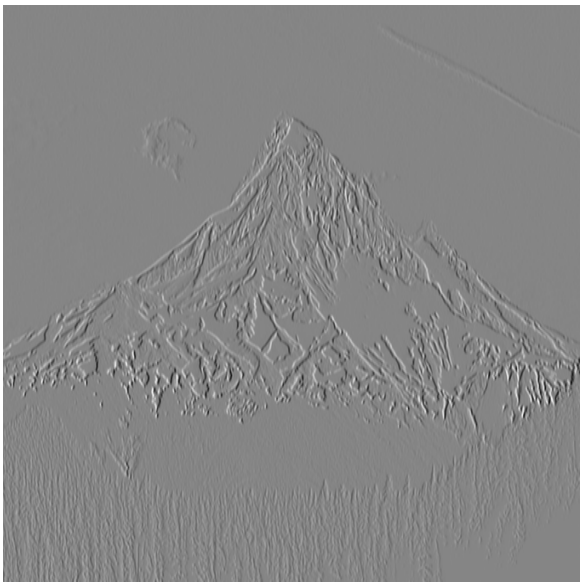

Figure 2. The image contains Sobel-Feldman operator

III. OpenMP

OpenMP (Open Multi-Processing)[4] is an application programming interface(APi) which supports most operating systems such as Linux, macOS and Windows. The purpose of the usage of OpenMP is to create multi-platform shared memory multiprocessing programming in C, C++ and Fortran. It contains a set of library routines, compiler directives and environment variables that affects runtime behavior. The management of OpenMP is implemented by the nonprofit technology consortium *OpenMP Architecture Review Board(or OPenMP ARB)*, jointly determined by a group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, Oracle Corporation and more.

OpenMP uses a portable, scalable model that creates a simple and flexible interface for programmers to develop parallel applications for platforms ranging from the standard desktop computer to the supercomputer. An application structured with the hybrid model of parallel programming can be implemented on a computer cluster using both OpenMP and Message Passing Interface(MPI), such that OpenMP is used for parallelism within a (multicore) node while MPI is used for parallelism between nodes.

There are several possible ways to build parallelization and OpenMP[5] is one of them which is an implementation of multithreading, a method of parallelizing whereby a master thread forks a specified number of slave threads and the system divides a task among them. The runtime environment allocates threads to different processors. Each thread has a unique id assignd to it which can be obtained using function called omp_get_thread_num(). The thread id is integer and the id of the master thread is 0. When the execution of the parallelized code finish, the threads join back into the master thread, which continues onward to the end of the program. By default, all sections of code are executed by threads independently. Using work-sharing constructs, a task is divided among threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be implemented using OpenMP in this way. The runtime environment assigns threads to processors depending on usage, machine load and other factors. The OpenMP methods are included in a header file labelled omp.h in C/C++.

The constructs for thread creation , workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables are the core elements of OpenMP. OpenMP uses pragmas in C/C++. Omp parallel is the pragma which is used to fork additional threads to carry out the work enclosed in the construct in parallel. he following code display simple "Hello, world" example with multiply threads.

```
#include <stdio.h>
int main(void){
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

It is used the following flag to compile the code using GCC 7.3 compiler.

```
gcc -fopenmp hello.c -o hello
```

Output on a computer with two cores, and thus two threads:

Hello, world.

Hello, world. , the output may also be garbled because of the race condition caused from the two threads sharing the standard output.

Hello, wHello, woorld.
rld.

IV. Parallelization

Work-sharing constructs are used to assign independent work to one or several threads.

- *open for* or *omp do* : to separate loop iterations among the threads.
- *sections*: to assign independent consecutive code blocks to various threads.
- *single*: to specify a code block which is executed by only one thread.
- *master*: the difference between single and master is that the code block is executed by the master thread.

As OpenMp is a shared memory programming model, most variables are visible to all threads by default. But depending on the structure, sometimes there is a need to pass values between threads. So in this situation, data sharing attribute clauses are used.

- *shared*: the data is shared within a parallel region in which the data is visible and accessible by all threads simultaneously.
- *private*: The data is open only for the given region and it is used as a temporary variable.
- *reduction*: to join work from all threads after construction.

While I implemented Sobel filtering, I considered both filtering using a serial program and parallel filtering. To generate the transformed image in parallelized version, I defined shared and private values using pragmas[10]. The transformation part of the code is implemented as following:

```
# pragma omp parallel shared
( min, max,image2,weight,image1 )
private ( y, x, i, j,pixel_value )
{
# pragma omp for
for (y = 0; y < y_size2; y++) {
for (x = 0; x < x_size2; x++) {
    image2[y][x] = 0;
}
```

```
}
# pragma omp for
for (y = 1; y < y_size1 - 1; y++) {
for (x = 1; x < x_size1 - 1; x++) {
  pixel_value = 0.0;
for (j = -1; j <= 1; j++) {
for (i = -1; i <= 1; i++) {
  pixel_value += weight[j + 1][i + 1]
  * image1[y + j][x + i];
}
}
  pixel_value = MAX_BRIGHTNESS
  * (pixel_value - min)/(max - min);
  image2[y][x] = (unsigned char)
  pixel_value;
}
}
}
```

I used *omp_get_wtime()* method to measure the implementation time. The elapsed time was 0.015821 in filtering using a serial program case. But in the parallel implementation, it was 0.00739366 in my computer which feature is Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz. So the parallel filtering work faster result than a serial program as expected.

I have also done offloading to GPU and measured the performance CPU and GPU. All the graphics calculations are handled by the integrated Intel HD Graphics 620 of the processor in my current laptop. The GPU can reach up to 1000 MHz which is less than CPU frequency. Using GPU, the elapsed time is 0.00997114 which is higher than CPU version.

I have also tried Jeff Hammond's code[9] which compares some processes with GPU offloading. According to outputs, GPU offloading is faster than others just in one case. The result is as following:

