Tutorial: Intel® Processor Graphics Optimization

Intel® SDK for OpenCL™ Applications

OpenCL Sample Application Code

Legal Information

Legal Information

Overview

Discover how to optimize OpenCL™ kernels for running on the Intel® Graphics device with the Intel Processor Graphics Optimization sample based on the Sobel Filter algorithm. The optimization tips, described in this tutorial, are also applicable to any other image processing algorithms targeting the Intel Graphics OpenCL device.

About This Tutorial	This tutorial demonstrates an end-to-end workflow you can ultimately apply to your own applications: • Discover the image processing kernel background: algorithm definition and OpenCL implementation • Apply optimizations typical for image processing
Estimated Duration	10-15 minutes.
Learning Objectives	After you complete this tutorial, you should be able to: • Reduce the overhead from the thread spawning by using vector data types instead of scalar • Decrease the number of memory read operations by switching to tile-based algorithm and reusing loaded data as much as possible • Use vload16/vstore16 functions
More Resources	Intel SDK for OpenCL Applications documentation: - Optimization Guide - User's Guide OpenCL Specification Version 1.2 http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

Prerequisites

Before you start with the tutorial, make sure your system meets the requirements.

To build and run the sample application, you need

- Intel Architecture processor with the supported version of the Intel Graphics device. See the list of supported processors in the SDK release notes.
- Intel Graphics Driver available at the OpenCL Drivers and Runtimes for Intel® Architecture page

- Microsoft Visual Studio* 2010 and higher
- Intel® SDK for OpenCL™ Applications 2013 and higher, available at http://software.intel.com/en-us/vcsource/tools/opencl-sdk

You can also use the Intel® VTune™ Amplifier XE 2013 and higher for OpenCL performance analysis (beyond the tools available with the SDK).

Navigation Quick Start

This tutorial includes a sample code that you can compile using the Microsoft Visual Studio Professional 2010 and higher. Find the relevant solution file in **sample root directory** > **ImgprocOpt** subfolder.

Building and Running Code Sample

To build the code sample,

1. Double-click the solution file (`*.sln`) relevant to your Visual Studio IDE version.
2. Select **Build** > **Build Solution**.

Then to run the application,

1. Select a project file in the **Solution Explorer**.
2. Right-click the project and select **Set as StartUp Project**.
3. Press **Ctrl+F5** to run the application.

To run the application in **Debug** mode, press **F5**.

You can also run the sample application using the command-line interface:

1. Run the command prompt.
2. Switch to the directory, where the solution file you used resides.
3. Then go to the directory according to the platform configuration:
    - \Win32 - for Win32 configuration
    - \x64 - for x64 configuration
4. Open the appropriate project configuration (Debug or Release).
5. Run the sample by entering the name of the executable.

Controlling the Sample Application

You can run the sample application with the following command-line options:

Command-Line Options		
Short Form	**Full Form**	**Description**
`-h`	`--help`	Shows command-line options with descriptions.
`-p`	`--platform`	Selects OpenCL platform by ID (0 to n-1, where n is the number of available platforms).
`-t`	`--type`	Selects an OpenCL device to run by *type.* First device of the specified type will be picked. The following options are available: - `all`

		- cpu - gpu - acc - default Combine the -t option with the -p option, which specifies OpenCL platform.
-d	--device number-or-string	Selects an OpenCL device by *number* (or name). This option combines well with the previous ones. For example, if you have multiple devices of the same *type* specified with –t, you can select the particular device to run using -d.
-W	--width <integer>	Sets the width of the processed image (128-8192 pixels).
-H	--height <integer>	Sets the height of the processed image (128-8192 pixels).

Reference (Native) Implementation

Reference implementation is done in `ExecuteSobelReference()` routine of `ImgprocOpt.cpp` file. This is single-threaded code that performs exactly the same Sobel filtering sequence as OpenCL implementation, but uses conventional loop over image in plain C.

Image Processing Algorithm Background

This section describes a typical image-processing algorithm that you can optimize for running on the Intel Graphics OpenCL device.

Image processing algorithms usually operate in a "streaming" mode, which is processing an input image to produce an output one. It is also typically, especially for simple filters, that the number of calculations per pixel is relatively small comparing to the data being fetched.

