Fastvideo Image & Video Processing SDK

TECHNICAL MANUAL

v.0.13.1

Fastvideo LLC

Academic B.M. Pontekorvo Str. 6-103, Dubna, Moscow Region 141986, Russia

Phone: +7 (495) 542-04-49

http://www.fastcompression.com

http://www.fastcinemadng.com

info@fastcompression.com

Table of Contents

Introduction....11
About this manual....11
About FASTVIDEO....11
Contact FASTVIDEO....11
Conformity....11
Helpful Links....12
FASTVIDEO Image & Video Processing SDK....13
Input Data Formats....13
RAW Data Unpacking and Transforms....13
Dark Frame Subtraction....13
Flat Field Correction (Shading Correction)....13
RGB coefficients for RAW data....14
Raw Curve as 1D LUT transform....14
Temporal Raw Denoiser (under development)....14
Spatial Raw Denoiser together with Splitter and Merger....14
Median Filter....14
Demosaic (Debayer)....14
Color Surface Converter....15
Spatial Denoiser for luma and chroma....15
Color Correction....15
1D LUTs....15
3D LUTs for RGB and HSV....15
Crop....15
Rotate....15
Resize....15
Remap....16
Sharp....16
Histogram....16
Parade (Waveform monitor)....16
OpenGL Output....16
CUDA Streams support....16
JPEG Compression and Decompression....17
JPEG2000 Encoder and Decoder....18

RAW Bayer Codec....20
Operation....21
Software Requirements....21
Hardware Requirements....21
System Configuration....21
Supported Image and Video Formats....22
IP protection....22
Technical Overview....22
GPU Test for Windows....23
Quick Start....23
Installation....23
Programming components....24
Library of components....26
Pipeline....27
Import and Export Adapters....29
Pipeline Surface Format....31
Pipeline Split and Merge....36
Component recreation....37
Threads, Streams and Performance....38
Debayer....40
Spatial Denoiser....41
JPEG Load and Store functions....42
JPEG Encoder/Decoder....44
JPEG CPU Decoder....46
Timer....47
MJPEG Reader/Writer....48
Affine Transforms....49
Crop component....50
Image Filter component....51
Gaussian sharpen filter....51
Median filter....51
MAD (multiply and add)....51
Bayer Black Shift....51
White Balance....51
Base Color Correction....52
Color Saturation....52

Tone Curve....52
LUT....52
RGB 3D LUT....54
HSV 2D/3D LUT....54
JPEG2000 Encoder/Decoder....63
Sample Applications....68
How to create your own applications with that SDK....69
Demo Applications....70
Fastvideo SDK API....71
Statuses....71
Master SDK and secondary library initialization....73
fastInit....73
fastGetSdkParametersHandle....73
fastLibraryInit....73
Trace and Auxiliary functions....74
fastGetDeviceSurfaceBufferInfo....74
fastEnableInterfaceSynchronization....74
fastTraceCreate....74
fastTraceClose....75
fastTraceEnableFlush....75
Memory management functions....76
fastMalloc....76
fastFree....76
fastGetDevices....76
Pipeline import functions....77
fastImportFromHostCreate....77
fastImportFromHostGetAllocatedGpuMemorySize....77
fastImportFromHostCopy....77
fastImportFromDeviceDestroy....79
Pipeline export functions....81
fastExportToHostCreate....81
fastExportToHostGetAllocatedGpuMemorySize....81
fastExportToHostChangeSrcBuffer....81
fastExportToHostCopy....82
fastExportToHostDestroy....82
fastExportToDeviceCreate....82

fastExportToDeviceCopy.....83
fastExportToDeviceGetAllocatedGpuMemorySize.....83
fastExportToDeviceChangeSrcBuffer.....83
fastExportToDeviceDestroy.....84
Debayer functions.....85
fastDebayerCreate.....85
fastDebayerGetAllocatedGpuMemorySize.....85
fastDebayerChangeSrcBuffer.....86
fastDebayerTransform.....86
fastDebayerDestroy.....86
Denoise functions.....88
fastDenoiseCreate.....88
fastDenoiseGetAllocatedGpuMemorySize.....89
fastDenoiseChangeSrcBuffer.....89
fastDenoiseTransform.....89
fastDenoiseTransformBayerPlanes.....90
fastDenoiseDestroy.....91
JPEG Encoder functions.....92
fastJpegEncoderCreate.....92
fastJpegEncoderGetAllocatedGpuMemorySize.....92
fastJpegEncoderChangeSrcBuffer.....93
fastJpegEncode.....93
fastJpegEncodeAsync.....93
fastJpegEncodeWithQuantTable.....94
fastJpegEncodeAsyncWithQuantTable.....94
fastJpegEncoderDestroy.....94
JPEG Decoder functions.....95
fastJpegDecoderCreate.....95
fastJpegDecoderGetAllocatedGpuMemorySize.....95
fastJpegDecode.....96
fastJpegDecoderDestroy.....96
JPEG CPU Decoder functions.....97
fastJpegCpuDecoderCreate.....97
fastJpegCpuDecoderGetAllocatedGpuMemorySize.....97
fastJpegCpuDecode.....98
fastJpegDecoderDestroy.....98

JPEG I/O functions....99
fastJfifLoadFromFile....99
fastJfifHeaderLoadFromFile....99
fastJfifBytestreamLoadFromFile....100
fastJfifLoadFromMemory....100
fastJfifLoadHeaderFromMemory....100
fastJfifLoadBytestreamFromMemory....101
fastJfifStoreToFile....101
fastJfifStoreToMemory....102
Affine functions....103
fastAffineCreate....103
fastAffineGetAllocatedGpuMemorySize....103
fastAffineChangeSrcBuffer....104
fastAffineTransform....104
fastAffineDestroy....104
Crop functions....106
fastCropCreate....106
fastCropGetAllocatedGpuMemorySize....106
fastCropChangeSrcBuffer....107
fastCropTransform....107
fastCropDestroy....108
Image Filter functions....109
fastImageFilterCreate....109
fastImageFiltersGetAllocatedGpuMemorySize....112
fastImageFiltersChangeSrcBuffer....113
fastImageFiltersTransform....113
fastImageFiltersDestroy....114
Resize functions....115
fastResizerCreate....115
fastResizerGetAllocatedGpuMemorySize....116
fastResizerChangeSrcBuffer....116
fastResizerTransform....116
fastResizerTransformStretch....117
fastResizerDestroy....118
Bayer Splitter functions....119
fastBayerSplitterCreate....119

fastBayerSplitterGetAllocatedGpuMemorySize....119
fastBayerSplitterChangeSrcBuffer....120
fastBayerSplitterSplit....120
fastBayerSplitterDestroy....120
Bayer Merger functions....122
fastBayerMergerCreate....122
fastBayerMergerGetAllocatedGpuMemorySize....122
fastBayerMergerChangeSrcBuffer....123
fastBayerMergerMerge....123
fastBayerMergerDestroy....123
Timer functions....125
fastGpuTimerCreate....125
fastGpuTimerStart....125
fastGpuTimerStop....125
fastGpuTimerGetTime....125
fastGpuTimerDestroy....126
Mux functions....127
fastMuxCreate....127
fastMuxSelect....127
fastMuxDestroy....127
SDI import and export....129
FastSDIImportFromHostCreate / fastSDIImportFromDeviceCreate....129
fastSDIExportToHostCreate / fastSDIExportToDeviceCreate....129
fastSDIImportFromHostGetAllocatedGpuMemorySize....130
fastSDIImportFromDeviceGetAllocatedGpuMemorySize....130
fastSDIExportToHostGetAllocatedGpuMemorySize....131
fastSDIExportToDeviceGetAllocatedGpuMemorySize....131
fastSDIImportFromHostCopy / fastSDIImportFromDeviceCopy....131
fastSDIExportToHostCopy / fastSDIExportToDeviceCopy....132
fastSDIImportToHostCopy3 / fastSDIImportToDeviceCopy3....132
fastSDIExportToHostCopy3 / fastSDIExportToDeviceCopy3....133
fastSDIImportFromHostDestroy / fastSDIImportFromDeviceDestroy....134
fastSDIExportToHostDestroy / fastSDIExportToDeviceDestroy....134
Surface converter....135
fastSurfaceConverterCreate....135
fastSurfaceConverterGetAllocatedGpuMemorySize....136

fastSurfaceConverterChangeSrcBuffer....136
fastSurfaceConverterTransform....137
fastSurfaceConverterDestroy....137
Histogram functions....138
fastHistogramCreate....138
fastHistogramGetAllocatedGpuMemorySize....139
fastHistogramChangeSrcBuffer....139
fastHistogramCalculate....139
fastHistogramDestroy....140
JPEG2000 Encoder functions....142
fastEncoderJ2kCreate....142
fastEncoderJ2kGetAllocatedGpuMemorySize....142
fastEncoderJ2kTransform....143
fastEncoderJ2kFreeSlotsInBatch....143
fastEncoderJ2kUnprocessedImagesCount....144
fastEncoderJ2kAddImageToBatch....144
fastEncoderJ2kTransformBatch....144
fastEncoderJ2kGetNextEncodedImage....145
fastEncoderJ2kDestroy....146
JPEG2000 Decoder functions....147
fastDecoderJ2kCreate....147
fastDecoderJ2kGetAllocatedGpuMemorySize....147
fastDecoderJ2kTransform....148
fastDecoderJ2kFreeSlotsInBatch....148
fastDecoderJ2kUnprocessedImagesCount....148
fastDecoderJ2kAddImageToBatch....149
fastDecoderJ2kTransformBatch....149
fastDecoderJ2kGetNextDecodedImage....150
fastDecoderJ2kDestroy....150
NppFilter functions....151
fastNppFilterCreate....151
fastNppFilterGetAllocatedGpuMemorySize....152
fastNppFilterChangeSrcBuffer....152
fastNppFilterFiltersTransform....153
fastNppFilterDestroy....153
NppGeometry functions....154

fastNppGeometryCreate....154
fastNppGeometryGetAllocatedGpuMemorySize....155
fastNppGeometryChangeSrcBuffer....156
fastNppGeometryTransform....156
fastNppGeometryDestroy....157
NppResize functions....158
fastNppResizeCreate....158
fastNppResizeGetAllocatedGpuMemorySize....158
fastNppResizeChangeSrcBuffer....159
fastNppResizeTransform....159
fastNppResizeTransformStretch....160
fastNppResizeDestroy....160
NppRotate functions....162
fastNppRotateCreate....162
fastNppRotateGetAllocatedGpuMemorySize....162
fastNppRotateChangeSrcBuffer....162
fastNppRotateGetRotateQuad....163
fastNppRotateTransform....163
fastNppRotateDestroy....164
Source Code for Sample Applications....165
Other Sample Applications....165
Examples of command line for DebayerSample application....165
Examples of command line for JpegSample application....165
Example of command line for DebayerJpegSample application....166
Examples of command line for SDIConverterSample application....166
Example of command line for PhotoHostingSample application....167
Troubleshooting....167
Disclaimer of Warranty....167
List of Trademarks....167
Application Notes....169

Introduction

About this manual

We sincerely hope that this manual can answer your questions, but should you have any further questions or if you wish to claim, please contact your local dealer or refer to the FASTVIDEO support on our website

The purpose of this document is to provide a description of the FASTVIDEO Image & Video Processing SDK and to describe the correct way to install related software and drivers and run it successfully. Please read this manual thoroughly before getting started the software for the first time. Please follow all instructions and observe the warnings.

This document is subject to change without notice.

About FASTVIDEO

FASTVIDEO is one of worldwide leaders in the field of high performance GPU image and video processing solutions. Based in Russia, FASTVIDEO offers their GPU software solutions worldwide. In close collaboration with customers FASTVIDEO has developed a broad spectrum of technologies and cutting-edge, highly competitive products.

FASTVIDEO software solutions find use in machine vision cameras, high speed imaging applications, broadcasting, heavy-loaded internet services, 3D and VR, other applications for image and video processing. The broad spectrum of solutions also includes medical applications and surveillance.

Contact FASTVIDEO

FASTVIDEO is a worldwide operating company

Headquarters: Academic B.M. Pontekorvo Str. 6-103, Dubna, Moscow Region 141986, Russia

Web: www.fastcompression.com

Email: info@fastcompression.com

Conformity

FASTVIDEO Image & Video Processing SDK has been tested at the following:

- Windows-7/8/10 (64-bit)
- Linux Ubuntu, OpenSUSE 12, SLC-6.5, CentOS, etc.
- NVIDIA GPUs with Compute Capability >= 3.0
- CUDA-9.2 for server, desktop, laptop and mobile GPUs
- NVIDIA drivers 398.36 or later
- All sample applications are prepared for MSVS 2015 and gcc

Helpful Links

- FASTVIDEO Homepage http://www.fastcompression.com
- NVIDIA CUDA and GPU drivers download: http://www.nvidia.com/Download/index.aspx
- JPEG Standard: http://www.w3.org/Graphics/JPEG/itu-t81.pdf
- Full Image Processing Pipeline on the GPU: http://on-demand.gputechconf.com/gtc/2014/presentations/S4151-full-gpu-image-processing-pipeline-camera-apps.pdf

FASTVIDEO Image & Video Processing SDK

Input Data Formats

Output formats for cameras could differ significantly. That's why we have created input module which handles various image formats acquired from cameras.

RAW Data Unpacking and Transforms

RAW data from cameras could have 8/10/12/14/16 bits per pixel. Quite often input data are packed and we need to do unpacking. For example, 12-bit data are usually transferred as 2 pixels in 3 Bytes. Unpacking module can extract RAW data from bytestream and convert it to standard 8-bit or 16-bit format.

Dark Frame Subtraction

For many image sensors this is important operation mode. One can either subtract one value from each pixel of the image or subtract the whole image (dark frame). For CMOS image sensors that mode could be called FPN (Fixed Pattern Noise) subtraction.

Flat Field Correction (Shading Correction)

It's well known that image intensity is decreasing near the edges of image sensor. It happens due to differences in amount of light falling on each pixel. To take that into account one have to multiply pixel values to correction coefficients.

RGB coefficients for RAW data

Four coefficients cR, cG1, cG2 and cB are necessary to balance different RAW color channels before demosaicing. Usually bayer RAW image looks green, which is not good for further demosaicing. That simple image color balancing is widely used for color cameras.

Raw Curve as 1D LUT transform

Standard or custom 1D look-up tables (LUT) could be applied to each channel of RAW data before debayering. Master curve has the same form all three channels. There are also individual curves for each channel of Bayer pattern.

Temporal Raw Denoiser (under development)

There is an opportunity to suppress temporal noise for image series (not applicable to single image). That algorithm is based on correlation of adjacent frames to check motion detection and to remove temporal noise.

Spatial Raw Denoiser together with Splitter and Merger

It is possible to suppress spatial noise for each channel of Raw Bayer image before applying demosaicing. One can split Raw Bayer image into 4 planes according to Bayer pattern and then remove spatial noise for each plane separately. Finally we need to merge these planes to get image with Bayer pattern. Parameters: Bayer pattern, wavelet name, threshold function, number of DWT resolution levels, array of denoising thresholds for each wavelet band.

Median Filter

Median filter can suppress impulse noise on images. GPU-based implementation of Median filter with window 3x3 is working very fast and efficient.

Demosaic (Debayer)

Demosaicing is a transformation of a 8/16-bit Raw Bayer image into the conventional 24/48-bit RGB format. Demosaicing is required because digital cameras normally don’t produce ready-to-go RGB images, instead they store visual information as a set of separate R, B, and G values derived from the image sensor of the camera and the actual color of a pixel in that array is determined by interpolating nearby pixel colors. Demosaicing software does the following:

- Algorithms: HQLI, DFPD, MG
- Converts 8/16-bit Raw Bayer images to 24/48-bit BMP/PPM
- Can do that really fast on NVIDIA GPU, much faster than on any CPU
- All Bayer mosaic patterns for input data are supported (RGGB, BGGR, GBRG, GRBG)
- High quality image demosaicing
- Significant moire suppression
- Immediate and precise time and performance measurements

Color Surface Converter

Color Surface Converter is intended to transform RGB data to separate planes of R, G and B. It also performs bit depth transform for the whole image.

Spatial Denoiser for luma and chroma

There is an opportunity to suppress luma and chroma spatial noise for each color image. One have to set algorithm type, threshold function, number of DWT resolution levels, and array of denoising thresholds for luma and chroma channels for each wavelet band.

Color Correction

That functionality is a must for color correction procedure. To get good color reproduction, we need to specify or calculate color correction matrix and apply it to the image.

1D LUTs

Standard or custom 1D look-up tables (LUT) could be utilized in the software. Gamma transform is also implemented via 1D LUT.

3D LUTs for RGB and HSV

Standard or custom 3D look-up tables (LUT) in .cube format could be utilized for color grading.

Crop

To cut an image with specified dimensions we utilize Crop function. We offer high performance implementation of Crop algorithm on GPU.

Rotate

To rotate or flip/flop an image with specified dimensions we utilize Rotate function. We offer high performance algorithm for real-time image rotation (90/180/270 degrees) and flip/flop on GPU. We've also implemented rotation to an arbitrary angle.

Resize

Quite often one has to show pictures with resolutions which are different from monitor or window resolution. Here comes a task of image resize. We offer high quality algorithm for real-time image resize on GPU (both downsampling and upsampling).

Remap

We offer remapping algorithm to perform the following actions: rotation to an arbitrary angle, affine and perspective transforms, various projections, arbitrary image mapping on GPU.

Sharp

To enhance image quality we have implemented Sharp function. We offer very fast and high quality algorithm for real-time image sharpening on GPU.

Histogram

Various histograms could be calculated for 8/16-bit grayscale images and for 24-bit color images to evaluate distribution of grayscale/color component values of particular image. There is an option to get a histogram for each color channel of RAW or DNG data as well.

Parade (Waveform monitor)

Parade is a set of histograms for vertical rows of each RGB component of color image. This is useful representation of color image data to evaluate image quality, white balance and color distribution.

OpenGL Output

After GPU image processing one has to show a picture on the screen. As soon as all computations on GPU are done, then OpenGL is the fastest way to show the image on monitor because all data are already in GPU memory.

CUDA Streams support

CUDA Streams technology offers an opportunity to perform computations on GPU at the same time with data copy to or from GPU. This is the way to improve total performance due to overlap between copy and computations.

JPEG Compression and Decompression

This is CUDA implementation of JPEG Baseline algorithm for compression and decompression on NVIDIA GPUs according to JPEG Standard.

Key Features

- Implementation is 100% compliant with JPEG Baseline standard
- Baseline JPEG compression and decompression for grayscale (8-bit) and color (24-bit) images with arbitrary width and height
- Extremely fast lossy image encoding and decoding with variable compression ratio
- Chroma subsampling modes: 4:4:4, 4:2:2, 4:2:0
- Minimum input image size: 1x1 for all chroma subsampling modes
- Maximum input image size is 16,000 x 16,000 or more (optional)
- JPEG image quality in the range from 1 to 100%
- Option to read/write any EXIF section
- Standard data input: 8/24-bit images from RAM/HDD/RAID/SSD/GPU
- Optional data input/output: 12/36-bit images from RAM/HDD/RAID/SSD/GPU
- Optional parameters: quantization tables
- Data output: final compressed/uncompressed image in RAM/HDD/RAID/SSD/GPU
- Continuous data mode (input one image after another)
- Standard set of computations for parallel implementation of Baseline JPEG compression and decompression
  - JPEG Encoding: Input data parsing, Color Transform, Level shift, 2D DCT, Quantization, Zig-zag, AC/DC, DPCM, RLE, Huffman, Byte stuffing, JFIF formatting
  - JPEG Decoding: JFIF parsing, Restart Marker search, Inverse Huffman, Inverse RLE, Inverse DPCM, AC/DC, Inverse Zig-zag, Inverse Quantization, IDCT, Inverse Level shift, Inverse Color Transform, Output formatting
- Optimized for the latest NVIDIA GPUs (Kepler, Maxwell and Pascal)
- Compatible with Windows-7/8/10, Linux Ubuntu, OpenSUSE, CentOS, L4T

We have succeeded to make all stages of JPEG algorithm parallel including entropy encoding and decoding. There was a widespread opinion that Huffman algorithm could be only serial and that's why this is a bottleneck of JPEG performance. In our solution Huffman coding is not a bottleneck anymore because it's fully parallel and is performed on GPU. We don't off-load anything from GPU to CPU to make JPEG codec faster. CUDA JPEG codec is extremely fast and is functioning completely on GPU.

JPEG2000 Encoder and Decoder

This is CUDA-based implementation of JPEG2000 algorithm for image encoding and decoding according to JPEG2000 Standard.