```
for time = 0.000002
OpenMP for time = 0.002933
(error=0.000000)
OpenMP offload for time = 0.005076
(error=0.000000)
_Cilk_for time = 0.000218
(error=0.000000)
offload _Cilk_for time = 0.000008
(error=0.000000)
junk=6201980.842140
for time = 0.000001
OpenMP for time = 0.006271
(error=0.000000)
OpenMP offload for time = 0.012593
(error=0.000000)
_Cilk_for time = 0.000016
(error=0.000000)
offload _Cilk_for time = 0.000006
(error=0.000000)
junk=6201980.927590
for time = 0.000001
OpenMP for time = 0.007580
(error=0.000000)
OpenMP offload for time = 0.000227
(error=0.000000)
_Cilk_for time = 0.000014
(error=0.000000)
offload _Cilk_for time = 0.000005
(error=0.000000)
junk=6201981.009529
for time = 0.000001
OpenMP for time = 0.000011
(error=0.000000)
OpenMP offload for time = 0.000219
(error=0.000000)
_Cilk_for time = 0.000009
(error=0.000000)
offload _Cilk_for time = 0.000005
(error=0.000000)
junk=6201981.026742
for time = 0.000001
OpenMP for time = 0.000010
(error=0.000000)
OpenMP offload for time = 0.000243
(error=0.000000)
_Cilk_for time = 0.000009
(error=0.000000)
offload _Cilk_for time = 0.000005
(error=0.000000)
junk=6201981.028717
for time = 0.000001
OpenMP for time = 0.000010
(error=0.000000)
OpenMP offload for time = 0.000167
(error=0.000000)
_Cilk_for time = 0.000007
(error=0.000000)
offload _Cilk_for time = 0.000004
```

```
(error=0.000000)
junk=6201981.030521
for time = 0.000001
OpenMP for time = 0.000017
(error=0.000000)
OpenMP offload for time = 0.000190
(error=0.000000)
_Cilk_for time = 0.000008
(error=0.000000)
offload _Cilk_for time = 0.000005
(error=0.000000)
junk=6201981.032217
for time = 0.000001
OpenMP for time = 0.000011
(error=0.000000)
OpenMP offload for time = 0.000164
(error=0.000000)
_Cilk_for time = 0.000007
(error=0.000000)
offload _Cilk_for time = 0.000004
(error=0.000000)
junk=6201981.033955
for time = 0.000001
OpenMP for time = 0.000010
(error=0.000000)
OpenMP offload for time = 0.000168
(error=0.000000)
_Cilk_for time = 0.000008
(error=0.000000)
offload _Cilk_for time = 0.000004
(error=0.000000)
junk=6201981.035516
for time = 0.000000
OpenMP for time = 0.000010
(error=0.000000)
OpenMP offload for time = 0.000183
(error=0.000000)
_Cilk_for time = 0.000008
(error=0.000000)
offload _Cilk_for time = 0.000004
(error=0.000000)
junk=6201981.037130
```

V. OpenCL

Open Computing Language (OpenCL)[6] is a framework to write programs which execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. OpenCL defines programming languages for programming these devices and application programming interfaces (APIs) to manage the platform and execute programs on the compute devices.

VI. Filtering in GStreamer

GStreamer uses a plug-in architecture to implement Gstreamer's functionality as shared libraries. GStreamer's base functionality provides several functions to register and load plug-ins. Plug-in libraries is loaded dynamically to support a wide range of input/output drivers, container formats and effects. GStreamer contains lots of plugins but it is possible to create new plugin modules to extend functionality such as parallelized filtering for video streams. To implement parallelized filtering, we can use functionalities of OpenMP or OpenCL. There's a GStreamer plug-in to utilize OpenCL within this popular Linux video framework so that an OpenCL kernel can be applied against a video stream. This support allows for an OpenCL kernel to be run against a video stream in the GStreamer pipeline.

VII. Conclusion

We implemented filtering using a serial program and parallel filtering for the image format and compared the result which parallelized implementation is faster than a serial program as expected. We also tried offloading GPU and it currently works a bit slower than CPU in the current machine. Using OpenMP, it seems possible to create a new Sobel Filtering plugin for Gstreamer, further work will examine this.

References

[1] Wikipedia, *GStreamer*, https://en.wikipedia.org/wiki/GStreamer
[2] GStreamer, *GStreamer*, https://gstreamer.freedesktop.org/documentation
[3] Wikipedia, *Sobel Filter*, https://en.wikipedia.org/wiki/Sobel_operator
[4] Wikipedia, *OpenMP*, https://en.wikipedia.org/wiki/OpenMP
[5] OpenMP, *OpenMP*, http://www.openmp.org/
[6] OpenCL, *OpenCL*, https://en.wikipedia.org/wiki/OpenCL
[7] Fabian Deutsch, *OpenCL Plugin for Gstreamer*, https://github.com/fabiand/gst-plugins-cl
[8] James Beyer, Jeff Larkin, *Targeting GPUs with OpenMP4.5 Device Directives*, http://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf
[9] Jeff Hammond, *Offloading GPU*, https://github.com/jeffhammond/HPCInfo/tree/master/openmp/offload
[10] Gunay Abdullayeva, *Serial and Parallel Implementation of Sobel Filtering*, https://github.com/AbGunay/DS-seminar