Intel® SDK for OpenCL* - Median Filter Sample

User's Guide

Document Number: 325264-003US

Revision: 1.3

World Wide Web: http://www.intel.com

Disclaimer and Legal Information

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

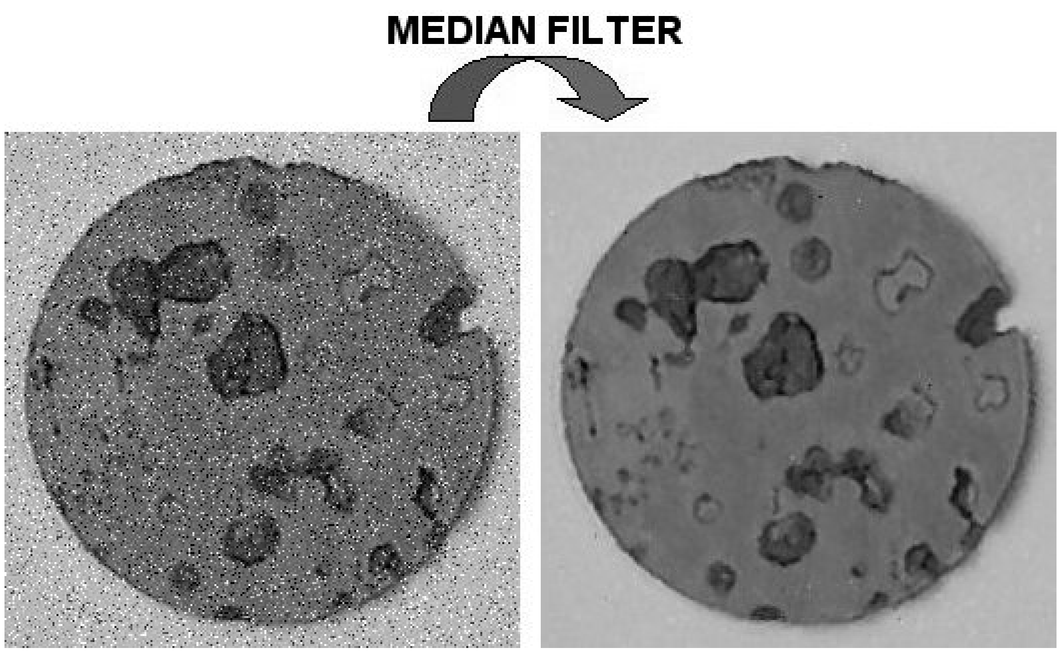
Notice revision #20110804

Contents

About Median Filter Sample

The Median Filter sample demonstrates how to use median filter in OpenCL*. This implementation optimizes filtration process using implicit Single Instruction Multiple Data (SIMD) code vectorization performed by build-in OpenCL* compiler vectorizer. Data-level parallelism of the underlying algorithm results in additional performance gain. The sample improves the performance of the method and reduces data loads.

The following figure illustrates median filtering [1]:



Path

Location	Executable
<INSTALL_DIR>samples\MedianFilter	Win32\Release\MedianFilter.exe – 32-bit executable x64\Release\MedianFilter.exe – 64-bit executable Win32\Debug\MedianFilter.exe – 32-bit debug executable x64\Debug\MedianFilter.exe – 64-bit debug executable

Introduction

Median filter is a non-linear filter that removes noise from an image or a signal. One of the advantages of this method is that it can preserve sharp edges while removing noise. To remove noise, the median filter algorithm processes element patterns of the input image or signal. For each pattern of neighboring elements called *window* or *support,* the algorithm finds the median value that is further used as filtering result for the central element of the window.

Motivation

In general, median filter effect requires a significant number of calculations and color buffer accesses. This sample implementation minimizes color buffer accesses, removes synchronization points, and uses data-level parallelism. This results in significant performance gain and better result quality as compared to applications that use the same filtration technique optimized for traditional GPU architectures. For an example, please see the algorithm implemented for traditional GPUs described in [2].

This sample demonstrates a CPU-optimized implementation of 2D image median filtration, showing how to:

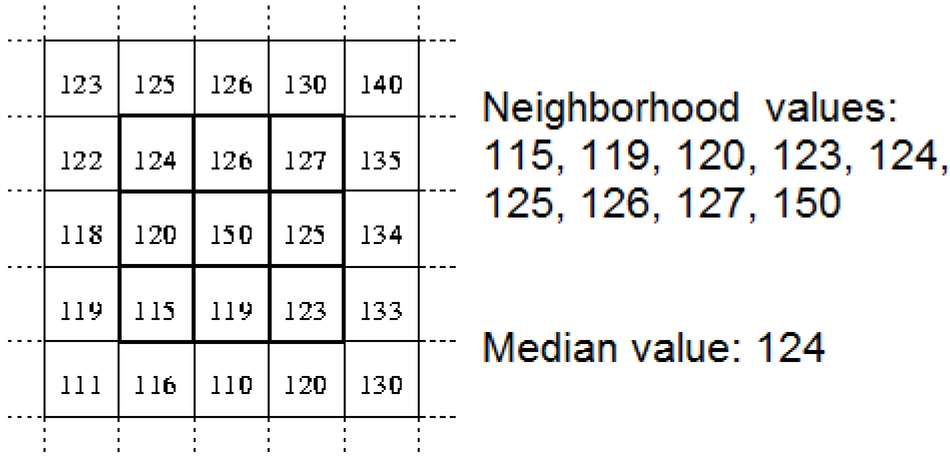
- Implement calculation kernels using OpenCL* C99
- Parallelize the kernels by running several work-groups in parallel
- Organize host-device data exchange with final image storage on the hard drive.