Key Features

- Implementation is compliant with JPEG2000 standard
- JPEG2000 encoding and decoding for grayscale and color images with arbitrary width and height
- Lossy (CDF 9/7) and lossless (CDF 5/3) image compression and decompression
- Bit depth: 8–16 bits per channel (optionally up to 24 bits per channel)
- Color spaces: RGB, YCbCr, XYZ
- Number of DWT resolution levels: 1–12
- Code-block size 16×16, 32×32 or 64×64
- Subsampling 4:4:4, 4:2:2, 4:2:0
- Image quality in the range of 1–100 (float value)
- Rate control option to set image compression ratio
- Support of tiling
- Data input: images from HDD/RAID/SSD/CPU/GPU
- Data output: final compressed or uncompressed image in HDD/RAID/SSD/CPU/GPU
- Modes of operation
  - Single image mode (minimum latency)
  - Batch mode (streaming processing with max throughput)
  - Multiple tile mode for big images
  - Other fast modes: massive parallelism with high performance and slightly less compression (option)
- CUDA Streams support to offer maximum performance
- Standard set of computations for JPEG2000 encoding and decoding on CUDA
  - **CUDA JPEG2000 Encoder**
    - Input data parsing
    - Color Transform (ICT/RCT)
    - 2D DWT with CDF 9/7 or 5/3
    - Quantization
    - EBCOT Tier-1 (Context modeler and Bit-plane MQ-Coder)
    - PCRD (Post Compression Rate Distortion)
    - Tier-2 (Tag Tree Coding)
    - Output formatting
  - **CUDA JPEG2000 Decoder**
    - Input parsing
    - Tag Tree Decoding
    - Binary Decoder
    - Inverse Quantization
    - IDWT
    - Inverse Color Transform

    - Output formatting

- Performance is much better than CPU-based JPEG2000 codecs JasPer, JJ2000, OpenJPEG, Kakadu, etc.
- Performance is significantly higher than CUDA-based JPEG2000 encoders CUJ2K and GPU JPEG2K

Please note that last stages of JPEG2000 encoding (Tier-2 and output formatting) and decoding algorithms are carried out on CPU, that's why powerful GPU and CPU are necessary to get maximum performance for J2K encoding and decoding.

RAW Bayer Codec

Performance of image processing in camera applications could be much higher is we apply image compression for raw data without using Debayer. That approach is usually called Raw Bayer Compression. Right after image acquisition, we can create four planes according to colors of Bayer pattern (RGGB). Then we can do JPEG/JPEG2000 data compression for each plane with high quality. It could be done very fast and without introducing significant image artifacts. In this way, we can temporary exclude debayer from image processing pipeline and increase performance for realtime applications.

<u>Pipeline description for image acquisition stage</u>

- Image acquisition and unpacking of raw data
- Image preprocessing: dark frame (black offset) and FFC
- Decomposition and alignment of real bayer pattern to 4 separate color planes (Split)
- JPEG compression (quality > 90%), optionally JPEG2000 (CDF 5/3 or 9/7)
- Data storage to SSD/HDD/RAID

<u>Image processing and visualizing for compressed RAW</u>

- Bayer image decompression (JPEG decoding)
- Image composition with real bayer pattern from 4 color planes
- White balance
- Debayer
- Denoiser
- Color correction
- LUT (gamma)
- Crop/Rotate/Resize/Sharp
- OpenGL output
- Optional output MJPEG compression to AVI

Operation

Software Requirements

FASTVIDEO Image & Video Processing SDK is compatible with the following OS (64-bit):

- Windows 7 SP1
- Windows 8
- Windows 10
- Linux Ubuntu
- Linux OpenSUSE 12.3
- Linux SLC-6.5
- Linux CentOS
- Linux4Tegra

Hardware Requirements

FASTVIDEO Image & Video Processing SDK is compatible with NVIDIA GPUs with Compute Capability 3.0 and more. These are Kepler, Maxwell and Pascal GPUs of GeForce family and Quadro/Tesla GPUs as well. We also support mobile Tegra GPU K1, X1 and X2 (Linux4Tegra).

To work on laptop, one have to download CUDA, which is designed for laptop GPUs. While working, laptop should be connected to mains to offer maximum performance. If you try to work from internal battery of laptop, the performance will degrade.

System Configuration

Minimum system configuration

For a basic operation of FASTVIDEO SDK the following minimum system configuration is required. Please note that bandwidth and processing performance are tied to the hardware configuration and the minimum hardware configuration could lead to reduced bandwidth and limited performance.

- CPU: Intel Core-i3 or better
- RAM: 4 GB RAM or more
- HDD/SSD: 200 MB of free disc space
- Video: NVIDIA GPU with Compute Capability >=3.0 (Kepler, Maxwell or Pascal), GPU memory 2 GB or more
- Ports: Motherboard with PCIe x16 Gen2 or Gen3 slot for GPU

One can also use FASTVIDEO Image & Video Processing SDK with a laptop which has NVIDIA GeForce GT series GPU with Compute Capability >=3.0.

Recommended standard system configuration

For good processing performance and bandwidth we recommend to use the following system configuration.

- CPU: Intel Core-i7, 4770K or better
- RAM: 8 GB RAM or more
- HDD/SSD: 1000 MB of free disc space
- Video: NVIDIA GeForce GTX 1080TI or Quadro 6000M, GPU memory 8-12 GB
- Ports: Motherboard with PCI-Express x16 Gen3 slot for GPU

For maximum throughput in addition to top-level GPUs we recommend to use motherboards with PCI-Express x16 Gen3 support and Intel IvyBridge CPUs to benefit from PCIe-3.0 technology.

To work with PCIE/Thunderbolt cameras or PCI-Express frame grabbers (which are custom for CameraLink, CoaxExpress and 10GigE high speed and high resolution cameras) we recommend to use motherboards with >40 PCIe lanes (chipset 2011 or better).

Recommended professional system configuration

- 1U / 2 x 2609v4 / 32 GB DDR4 ECC REG / 960 GB SSD Intel S4600 2.5"
- NVIDIA Tesla P40, GPU memory 24 GB

Supported Image and Video Formats

Standard image formats: BMP, PGM, PPM, YUV, YCbCr, JPG, J2K, JP2

Optional image formats: DNG, CinemaDNG, OpenGL texture or PBO, RAW

Video format for MJPEG codec: AVI

IP protection

Demo version of SDK is protected with limitation of time usage and with watermarks. We could also supply hardware dongles which allow to work without the above mentioned restrictions. Demo version has almost the same performance and one can utilize demo SDK to build your own demo application or integrate SDK into your software and test it in corresponding environment.

Apart from hardware dongles we also have other licensing options.

Technical Overview

- C language API provides total control over image processing functions in SDK
- API in C/C++
- Thread-safety allows multi-threaded client applications
- Image data input/output provided by buffers offers maximum flexibility
- Both static and dynamic libraries are available
- Designed for CUDA-9.2 (64-bit)
- Compatible with the latest NVIDIA GPUs (Kepler, Maxwell and Pascal)
- Available on multiple platforms including Win-7/8/10, Linux, L4T

GPU Test for Windows

To check available GPUs on PC, one could download freeware software TechPowerUp GPU-Z from the following link: https://www.techpowerup.com/gpuz/

GPU-Z is a lightweight utility designed to provide you with all information about your graphics card and GPU. You can also find out the info about PCI-Express connection to PC.

Quick Start

Download SDK, create a directory and unzip the Software Development Kit (SDK) into the directory.

Run sample applications without any parameters. This will display all parameters.

In `sdkReadme.txt` file you can find general info about the software.

In `release_log.txt` file you can see all changes that we've done.

Installation

1. Read the Standard License Agreement (SLA).
2. Download the latest drivers for your NVIDIA GPU from the following link: http://www.nvidia.com/Download/index.aspx
3. Download the latest Senselock dongle drivers here: http://senselock.ru/files/senselock_windows_2.52.1.0.rar
4. Start the installer. Be sure that you have administrator privileges or start the Installer with administrator rights (right click and select “Run as Administrator”).
5. Unzip FASTVIDEO Image & Video Processing SDK

Programming components

This is brief list of components from **Fastvideo Image & Video Processing SDK**:

- ImportAdapter and ExportAdapter
- Raw data unpacking
- Serial Digital Interface (SDI) importer/exporter
- Image Preprocessing (dark frame subtraction, vignetting removal, etc.)
- Raw per-channel coefficients for RGGB
- Raw Tone Curve (master and per-channel)
- Temporal Raw Denoiser (under development)
- Raw Bayer Splitter and Merger
- Raw Bayer Codec
- Median filter
- Spatial Raw Denoiser
- Debayer (HQLI, DFPD and MG algorithms)
- Spatial Denoiser for luma and chroma
- Color Surface Converter (RGB to R, G, B; bit depth converter)
- RGB to Gray
- Color Transforms (RGB, HSV, YCbCr)
- Color Correction with matrix profile
- 1D LUT and Gamma
- 3D LUT for HSV and RGB
- Affine Transforms (Rotate, Flip, Flop)
- Crop
- Resize
- Remap (rotation to arbitrary angle, perspective transforms, etc.)

- Sharp
- Histogram
- Parade (Waveform monitor)
- OpenGL output
- JPEG Encoder/Decoder, MJPEG Reader/Writer
- JPEG2000 Encoder/Decoder, MXF Reader/Writer
- H.264 Encoder
- CUDA Streams support
- Trace
- Time, performance and quality measurements
- Multiplexor
- Sample applications with source codes

All functions and types are declared in `fastvideo_sdk.h`

Library of components

There are multiple libraries in that SDK. FASTVIDEO SDK is a primary library of SDK. Other libraries of components are the secondary libraries. Secondary library contain optional, experimental or wrapper components. Wrapper component wraps some external library to use it together with FASTVIDEO components.

Primary library has to be initialized first. After that any secondary library can be initialized. Primary library contains global options that affects some functionality of all components. Global options have to be passed to the secondary library to allow primary library control and manage secondary library.

Primary library of SDK is initialized by `fastInit` function call. It takes affinity mask of GPU devices and attach CUDA context to defined GPU. If SDK will be used with OpenGL, integration `openGlMode` flag in `fastInit` has to be `true`. In this case SDK calls `cudaGLSetGLDevice` to initialize device instead of `cudaSetDevice`. In all other cases `openGlMode` flag has to be `false`.

After primary library initialization function `fastGetSdkParametersHandle` has to be called to get handle of global options. Any secondary library has to be initialized by this handle. Initialization function of any secondary library has format `fast{library_name}LibraryInit`, where `library_name` is the name of the library. Initialization function takes handles for global options as parameters.

In case if user application does not use secondary library, it is not necessary to call `fastGetSdkParametersHandle` function.

Pipeline

In computing, a pipeline is a set of data processing elements connected sequentially, where an output of one element is the input for the next one. FASTVIDEO SDK uses pipeline paradigm for data processing. Actually, SDK is a set of data processing components. Each component in general has one input and one output. Input of one component has to be connected to output of another component.

Every pipeline has the first component, several central components and the last component. Each central component has input and output. The first component has only output. The last component has only input.

Link between the components is aggregated by `fastDeviceSurfaceBufferHandle_t` structure or Link Buffer. In general, `fastDeviceSurfaceBufferHandle_t` is a buffer together with some additional service information. Its details are hidden from user. The buffer is always allocated by the previous component and is used by the next component.

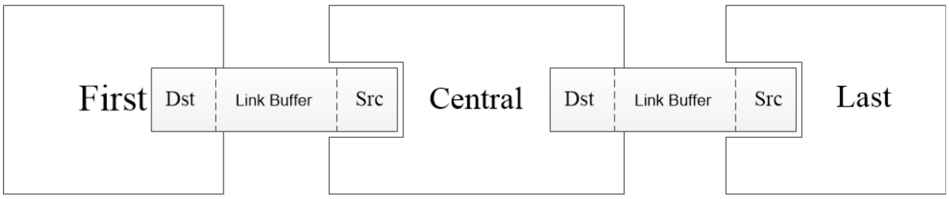

Fig. 1 *Pipeline diagram*

There are three stages of life cycle of every component: creation, working, destroying. Every component has three general methods for corresponding live cycle stages. Method with suffix “Create” corresponds to creation stage. Method with suffix “Destroy” corresponds to destroying stage. Method with suffix “Transform” corresponds to working stage for most components, but for some components working methods have another names that more accurately define the essence of component's operation (e.g. Encode, Decode, Copy, etc.).

On creation stage all necessary components are instantiated and linked to each other. Every central component has two parameters in `Create` method:
`fastDeviceSurfaceBufferHandle_t srcBuffer,`
`fastDeviceSurfaceBufferHandle_t *dstBuffer.`

`dstBuffer` is a pointer to output buffer for the current component. `srcBuffer` is an output buffer of the previous component. So the First component has only `dstBuffer` and Last component has only `srcBuffer`. Components of the pipeline are instantiated in the order from the First to the Last. For component creation it's necessary that source buffer already has been instantiated. So when none of components exists, only the First component can be created.

Main functionality of the First component is to prepare data from an external source and to import it to the pipeline. Main functionality of the Last component is to export data from the pipeline to external destination. In general, external source and destination could be both GPU and CPU buffers. Components that import from CPU (Host) and GPU (Device) buffer data to pipeline are `ImportFromHost` and `ImportFromDevice`, respectively. Components that export data from pipeline to Host and Device buffer are `ExportToHost` and

`ExportToDevice`, respectively. External source and destination buffers can have quite complicated format (e.g. JPEG). So `JpegDecoder` is the First component of the pipeline and `JpegEncoder` is the Last component of the pipeline.

As soon as the pipeline is created, it is ready to process the data. Any pipeline is intended to process not just one, but many images. Array of images should be organized by user application and they could be processed in a loop. Working method of every stage of the pipeline has to be called sequentially to process every image. Working method takes data from the input buffer, processes it and copies to the output buffer, so working method of the next component can be called.

During creation, every component of a pipeline takes maximum size of processing image. Accordingly, all internal buffers are allocated according to max size of the image. Setting up maximum image size is the task for user application. Any pipeline can process images with sizes less than maximum at each dimension. Working method returns `FAST_INVALID_SIZE` if image size is greater than maximum value.

As soon as data processing is finished, the pipeline should be destroyed. Each component is destroyed by calling `Destroy` method. It doesn't matter in which order you are going to destroy the components.

Some central components of the pipeline are containers for multiple subcomponents or methods. For example, Image Filter component is a container for a set of image filters. Container provides common interface for subcomponents with individual parameters.

Import and Export Adapters

There is no way to offer direct feed for central component of pipeline by user data. In order to import user data to the pipeline, we need to run Import Adapter. To export data from the pipeline to user application, we need Export Adapter. That's why the shortest pipeline contains at least three components: Import Adapter, Central Component (e.g. Debayer), and Export Adapter.

`ImportFromHost`, `ExportToHost` adapters allow to communicate with any pipeline through CPU (Host) buffer. It is a common case for most applications. Host buffer should be allocated by `fastMalloc`. If the buffer is allocated by original `malloc`, then it also can be used, but in that case the performance of copy will degrade.

Function `fastMalloc` calls `cudaMallocHost` to allocate page-locked memory in CUDA context.

Some info concerning `cudaMallocHost` from CUDA documentation: the driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as `cudaMemcpy*()`. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as `malloc()`.

To lock in CUDA context already allocated host buffer user application should call `cudaHostRegister`. Function `cudaHostRegister` page-locks the memory range specified by ptr and size and maps it for the device(s) as specified by flags. This memory range also is added to the same tracking mechanism as `cudaHostAlloc()`. The pointer ptr and size must be aligned to the host page size (4 KB).

OS specific page lock functions (like `VirtualLock` in Windows) do not do the same as `cudaHostRegister`. OS specific page lock functions lock memory for system but CUDA context knows nothing about this memory. So that memory cannot be accessed directly by the device because device knows nothing about it.

`ImportFromDevice`, `ExportToDevice` adapters are used in the case when user application has its own GPU kernels. And it is necessary to communicate between user kernels and SDK components. Device Adapters also allow to insert user-specific kernels in a pipeline. Device buffer has to be allocated by `cudaMalloc`.

Import and Export Adapters make some image transforms. Pipeline supports only RGB format of color pixel. To process color image with BGR pixel format, Import Adapter transforms it to RGB. Export Adapter allows to transform internal RGB format to BGR.

Format of external buffer is shown on Fig.2.

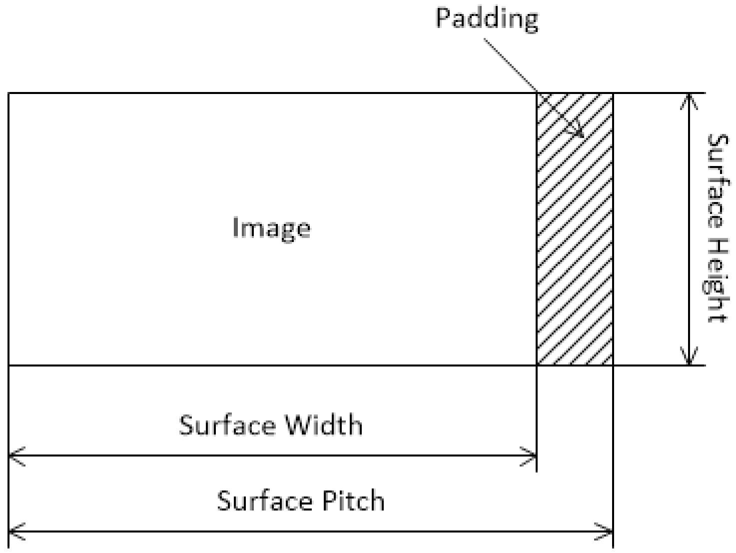

Fig. 2 *Surface buffer*

Pitch means the size of image row in Bytes. It is allowable to have padding after row pixels. To get good performance, it is necessary that every image row has to be aligned on 4-Byte boundary. This is important for both Host and Device buffers. If user application uses bigger alignments (e.g. 8, 16, 32 or more) it will not influence on performance.

Pipeline Surface Format

There are several surface types supported in FASTVIDEO SDK: FAST_I8, FAST_RGB8, FAST_I10, FAST_I12, FAST_RGB12, FAST_I14, FAST_I16, FAST_RGB16, FAST_BGR8, FAST_BGRX8. Surface FAST_I is grayscale or Bayer filtered image, surface FAST_RGB is color image with components order as $R_0G_0B_0$ $R_1G_1B_1$ $R_2G_2B_2$ etc. Number in surface format means bits per channel.

Next table describes all supported formats in terms of number of channels, bits per channel, max channel values, Bytes per channel.

Surface Type	Number of Channels	Bits per channel	Max Channel Value	Bytes per channel
FAST_I8	1	8	255	1
FAST_RGB8	3	8	255	1
FAST_I10	1	10	4095	2
FAST_I12	1	12	4095	2
FAST_RGB12	3	12	4095	2
FAST_I14	1	14	65535	2
FAST_I16	1	16	65535	2
FAST_RGB16	3	16	65535	2

Bytes per pixels = Bytes per channel * Number of Channels.

Surface format FAST_BGR8 is supported by SDK but it is converted by importer or exporter (ImportFrom or ExportTo) to FAST_RGB8.

Surface format FAST_BGRX8 is supported by SDK but it is converted by importer or exporter (ImportFrom or ExportTo) to FAST_RGB8. Alpha channel of FAST_BGRX8 is lost during import and populated by zero during export.

Surface formats FAST_CrCbY8, FAST_YCbCr8 are defined but have not supported yet.

Surface FAST_I10, FAST_I14 is alliase for FAST_I12, FAST_I16 respectively. Only few components support these formats. Main role of these surfaces is to import raw 10- and 14 bits-per-channel image to pipeline and automatic conversation.

Pipeline surface format is initiated by the First component (ImportFrom* or JpegDecoder) which has surface format as input parameter. Every Central component gets surface format from the previous component through Link Buffer. Some components remain surface format unchanged and other components have different output surface formats. So output surface format of the pipeline may differ from initial one. Last components ExportTo* have output surface format as output parameter in create method. It allows to know the type of the surface and to allocate appropriate buffer.

8-bit surfaces are supported by all components in current version of SDK. 12/16-bit surfaces are supported by a few components. So general approach for 12/16-bit surfaces is the following:

1. At the beginning of the pipeline, 12/16-bit components could be used.

2. If the next component supports only 8-bit surfaces, then LUT component should be used to transform 12/16-bit data to 8-bit format.

3. At the end of the pipeline 8-bit components should be used.

If the component doesn't support input surface, then its create method returns `FAST_UNSUPPORTED_FORMAT`.