Translating this into the OpenCL notion, you get millions of tiny work-items with just a few clocks of actual execution, while producing intensive memory traffic to read input pixel and its neighborhood, and to write back the result. So in many cases OpenCL kernel optimizations actually target improving usage of the available memory bandwidth (as modern image resolutions are too high to fit a cache), and amortizing work-items scheduling overheads.

Brief Definition of the Sobel Filter

The Intel Processor Graphics Optimization sample comprises a naïve Sobel kernel - a straightforward OpenCL implementation of the Sobel operator. The operator uses two 3×3 kernels that are convolved with the original picture to calculate approximations of the image derivatives - one for image change in the horizontal dimension, another – for the vertical dimension.

In the scheme below, **A** is the source (input) image, and $\mathbf{G}_x$ and $\mathbf{G}_y$ are two images holding the horizontal and vertical derivatives. Then the computations are performed according to the following scheme:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A \quad \text{and} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * A$$

where <*> denotes the 2-dimensional convolution operation.

The x-coordinate is defined here as increasing in the "right" direction, and the y-coordinate is defined as increasing in the "down" direction (so image origin is the top-left corner).

For each pixel in the image, the magnitude of the resulting gradient approximations is the following:

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

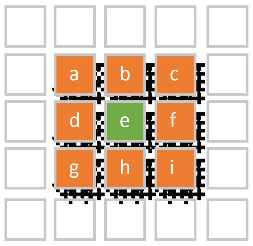
After calculation, the **G** value is written to the corresponding pixel of the destination image.

Naïve OpenCL™ Implementation of the Sobel Filter

The naïve kernel processes each byte of the input grayscale single-channel picture (stored as OpenCL 2D buffer of `uchars`) separately. So each work-item processes a single pixel and performs the following steps in the OpenCL kernel:

- Loads 3x3 unsigned `uchar` pixels from the source (input) buffer.
- Performs horizontal and vertical derivative calculations (described in the previous section).
- Stores single `uchar` value for the result (gradient magnitude) in the output buffer.

9x9 pixels, required to calculate the gradient, are loaded according to the following scheme:



The pixel of interest - for which the gradient value is computed - is marked with green (in center), neighboring pixels are marked with orange.

The ‘a’, ‘b’, ‘c’, and so on values match the variable names in the first version of the Sobel kernel (`Sobel_uchar`). See the `ImgprocOpt.cl` file for details.

Problems of the Naïve Kernel

The naïve kernel exhibits some major performance inefficiencies when executing on the Intel Graphics OpenCL devices. For example, when running over **2048*2048** image on the Intel HD Graphics 5200 device, the kernel has to launch **2048*2048 = 4M** work-items, which (taking SIMD16 into account) translates into more than 200k of thread launches, as the Intel HD Graphics 5200 device features **40**(Execution Units)**x7**(threads per EU)**= 280** hardware threads.

In the particular case of Sobel with a handful of arithmetic per work-item, performance is heavily limited by the number of threads the scheduler can spawn per second. So you need to amortize this thread dispatching overhead by doing more work in each thread.

In addition, the available memory read/write bandwidth is heavily underutilized in this kernel, because memory subsystem is optimized to 128-bit data path (so by requesting single uchar per data read, we waste most of the bandwidth and trigger costly routine to unpack the individual byte). Similarly vector units that are optimized for work with 32-bit values are also underutilized (again, as we operate on uchars). Finally math execution is generally less efficient for integer data types, for example, comparing to floating point data, which also enables multiply-add.

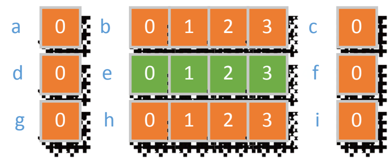
This tutorial describes Intel GPU-specific optimizations that can be applied to overcome those issues. Please refer to Memory Access Considerations and Check-list for OpenCL* Optimizations chapters from Intel SDK for OpenCL Applications Optimization Guide for more detailed explanations.

Optimizing OpenCL Kernel for Image Processing

The following sections describe the optimizations you can utilize to improve OpenCL kernel performance on the Intel Graphics OpenCL device when using image processing algorithms.