Algorithm

The median filter processes each pixel in the image and compares it to its neighbors to determine whether this pixel can represent the window entries. It replaces the central pixel value with the *median* of the pixel values in the window.

To define the median of a window, sort the entries of the window numerically. For windows with an odd number of entries, the median is the value of the middle entry. For windows with an even number of entries, several options are possible.

The following figure illustrates a sample calculation of the median value for a pixel neighborhood:



This example illustrates a 3×3 square window. As the central pixel value of 150 does not represent the surrounding values well, it is replaced with the median value of 124. Please note that larger windows produce greater smoothing.

The advantage of the median filtering is that unrepresentative pixels in a window cannot have significant effect on the median value. Since the median value must be an actual value of one of the window entries, the median filter does not create new unrealistic pixel values when the filter processes an edge region. Thus, median filtering permits to preserve sharp edges. For details, see [3].

OpenCL* Implementation

This sample applies the following algorithm stages to a 2D image:

- 3x3 pixels patch load
- partial bitonic sorting
- result storage in 4-channel 32-bit integer format.

In this implementation, `MedianFilterBitonic` OpenCL* kernel of `MedianFilter.cl` file uses partial bitonic sorting to perform median filtering. Every input array pixel corresponds to a unique global ID that the kernel uses for their identification. The full median filtering sequence consists of OpenCL* kernel call performed in `ExecuteMedianFilterKernel()` function of `MedianFilter.cpp` file.

This algorithm implementation consists of the Pixels Load and Partial Sort parts.

Pixels Load

This sample uses 32-bit red, green, blue, and alpha (RGBA) pixels, with 8-bit unsigned `char` values representing pixel individual color channel. For further processing, 3x3=9 pixels are preloaded into temporary storage.

Partial Sort

This sample uses an algorithm that operates on 3x3 box-shaped support and performs partial sort to find the fifth sorted value out of 3x3 = 9 values. Partial sort performs 19 MIN and 20 MAX operations to find median value for 3x3 support for each color channel. The algorithm operates with 32-bit unsigned integer values. For details on this algorithm, please see [4] and [5].

Understanding OpenCL* Performance Characteristics

Benefits of Implicit Compiler Vectorization

The kernel structure enables implicit vectorization performed by the Intel® OpenCL* Offline Compiler when work-group size is multiple of four. Consequently, you can achieve ~2x speedup for the current versions of kernel and vectorizer.

Work-group Size Considerations

You can specify any work-group size for the kernel. However, OpenCL* implementation of the median filter achieves peak performance for 1024x1024 two-dimensional array with work-group size ranging from 4 to 64 elements.

Limitations

The current version of the sample requires scalar data types and a fixed kernel size (3x3 median filter).

Future Work and Enhancements

The sample performs all calculations in integer values. Each image pixel consists of one 32-bit integer value representing RGBA image channels. You can improve this sample performance by introducing the following:

- arbitrary window size
- alternative implementation of sorting part (binary search)
- further filter extension (bilateral filter)

Project Structure

This sample project has the following structure:

- `MedianFilter.cpp` - the host code, with OpenCL* initialization and processing functions. This source file also includes native implementation of the algorithm.
- `MedianFilter.cl` – source code of the OpenCL* median filter kernel.
- `MedianFilter.vcproj` - Microsoft Visual Studio* 2008 project file.
- `MedianFilter.vcxproj` - Microsoft Visual Studio* 2010 project file.

APIs Used

This sample uses the following APIs:

- `clKreateKernel`
- `clCreateContextFromType`
- `clGetContextInfo`
- `clCreateCommandQueue`
- `clCreateProgramWithSource`
- `clBuildProgram`
- `clCreateBuffer`
- `clSetKernelArg`
- `clEnqueueNDRangeKernel`
- `clEnqueueReadBuffer`
- `clReleaseMemObject`
- `clReleaseKernel`
- `clReleaseProgram`
- `clReleaseCommandQueue`
- `clReleaseContext.`

Reference (Native) Implementation

Reference implementation is done in `ExecuteMedianFilterReference()` routine of `MedianFilter.cpp` file. This is single-threaded code that performs exactly the same median filtering sequence as OpenCL* implementation, but uses conventional C nested loop.

Controlling the Sample

The sample executable is a console application. You can set the input array size and choose device using command line arguments.

If you do not specify the array size, the sample uses the default value of 1024x1024=1048576 items.

- **--**h command line argument prints help information;
- `-h <height>` command line argument setups input array size height;
- `-w <width>` command line argument setups input array size width;
- -g run sample on the Intel® Processor Graphics device.

References

[1] http://tracer.lcc.uma.es/problems/mfp/mfp.html

[2] http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/OpenCL/website/samples.html

[3] http://homepages.inf.ed.ac.uk/rbf/HIPR2/median.htm

[4] http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm

[5] Frederick M. Waltz, Ralf Hack, and Bruce G. Batchelor. Fast, efficient algorithms for 3x3 ranked filters using finite-state machines. http://www.engin.umd.umich.edu/~jwvm/ece581/18_RankedF.pdf