Table below lists all components and subcomponents with supported types

Component/subcomponent	Input surface formats	
	Grayscale	Color
JPEG Encoder	8,12	8,12
JPEG Decoder	8	8
Debayer HQLI	8, 12, 16	
Debayer DFPD	8, 12, 16	
Debayer MG	12,16	
Affine	8,12,16	8,12,16
Crop	8,10,12,14,16	8,12,16
Image Filter: Gaussian sharpen	8	8
Image Filter: MAD	8,10,12,14,16	
Image Filter: MAD16	10,12,14,16	
Image Filter: Base Color Correction	8, 12, 16	8, 12, 16
Image Filter: White Balance	8, 12, 16	
Image Filter: LUT_8_{8,12,16}	8	8
Image Filter: LUT_8_{8,12,16}_C		8
Image Filter: LUT_12_{8,12,16}	12	12
Image Filter: LUT_12_{8,12,16}_C		12
Image Filter: LUT_16_{8,16}	16	16
Image Filter: LUT_16_{8,16}_C		16
Image Filter: Color Saturation {HSL, HSV}		8, 12, 16
Image Filter: Tone Curve		16
Image Filter: Median Filter	12,16	12,16
Image Filter: RGB LUT 3D		12,16
Image Filter: HSV LUT 3D		12,16
Image Filter: LUT_8_16_BAYER	8	
Image Filter: LUT_10_16_BAYER	10	
Image Filter: LUT_12_16_BAYER	12	
Image Filter: LUT_14_16_BAYER	14	

Image Filter: LUT_16_16_BAYER	16	
Image Filter: BAYER_BLACK_SHIFT	12,16	
Resizer	8, 12, 16	8, 12, 16
Bayer Splitter	8, 12, 16	
Bayer Merger	8, 12, 16	
Surface Converter: BIT_DEPTH	8,10,12,14,16	8,12,16
Surface Converter: SELECT_CHANNEL		8,12,16
Surface Converter: RGB_TO_GRAYSCALE		8,12,16

That table lists all components and subcomponents that transform pipeline surface types

Component/subcomponent	Transformation	
	Color	Bits
Debayer	I → RGB	
Image Filter: {LUT_8_12, LUT_8_12_C}		8 → 12
Image Filter: {LUT_8_16, LUT_8_16_C}		8 → 16
Image Filter: {LUT_12_8, LUT_12_8_C}		12 → 8
Image Filter: {LUT_12_16, LUT_12_16_C}		12 → 16
Image Filter: {LUT_16_8, LUT_16_8_C}		16 → 8
Image Filter: LUT_{8,10,12,14}_16		8,10,12,14→ 16
SurfaceConverter: BIT_DEPTH		8,10,12,14,16→ 8,12,16
Surface Converter: SELECT_CHANNEL	RGB → I	
Surface Converter: RGB_TO_GRAYSCALE	RGB → I	

Pipeline Split and Merge

In terms of SDK pipeline, split is a case when two or more components use the same buffer as an input. That operation allows to split data processing for multiple path. For example split can be used for debug purposes. Export to Host component could be attached to output buffer of debugging component together with the following central component.

Central component in any pipeline does not modify input buffer. But there are some output components which change input buffer: JPEG Encoder and Debayer. If after splitting one of components is changing its input buffer, then this transform method should be called last.

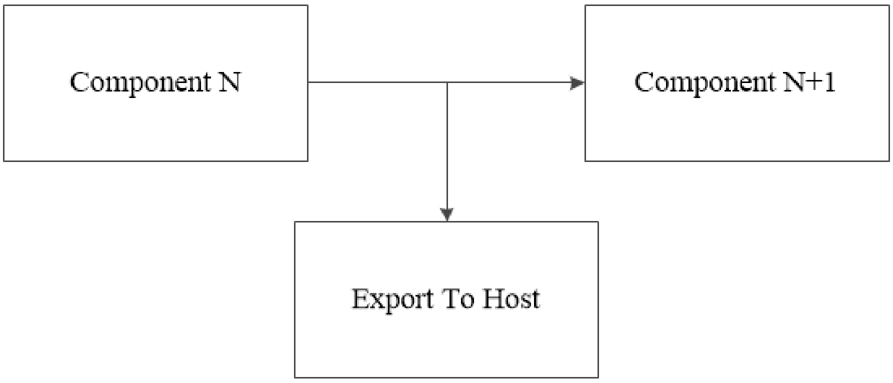

Fig. 3 Debug Pipeline

Sometimes it is necessary to bypass certain component in the pipeline. In this case, multiplexer (Mux) component should be used. Mux selects one of its inputs and pass data to output. Transform method of Mux takes number of inputs that will be passed to output.

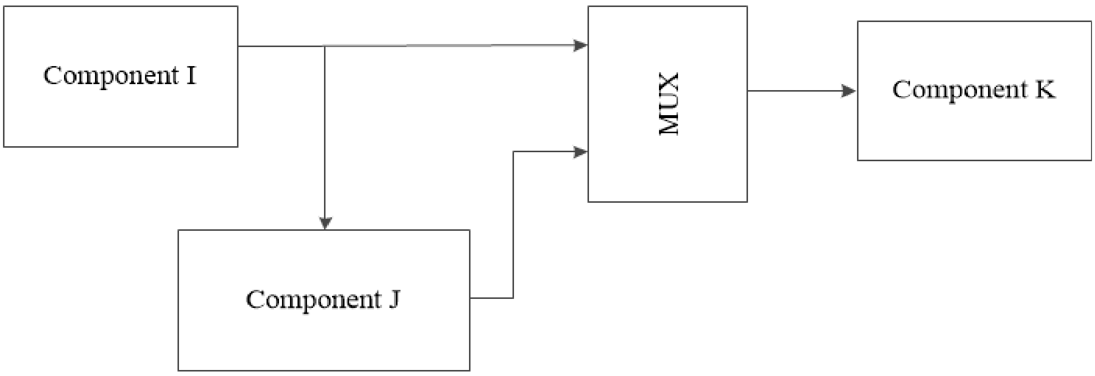

Fig. 4 Bypass Pipeline

Component recreation

There are tasks conducting to recreation some component of existing pipeline. Functions ChangeSrcBuffer allows to avoid whole pipeline recreation in this case. ChangeSrcBuffer function sets new source buffer for component. So to insert new component instead existing component of pipeline destination buffer of created component has to be set as source for following component through ChangeSrcBuffer function.

Threads, Streams and Performance

SDK is compiled with CUDA option "--default-stream per-thread". It means that every CPU thread has its own CUDA stream. SDK does not use CUDA default stream. So SDK pipeline has no influence on CUDA kernel in either default or custom stream. User kernel started without stream specification in the same thread shares stream with SDK kernels.

Individual stream for each thread gives an ability to run multiple pipelines concurrently. In general, each pipeline has import, computation, and export components. Concurrent execution of pipelines allows to overlap computations from one pipeline with data transfer from another pipeline.

Concurrent pipelines also increase GPU utilization by decreasing time when GPU is idle. GPU that has two copy engines (some high range Geforce, Titan, Tesla and Quadra) can overlap host-to-device transfer and device-to-host transfer from concurrent pipelines. Kernels with small amount workloads can be overlapped also. Figure 5 is screenshot of NVIDIA Visual Profiler shown overlapping device-to-host, host-to-device transfers and computation. GPU has two copy engines.

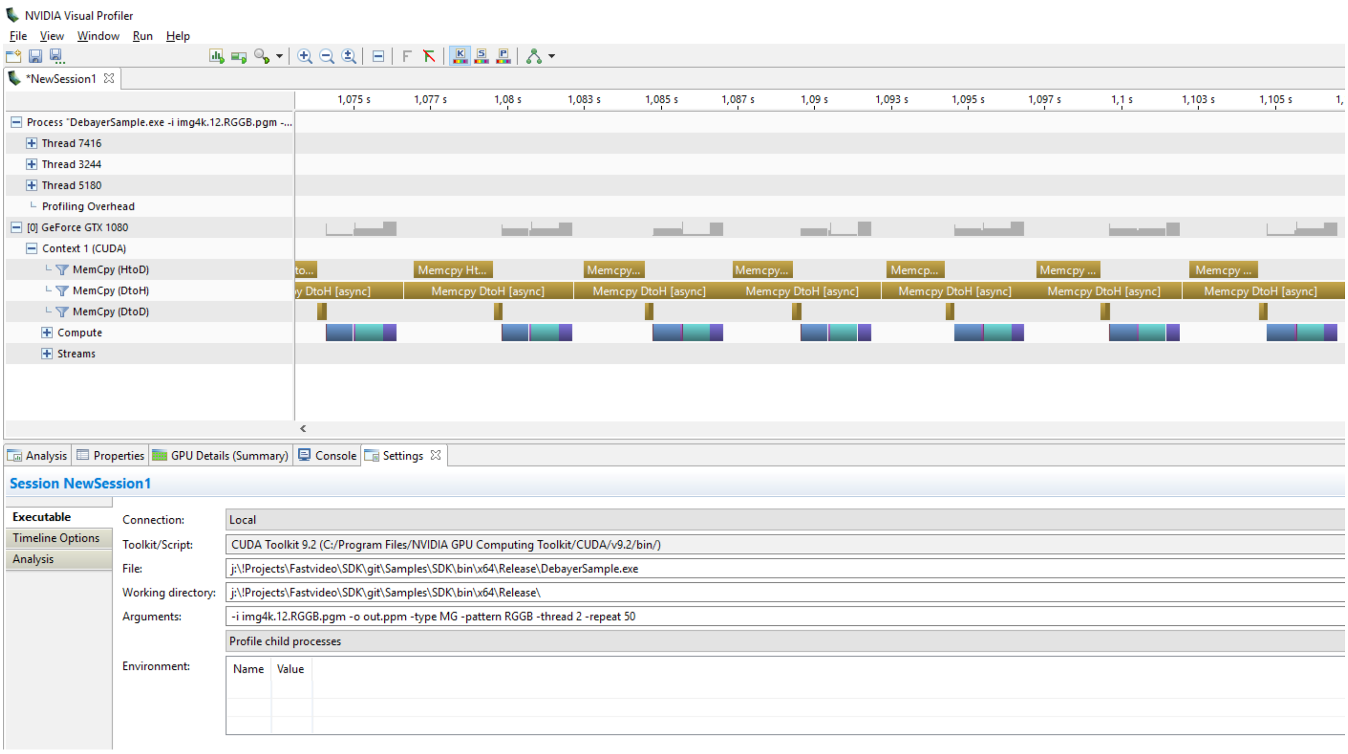

Fig. 5 Overlapping device-to-host, host-to-device transfers and computation.

Concurrent pipelines increase overall system performance but also increase latency. If you need get pipeline result as quickly as possible concurrent pipelines and multi threading is not a case.

Also concurrent pipelines is increasing device memory usage. Each pipeline has its own data set. So four pipelines use almost four times more memory than just one. But some service information is shared between pipelines. So four separate processes with one pipeline consume even more device memory.

Some performance notes. In most case better practise is two concurent pipelines. It overlaps device-to-host, host-to-device transfers and computation and get most significan performance increase. More concurent pipeline (3 or greater) on one GPU can be beter in case of processing small image (FullHD or less).

Debayer

Debayer module restores image colors from Bayer filtered source image according to the pattern. There are 4 supported patterns – RGGB, BGGR, GRBG and GBRG; where R stands for red, G – green, B – blue. Enum `fastBayerPattern_t` contains all supported patterns.

At the moment there are three implemented Debayer algorithms (types): HQLI, DFPD and MG. These algorithms are enumerated in `fastDebayerType_t`. MG has better quality than HQLI and DFPD, but HQLI shows the best performance.

HQLI debayer algorithm is working with window 5x5, DFPD – 11x11, MG – 23x23.

Debayer supports 8/12/16-bit formats (MG supports 12/16 only). All debayers support only image with even width. User has to crop odd width by 1 pixel.

Debayer modules changes pipeline surface format. Initial surface format is `FAST_I{8,12,16}` before Debayer, and after Debayer format becomes `FAST_RGB{8,12,16}` respectively. Debayer changes format from gray to color, and after debayering we finally get three color components (RGB) per pixel instead of one (Gray). At the same time, number of bits per each color component is not changed. For example, if input format has 16 bits per color component, then output format will have 16 bits per each color component of the pixel as well.

Function `fastDebayerCreate` creates Debayer component. Input parameters for this function are the following: maximum image size and name (type) of Debayer algorithm. Instance of Debayer component is associated with a handle which returns `fastDebayerCreate` function. Function `fastDebayerCreate` also allocates all necessary buffers in GPU memory. In case GPU does not have enough free memory, `fastDebayerCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY` status. Customer application has to detect and to process that status correctly. Enum `fastStatus_t` contains all statuses.

Function `fastDebayerTransform` performs debayering according to initial parameters. That function takes image size and bayer pattern as parameters.

Spatial Denoiser

Spatial Denoiser module is based on Discrete Wavelet Transform (DWT) and removes spatial noise from images according to specified parameters: wavelet name, thresholding function, number of DWT resolution levels, array of threshold values, etc. Denoise support grayscale and color images. Also it supports separated planes of Bayer filtered image.

For color images, Spatial Denoiser converts data to YCbCr and performs denoising for these components separately. We recommend to apply the same parameters both to Cb and Cr components to avoid false colors after denoising.

At the moment there are two implemented wavelets for Spatial Denoiser: `CDF53` and `CDF97`. These wavelets are enumerated in `fastWaveletType_t`. They have comparable quality, but `CDF53` shows better performance.

Thresholding function could be `Hard`, `Soft` or `Garrote`. These functions correspond to different algorithms of threshold approximation.

Spatial Denoiser supports the following surfaces: `FAST_RGB8, FAST_I8, FAST_RGB12, FAST_I12, FAST_RGB16, FAST_I16`.

Function `fastDenoiserCreate` creates Spatial Denoiser component. Input parameters for this function are the following: maximum image size and static parameters. Instance of Spatial Denoiser component is associated with a handle which returns `fastDenoiserCreate` function.

Static parameters for denoiser are incapsulated by `denoise_static_parameters_t` struct. It contains wavelet name and type of threshold function. Types of threshold function are enumerated in `fastDenoiseThresholdFunctionType_t`.

Function `fastDenoiserTransform` performs spatial denoising according to initial static and current dynamic parameters. Dynamic parameters for denoiser are the following:

dwt_levels - number of DWT transforms (maximum is 11)

enhance[3] — gain coefficients for each channel of YCbCr. Applied after threshold function to wavelet coefficient.

threshold[3] — basic threshold for Y, Cb and Cr channels respectively.

threshold_per_level[33] — individual relative thresholds for each wavelet band. The first three values in the array correspond to values of Y, Cb and Cr of the first band. Resulting threshold for each wavelet band of channel is multiplication of *threshold* by respective *threshold_per_level.*

Resulting threshold is applied to wavelet coefficient through threshold function multipliers for threshold_per_level for Y, Cb and Cr. Total threshold is equal to the result of (threshold_per_level * threshold) for each band and each color channel.

The `fastDenoiserTransformBayerPlanes` processes distinct color planes of Bayer filtered image prepared by Bayer Splitter component. It filters each planes separately. Resulted planes have to be merged to one color image by Bayer Merger component. Calling the function on normal grayscale image will cause image destruction. So the only way of using denoise component with the function is as part of pipeline, where preceding component is Bayer Splitter and the following component is Bayer Merger component.

JPEG Load and Store functions

SDK includes set of functions to load and store JPEG files. These are six functions for load and two functions for store: `fastJfifLoadFromFile`, `fastJfifHeaderLoadFromFile`,`fastJfifBytestreamLoadFromFile`, `fastJfifLoadFromMemory`, `astJfifHeaderLoadFromMemory`,`fastJfifBytestreamLoadFromMemory`, `fastJfifStoreToFile`, `fastJfifStoreToMemory`.

These functions are not a pipeline components because they work on CPU only. You can find them in \common\helper_jpeg\ folder.

Load functions parse JPEG file and save data to `fastJfifInfo_t` struct the following JPEG entities:

image width and height
restart interval
quantization tables
Huffman tables
EXIF sections
Huffman bytestream

Functions `fastJpegHeaderLoad*` allow decode only header to populate `fastJfifInfo_t`. For example to get JPEG file width and hight. Functions `fastJpegBytestreamLoad*` extract Huffman bytestream to `fastJfifInfo_t`.

Load functions can parse only JPEG files supported by SDK JPEG Decoder. If format of JPEG file is not supported (for example, 12-bit JPEG, JPEG with arithmetic coding or progressive JPEG), then function returns **`FAST_UNSUPPORTED_FORMAT`**. If any errors occur during file parsing, then function returns `FAST_INVALID_FORMAT` and puts error description to stderr.

There are two storages for Load/Store functions: file or memory buffer. Memory buffer as a storage is useful for applications that sends/receives JPEG images via network.

Memory buffers for `fastJfifLoadFromMemory` and `fastJfifStoreToMemory` are allocated by original C `malloc`.

Specially note that buffer for Huffman bytestream in `fastJfifInfo_t` has to be allocated by user application with `fastMalloc`. Size of allocated buffer is set to `bytestreamSize` in `fastJfifInfo_t`. If size of `h_Bytestream` is smaller than size of bytestream of loaded image, then the function returns `FAST_INVALID_SIZE`.

Store functions serialize `fastJfifInfo_t` struct to file or memory buffer.

There are two JPEG file formats: JFIF and EXIF. EXIF format allows to store camera meta data, copyright, author information and other.

Load and Store functions can read and write EXIF sections. In general JPEG file can include multiple EXIF sections, so in `fastJfifInfo_t` there is a pointer to EXIF section array `exifSections`. Count element in array is stored in `exifSectionsCount`. Struct `fastJpegExifSection_t` contains EXIF section code, buffer for EXIF data that starts with EXIF section code and size of EXIF data.

Load functions allocate all buffers for EXIF sections. So after EXIF sections no longer need, then user application has to free all EXIF section buffers by `free`. In other case memory will leak.

Load/store functions work with entire section. To extract tag we propose use existing library, for example libexif. Example of using libexif with SDK load/store functions you can find in JpegSample and BayerCompressionSample.

Allocated `fastJfifInfo_t` for JPEG encoder has to be initialized by `exifSectionsCount =0` and `exifSections = NULL`.

JPEG Encoder/Decoder

JPEG Encoder compresses grayscale and color images into JPG format with different options. Current Encoder supports images with 8 and 12 bits-per-channel. GPU JPEG Decoder supports only 8 bits-per-channel images. JPEG CPU Decoder component is a temporary solution for decoding of images with 12 bits-per-channel. This component is a wrapper around open source libjpeg-turbo library.

It is important to note that there is no additional parameters to enable 12-bit encoder. The same encoder component and interface functions are used for both 8-bit and 12-bit encoders. Encoder type is selected automatically by bit depth of input surface.

JPEG Encoder supports three color subsampling formats: JPEG_444, JPEG_422, and JPEG_420. The ratios at which the chroma subsampling are usually done for JPEG images are 4:4:4 (no chroma downsampling), 4:2:2 (reduction by a factor of 2 in the horizontal direction for components Cb and Cr), or (most commonly) 4:2:0 (chroma reduction by a factor of 2 in both horizontal and vertical directions). Enum `fastJpegSubsamplingFormat_t` contains all supported subsampling formats.

To compress grayscale image, subsampling format has to be `JPEG_Y`. Also there is an option to compress grayscale image as color. In this case JPEG encoder duplicates gray channel to all color channels. To enable this feature, input image should be gray and subsampling format should be `JPEG_420`.

Quality parameter adjusts output JPEG file size and image quality. Quality is an integer value from 1 to 100, where 100 means the best quality and maximum image size. Recommended value for JPEG Quality is 90 (this is so called visually lossless compression).

Restart Interval is an integer number of MCUs (Minimum Coded Unit) processed as an independent sequence within a scan. Current JPEG Encoder has fixed restart interval for each subsampling mode. For subsampling 4:2:0 restart interval is 5, subsampling 4:2:2 has restart interval 8, subsampling 4:4:4 has restart interval 10, for grayscale images this is 32. Please note that Restart Interval = 5 means one restart marker after each five MCUs. These restart intervals are optimal to achieve best JPEG Encoder performance.

If any error occurs during procedure calls, it returns status not equal to `JPEG_OK` and which will appear in `stderr` description of error.

JPEG Encoder is the Last component in terms of SDK pipeline. Its output is structure `fastJfifInfo_t` that resides in CPU memory.

Function `fastJpegEncoderCreate` creates JPEG Encoder. It takes maximum image size. Instance of created JPEG encoder is associated with a handle which returns `fastJpegEncoderCreate` function. Function `fastJpegEncoderCreate` also allocates all necessary buffers in GPU memory. So in case when GPU does not have enough free memory, `fastJpegEncoderCreate` returns status `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Function `fastJpegEncode` compresses images taken from previous component of the pipeline with defined quality and populates `fastJfifInfo_t` with JPEG bytestream and JPEG tables. Buffer for JPEG bytestream in `fastJfifInfo_t` has to be allocated before call. Its recommended size is *surfaceHeight*surfacePitch4.* Real JPEG bytestream size is calculated during compression and put to `bytestreamSize` in `fastJfifInfo_t`. If size of `h_Bytestream` is not enough, procedure returns status `FAST_INTERNAL_ERROR`.