Use uchar4 Instead of uchar

You can reduce the overhead from thread spawning by four times, as four times less threads have to be launched for the same image size when using uchar4 instead of uchar. With such optimization, the kernel processed four uchar pixels per work-item in contrast to the single uchar pixel per OpenCL work-item when using the uchar data type. In such case, the source pixels are loaded like the following (numbered according to the indices of uchar4 vector elements):

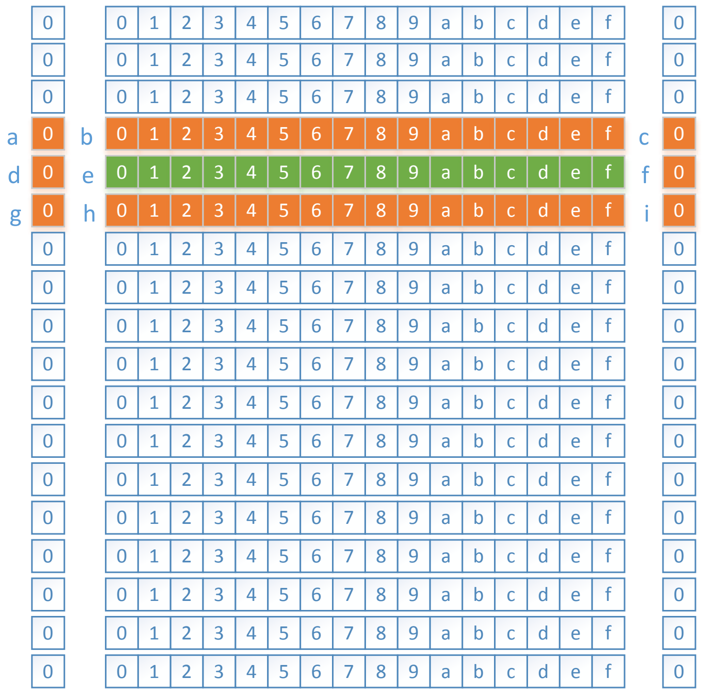


Central pixels are read just once (as a single uchar4), which leads to an increase in read memory bandwidth – this is a noticeable factor for small kernels like Sobel.

This optimization is implemented in the Sobel_uchar4 kernel of the ImgprocOpt.cl file.

Use float16x16 Tiles and Use vload16/vstore16 Functions

You can decrease the number of memory read operations per pixel if you process an image by 16x16 tiles per work-item. In such case 16 pixels in a line are processed using 16-way vector data type, and 16 lines of the tile are processed



one by one in a loop. That means that nine read operations are issued only for the first iteration of the loop, and just three reads for the rest of the block, since upper six elements are reused from the previous iteration. The following scheme demonstrates one loop iteration for the 4th line (loaded pixels shown in color):

Improvements this processing scheme provides:

- Each thread does more work that in the original implementation (without the optimization applied). The number of thread launches and the number of associated overhead is reduced.
- The memory read bandwidth increases, thus less memory-related EUs stalls occur.

Additionally, you can

- Switch the kernel to using the `vload16`/`vstore16` operations, which makes data read and write operations easier for compiler optimizations.
- Use floating point math, which is currently the fastest way to perform calculations on Intel Graphics, since the Intel Graphics EUs have higher floating point compute throughput than integer.

This pack of optimizations is implemented in the `Sobel_uchar16_to_float16_vload_16` kernel of `ImgprocOpt.cl` file.

Sample Limitations to Pay Attention to

For the sake of simplicity, the sample code does not include routines for reading from popular file formats, while concentrating on the optimizations. The default input image is filled by random pixels.

Since the original Sobel Filter operates on image intensities, the OpenCL kernels in the sample also accept the monochrome (single-channel) inputs. Still, the optimizations are applicable to filters for the three- and four-channel images. For example, the uchar4-based version is a natural fit for R8G8B8A8 images, while 16-way processing from the “Kernel Optimizations” section just packs four of RGBA pixels.

Finally, to avoid border conditions when reading neighboring pixel, the image is also padded on the host:

- In vertical direction by 1 extra line on top and bottom
- In horizontal direction by 16 pixel boundaries on both sides - this is the easiest way to satisfy data alignment, required for 16-way vector load operations used in the optimized kernels

Summary

This tutorial demonstrated an end-to-end workflow you can apply to your own application.

Step	Key Tutorial Take-aways
Use vector data types instead of scalar	Intel Graphics OpenCL device can process more data if you improve memory bandwidth.
Use tiles	Decrease the number of memory read operations per pixel via processing an image by 16x16 tiles per work-item.
Use floating point math instead of integers	Use floating point operations to employ higher compute throughput of FP units.
Use `vload16/vstore16` Functions	Use `vload16/vstore16` operations in kernels, which makes data read and write operations easier for compiler optimizations.