Function `fastJpegEncodeAsync` acts the same as `fastJpegEncode` but store compressed bytestream to device memory. It takes `fastJfifInfoAsync_t` instead of fastJfifInfo_t as a parameter. Structure fastJfifInfoAsync_t is the same as fastJfifInfo_t except d_Bytestream that point on device memory. Important that d_Bytestream is not allocated by user. It points on internal buffer in jpeg encoder. So data from d_Bytestream should be coped from before next fastJpegEncodeAsync call. To store bytestream to file user should copy all fastJfifInfoAsync_t fields to fastJfifInfo_t. Also it has to copy data from device to host by standard CUDA copy functions (see JpegAsyncSample)

JPEG Decoder is the First component in terms of SDK pipeline. Its input is struct fastJfifInfo_t that resides in CPU memory.

Function fastJpegDecoderCreate creates JPEG Decoder. It takes maximum image size and output surface format as parameters (FAST_I8 or FAST_RGB8). Instance of created JPEG Decoder is associated with a handle which returns fastJpegDecoderCreate function. Function fastJpegDecoderCreate also allocates all necessary buffers in GPU memory. So in case when GPU does not have enough free memory fastJpegDecoderCreate returns status FAST_INSUFFICIENT_DEVICE_MEMORY.

Function fastJpegDecode uncompresses image from fastJfifInfo_t taken from previous JPEG Load function.

Restart Interval has great impact on decoding performance. Optimal restart interval for decoder is 1 (one restart marker after each MCU). So even jpeg from fastvideo encoder has not enough markers. If jpeg does not contains marker or restart interval greater than 255 then CPU version of Huffman decoder will be used.

Utility jpegtran can change number of restart markers in existing JPEG image. Jpegtran is part of libjpeg library.

The first step: we need to clear up the image from existing restart markers

jpegtran.exe -copy none image_src.jpg image_dest.jpg

At the end we get the image without restart markers.

The second step: we need to insert necessary amount of restart markers

jpegtran.exe -restart 1B image.jpg image_new.jpg

At the end we get the new image with 1 restart marker after each block (MCU).

JPEG CPU Decoder

JPEG CPU Decoder component is a temporary solution for decoding of 12 bits-per-channel images. This component is a wrapper around open source libjpeg-turbo library that is compiled with BITS_IN_JSAMPLE =12. That's why CPU Decoder does not support 8-bit JPEG images.

JPEG CPU Decoder is not included in primary SDK library. It is delivered in separate dll – fastvideo_jpegCpuDecoder.dll. Please note that libjpeg interface of CPU Decoder is different from GPU Decoder.

Function `fastJpegCpuDecoderCreate` creates JPEG CPU Decoder. It takes maximum image size and output surface format as parameters (`FAST_I12` or `FAST_RGB12`). Function returns `FAST_UNSUPPORTED_SURFACE` if surface parameter was set `FAST_I8` or `FAST_RGB8`.

Function `fastJpegCpuDecode` decompresses JPEG images. It takes buffer on entire file with header. Function puts uncompressed surface to its device surface buffer and puts other JPEG parameters to `fastJfifInfo_t` struct. Decoder returns `FAST_IO_ERROR` if 8-bit JPEG image is supplied as an input.

Timer

Timer component allows to measure time intervals for GPU kernels. For components that executed only on GPU and does not copy anything to CPU, the timer component is the only way to measure timing. CPU timer does not get adequate results in this case because GPU kernel calls are asynchronous.

Timer component is a wrapper for CUDA event. Function `fastGpuTimerCreate` creates two CUDA events (start and stop). Function `fastGpuTimerStart` calls `cudaEventRecord` for start event. Function `fastGpuTimerStop` calls `cudaEventRecord` for stop event. Function `fastGpuTimerGetTime` calls `cudaEventSyncronize` for stop event and then call `cudaEventElapsedTime` for start and stop event.

For time economy reasons `fastGpuTimerGetTime` should be called after all calculations are completed. Otherwise CPU in `fastGpuTimerGetTime` will have to wait until all GPU calculations are done instead of running next kernels that affects total execution time.

GPU Timer has not to be used in case of concurrent piplines becouse great performance degradation.

MJPEG Reader/Writer

MJPEG Reader/Writer is a simple wrapper of FFmpeg to read/write MJPEG file in AVI container. This component is not a part of FASTVIDEO SDK library. It is deployed as separate hpp/cpp files that placed in ffmpeg_wrapper folder.

The Reader is designed to read MJPEG files by frames. On initialization, that component gets MJPEG file path and tries to open it. If the file is opened successfully, then user can get info about frame dimensions and total number of frames in the file. Frames from file are read sequentially by function `GetNextFrame`. When the function reaches the end of the file, it returns status not equal to `FAST_OK`.

Frame read by `GetNextFrame` is a standard JPEG file stream. It has header and can be decoded by any JPEG Decoder. In our samples FASTVIDEO JPEG Decoder is used.

The Writer is designed to serialize JPEG file to MJPEG container. On initialization, the component gets MJPEG file path, frame size, frame rate and sampling format. MJPEG supports only 420 sampling format. It is possible to store other samplings, but in the header only 420 is stored. All written frames have to be of the same size. Frame written by `WriteFrame` function has to be normal JPEG file stream with a header. JPEG stream is stored to file without any transformation, so this is fast process which is limited only by I/O capabilities of the system.

The `FfmpegSample` demonstrates how to work with MJPEG Reader/Writer.

FASTVIDEO SDK is working with FFmpeg which is under the LGPL v2.1.

FFmpeg is compiled without "--enable-gpl" and without "--enable-nonfree".

Affine Transforms

FASTVIDEO SDK supports the following Affine transforms: flip, flop, rotation 180, rotation left 90, rotation right 90. Enum `fastAffineType_t` contains all supported affine transforms. Flip transform reverses order of image columns. Flop transform reverses order of image rows. Rotation 180 degrees reverses order of columns and rows. Rotation 90 left and 90 right changes image dimension: width becomes equal to height, height becomes equal to width. So `maxWidth` and `maxHeight` of the next component in a pipeline have to be properly adjusted.

That component supports `FAST_RGB8` and `FAST_I8` surfaces.

Crop component

The component crops image. Region of interest is defined by coordinates of left top corner of rectangle and its width and height.

That component supports `FAST_RGB8, FAST_I8, FAST_RGB12, FAST_I12, FAST_RGB16, FAST_I16` surfaces.

Image Filter component

Image Filter is a component which includes multiple filters. Image filter in terms of SDK means any operation that transforms every pixel of image, but does not change image geometry, size, and bits-per-pixel value. There are five filter groups: LUT, Sharpen filter, Vector operation, Color correction, Median filter. All filters are listed in `fastImageFilterType_t` enum. Every filter can have two sets of parameters: static and dynamic. Static parameters are assigned on creation and can not be changed while filter is working. Dynamic parameters could be set individually for every processed image.

Some filters have the same static and dynamic parameters. In this case static parameters are initial parameters for filter. Dynamic parameters overload initial parameters and are locked up in the filter until new dynamic parameters will not be passed. Null pointer for dynamic parameters means to use previously set parameters.

Filter that has same static and dynamic parameters can be initialized both in filter creation and in first transform. If filter is not initialized neither in creation nor in first transform then first transform return `FAST_INVALID_VALUE` exception.

Gaussian sharpen filter.

Sharpen filter group contains one Gaussian filter with window 3x3. There is one dynamic parameter – sigma (it's so called “Gaussian kernel radius”). The subcomponent supports `FAST_RGB8` and `FAST_I8` surfaces.

`Median filter.`

Median filter group contains one Median filter with window 3x3. There is no parameters for filter. The subcomponent supports `FAST_RGB12, FAST_RGB16, FAST_I12, FAST_I16` surfaces.

MAD (multiply and add).

Vector operation contains MAD (multiply and add) filter that transforms every pixel according to the equation:

$A \times x + B$, where x – pixel value, A and B are matrices with the same dimensions as image size. Matrix B is called BlackShiftMatrix, matrix A is called CorrectionMatrix.

There are two MAD filter: MAD and MAD16. Structures `fastMad_t and fastMad16_t` are a static / dynamic parameters for appropriate filters. It contains pointers on both matrices. Black matrix has `unsigned char` type in `fastMad_t` and `unsigned short` type in `fastMad_t.` Component MAD supports all grayscale formats. Component MAD16 supports all grayscale formats except `FAST_I8`.

Color correction group contains White Balance, Base Color Correction, Color Saturation Filters.

Bayer Black Shift

This filter is applied to Bayer filtered image. It simple subtract constant value from pixel. There are three constants for each RGB channels.

White Balance

White Balance filter supports `FAST_I8, FAST_I12, FAST_I16` surfaces for bayer filtered image. Structure `fastWhiteBalance_t` defines static and dynamic parameters for the filter. It contains white balance matrix and bayer format. White balance matrix defines by four values: R, G1, G2, B. Bayer patter contains two green values. G1 defines value for green pixel in first row, G2 defines value for green value in second row. Filter multiplies values from white balance matrix on respective values in image.

Base Color Correction

Base Color Correction filter supports `FAST_I8`, `FAST_I12`, `FAST_I16`, `FAST_RGB8`, `FAST_RGB12`, `FAST_RGB16` surfaces. Structure `fastBaseColorCorrection_t` defines static and dynamic parameters for the filter. It contains color correction matrix. Color Correction matrix has 3 rows and 4 columns. In general for RGB pixel, that operation is defined in matrix form as the following:

$$\begin{pmatrix} R_R & R_G & R_B & R_0 \\ G_R & G_G & G_B & G_0 \\ B_R & B_G & B_B & B_0 \end{pmatrix} \times \begin{pmatrix} R \\ G \\ B \\ -1 \end{pmatrix}$$

for grayscale image:

$$\begin{pmatrix} I & 0 & 0 & I_0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} I \\ 0 \\ 0 \\ -1 \end{pmatrix}$$

Color Saturation

Color Saturation filter includes three steps: RGB to {HSL, HSV} transformation, LUT-based {HSL, HSV} transformation, {HSL, HSV} to RGB transformation. User defines three separate LUTs for each {HSL, HSV} channel. Each LUT has 1024 float values. Interval for H LUT is [0,360], intervals for S and L LUTs are [0, 1].

LUT-based {HSL, HSV} transformation depends on individual operation for each channel. There are three type of operation: replace, addition and multiplication. In case of replace operation value from input replaced by new value from LUT. In case of addition operation new value is sum of input value and value from LUT. In case of multiplication operation new value is product of input value and value from LUT.

Structure `fastColorSaturation_t` is a static and dynamic parameters for the filter. It contains three LUTs and three operations. Component supports `FAST_RGB8`, `FAST_RGB12`, `FAST_RGB16` surfaces.

Tone Curve

Tone Curve filter change image tone according tone curve. Filter algorithm is taken from Adobe DNG SDK. Tone curve is defined as a LUT table with 1024 values. Interval for tone curve is [0,1]. Structure `fastToneCurve_t` is a static and dynamic parameters for the filter. It contains LUT of tone curve. Component supports only `FAST_RGB16` surfaces. Example of linear tone curve is defined in defaultToneCurve.h in ComponentsSample folder.

LUT

LUT group contains multiple Lookup Table filters. Lookup Table filter uses input value as an index in table of new (output) values. Two main parameters for Lookup table are bit-depth of input and output values. Lookup filter contains one table, that is common for all color channels, or separate Lookup tables for each color channels Total number of Bytes for Lookup table is the following:

$$2^{bit\,depth\,of\,input\,value} * \text{size in byte of output value}$$

There are following types of Lookup filters: LUT_8_{8,12,16}, LUT_8_{8,12,16}_C, LUT_12_{8,12,16}, LUT_12_{8,12,16}_C, LUT_16_{8,16}, LUT_16_{8,16}_C, LUT_{8,10,12,14,16}_16_BAYER. The first number defines bit depth of input value, the second number defines bit depth of output value. Suffix _C means that filter has three Lookup tables to apply to RGB channels. Suffix _BAYER means that LUT is applied to Bayer filtered image. Bayer LUTs also have three tables to apply to RGB channels. Bayer LUTs take additional parameter to identify bayer type {RGGB, GRBG and etc}. All LUTs have static and dynamic parameters.

Parameters and structures for all LUTs are listed in the table below. Column “tables” defines number of tables in a structure. Column “elements” defines number of elements in LUT table. Column “type of element” means type of LUT table elements.

LUT	Static and dynamic parameters	Tables	Elements	Type of element
`LUT_8_8`	`fastLut_8_t`	1	256	uchar
`LUT_8_8_C`	`fastLut_8_C_t`	3	256	uchar
`LUT_8_{12,16}`	`fastLut_8_16_t`	1	256	ushort
`LUT_8_{12,16}_C`	`fastLut_8_16_C_t`	3	256	ushort
`LUT_8_16_BAYER`	`fastLut_8_16_Bayer_t`	3	256	ushort
`LUT_10_16_BAYER`	`fastLut_10_16_Bayer_t`	3	1024	ushort
`LUT_12_8`	`fastLut_12_8_t`	1	4096	uchar
`LUT_12_8_C`	`fastLut_12_8_C_t`	3	4096	uchar
`LUT_12_{12,16}`	`fastLut_12_t`	1	4096	ushort
`LUT_12_{12,16}_C`	`fastLut_12_C_t`	3	4096	ushort
`LUT_12_16_BAYER`	`fastLut_12_16_Bayer_t`	3	4096	ushort
`LUT_14_16_BAYER`	`fastLut_14_16_Bayer_t`	3	16384	ushort
`LUT_16_8`	`fastLut_16_8_t`	1	16384	uchar
`LUT_16_8_C`	`fastLut_16_8_C_t`	3	16384	uchar
`LUT_16_16`	`fastLut_16_t`	1	16384	ushort
`LUT_16_16_C`	`fastLut_16_C_t`	3	16384	ushort
`LUT_16_16_BAYER`	`fastLut_16_16_Bayer_t`	3	16384	ushort

LUT changes bit depth of pipeline surface if LUT's input bit depth is not equal to output bit depth.

16-bit LUTs have only 16384 values in table. Size of LUT's table is limited by resources of GPU. 14 significant bits of input value are used to identify element in LUT, and two less significant bits of output are calculated via linear interpolation.

RGB 3D LUT

RGB 3D LUT is used to map one color space to another. In general 3D LUT is uniform 3D grid of RGB values. Often this grid is called cube because it has same number of elements in all directions. Input RGB value is defined elementary cube in grid. Trilinear interpolation is used to calculate output RGB value based on the elementary cube and input RGB value.

Current Fastvideo SDK supports arbitrary RGB 3D LUT size with up to 65 elements in one dimension and even more. Structure `fastRGBLut_3D_t` could be both static and dynamic parameter for the filter. It contains cube 1d size and pointer to LUT. Memory ordering of elements is according Adobe CUBE format. This ordering is the opposite of the typical in-memory order of multi-dimensional tables. An equivalent index would be r + N * g + N * N * b, where r, g, b are the Red, Green, and Blue indices in the range 0 to N–1.

Component supports `FAST_RGB12, FAST_RGB16` surfaces. Example of RGB 3D LUT you can find in `ComponentSample`.

HSV 2D/3D LUT

HSV 3D LUT is used to map one color space to another through HSV transformation. Component is compatible with lookup table specified in Adobe Digital Negative Format.

In general HSV LUT is uniform 2D/3D grid of 3 components vectors. Each vector contains transformation value for channels H, S, V, one element for one channel. Type of transformation value is float. Type of transformation is defined per channel (H,S,V) globally. There are three type of transformations: replace, addition, multiplication. In case of replace operation value from input is replaced by new value from appropriate element from the vector. In case of addition operation new value is sum of input value and value from the vector. In case of multiplication operation new value is the product of input value and value from the vector.

Unlike RGB 3D LUT, HSV 3D LUT grid is not a cube. Number of elements per each axis can be different. Also there is 2D case when number of elements per V axis is 1. 2D LUTs with H = 1 or S = 1 are not supported.

Process steps:

1. Convert RGB pixel to HSV, where H in [0,360], S in [0,1], V in [0,1].
2. Find elementary cube in 3D LUT included input value.
3. Calculate transformation vector by trilinear interpolation of the elementary cube and input RGB value.
4. Apply transformation operation to input HSV pixel with transformation vector.
5. Convert new HSV pixel to RGB.

Structure `fastHsvLut3D_t` could be both static and dynamic parameter for the filter. It contains LUT size for each axis, LUT pointer and transformation operation for each channel. Memory ordering of elements in LUT is according to lookup table ordering at Adobe DNG format. This ordering is the opposite of the typical in-memory order of multi-dimensional tables. An equivalent index would be V + dimV * H + dimV * dimH * S.

Component supports `FAST_RGB12, FAST_RGB16` surfaces. Example of RGB 3D LUT you can find in `ComponentSample`.

Resizer component

The component downscales and upscales an image. Image scale factor defines in two ways:

1. By defining width of resized image with preserved image aspect ratio.
2. By defining any width and height of resized image.

In the first case, hight of resized image is automatically calculated by image height and scale factor. Resized height is returned by component to unify rounding process.

In the second case, image aspect ratio is not preserved. User has to define resized image height as well.

Both cases are used both for downscale and upscale.

The only supported algorithm of resize is Lanczos. Component supports `FAST_RGB8`, `FAST_I8`, `FAST_RGB12`, `FAST_I12`, `FAST_RGB16`, `FAST_I16` surfaces.

Bayer Splitter and Bayer Merger components

The main reason why these components were included in SDK is better performance in common task of saving frames from a camera in real time applications. Camera frame is Bayer filtered so in common case color has to be restored by Debayer component and encoded by color JPEG Encoder. Encoding each Bayer frame to JPEG directly is not a good idea. In general it saves time by excluding Debayer from the pipeline but quality suffers greatly. JPEG compression algorithm utilizes the fact that brightness in real picture changed smoothly. This allows to drop height frequency data to get better image compression. Bayer image has a lot of high frequency data because in one block we have merged data from three color channels. As a results, JPEG compressed Bayer image has lower compression rate and worse quality. To solve that issue, we have to split Bayer image on four planes with pixels from appropriate channels (R, G1, G2, B). Thus one Bayer plane is similar to color channel downsampled by two.

The Bayer Splitter component splits Bayer image on four planes and concatenates them in one column (from top to bottom). Now splitted Bayer image is ready be compressed to JPEG. Height of every concatenated plane is aligned to 8. It is important for JPEG compression because each plane does not interfere with another. Position of plane in columns depends on pixel's position in the pattern and doesn't depend on Bayer pattern.

Height of original image can not be taken from height of splitted Bayer filtered image, so the original height has to be stored to JPEG file in EXIF section. EXIF section is defined by `SplitterExif_t` structure in `ExifInfo.hpp`. It contains original image width, height and bayer pattern. EXIF section ID is `0xFFE1`.

The Bayer Merger component restores original image from splitted Bayer image and EXIF section information.

SDI import and export components

Serial Digital Interface (SDI) is a family of digital video interfaces. They are used for transmission of uncompressed, unencrypted digital video signals.

SDI Import component of FASTVIDEO SDK allows to feed pipeline with data directly from video interface. There are two types of SDI Import component one import data from host another from device memory. SDI interface supports multiple video formats. Current version of that component supports some of them. In the future the number of supported formats will be increased by user demand. SDI Import component copies data from host to device memory and converts format to appropriate RGB.

Also SDI component allows convert to / from other formats used in video processing software.

List of supported SDI formats for import (enum `fastSDIFormat_t`):

```
FAST_SDI_YV12_BT601
FAST_SDI_YV12_BT601_FR
FAST_SDI_YV12_BT709
FAST_SDI_422_8_CbYCr_BT709
FAST_SDI_422_8_CbYCr_BT601
FAST_SDI_422_8_CbYCr_BT601_FR
FAST_SDI_422_8_CrYCb_BT709
FAST_SDI_422_8_CrYCb_BT601
FAST_SDI_422_8_CrYCb_BT601_FR
```

SDI Export component of SDK allows to store data from pipeline in appropriate SDI format. There are two types of SDI Export components: one export data to host and another to device memory.

List of supported SDI formats for export (enum `fastSDIFormat_t`):

```
FAST_SDI_YV12_BT601
FAST_SDI_YV12_BT601_FR
FAST_SDI_YV12_BT709
FAST_SDI_422_8_CbYCr_BT709
FAST_SDI_422_8_CbYCr_BT601
FAST_SDI_422_8_CbYCr_BT601_FR
FAST_SDI_422_8_CrYCb_BT709
FAST_SDI_422_8_CrYCb_BT601
FAST_SDI_422_8_CrYCb_BT601_FR
FAST_SDI_RGBA_ZERO_ALPHA,
FAST_SDI_RGBA_FF_ALPHA,
FAST_SDI_NV12_BT601_FR
```

Name of SDI format consists of three parts: layout, color order, color transformation. Layout means how pixels or blocks of pixels reside in memory and which pixels/subpixels are contained by block of pixels. Color order means order of color subpixels in block of pixels. Color transformation means name of color transformation to convert source pixels to RGB.

There are three layouts: 422_8, YV12, NV12.

Layout 422_8 is a simple YCbCr format with 422 subsampling. Block of pixels contains 2 pixels (2 columns per 1 row) or 4 Bytes. We have individual Y component for each source pixel and average color component – Cb and Cr.

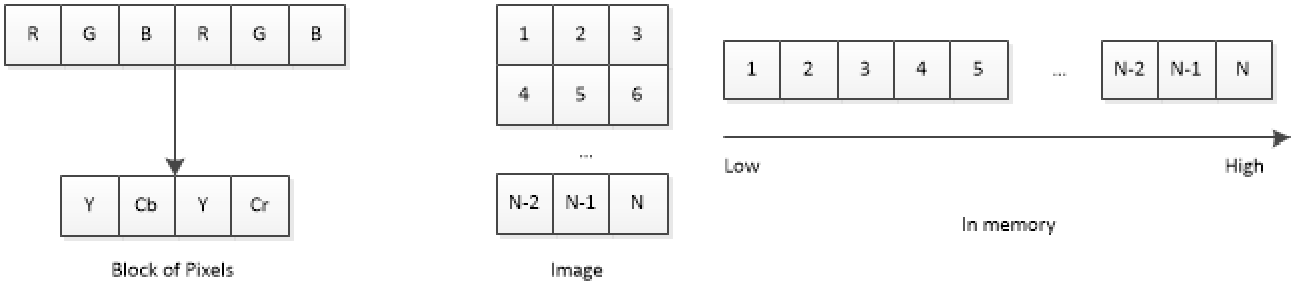

Fig. 5 *Format 422_8*

Figure 5 shows block of pixels in source image and its position in memory array.

Notation CbYCr means that 4 Bytes in block from less to most significant Bytes contain CbY1CrY2 subpixels, CrYCb respectively CrY1CbY2.

Layout YV12 is YCbCr planar format with 420 subsampling. There are three planes: Y, Cb, Cr. Luminance plane (Y) has the same size as original image. Chrominance planes (Cb and Cr) are downscaled by two times both for width and height. Each chrominance value corresponds to one block (2 rows by 2 columns) of original image.

Function with suffix `Copy3` for YV12 format allows to copy each plane to/from separate memory blocks.

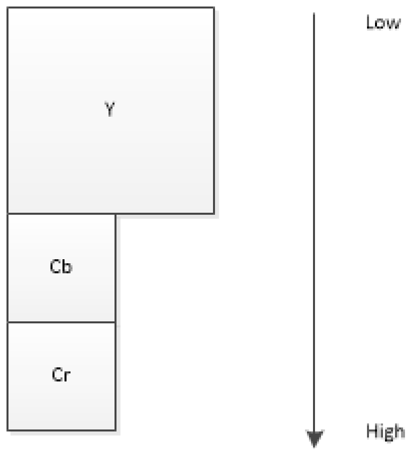

Fig. 6 *Format YV12*

Figure 6 shows color planes in memory array.

Layout NV12 is YCbCr hybrid format with 420 subsampling. There are two planes: Y, Cb/Cr. Luminance plane (Y) has the same size as original image. Each chrominance value corresponds to one block (2 rows by 2 columns) of original image. Combined Cb/Cr plane contains alternated Cb and Cr pixels. So Cb/Cr plane width is equal to original image width and its height is downscaled by two times. Figure 7 shows color planes in memory array.

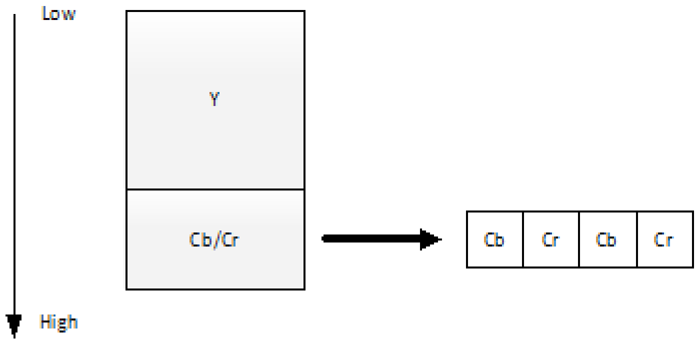

Fig. 7 *Format NV12*

All supported SDI formats are mapped to `FAST_RGB8` surface format. There is no way to export grayscale or color format with more than 8 bits per channel.

There are two types of color transformations to convert RGB to YCbCr and vice versa: BT601 and BT709. Also there are two modes: first for digital television second for video processing on PC. Mods are differed by output value range. For digital television Y is in [16,235], Cb/Cr is in [16,240]. For PC (other name for the mods is full range) Y,Cb,Cr are in [0,255].

BT601

RGB to YCbCr

Y = 0.257 * R + 0.504 * G + 0.098 * B + 16

Cb = -0.148 * R – 0.291 * G + 0.439 * B + 128

Cr = 0.439 * R – 0.368 * G + 0.071 * B + 128

YCbCr to RGB

R = 1.164 * (Y - 16) + 1.596 * (Cr - 128)

G = 1.164 * (Y - 16) – 0.813 * (Cr - 128) - 0.391 * (Cb - 128)

B = 1.164 * (Y - 16) + 2.018 * (Cb - 128)

BT709

RGB to YCbCr

Y = 0.183 * R + 0.614 * G + 0.062 * B + 16

Cb = -0.101 * R - 0.338 * G + 0.439 * B + 128

Cr = 0.439 * R - 0.399 * G - 0.040 * B + 128

YCbCr to RGB

R = 1.164 * (Y - 16) + 1.793 * (Cr - 128)

G = 1.164 * (Y - 16) - 0.534 * (Cr - 128) – 0.213 * (Cb - 128)

B = 1.164 * (Y - 16) + 2.115 * (Cb – 128)

BT601 (full range, FR)

RGB to YCbCr

Y = 0.299 * R + 0.587 * G + 0.114 * B

Cb = -0.169 * R – 0.331 * G + 0.499 * B + 128

Cr = 0.499 * R – 0.418 * G + 0.071 * B + 128

YCbCr to RGB

R = 1.0 * Y + 1.40 * (Cr - 128)

G = 1.0 * Y – 0.711 * (Cr - 128) - 0.343 * (Cb - 128)

B = 1.0 * Y + 1.765 * (Cb - 128)

RGBA is simple format with 4 Bytes per pixel for better OpenGL integration. Less significant byte is R then G and B. Most significant Byte in it is Alpha channel. Alpha channel value can be zero or FF. It is defined by format type.

Surface converter component

Component converts bit depth and surface type. There are three type of conversions: bit depth, RGB channel select, RGB to grayscale by brightness.

Bit depth conversion changes bit depth of surface by left or right shift input pixels. When source value has smaller bit depth than destination, then left shift is used. Inserted bits are populated by zero. When source value has greater bit depth than destination, right logical shift is used.

Acceptable conversions `FAST_I{8,10,12,14,16}->FAST_I{8,12,16}`, `FAST_RGB{8,12,16}->FAST_RGB{8,12,16}`.

Select channel filter allows to get any channel from RGB and save it as grayscale image. RGB to grayscale filter takes all RGB channels with appropriate coefficient to create RGB image. Coefficients are user defined, so any type of RGB to grayscale can be used.

Histogram component

An image histogram is a type of histogram that acts as a graphical representation of the tonal distribution in a digital image. Histograms are widely used in image processing and computer vision. Histogram component calculates three types of histograms: common, bayer, parade. Enum `fastHistogramType_t` contains all supported types.

Common types defines ordinary histogram, calculated for a whole image. Component generates one histogram for grayscale image and three histograms for RGB image.

Number of bins is usually equal to number of tone levels. But number of bins has significant influence on performance. So use less bins as possible.

Bayer type histogram interprets grayscale image as Bayer filtered image with defined pattern. Bayer pattern is additional input parameter. Component generates three histograms for Bayer image, for each color in pattern, respectively. Also there is an option when each green pixel in pattern has its own histogram (`FAST_HISTOGRAM_BAYER_G1G2`).

Parade contains multiple histograms for each color. In general case, it gives three color histograms for each image columns. Also there is an option to calculate parade with predefined step to skip several columns.

Structure `fastHistogramBayer_t` could be both static and dynamic parameter for bayer type histogram. Structure `fastHistogramParade_t` could be both static and dynamic parameter for parade histogram.

JPEG2000 Encoder/Decoder

JPEG2000 Encoder compresses grayscale and color images into JP2 format using lossy or lossless algorithm with different options. Current encoder and decoder support images with 8–16 bits per channel.

JPEG2000 Decoder supports three popular color subsampling formats: 4:4:4 (no chroma downsampling), 4:2:2 (reduction by a factor of 2 in the horizontal direction for components Cb and Cr), or (most commonly) 4:2:0 (chroma reduction by a factor of 2 in both horizontal and vertical directions).

Quality parameter adjusts output JP2 file size and image quality. Quality is a floating-point value from 0 to 100 (which is nonlinearly related to the quantization coefficient), where 100 means no quantization and maximum image size. Recommended value for Quality parameter is 85 (visually lossless compression).

If any error occurs during procedure call, it returns status not equal to `FAST_OK` and description of the error will appear in `stderr`.

JPEG2000 Encoder is the last component in terms of SDK pipeline. Its output resides in CPU memory.

Function `fastEncoderJ2kCreate` creates JPEG2000 Encoder. It takes maximum image size. Instance of created JPEG2000 encoder is associated with a handle which is returned by `fastEncoderJ2kCreate` function. Function `fastEncoderJ2kCreate` also allocates all necessary buffers in GPU memory. So, in case when GPU does not have enough free memory, `fastEncoderJ2kCreate` returns status `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Function `fastEncoderJ2kTransform` compresses image taken from previous component of the pipeline with defined quality, returns `fastEncoderJ2kOutput_t` structure with JPEG2000 bytestream and `fastEncoderJ2kReport_t` structure with report. Buffer for JPEG2000 bytestream in `fastEncoderJ2kOutput_t` has to be allocated before the call. Real JPEG2000 bytestream size is calculated during compression and put to `streamSize` in `fastEncoderJ2kOutput_t`. If `bufferSize` is too small, bytestream is truncated.

Functions `fastEncoderJ2kAddImageToBatch`, `fastEncoderJ2kTransformBatch` and `fastEncoderJ2kGetNextEncodedImage` are used to compress multiple images simultaneously in batch mode. First, images are added one by one to the batch using `fastEncoderJ2kAddImageToBatch` function until `maxBatchSize` is reached. After that, `fastEncoderJ2kTransformBatch` is used to process the batch and to get the first compressed image. The rest of the images are returned sequentially by `fastEncoderJ2kGetNextEncodedImage`. When all compressed images are returned, the next batch can be passed for processing. Function `fastEncoderJ2kFreeSlotsInBatch` can be used to determine how many images can be added to the batch. Function `fastEncoderJ2kUnprocessedImagesCount` can be used to determine how many images are waiting to be compressed.

JPEG2000 Decoder is the first component in terms of SDK pipeline. Its input must reside in CPU memory.

Function `fastDecoderJ2kCreate` creates JPEG2000 Decoder. It takes maximum image size and output surface format as parameters. Instance of created JPEG2000 decoder is associated with a handle which is returned by `fastDecoderJ2kCreate` function. Function `fastDecoderJ2kCreate` also allocates all necessary buffers in GPU memory. So, in case when GPU does not have enough free memory `fastDecoderJ2kCreate` returns status `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Function `fastDecoderJ2kTransform` decompresses image from CPU memory and returns `fastDecoderJ2kReport_t` structure with report.

Functions `fastDecoderJ2kAddImageToBatch`, `fastDecoderJ2kTransformBatch` and `fastDecoderJ2kGetNextEncodedImage` are used to decompress multiple images simultaneously in batch mode. First, images are added one by one to the batch using `fastDecoderJ2kAddImageToBatch` function until `maxBatchSize` is reached. After that, `fastDecoderJ2kTransformBatch` is used to process the batch and to get the first decompressed image. The rest of the images are returned sequentially by `fastDecoderJ2kGetNextEncodedImage`. When all decompressed images are returned, the next batch can be passed for processing. Function `fastDecoderJ2kFreeSlotsInBatch` can be used to determine how many images can be added to the batch. Function `fastDecoderJ2kUnprocessedImagesCount` can be used to determine how many images are waiting to be decompressed.

NPP component

NVIDIA Performance Primitive (NPP) is a large library which provides GPU-accelerated image, video, and signal processing functions. With over 5000 primitives for image and signal processing, you can easily perform multiple tasks.

In Fastvideo SDK exits some components, which are based on NPP functions. Some components are wrapper for NPP functions, some components add additional functionality.

There are four NPP-based SDK components: NppFilter, NppGeometry, NppResize, NppRotate.

NppFilter component currently supports gaussian sharpen and unsharp mask filters. Gaussian sharpen filter based on gaussian blur kernel. It is applied to all RGB channels separately. Filter gaussian sharpen is a wrapper for nppiFilterGaussAdvancedBorder*.

Unsharp mask is smart sharpening filter. It's also based on gaussian blur but it is applied only to Y channel of YCbCr image. Chrominance channels are not changed. Also it has additional parameters: threshold, amount, envelope. Threshold controls the minimum brightness change that will be sharpened or how far apart adjacent tonal values have to be before the filter does anything. Amount is how much contrast is added at the edges. Envelope function allows to make Amount parameter depend on pixel brightness. Amount has maximum value in the center of pixel brightness range. And minimum value on the border of range. This is to reduce over saturation for bright and dark areas.

Envelop function is the following:

$$envelope = exp(coef * ((value - median)/sigma)^{rank})$$

where *value* – pixel brightness in range [0;1], *median* – median value in range [0;1], *sigma - variance value in range [0;], rank* has to be even, *coef* is negative value.

NppGeometry component currently supports image remap operation. Remap operation is a wrapper for nppiRemap* functions.

NppResize component resize image with selected interpolation mode. This component is a wrapper for nppiRotate* functions.

NppRotate component rotates an image around the origin (0,0) and then shifts it. This component is a wrapper for nppiRotate* functions.

Auxiliary functions

Auxiliary functions give access to internal pipeline information to help user to identify and solve some software issues.

Function `fastGetDeviceSurfaceBufferInfo` extracts information from device surface buffer about surface format, maximal image width, height and pitch, current image width, height and pitch. This information allows to control surface and image size, which could be changed in the pipeline. Surface format and maximal image size in device surface buffer are initialized after component creation. Image size and pitch are initialized after call of transform function.

If create function of component returns `FAST_UNSUPPORTED_SURFACE` then one should call `fastGetDeviceSurfaceBufferInfo` and check whether the component supports current surface format.

If transform function of component returns `FAST_INVALID_SIZE` this means that image size is lager than maxWidth and maxHeight, which were assigned to component on creation. User has to call `fastGetDeviceSurfaceBufferInfo` and check image size.

Most functions of SDK are asynchronous. Only functions that copy from device memory to host are synchronous. Such behavior is result of asynchronous nature of CUDA kernel call. Asynchronous call allows increase performance but result in problem with error handling. Failed function can return not own error but error that raised some function before. So it is impossible to determine what function is crashed.

Global option Interface Synchronization add to the end of all interface method `cudaDeviceSyncronize` call. This localizes problem in one interface method. Error can not pass bound between calls. Function `fastEnableInterfaceSynchronization` enables and disables Interface Synchronization. By default InterfaceSynchronization is disabled. If InterfaceSynchronization is enabled performance is degraded greatly. So use this option only for debug purpose.

Trace functions

SDK Trace stores to file all parameters of all called SDK functions and its returned statuses. All public fields of device surface buffer are serialized in trace module as well.

Trace helps user to check all parameters and to found errors during creation or transform phase. Fastvideo support team can request trace file to reproduce a problem. Trace file is an ordinary text file so user can verify that there is no private information in its content.

To enable trace function `fastTraceCreate` has to be called with trace file name/path. Function `fastTraceClose` flushes buffers and closes trace file. Name of trace file is a global option.

Some of failure is raised by structural exception like segmentation fault or stack destroying or other and cannot be catch by C++ try catch. In this case user cannot call `fastTraceClose` to close trace file correctly and will loose some trace information. To avoid loosing trace information user has to enable trace flush global option. If trace flush is enabled, every trace write will be stored to file. Function `fastTraceEnableFlush` enables and disables trace flush option.

Sample Applications

There are the following sample applications in SDK: `BayerCompressionSample`, `CameraMultiSample`, `CameraSample`, `ComponentsSample`, `DebayerJpegSample`, `DebayerSample`, `DenoiseSample`, `FfmpegSample`, `ImageConverterSample`, `JpegAsyncSample`, `JpegSample`, `MtDebayerSample`, `MuxSample`, `PhotoHostingSample`, `RawImportSample`, `ResizeSample`, `SDIConverterSample`. All sample applications can process single file or folder. In case of folder, multiple files are processed in a loop.

`BayerCompressionSample` demonstrates how to split, compress, decompress and merge Raw Bayer data for camera application.

`CameraSample` demonstrates how to implement full image processing pipeline for data from a camera, including visualizing via OpenGL. The application also demonstrates pixel correction operation and LUT transform, etc. CameraSample is the only demo project that need CUDA SDK to be installed

`DebayerSample` demonstrates how to work with API of Debayer component. Class `Debayer` (`Debayer.h`, `Debayer.cpp`) encapsulates API calls for Debayer.

`DebayerJpegSample` demonstrates how to integrate Debayer with JPEG Encoder.

`JpegSample` demonstrates how to work with API of JPEG Encoder/Decoder components of SDK. Classes `Encoder` (`Encoder.h`, `Encoder.cpp`) and `Decoder` (`Decoder.h`, `Decoder.cpp`) encapsulate API calls for JPEG Encoder and JPEG Decoder respectively. In addition, application demonstrates pixel correction operation and LUT transform for JPEG Encoder.

`PhotoHostingSample` demonstrates how to integrate in one pipeline the following five components: JPEG Decoder, Crop, Resizer, Image Filter, JPEG Encoder.

`SDIConverterSample` demonstrates how to import SDI image into pipeline and how to export SDI image from pipeline to host memory.

How to create your own applications with that SDK

It's a good idea to start from doing some tests with command-line sample applications, which come with SDK. This is important to be sure that you understand all parameters, have right input image and final image is fine in terms of quality.

Then one could start from source code analysis of our sample applications to build your own software. At this point one can work with images on HDD/SSD to reproduce working solution with good image quality.

Next step is an attempt to work with your data which reside in system memory or GPU memory. You can copy data from HDD/SSD to system memory and check the performance.

In the case if you have packed data from a camera, we suggest to switch to 8-bit camera mode to exclude packing, at least at the beginning. At this stage you need to create your software which is working with camera data in system memory. Unpacking could be added later, as soon as your application is working.

Demo Applications

Apart from sample applications in SDK, you can get ready-to-use demo applications from the following site www.fastcompression.com at download section. You can find there the following software for Windows:

CUDA Debayer – command-line application for debayering (all bayer patterns, 8/12/16-bit data, demosaicing algorithms HQLI, DFPD, MG

CUDA JPEG Codec – command-line application for JPEG compression and decompression

CUDA J2K Codec – command-line application for J2K encoding and decoding

CUDA Resize – command-line application for resize

Fast CinemaDNG Processor (for Windows) – GUI application for realtime processing of CinemaDNG image series and MLV video on GPU.

You can get more info at www.fastcinemadng.com

Fastvideo SDK API

Statuses

All SDK functions return result status of `fastStatus_t` type.

`FAST_OK` – There is no error during function execution.

`FAST_TRIAL_PERIOD_EXPIRED` – Trial period expired for demo version of SDK. All functions are shut down.

`FAST_INVALID_DEVICE` – Device with selected index does not exist or device is non NVIDIA device or device is non CUDA-compatible device.

`FAST_INCOMPATIBLE_DEVICE` – Device is CUDA-compatible, but its compute compatibility is below 3.0, thus device is considered to be incompatible with SDK.

`FAST_INSUFFICIENT_DEVICE_MEMORY` – Available device memory is not enough to allocate new buffer.

`FAST_INSUFFICIENT_HOST_MEMORY` – Available host memory is not enough to allocate new buffer.

`FAST_INVALID_HANDLE` – Component handle is invalid or has inappropriate type.

`FAST_INVALID_VALUE` – Some parameter of the function called is invalid or combination of input parameters are unacceptable.

`FAST_UNAPPLICABLE_OPERATION` – This operation can not be applied to the current type of data.

`FAST_INVALID_SIZE` – Image dimension is invalid.

`FAST_UNALIGNED_DATA` – Buffer base pointers or pitch are not properly aligned.

`FAST_INVALID_TABLE` – Invalid quantization / Huffman table.

`FAST_BITSTREAM_CORRUPT` – JPEG bitstream is corrupted and can not be decoded.

`FAST_EXECUTION_FAILURE` – Device kernel execution failure.

`FAST_INTERNAL_ERROR` – Internal error, non-kernel software execution failure.

`FAST_UNSUPPORTED_SURFACE` – Current component does not support this type of surface. Check documentation.

`FAST_IO_ERROR` – Failed to read/write file.

`FAST_INVALID_FORMAT` – Invalid file format.

FAST_UNSUPPORTED_FORMAT – File format is not supported by the current version of SDK.

FAST_END_OF_STREAM – Error related motion JPEG decoding. Unexpected end of stream.

FAST_MJPEG_THREAD_ERROR – Error related motion JPEG encoding/decoding. General error in worker thread.

FAST_TIMEOUT – Error related motion JPEG encoding/decoding or other video coding. Timeout in worker thread.

FAST_MJPEG_OPEN_FILE_ERROR – Error related motion JPEG encoding/decoding. Could not open file.

FAST_UNKNOWN_ERROR – Unrecognized error.

Master SDK and secondary library initialization

fastInit

```
fastStatus_t fastInit (unsigned affinity, bool openGlMode)
```

Sets GPU device to work with.

Parameters:

affinity [in] - affinity mask. “1” in less significant bit of affinity mask denotes to use GPU with device Id = 1. “1” in next bit denotes to use GPU with device Id = 2 and so on.

openGlMode [in] – if *openGlMode* set to be true, then FASTVIDEO SDK is initialized to work with OpenGL application. In other case FASTVIDEO SDK is initialized to work with ordinary CUDA application.

Notes:

If device is not found or device is not NVIDIA device or device does not support CUDA, function will return status `FAST_INVALID_DEVICE`.

If device has compute compatibility below 3.0, function will return status ***FAST_INCOMPATIBLE_DEVICE***.

Statuses:

`FAST_OK, FAST_INVALID_DEVICE, FAST_INTERNAL_ERROR, FAST_INCOMPATIBLE_DEVICE`

fastGetSdkParametersHandle

```
fastStatus_t fastGetSdkParametersHandle (fastSdkParametersHandle_t *handle)
```

Get handle of SDK global options.

Parameters:

handle [out] - pointer to global options handle

Statuses:

`FAST_OK`

fastLibraryInit

```
fastStatus_t fast*LibraryInit (fastSdkParametersHandle_t handle)
```

Init secondary library of component by SDK global options.

Parameters:

handle [in] - global options handle

Statuses:

`FAST_OK`

Trace and Auxiliary functions

fastGetDeviceSurfaceBufferInfo

```
fastStatus_t fastGetDeviceSurfaceBufferInfo (
      fastDeviceSurfaceBufferHandle_t buffer,
            fastDeviceSurfaceBufferInfo_t *devBuffer)
```

Get public information from device surface buffer.

Parameters:

buffer [in] – handle of device surface buffer.

devBuffer [out] – pointer to structure with public information.

Notes:

Structure fastDeviceSurfaceBufferInfo_t contains the following fields:

surfaceFmt – type of surface. Value is `fastSurfaceFormat_t` enum integer values. Values 12 and 13 are internal representations of FAST_RGB12 and FAST_RGB16 formats respectively. Value is defined after component creation.

maxWidth – maximum width of processed image. Value is defined after component creation.

maxHeight – maximum height of processed image. Value is defined after component creation.

maxPitch – pitch for maximal image. Value is defined after component creation.

width – width of currently processed image. Value is defined after call of transform function.

height – height of currently processed image. Value is defined after call of transform function.

pitch – pitch of currently processed image. Value is defined after call of transform function.

Statuses:

`FAST_OK`

fastEnableInterfaceSynchronization

```
fastStatus_t fastEnableInterfaceSynchronization(bool isEnabled)
```

Set value of interface synchronization in global option.

Parameters:

isEnabled [in] – option value

Notes:

Global option Interface Synchronization adds to the end of all interfaces method `cudaDeviceSyncronize` call. This localizes problem in one interface method. Asynchronous error can not pass bound between calls.

Statuses:

`FAST_OK`

fastTraceCreate

```
fastStatus_t fastTraceCreate(const char *fileName)
```

Open trace file.

Parameters:

fileName [in] – file path or file name for trace file

Notes:

If file cannot be opened or created function returns `FAST_IO_ERROR`.

Statuses:

`FAST_OK, FAST_IO_ERROR`

fastTraceClose

```
fastStatus_t fastTraceClose()
```

Close current trace file.

Statuses:

`FAST_OK`

fastTraceEnableFlush

```
fastStatus_t fastTraceEnableFlush(bool enableFlush)
```

Set value of trace flush in global option.

Parameters:

enableFlush [in] – option value

Notes:

If trace flush is enabled, every trace write will be stored to file immediately.

Statuses:

`FAST_OK`

Memory management functions

fastMalloc

*fastStatus_t fastMalloc(void **buffer, size_t size)*

Allocates page-locked memory on CPU.

Parameters:

buffer [out] – pointer to allocated memory

size [in] – size of allocated memory in Bytes

Notes:

Page-locked memory cannot be moved from RAM to swap file. It increases PCI-Express I/O speed of GPU over conventional memory.

If system can not allocate page-locked memory, then function will return status `FAST_INSUFFICIENT_HOST_MEMORY`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_HOST_MEMORY`

fastFree

*fastStatus_t fastFree(void *buffer)*

Frees page-locked memory.

Parameters:

buffer [in] – pointer to allocated memory

Statuses:

`FAST_OK, FAST_INTERNAL_ERROR`

fastGetDevices

*fastStatus_t fastGetDevices(fastDeviceProperty **devices, int *count)*

Pipeline import functions

fastImportFromHostCreate

```
fastStatus_t fastImportFromHostCreate(
      fastImportFromHostHandle_t     *handle,
      fastSurfaceFormat_t      surfaceFmt,
      unsigned     maxWidth,
      unsigned     maxHeight,
      fastDeviceSurfaceBufferHandle_t *dstBuffer)
```

Creates ImportFromHostAdapter and returns associated handle.

Parameters:

handle [out] – pointer to created ImportFromHostAdapter handle

surfaceFmt [in] – defines input pixel format

maxWidth [in] – maximum image width in pixels

maxHeight [in] – maximum image height in pixels

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastImportFromHostCreate` allocates all necessary buffers in GPU memory. In case GPU does not have enough free memory, `fastImportFromHostCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY`

fastImportFromHostGetAllocatedGpuMemorySize

```
fastStatus_t fastImportFromHostGetAllocatedGpuMemorySize(
      fastImportFromHostHandle _t    handle,
      unsigned     *allocatedGpuSizeInBytes)
```

Returns requested GPU memory for ImportFromHostAdapter.

Parameters:

handle [in] – ImportFromHostAdapter handle

allocatedGpuSizeInBytes [out] – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for ImportFromHostAdapter.

Statuses:

`FAST_OK`

fastImportFromHostCopy

```
fastStatus_t fastImportFromHostCopy(
      fastImportFromHostHandle_t     handle,
            void* h_src,
            unsigned     width,
            unsigned     pitch,
            unsigned     height)
```

Copy image from CPU buffer to pipeline.

Parameters:

handle [in] – ImportFromHostAdapter handle

h_src [in] – pointer on CPU buffer with image data

width [in] – image width in pixels

pitch [in] – size of image row in Bytes

height [in] – image height in pixels

Notes:

Buffer *h_src* has to be allocated with `fastMalloc`. Buffer allocated by original `malloc` also can be used, but copy speed will degrade.

If image size is greater than maximum value on creation, then error status `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastImportFromHostDestroy

```
fastStatus_t fastImportFromHostDestroy(
      fastImportFromHostHandle_t     handle)
```

Destroy ImportFromHostAdapter.

Parameters:

handle [in] – ImportFromHostAdapter handle

Statuses:

`FAST_OK`

fastImportFromDeviceCreate

```
fastStatus_t fastImportFromDeviceCreate(
      fastImportFromDeviceHandle_t    *handle,
      fastSurfaceFormat_t      surfaceFmt,
      unsigned     maxWidth,
      unsigned     maxHeight,
      fastDeviceSurfaceBufferHandle_t      *dstBuffer)
```

Creates ImportFromDeviceAdapter and returns associated handle.

Parameters:

handle [out] – pointer to created ImportFromDeviceAdapter handle

surfaceFmt [in] – defines input pixel format

maxWidth [in] – maximum image width in pixels

maxHeight [in] – maximum image height in pixels

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastImportFromDeviceCreate` allocates all necessary buffers in GPU memory. In case GPU does not have enough free memory, then `fastImportFromDeviceCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY`

fastImportFromDeviceGetAllocatedGpuMemorySize

```
fastStatus_t fastImportFromDeviceGetAllocatedGpuMemorySize(
      fastImportFromDeviceHandle _t  handle,
      unsigned    *allocatedGpuSizeInBytes)
```

Returns requested GPU memory for ImportFromDeviceAdapter.

Parameters:

handle [in] – ImportFromDeviceAdapter handle

allocatedGpuSizeInBytes [out] – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for ImportFromDeviceAdapter.

Statuses:

`FAST_OK`

fastImportFromDeviceCopy

```
fastStatus_t fastImportFromDeviceCopy(
      fastImportFromDeviceHandle_t   handle,
      void* d_src,
      unsigned    width,
      unsigned    pitch,
      unsigned    height)
```

Copy image from GPU buffer to pipeline.

Parameters:

handle [in] – Import From Device Adapter handle

d_src [in] – pointer on Device buffer with image

width [in] – image width in pixels

pitch [in] – size of image row in Bytes

height [in] – image height in pixels

Notes:

Buffer d_*src* has to be allocated in Device memory by `cudaMalloc`.

If image size is greater than maximum value on creation, then error status `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastImportFromDeviceDestroy

```
fastStatus_t fastImportFromDeviceDestroy(
      fastImportFromDeviceHandle_t handle)
```

Destroy ImportFromDeviceAdapter.

Parameters:

handle [in] – ImportFromDeviceAdapter handle

Statuses:

`FAST_OK`

Pipeline export functions

fastExportToHostCreate

```
fastStatus_t fastExportToHostCreate(
      fastExportToHostHandle_t        *handle,
      fastSurfaceFormat_t             *surfaceFmt,
      fastDeviceSurfaceBufferHandle_t srcBuffer)
```

Creates ExportToHostAdapter and returns associated handle.

Parameters:

handle [out] – pointer to created ExportToHostAdapter handle

surfaceFmt [out] – pipeline output surface format

srcBuffer [in] – linked buffer from previous component

Statuses:

`FAST_OK`

fastExportToHostGetAllocatedGpuMemorySize

```
fastStatus_t DLL fastExportToHostGetAllocatedGpuMemorySize(
      fastExportToHostHandle_t handle,
      unsigned *requestedGpuSizeInBytes
);
```

Get ExportToHostAdapter GPU memory usage.

Parameters:

handle [in] – ExportToHostAdapter handle

requestedGpuSizeInBytes [out] – memory usage in byte

Statuses:

`FAST_OK`

fastExportToHostChangeSrcBuffer

```
fastStatus_t DLL fastExportToHostChangeSrcBuffer(
      fastExportToHostHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – ExportToHostAdapter handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

FAST_OK,FAST_INVALID_SIZE

fastExportToHostCopy

```
fastStatus_t fastExportToHostCopy(
      fastExportToHostHandle_t       handle,
      void* h_dst,
      unsigned    width,
      unsigned    pitch,
      unsigned    height,
      fastExportParameters_t   *parameters)
```

Copies image from pipeline to CPU buffer.

Parameters:

handle [in] – ExportToHostAdapter handle

h_dst [in] – pointer on Host buffer for image

width [in] – image width in pixels

pitch [in] – size of image row in Bytes

height [in] – image height in pixels

parameters [in] – export parameters

Notes:

Buffer `h_dst` has to be allocated with `fastMalloc`. Buffer, which is allocated by original `malloc` also can be used, but copy speed will degrade. If size of `h_dst` is not enough, then the function will fail with segmentation fault.

Export Parameters allows to convert RGB color format to BGR color format. In other case `parameters` have to be `null`. To convert color format to BGR `convert` member of `fastExportParameters_t` have to set in `FAST_CONVERT_BGR`.

Statuses:

FAST_OK

fastExportToHostDestroy

```
fastStatus_t fastExportToHostDestroy(
      fastExportToHostHandle_t handle)
```

Destroy ExportToHostAdapter.

Parameters:

handle [in] – ExportToHostAdapter handle

Statuses:

FAST_OK

fastExportToDeviceCreate

```
fastStatus_t fastExportToDeviceCreate(
      fastExportToDeviceHandle_t      *handle,
      fastSurfaceFormat_t       *surfaceFmt,
      fastDeviceSurfaceBufferHandle_t       srcBuffer)
```

Creates ExportToDeviceAdapter and returns associated handle.

Parameters:

handle [out] – pointer to created ExportToDeviceAdapter handle *surfaceFmt [out]* – pipeline output surface format

srcBuffer [in] – linked buffer from previous component

Statuses:

`FAST_OK`

fastExportToDeviceCopy

```
fastStatus_t fastExportToDeviceCopy(
      fastExportToDeviceHandle_t     handle,
      void* d_dst,
      unsigned     width,
      unsigned     pitch,
      unsigned     height,
      fastExportParameters_t   *parameters)
```

Copy image from pipeline to GPU buffer.

Parameters:

handle [in] – ExportToDeviceAdapter handle

d_dst [in] – pointer on Device buffer for image

width [in] – image width in pixels

pitch [in] – size of image row in Bytes

height [in] – image height in pixels

parameters [in] – export parameters

Notes:

Buffer `d_dst` has to be allocated in Device memory by `cudaMalloc`. If size of `d_dst` is not enough, then function will fail with segmentation fault.

Export Parameters allows to convert RGB color format to BGR color format. In other case `parameters` have to be `null`. To convert color format to BGR `convert` member of `fastExportParameters_t` have to set in `FAST_CONVERT_BGR`.

Statuses:

`FAST_OK`

fastExportToDeviceGetAllocatedGpuMemorySize

```
fastStatus_t DLL fastExportToDeviceGetAllocatedGpuMemorySize(
      fastExportToDeviceHandle_t     handle,
      unsigned *requestedGpuSizeInBytes
);
```

Get ExportToDeviceAdapter GPU memory usage.

Parameters:

handle [in] – ExportToDeviceAdapter handle

requestedGpuSizeInBytes [out] – memory usage in Bytes

Statuses:

`FAST_OK`

fastExportToDeviceChangeSrcBuffer

```
fastStatus_t DLL fastExportToDeviceChangeSrcBuffer(
      fastExportToDeviceHandle_t handle,
```

```
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – ExportToDeviceAdapter handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastExportToDeviceDestroy

```
fastStatus_t fastExportToDeviceDestroy(
      fastExportToDeviceHandle_t handle)
```

Destroy ExportToDeviceAdapter.

Parameters:

handle [in] – ExportToDeviceAdapter handle

Statuses:

`FAST_OK`

Debayer functions

fastDebayerCreate

```
fastStatus_t fastDebayerCreate(
    fastDebayerHandle_t   *handle,
    fastDebayerType_t     debayerType,
    unsigned        maxWidth,
    unsigned        maxHeight,
    fastDeviceSurfaceBufferHandle_t   srcBuffer,
    fastDeviceSurfaceBufferHandle_t   *dstBuffer)
```

Creates Debayer and returns associated handle.

Parameters:

handle [out] – pointer to created Debayer handle

debayerType [in] – debayer algorithm (HQLI, DFPD, MG)

maxHeight [in] – maximum image height in pixels

maxWidth [in] – maximum image width in pixels

srcBuffer [in] – linked buffer from previous component.

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastCreateDebayerHandle` allocates all necessary buffers in GPU memory. In case GPU does not have enough free memory, then `fastCreateDebayerHandle` will return `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Maximum dimensions of the image are set to Debayer during creation. Thus if transformation result exceeds the maximum value, then error status `FAST_INVALID_SIZE` will be returned.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

Statuses:

FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNSUPPORTED_SURFACE

fastDebayerGetAllocatedGpuMemorySize

```
fastStatus_t fastDebayerGetAllocatedGpuMemorySize(
      fastDebayerHandle_t      handle,
      unsigned     * requestedGpuSizeInBytes)
```

Returns requested GPU memory for Debayer.

Parameters:

handle [in] – Debayer handle

requestedGpuSizeInBytes [out] – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for Debayer.

Statuses:

FAST_OK

fastDebayerChangeSrcBuffer

```
fastStatus_t DLL fastDebayerChangeSrcBuffer(
      fastDebayerHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Sets new source buffer

Parameters:

handle [in] – Debayer handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

FAST_OK, FAST_INVALID_SIZE

fastDebayerTransform

```
fastStatus_t fastDebayerTransform(
      fastDebayerHandle_t      handle,
      fastBayerPattern_t bayerFmt,
      unsigned     width,
      unsigned     height)
```

Restores image colors.

Parameters:

handle [in] – Debayer handle

bayerFmt [in] – bayer pattern

height [in] – image height in pixels

width [in] – image width in pixels

Notes:

The procedure takes Bayer image from input linked buffer, restores colors based on pattern and algorithm and then stores color image to output linked buffer.

If image size is greater than maximum value on creation, then error status `FAST_INVALID_SIZE` will be returned.

Statuses:

FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE

fastDebayerDestroy

```
fastStatus_t fastDebayerDestroy(debayerHandle_t handle)
```

Destroys Debayer handle.

Parameters:

handle [in] – Debayer handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

Denoise functions

fastDenoiseCreate

```
fastStatus_t fastDenoiseCreate(
	fastDenoiseHandle_t *handle,
	fastSurfaceFormat_t surfaceFmt,
	void *staticDenoiseParameters,
	unsigned maxWidth,
	unsigned maxHeight,
	fastDeviceSurfaceBufferHandle_t srcBuffer,
	fastDeviceSurfaceBufferHandle_t *dstBuffer)
```

Creates Denoise and returns associated handle.

Parameters:

handle [out] – pointer to created Denoise handle

surfaceFmt [in] – image surface format

staticDenoiseParameters [in] – static parameters for Denoise

maxHeight [in] – maximum image height in pixels

maxWidth [in] – maximum image width in pixels

srcBuffer [in] – linked buffer from previous component.

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastCreateDenoiseHandle` allocates all necessary buffers in GPU memory. In case GPU does not have enough free memory, then `fastCreateDenoiseHandle` will return `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Maximum dimensions of the image are set to Denoise during creation. Thus if transformation result exceeds the maximum value, then error status `FAST_INVALID_SIZE` will be returned.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

Structure `denoise_static_parameters_t` contains static parameters for Denoise.

```
struct {
	fastDenoiseThresholdFunctionType_t function;
	fastWaveletType_t wavelet;
} denoise_static_parameters_t
```

Where

function – type of threshold function. All threshold functions are enumerated in `fastDenoiseThresholdFunctionType_t`.

wavelet – type of used wavelet. All wavelets are enumerated in `fastWaveletType_t`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNSUPPORTED_SURFACE`

fastDenoiseGetAllocatedGpuMemorySize

```
fastStatus_t fastDenoiseGetAllocatedGpuMemorySize(
      fastDenoiseHandle_t      handle,
      unsigned     *allocatedGpuSizeInBytes)
```

Returns requested GPU memory for Denoise.

Parameters:

handle [in] – Denoise handle

allocatedGpuSizeInBytes [out] – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for Denoise.

Statuses:

FAST_OK

fastDenoiseChangeSrcBuffer

```
fastStatus_t DLL fastDenoiseChangeSrcBuffer(
      fastDenoiseHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – Denoise handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise FAST_INVALID_SIZE will be returned.

Statuses:

FAST_OK, FAST_INVALID_SIZE

fastDenoiseTransform

```
fastStatus_t DLL fastDenoiseTransform(
      fastDenoiseHandle_t handle,
      void *denoiseParameters,
      unsigned width,
      unsigned height
);
```

Denoise image.

Parameters:

handle [in] – Denoise handle

denoiseParameters [in] – dynamic parameters for Denoise

height [in] – image height in pixels

width [in] – image width in pixels

Notes:

The procedure takes the image from input linked buffer, filters the image based on dynamic parameters and then stores the image to output linked buffer.

If image size is greater than maximum value on creation, then error status `FAST_INVALID_SIZE` will be returned.

Structure `denoise_parameters_t` contains dynamic parameters for Denoise.

```
struct {
      int dwt_levels;
      float enhance[3];
      float threshold[3];
      float threshold_per_level[33];
} denoise_parameters_t;
```

Where

dwt_levels - number of DWT transforms (maximum is 11)

enhance[3] - gains for each channel of YCbCr. Applied after threshold function to wavelet coefficient.

threshold[3] — basic thresholds for Y, Cb and Cr channels respectively.

threshold_per_level[33] — individual relative thresholds for each wavelet band. The first three values in the array correspond to values of Y, Cb and Cr of the first band. Resulting threshold for each wavelet band of channel is multiplication of *threshold* by respective *threshold_per_level.*

Resulting threshold is applied to wavelet coefficient through threshold function multipliers for threshold_per_level for Y, Cb and Cr. Total threshold is equal to the result of (threshold_per_level * threshold) for each band and each color channel.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastDenoiseTransformBayerPlanes

```
extern fastStatus_t DLL fastDenoiseTransformBayerPlanes(
      fastDenoiseHandle_t handle,
      void *denoiseParameters,
      unsigned width,
      unsigned height)
```

Denoise separated planes of Bayer filtered image.

Parameters:

handle [in] – Denoise handle

denoiseParameters [in] – dynamic parameters for Denoise

height [in] – image height in pixels

width [in] – image width in pixels

Notes:

The procedure process separated planes of Bayer filtered image prepared by Bayer Splitter component. It filters each planes separately. Resulted planes have to be merged to the one image by Bayer Merger component. Calling the function on normal gray scale image will cause image destruction. So the only way of using denoise component with the function is as part of pipeline where preceding component is Bayer Splitter and following component is Bayer Merger component.

Dynamic parameters and generated error

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastDenoiseDestroy

fastStatus_t fastDenoiseDestroy(fastDenoiseHandle_t handle)

Destroys Denoise handle.

Parameters:

handle [in] – Denoise handle

Notes:

Procedure frees component's device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

JPEG Encoder functions

fastJpegEncoderCreate

```
fastStatus_t fastJpegEncoderCreate(
      fastJpegEncoderHandle_t  *handle,
      unsigned    maxHeight,
      unsigne     maxWidth,
      fastDeviceSurfaceBufferHandle_t srcBuffer)
```

Creates JPEG 8 or 12 bits Encoder and returns associated handle.

Parameters:

handle [out] – pointer to created JPEG Encoder handle

maxHeight [in] – maximum image height in pixels

maxWidth [in] – maximum image width in pixels

srcBuffer [in] – linked buffer from previous component

Notes:

It is important to note that there is no additional parameters to enable 12-bit encoder. The same encoder component and interface functions are used for both 8 and 12 bit encoder. Encoder type is selected automatically by bit depth of input surface.

Function `fastJpegEncoderCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastJpegEncoderCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Maximum dimensions of the image are set to Encoder during creation. Thus if encoded image exceeds the maximum value, then error status `FAST_INVALID_SIZE` will be returned.

Gray image (`FAST_I8`) can be encoded by color encoder (`FAST_RGB8`). In this case Encoder converts gray image to color image by duplicating gray channel to all color channels.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNSUPPORTED_SURFACE`

fastJpegEncoderGetAllocatedGpuMemorySize

```
fastStatus_t fastJpegEncoderGetAllocatedGpuMemorySize(
      fastJpegEncoderHandle_t  handle,
      unsigned    *allocatedGpuSizeInBytes)
```

Returns requested GPU memory for JPEG Encoder.

Parameters:

handle [in] – JPEG Encoder handle

allocatedGpuSizeInBytes [out] – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for JPEG Encoder.

Statuses:

FAST_OK

fastJpegEncoderChangeSrcBuffer

```
fastStatus_t DLL fastJpegEncodeChangeSrcBuffer(
	fastJpegEncoderHandle_t handle,
	fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – JPEG Encoder handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastJpegEncode

```
fastStatus_t fastJpegEncode(
	fastJpegEncoderHandle_t  handle,
	unsigned    quality,
	fastJfifInfo_t     *jfifInfo)
```

Encodes surface to JPEG and store to host memory.

Parameters:

handle [in] – JPEG Encoder handle

quality [in] – adjusts output JPEG file size and quality. Quality is an integer value from 1 to 100 where 100 means the best quality and maximum file size of compressed image.

jfifInfo [in] – pointer to `fastJfifInfo_t` struct that contains all necessary information for JPEG encoding. For more detail see JPEG Encoder Description.

Notes:

The procedure takes surface from previous component of the pipeline through input linked buffer and encodes it accordingly addition parameters from `jfifInfo`. JPEG bytestream is placed to `h_Bytestream` buffer in `jfifInfo`.
Buffer for JPEG bytestream in `jfifInfo` has to be allocated before call. Its recommended size is *surfaceHeight*surfacePitch4.* Real JPEG bytestream size is calculated during compression and put to `bytestreamSize` in `jfifInfo`. If size of `h_Bytestream` is not enough, then procedure returns status `FAST_INTERNAL_ERROR`.

Members of `jfifInfo exifSectionsCount` and `exifSections` have to be initialized by 0.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNALIGNED_DATA, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastJpegEncodeAsync

```
fastStatus_t fastJpegEncode(
```

```
      fastJpegEncoderHandle_t  handle,
      unsigned     quality,
      fastJfifInfoAsync_t      *jfifInfo)
```

Encodes surface to JPEG and store to device memory.

Parameters:

handle [in] – JPEG Encoder handle

quality [in] – adjusts output JPEG file size and quality. Quality is an integer value from 1 to 100 where 100 means the best quality and maximum file size of compressed image.

jfifInfo [in] – pointer to `fastJfifInfoAsync_t` struct that contains all necessary information for JPEG encoding. For more detail see JPEG Encoder description.

Notes:

The procedure takes surface from previous component of the pipeline through input linked buffer and encodes it accordingly addition parameters from `jfifInfo`. JPEG bytestream is placed to `d_Bytestream` buffer in `jfifInfo`. Memory for `d_Bytestream` is allocated by jpeg encoder.
Members of `jfifInfo exifSectionsCount` and `exifSections` have to be initialized by 0.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNALIGNED_DATA, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastJpegEncodeWithQuantTable

```
fastStatus_t fastJpegEncodeWithQuantTable(
      fastJpegEncoderHandle_t handle,
      fastJpegQuantState_t *quantTable,
      fastJfifInfo_t *jfifInfo
);
```

Encodes surface to JPEG with defined quantization table and store to host memory.

Parameters:

handle [in] – JPEG Encoder handle

quantTable [in] – user defined quantization table.

jfifInfo [in] – pointer to `fastJfifInfo_t` struct that contains all necessary information for JPEG encoding. For more detail see JPEG Encoder Description.

Notes:

The procedure takes surface from previous component of the pipeline through input linked buffer and encodes it accordingly addition parameters from `jfifInfo`. JPEG bytestream is placed to `h_Bytestream` buffer in `jfifInfo`.
Buffer for JPEG bytestream in `jfifInfo` has to be allocated before call. Its recommended size is *surfaceHeight*surfacePitch4.* Real JPEG bytestream size is calculated during compression and put to `bytestreamSize` in `jfifInfo`. If size of `h_Bytestream` is not enough, then procedure returns status `FAST_INTERNAL_ERROR`.

Members of `jfifInfo exifSectionsCount` and `exifSections` have to be initialized by 0.

Statuses:

FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNALIGNED_DATA, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE

fastJpegEncodeAsyncWithQuantTable

```
fastStatus_t fastJpegEncodeAsyncWithQuantTable(
      fastJpegEncoderHandle_t handle,
      fastJpegQuantState_t *quantTable,
      fastJfifInfoAsync_t *jfifInfo
      );
```

Encodes surface to JPEG with defined quantization table and store to device memory.

Parameters:

handle [in] – JPEG Encoder handle

quantTable [in] – user defined quantization table.

jfifInfo [in] – pointer to fastJfifInfoAsync_t struct that contains all necessary information for JPEG encoding. For more detail see JPEG Encoder description.

Notes:

The procedure takes surface from previous component of the pipeline through input linked buffer and encodes it accordingly addition parameters from jfifInfo. JPEG bytestream is placed to d_Bytestream buffer in jfifInfo. Memory for d_Bytestream is allocated by jpeg encoder.
Members of jfifInfo exifSectionsCount and exifSections have to be initialized by 0.

Statuses:

FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNALIGNED_DATA, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE

fastJpegEncoderDestroy

```
fastStatus_t fastJpegEncoderDestroy(fastJpegEncoderHandle_t handle)
```

Destroys JPEG encoder.

Parameters:

handle [in] – JPEG encoder handle

Notes:

Statuses:

FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR

JPEG Decoder functions

fastJpegDecoderCreate

```
fastStatus_t fastJpegDecoderCreate(
      fastJpegDecoderHandle_t  *handle,
      fastSurfaceFormat_t      surfaceFmt,
      unsigned    maxWidth,
      unsigned    maxHeight,
      bool checkBytestream,
      fastDeviceSurfaceBufferHandle_t *dstBuffer)
```

Creates JPEG Decoder and returns associated handle.

Parameters:

handle [out] – pointer to created JPEG Decoder

surfaceFmt [in] – type of surface (decoded image). Surface is output for decoder.

maxWidth [in] – maximum image width in pixels

maxHeight [in] – maximum image height in pixels

checkBytestream [in] –

dstBuffer [out] – pointer for linked buffer for next component (output buffer of current component)

Notes:

Function `fastJpegDecoderCreate` allocates all necessary buffers in GPU memory. Thus in case GPU does not have enough free memory, then `fastJpegDecoderCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Maximum dimensions of the image are set to Decoder during creation. Thus if transformation result exceeds the maximum value, then error status `FAST_INVALID_SIZE` will be returned.

Only `FAST_RGB8` and `FAST_I8` surface formats are supported in other case `FAST_UNSUPPORTED_SURFACE` will be returned.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNSUPPORTED_SURFACE`

fastJpegDecoderGetAllocatedGpuMemorySize

```
fastStatus_t fastJpegDecoderGetAllocatedGpuMemorySize(
       fastJpegDecoderHandle_t       handle,
       unsigned                      *requestedGpuSizeInBytes)
```

Returns requested GPU memory for JPEG Decoder.

Parameters:

handle [in] – JPEG Decoder handle

allocatedGpuSizeInBytes [out] – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for JPEG Decoder.

Statuses:

`FAST_OK`

fastJpegDecode

```
fastStatus_t fastJpegDecode(
      fastJpegDecoderHandle_t  handle,
      fastJfifInfo_t     *jfifInfo)
```

Decodes JPEG to surface.

Parameters:

handle [in] – pointer to JPEG Decoder.

jfifInfo [in] – pointer to `fastJfifInfo_t` struct that contains all necessary information for JPEG decoding. For more detail see JPEG Encoder description.

Notes:

The procedure takes JPEG bytestream from `h_Bytestream` buffer in `jfifInfo`. Additional parameters for decoding are also taken from `jfifInfo`. Decoded surface is placed to output linked buffer and the following component of the pipeline consumes it. Struct `fastJfifInfo_t` for Decoder is populated by JfifLoad fuctions: `fastJfifLoadFromFile` and `fastJfifLoadFromMemory`.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_BITSTREAM_CORRUPT`

fastJpegDecoderDestroy

```
fastStatus_t fastJpegDecoderDestroy(fastJpegDecoderHandle_t handle)
```

Destroys JPEG Decoder.

Parameters:

handle [in] – pointer to JPEG Decoder

Notes:

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

JPEG CPU Decoder functions

fastJpegCpuDecoderCreate

```
fastStatus_t fastJpegDecoderCreate(
       fastJpegCpuDecoderHandle_t     *handle,
       fastSurfaceFormat_t      surfaceFmt,
       unsigned     maxWidth,
       unsigned     maxHeight,
       fastDeviceSurfaceBufferHandle_t *dstBuffer)
```

Creates JPEG CPU Decoder and returns associated handle.

Parameters:

handle [out] – pointer to created JPEG CPU Decoder

surfaceFmt [in] – type of surface (decoded image). Surface is output for decoder.

maxWidth [in] – maximum image width in pixels

maxHeight [in] – maximum image height in pixels

dstBuffer [out] – pointer for linked buffer for next component (output buffer of current component)

Notes:

Function `fastJpegCpuDecoderCreate` allocates all necessary buffers in GPU memory. Thus in case GPU does not have enough free memory, then `fastJpegDecoderCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Only `FAST_RGB12` and `FAST_I12` surface formats are supported in other case `FAST_UNSUPPORTED_SURFACE` will be returned.

Maximum dimensions of the image are set to Decoder during creation. Thus if transformation result exceeds the maximum value, then error status `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNSUPPORTED_SURFACE`

fastJpegCpuDecoderGetAllocatedGpuMemorySize

```
fastStatus_t fastJpegCpuDecoderGetAllocatedGpuMemorySize(
        fastJpegDecoderHandle_t       handle,
        unsigned                      *requestedGpuSizeInBytes)
```

Returns requested GPU memory for JPEG CPU Decoder.

Parameters:

handle [in] – JPEG Cpu Decoder handle

requestedGpuSizeInBytes [out] – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for JPEG Decoder.

Statuses:

`FAST_OK`

fastJpegCpuDecode

```
fastStatus_t fastJpegCpuDecode(
      fastJpegDecoderHandle_t  handle,
      unsigned char      *srcJpegStream,
      const long   jpegStreamSize,
      fastJfifInfo_t     *jfifInfo)
```

Decodes JPEG to surface.

Parameters:

handle [in] – pointer to JPEG Decoder.

srcJpegStream[in] – pointer to buffer with entire jpeg file.
jpegStreamSize[in] – buffer size in bytes.
jfifInfo [out] – pointer to `fastJfifInfo_t` struct that takes jpeg parameters of decoded file.

Notes:

The procedure takes JPEG file from `srcJpegStream`. There are no additional parameters necessary for decoding. Decoded surface is placed to output linked buffer and the following component of the pipeline consumes it. Struct `fastJfifInfo_t` is populated by `fastJpegCpuDecode` with parameters of decoded jpeg file.

Fields populated in `fastJfifInfo_t` are `width, height, bitsPerChannel, huffmanState (for luminance and chrominance), quantState (for luminance and chrominance), jpegFmt, restartInterval.`

Decoder returns `FAST_IO_ERROR` if 8-bit jpeg will be supplied as input.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_BITSTREAM_CORRUPT`

fastJpegDecoderDestroy

```
fastStatus_t fastJpegCpuDecoderDestroy(fastJpegCpuDecoderHandle_t handle)
```

Destroys JPEG CPU Decoder.

Parameters:

handle [in] – pointer to JPEG CPU Decoder

Notes:

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

JPEG I/O functions

fastJfifLoadFromFile

```
fastStatus_t fastJfifLoadFromFile(
      const char   *filename,
      fastJfifInfo_t     *jfifInfo)
```

Loads JPEG image from disk to memory.

Parameters:

filename [in] – path to JPEG file

jfifInfo [in] – pointer to struct with parsed JPEG file

Notes:

If JPEG file is not found, procedure returns ***FAST_IO_ERROR***. If format of JPEG file is not supported (for example 12-bit JPEG) function returns **FAST_UNSUPPORTED_FORMAT**. If any errors occur during file parsing function returns FAST_INVALID_FORMAT and puts the error description to stderr.

Buffer h_Bytestream in jfifInfo should be allocated before the procedure call and its size in Bytes should be set to bytestreamSize from jfifInfo. Buffer h_Bytestream should be allocated by fastMalloc. If size of h_Bytestream is smaller than size of bytestream of loaded image, then the function will return FAST_INVALID_SIZE.

Statuses:

FAST_OK, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_IO_ERROR, FAST_UNSUPPORTED_FORMAT, FAST_INSUFFICIENT_HOST_MEMORY.

fastJfifHeaderLoadFromFile

```
fastStatus_t fastJpegLoadFromFile(
      const char   *filename,
      fastJfifInfo_t     *jfifInfo)
```

Loads JPEG image header from disk to memory.

Parameters:

filename [in] – path to JPEG file

jfifInfo [in] – pointer to structure with parsed JPEG file

Notes:

If JPEG file is not found, procedure returns ***FAST_IO_ERROR***. If format of JPEG file is not supported (for example 12-bit JPEG) function returns **FAST_UNSUPPORTED_FORMAT**. If any errors occur during file parsing function returns FAST_INVALID_FORMAT and puts the error description to stderr.

Statuses:

FAST_OK, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_IO_ERROR, FAST_UNSUPPORTED_FORMAT, FAST_INSUFFICIENT_HOST_MEMORY.

fastJfifBytestreamLoadFromFile

```
fastStatus_t fastJpegLoadFromFile(
      const char   *filename,
      fastJfifInfo_t     *jfifInfo)
```

Loads JPEG image bytestream from disk to memory.

Parameters:

filename [in] – path to JPEG file

jfifInfo [in] – pointer to structure with parsed JPEG file

Notes:

Structure `fastJfifInfo_t` has to be populated by previous call `fastJpegLoadHeader*`.

Buffer `h_Bytestream` in `jfifInfo` should be allocated before the procedure call and its size in Bytes should be set to `bytestreamSize` from `jfifInfo`. Buffer `h_Bytestream` should be allocated by `fastMalloc`. If size of `h_Bytestream` is smaller than size of bytestream of loaded image, then the function will return `FAST_INVALID_SIZE`.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_IO_ERROR, FAST_UNSUPPORTED_FORMAT, FAST_INSUFFICIENT_HOST_MEMORY.`

fastJfifLoadFromMemory

```
fastStatus_t fastJfifLoadFromMemory(
      unsigned char      *inputStream,
      unsigned     inputStreamSize,
      fastJfifInfo_t     *jfifInfo)
```

Loads JPEG image from buffer into memory.

Parameters:

inputStream [in] – pointer to buffer with JPEG file

inputStreamSize [in] – size of buffer with JPEG file in Bytes

jfifInfo [in] – pointer to struct with parsed JPEG file

Notes:

If JPEG file format is not supported (for example 12-bit JPEG) function will return **`FAST_UNSUPPORTED_FORMAT`**. If any errors occurs during file parsing function will return `FAST_INVALID_FORMAT` and error description is put to `stderr`.

Buffer `h_Bytestream` in `jfifInfo` should be allocated before the procedure call and its size in Bytes should be set to `bytestreamSize` in `jfifInfo`. Buffer `h_Bytestream` should be allocated by `fastMalloc`. If size of `h_Bytestream` is smaller than size of bytestream of loaded image, then the function will return `FAST_INVALID_SIZE`.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNSUPPORTED_FORMAT, FAST_INSUFFICIENT_HOST_MEMORY.`

fastJfifLoadHeaderFromMemory

```
fastStatus_t fastJfifLoadFromMemory(
      unsigned char      *inputStream,
```

```
	unsigned	inputStreamSize,
	fastJfifInfo_t	*jfifInfo)
```

Loads JPEG image header from buffer into memory.

Parameters:

inputStream [in] – pointer to buffer with JPEG file

inputStreamSize [in] – size of buffer with JPEG file in Bytes

jfifInfo [in] – pointer to struct with parsed JPEG file

Notes:

If JPEG file format is not supported (for example 12-bit JPEG) function will return **FAST_UNSUPPORTED_FORMAT.** If any errors occurs during file parsing function will return FAST_INVALID_FORMAT and error description is put to stderr.

Statuses:

FAST_OK, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNSUPPORTED_FORMAT, FAST_INSUFFICIENT_HOST_MEMORY.

fastJfifLoadBytestreamFromMemory

```
fastStatus_t fastJfifLoadFromMemory(
	unsigned char	*inputStream,
	unsigned	inputStreamSize,
	fastJfifInfo_t	*jfifInfo)
```

Loads JPEG image bytestream from buffer into memory.

Parameters:

inputStream [in] – pointer to buffer with JPEG file

inputStreamSize [in] – size of buffer with JPEG file in Bytes

jfifInfo [in] – pointer to struct with parsed JPEG file

Notes:

Structure fastJfifInfo_t has to be populated by previous call fastJpegLoadHeader*.

Buffer h_Bytestream in jfifInfo should be allocated before the procedure call and its size in Bytes should be set to bytestreamSize in jfifInfo. Buffer h_Bytestream should be allocated by fastMalloc. If size of h_Bytestream is smaller than size of bytestream of loaded image, then the function will return FAST_INVALID_SIZE.

Statuses:

FAST_OK, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNSUPPORTED_FORMAT, FAST_INSUFFICIENT_HOST_MEMORY.

fastJfifStoreToFile

```
fastStatus_t fastJfifStoreToFile(
	const char	*filename,
	fastJfifInfo_t	*jfifInfo)
```

Serializes JPEG bytestream to file.

Parameters:

filename [in] – path to JPEG file

jfifInfo [in] – pointer to struct with parsed JPEG file

Statuses:

```
FAST_OK, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR,
FAST_UNKNOWN_ERROR,FAST_IO_ERROR
```

fastJfifStoreToMemory

```
fastStatus_t fastJfifStoreToMemory(
      unsigned char     *outputStream,
      unsigned     *outputStreamSize,
      fastJfifInfo_t     *jfifInfo)
```

Serializes JPEG bytestream to memory buffer.

Parameters:

outputStream [out] – pointer to buffer for JPEG format serialization. Buffer is allocated by customer application.

outputStreamSize [inout] – size of buffer for JPEG format serialization in Bytes

jfifInfo [in] – pointer to struct with parsed JPEG file

Notes:

Buffer for serialization has to be allocated before call. Its size is *surfaceHeight*surfaceWidth*numberOfChannels.*

Real size of serialized file will be returned in *outputStreamSize*.

Statuses:

```
FAST_OK, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR
```

Affine functions

fastAffineCreate

```
fastStatus_t fastAffineCreate(
      fastAffineHandle_t *handle,
      fastAffineType_t   affineType,
      unsigned    maxWidth,
      unsigned    maxHeight,
      fastDeviceSurfaceBufferHandle_tsrcBuffer,
      fastDeviceSurfaceBufferHandle_t*dstBuffer)
```

Creates Affine transformation component and returns associated handle.

Parameters:

handle [out] – pointer to created Affine component

affineType [in] – type of affine transformation

maxWidth [in] – maximum input image width in pixels

maxHeight [in] – maximum input image height in pixels

srcBuffer [in] – linked buffer from previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastAffineCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastAffineCreate` will return `FAST_INSUFFICIENT_DEVICE_MEMORY`.

There are 5 currently supported affine transformations: Flip, Flop, Rotate 180, Rotate 90 to left, Rotate 90 to right. All affine transformations are applicable for gray and for color images. Rotation 90 left and Rotation 90 right change image dimensions: width becomes height, height becomes width. So `maxWidth` and `maxHeight` of the following component have to be properly adjusted.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INTERNAL_ERROR, FAST_INVALID_SIZE, FAST_UNSUPPORTED_SURFACE`

fastAffineGetAllocatedGpuMemorySize

```
fastStatus_t fastAffineGetAllocatedGpuMemorySize(
      fastAffineHandle_t handle,
      unsigned    *allocatedGpuSizeInBytes)
```

Returns requested GPU memory for Affine transformation component.

Parameters:

handle [in] – Affine component handle

allocatedGpuSizeInBytes [out] – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for Affine component.

Statuses:

`FAST_OK`

fastAffineChangeSrcBuffer

```
fastStatus_t DLL fastAffineChangeSrcBuffer(
      fastAffineHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – Affine component handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastAffineTransform

```
fastStatus_t fastAffineTransform(
      fastAffineHandle_t handle
      unsigned     width,
      unsigned     height)
```

Performs current Affine transformation.

Parameters:

handle [in] – Affine component handle

width [in] – image width in pixels

height [in] – image height in pixels

Notes:

If image size is greater than maximum value on creation, then error status `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastAffineDestroy

```
fastStatus_t fastAffineDestroy(fastAffineHandle_t handle)
```

Destroys Affine component handle.

Parameters:

handle [in] – Affine component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

Crop functions

fastCropCreate

```
fastStatus_t fastCropCreate(
      fastCropHandle_t   *handle,
      unsigned     maxSrcWidth,
      unsigned     maxSrcHeight,
      unsigned     maxDstWidth,
      unsigned     maxDstHeight,
      fastDeviceSurfaceBufferHandle_tsrcBuffer,
      fastDeviceSurfaceBufferHandle_t *dstBuffer)
```

Creates Crop component and returns associated handle.

Parameters:

handle [out] – pointer to created Crop component

maxSrcWidth [in] – maximum input image width in pixels

maxSrcHeight [in] – maximum input image height in pixels

maxDstWidth [in] – maximum destination (cropped) image width in pixels

maxDstHeight [in] – maximum destination (cropped) image height in pixels

srcBuffer [in] – linked buffer from previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastCropCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastCropCreate` will return `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Parameter `maxDstWidth` *has to be* not more than `maxSrcWidth`, *and* `maxDstHeight` has to be not more than `maxSrcHeight`. In other case `fastCropCreate` will return `FAST_INVALID_SIZE`.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INTERNAL_ERROR, FAST_INVALID_SIZE, FAST_UNSUPPORTED_SURFACE`

fastCropGetAllocatedGpuMemorySize

```
fastStatus_t fastCropGetAllocatedGpuMemorySize(
      fastCropHandle_t   handle,
      unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory for Crop component.

Parameters:

handle [in] – Crop component handle

allocatedGpuSizeInBytes [out] – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for Crop component.

Statuses:

`FAST_OK`

fastCropChangeSrcBuffer

```
fastStatus_t DLL fastCropChangeSrcBuffer(
      fastCropHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – Crop component handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastCropTransform

```
fastStatus_t fastCropTransform(
      fastCropHandle_t   handle,
      unsigned     width,
      unsigned     height,
      unsigned     leftTopCoordsX,
      unsigned     leftTopCoordsY,
      unsigned     croppedWidth,
      unsigned     croppedHeight)
```

Performs current Crop transformation.

Parameters:

handle [in] – Crop component handle

width [in] – input image width in pixels

height [in] – input image height in pixels

leftTopCoordsX [in] – coordX in pixels for left top corner of cropped image in input image

leftTopCoordsY [in] – coordY in pixels for left top corner of cropped image in input image

croppedWidth [in] – cropped image width in pixels

croppedHeight [in] – cropped image width in pixels

Notes:

If size of input image or size of cropped image greater than maximum value on creation error status `FAST_INVALID_SIZE` will be returned.

Also `leftTopCoordsX + croppedWidth` has to be not more than `width` and `leftTopCoordsY + croppedHeight` has to be not more than `height`. In other case function returns `FAST_INVALID_SIZE`

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastCropDestroy

```
fastStatus_t fastCropDestroy(
      fastCropHandle_t handle)
```

Destroys Crop component handle.

Parameters:

handle [in] – Crop component handle

Notes:

Procedure frees all device memory

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

Image Filter functions

fastImageFilterCreate

```
fastStatus_t fastImageFilterCreate(
            fastImageFiltersHandle_t *handle,
            fastImageFilterType_t    filterType,
            void  *staticFilterParameters,
            unsigned     maxWidth,
            unsigned     maxHeight,
            fastDeviceSurfaceBufferHandle_tsrcBuffer,
            fastDeviceSurfaceBufferHandle_t*dstBuffer)
```

Creates ImageFilter component and returns associated handle.

Parameters:

handle [out] – pointer to created ImageFilter component

filterType [in] – type of image filter

staticFilterParameters [in] – static parameters for image filter

maxWidth [in] – maximum image width in pixels

maxHeight [in] – maximum image height in pixels

srcBuffer [in] – linked buffer from previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastImageFilterCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastImageFilterCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

Next table shows structure type for `staticFilterParameters` depends on filter type . It also shows can static parameter be null or not.

Filter Type	Static Parameter Type	Can be NULL
GAUSSIAN_SHARPEN	no	yes
MAD	fastMad_t	yes
MAD16	fastMad16_t	yes
BAYER_BLACK_SHIFT	fastBayerBlackShift_t	yes
BASE_COLOR_CORRECTION	fastBaseColorCorrection_t	yes
WHITE_BALANCE	fastWhiteBalance_t	yes
TONE_CURVE	fastToneCurve_t	yes
COLOR_SATURATION_{HSL,	fastColorSaturation_t	yes

HSV}		
LUT_8_8	fastLut_8_t	yes
LUT_8_8_C	fastLut_8_C_t	yes
LUT_8_{12,16}	fastLut_8_16_t	yes
LUT_8_{12,16}_C	fastLut_8_16_C_t	yes
LUT_12_8	fastLut_12_8_t,	yes
LUT_12_8_C	fastLut_12_8_C_t,	yes
LUT_12_{12,16}	fastLut_12_t,	yes
LUT_12_{12,16}_C	fastLut_12_C_t	yes
LUT_16_8	fastLut_16_8_t	yes
LUT_16_8_C	fastLut_16_8_C_t	yes
LUT_16_16	fastLut_16_t	yes
LUT_16_16_C	fastLut_16_C_t	yes
RGB_LUT_3D	fastRGBLut_3D_t	yes
HSV_LUT_3D	fastHSVLut_3D_t	yes

Filter `FAST_GAUSSIAN_SHARPEN` has no static parameters, so `staticFilterParameters` has to be null.

Structure `fastMad_t` is a static parameter for `FAST_MAD` filter.

```
typedef struct{
      unsigned char      *blackShiftMatrix;
      float *correctionMatrix;
} fastMad_t

typedef struct{
      unsigned short     *blackShiftMatrix;
      float *correctionMatrix;
} fastMad16_t
```

Where

blackShiftMatrix – pointer to black shift matrix (or B matrix)

correctionMatrix – pointer to correction matrix (or A matrix)

Matrices have to be allocated by `fastMalloc`. Matrix's height is equal to image height. Matrix's width is equal to image width snapped up to the nearest four fold value. After image filter destroy matrices, they have to be deallocated by `fastFree`.

Structure `fastBayerBlackShift_t` is a static parameter for `BASE_COLOR_CORRECTION filter.`

```
typedef struct {
```

```
      float R;
      float G;
      float B;
      fastBayerPattern_t bayerPattern;
} fastBayerBlackShift_t;
```

where

R – shift constant for R channel

G – shift constant for G channel

B – shift constant for B channel

bayerPattern – bayer pattern

Structure `fastBaseColorCorrection_t` is a static parameter for `FAST_BASE_COLOR_CORRECTION` filter.

```
typedef struct{
      float matrix[12];
      int whiteLevel[3];
} fastBaseColorCorrection_t
```

where

matrix – color correction matrix

whiteLevel – white level for RGB. If current pixel value greater than white value then white value will be taken.

Structure `fastWhiteBalance_t` is a static parameter for `FAST_WHITE_BALANCE` filter.

```
typedef struct{
      float R;
      float G1;
      float G2;
      float B;
      fastBayerPattern_t bayerPattern;
} fastWhiteBalance_t
```

where

R,G1,G2,B – values of white balance matrix

bayerPattern – bayer pattern

Structure `fastColorSaturation_t` is a static parameter for `COLOR_SATURATION_{HSL, HSV}` filter.

```
typedef struct {
      float Lut[3][1024];
      fastColorSaturationOperationType_t operation[3];
      fastColorSaturationChannelType_t sourceChannel[3];
} fastColorSaturation_t
```

where

sourceChannel[3] – define channel associated with index. If *sourceChannel[0]* == *FAST_CHANNEL_H,* then *operation[0], Lut[0]* contain information for H channel.

Lut[3] – transformation Luts.

operation[3] – transformation operation

Structure `fastRGBLut_3D_t` is a static parameter for `FAST_RGB_LUT_3D` filter.

```
typedef struct {
	fastRGB16_t *lut;
	unsigned size1D;
} fastRGBLut_3D_t
```

where

size1D – linear cube size

lut – pointer to 3D LUT.

Ordering of elements in 3D LUT is the opposite to the typical in-memory order of multi-dimensional tables. An equivalent index would be r + N * g + N * N * b, where r, g, b are the Red, Green, and Blue indices in the range from 0 to N–1.

Memory for 3D LUT have to be allocated by `fastMalloc`. After image filter destroy matrices, they have to be deallocated by `fastFree`.

Structure `fastHSVLut_3D_t` is a static parameter for `FAST_HSV_LUT_3D` filter.

```
typedef struct {
	unsigned int dimH;
	unsigned int dimS;
	unsigned int dimV;

	fastColorSaturationOperationType_t operationH;
	fastColorSaturationOperationType_t operationS;
	fastColorSaturationOperationType_t operationV;
	fastHSVfloat_t *Lut;
} fastHsvLut3D_t
```

where

dimH – number of elements per H axis

dimS – number of elements per S axis

dimV – number of elements per V axis. To define 2D LUT, *dimV* has to be 1.

operationH – transformation operation for H channel

operationS – transformation operation for S channel

operationV – transformation operation for V channel

lut – pointer to 2D/3D LUT.

Ordering elements in 3D LUT is the opposite to the typical in-memory order of multi-dimensional tables. An equivalent index would be V + dimV * H + dimV * dimH * S.

Memory for 3D LUT has to be allocated by `fastMalloc`. After image filter destroy matrices, they have to be deallocated by `fastFree`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNSUPPORTED_SURFACE`

fastImageFiltersGetAllocatedGpuMemorySize

```
fastStatus_t fastImageFiltersGetAllocatedGpuMemorySize(
      fastImageFiltersHandle_t handle,
            unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory for ImageFilter component.

Parameters:

handle [in] – ImageFilter handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for ImageFilter component.

Statuses:

`FAST_OK`

fastImageFiltersChangeSrcBuffer

```
fastStatus_t DLL fastImageFiltersChangeSrcBuffer(
      fastImageFiltersHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – Image Filter handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastImageFiltersTransform

```
fastStatus_t fastImageFiltersTransform(
            fastImageFiltersHandle_t handle,
            void   *filterParameters,
            unsigned     width,
            unsigned     height)
```

Perform current ImageFilter transformation.

Parameters:

handle [in] – ImageFilter component handle

filterParameters [in] – filter parameters for current image

width [in] – image width in pixels

height [in] – image height in pixels

Notes:

If image size is greater than maximum value on creation error status `FAST_INVALID_SIZE` will be returned.

Pointer `filterParameters` can point to the following structures: `fastGaussianFilter_t`, `fastBaseColorCorrection_t`, `fastWhiteBalance_t`, `fastToneCurve_t`, `fastColorSaturation_t`, `fastMad_t`, `fastMad16_t` and to all LUT structures.

Structure `fastGaussianFilter` is dynamic parameter for `FAST_GAUSSIAN_SHARPEN` filter.

```
typedef struct{
      double sigma;
} fastGaussianFilter_t
```

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastImageFiltersDestroy

```
fastStatus_t fastImageFiltersDestroy(
      fastImageFiltersHandle_t handle)
```

Destroys ImageFilter component handle.

Parameters:

handle [in] – ImageFilter component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

Resize functions

fastResizerCreate

```
fastStatus_t fastResizerCreate(
            fastResizerHandle_t      *handle,
            unsigned    maxSrcWidth,
            unsigned    maxSrcHeight,
            unsigned    maxDstWidth,
            unsigned    maxDstHeight,
            double      maxScaleFactor,
            float shiftX,
            float shiftY,
            fastDeviceSurfaceBufferHandle_t srcBuffer,
            fastDeviceSurfaceBufferHandle_t *dstBuffer)
```

Creates Resize and returns associated handle.

Parameters:

handle [out] – pointer to created Resizer component

maxSrcWidth [in] – maximum input image width in pixels

maxSrcHeight [in] – maximum input image height in pixels

maxDstWidth [in] – maximum destination (cropped) image width in pixels

maxDstHeight [in] – maximum destination (cropped) image height in pixels

maxScaleFactor [in] – maximum scale factor (relation between source and destination dimensions)

shiftX [in] – shift between source and destination grids by x coordinate. Currently ignored, should be 0,0.

shiftY [in] – shift between source and destination grids by y coordinate. Currently ignored, should be 0,0.

srcBuffer [in] – linked buffer from previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastResizeCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastResizeCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

Parameter `maxDstWidth` has to be not more than `maxSrcWidth`, *and* `maxDstHeight` has to be not more than `maxSrcHeight`. In other case `fastResizeCreate` returns `FAST_INVALID_SIZE`. Also `maxSrcWidth/maxDstWidth` and `maxSrcHeight/maxDstHeight` have to be less or equal to `maxScaleFactor`. In other cases `fastResizeCreate` returns `FAST_INVALID_SIZE`.

If resize component is used only for upscaling `maxScaleFactor` should be set at any value greater than one.

Min values for `maxDstWidth` and `maxDstHeight` are 32.

Max value for `maxScaleFactor` is 40.

Max values for `maxSrcWidth` and `maxSrcHeight` are 65536.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INTERNAL_ERROR, FAST_INVALID_SIZE, FAST_UNSUPPORTED_SURFACE`

fastResizerGetAllocatedGpuMemorySize

```
fastStatus_t fastResizerGetAllocatedGpuMemorySize(
      fastResizerHandle_t      handle,
      unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory for Resizer component.

Parameters:

handle [in] – Resizer handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for Resizer component.

Statuses:

`FAST_OK`

fastResizerChangeSrcBuffer

```
fastStatus_t DLL fastResizerChangeSrcBuffer(
      fastResizerHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – Resizer handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastResizerTransform

```
fastStatus_t fastResizerTransform(
            fastResizerHandle_t      handle,
            fastResizeType_t   resizeType,
            unsigned     width,
            unsigned     height,
            unsigned     resizedWidth,
            unsigned     *resizedHeight)
```

Resizes current image with preserving aspect ratio.

Parameters:

handle [in] – Resizer handle

resizeType [in] – type of resize. Currently only `FAST_LANCZOS` resize is supported.

width [in] – input image width in pixels

height [in] – input image height in pixels

resizedWidth [in] – width of resized image in pixels

resizedHeight [out] – height of resized image in pixels

Notes:

If size of input image or size of resized image are greater than maximum value on creation error status `FAST_INVALID_SIZE` will be returned.

Height of resized image is calculated by the function and then customer application gets it in `resizedHeight`.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastResizerTransformStretch

```
fastStatus_t fastResizerTransformStretch(
      fastResizerHandle_t      handle,
      fastResizeType_t   resizeType,
      unsigned    width,
      unsigned    height,
      unsigned    resizedWidth,
      unsigned    resizedHeight)
```

Resizes current image without preserving aspect ratio.

Parameters:

handle [in] – Resizer handle

resizeType [in] – type of resize. Currently only `FAST_LANCZOS` resize is supported.

width [in] – input image width in pixels

height [in] – input image height in pixels

resizedWidth [in] – width of resized image in pixels

resizedHeight [in] – height of resized image in pixels

Notes:

If size of input image or size of resized image are greater than maximum value on creation error status `FAST_INVALID_SIZE` will be returned.

Function allows upscale one dimension and downscale other dimension.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastResizerDestroy

```
fastStatus_t fastResizerDestroy(fastResizerHandle_t handle)
```

Destroys Resizer component.

Parameters:

handle [in] – Resizer component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE`

Bayer Splitter functions

fastBayerSplitterCreate

```
fastStatus_t fastBayerSplitterCreate(
      fastBayerSplitterHandle_t      *handle,
      unsigned     maxSrcWidth,
      unsigned     maxSrcHeight,
      unsigned     *maxDstWidth,
      unsigned     *maxDstHeight,
      fastDeviceSurfaceBufferHandle_t srcBuffer,
      fastDeviceSurfaceBufferHandle_t *dstBuffer)
```

Creates BayerSplitter component and returns associated handle.

Parameters:

handle [out] – pointer to created BayerSplitter component

maxSrcWidth [in] – maximum input image width in pixels

maxSrcHeight [in] – maximum input image height in pixels

maxDstWidth [out] – maximum width of splitted Bayer image in pixels

maxDstHeight [out] – maximum height of splitted Bayer image in pixels

srcBuffer [in] – linked buffer from previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastBayerSplitterCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, `fastBayerSplitterCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

Output parameters `maxDstWidth` and `maxDstHeight` are used by next pipeline component to determine its maxWidth and maxHeight. Intended next component is JPEG Encoder.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INTERNAL_ERROR, FAST_INVALID_SIZE, FAST_UNSUPPORTED_SURFACE`

fastBayerSplitterGetAllocatedGpuMemorySize

```
fastStatus_t fastBayerSplitterGetAllocatedGpuMemorySize(
      fastBayerSplitterHandle_t      handle,
      unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory for BayerSplitter component.

Parameters:

handle [in] – BayerSplitter component handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for BayerSplitter component.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE`

fastBayerSplitterChangeSrcBuffer

```
fastStatus_t DLL fastBayerSplitterChangeSrcBuffer(
      fastBayerSplitterHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – BayerSplitter component handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastBayerSplitterSplit

```
fastStatus_t fastBayerSplitterSplit(
      fastBayerSplitterHandle_t      handle,
      unsigned     srcWidth,
      unsigned     srcHeight,
      unsigned     *dstWidth,
      unsigned     *dstHeight)
```

Function splits Bayer image on four planes.

Parameters:

handle [in] – BayerSplitter component handle

srcWidth [in] – maximum input image width in pixels

srcHeight [in] – maximum input image height in pixels

dstWidth [out] – width of splitted Bayer image in pixels

dstHeight [out] – height of splitted Bayer image in pixels

Notes:

If image size is greater than maximum value on creation error status, `FAST_INVALID_SIZE` will be returned.

Output parameters `dstWidth` and `dstHeight` are passed to the next component.

Statuses:

`FAST_OK FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastBayerSplitterDestroy

```
fastStatus_t fastBayerSplitterDestroy(fastBayerSplitterHandle_t handle)
```

Destroys BayerSplitter component.

Parameters:

handle [in] – BayerSplitter component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

Bayer Merger functions

fastBayerMergerCreate

```
fastStatus_t fastBayerMergerCreate(
      fastBayerMergerHandle_t  *handle,
      unsigned    maxDstWidth,
      unsigned    maxDstHeight,
      fastDeviceSurfaceBufferHandle_tsrcBuffer,
      fastDeviceSurfaceBufferHandle_t*dstBuffer)
```

Creates BayerMerger component and returns associated handle.

Parameters:

handle [out] – pointer to created BayerMerger component

maxDstWidth [in] – maximum width of restored Bayer image in pixels

maxDstHeight [in] – maximum height of restored Bayer image in pixels

srcBuffer [in] – linked buffer from the previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastBayerMergerCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastBayerMergerCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

The function is different from other create functions because it takes maximum size of output image. BayerMerger takes splitted Bayer image and transforms it to normal Bayer image. It is more convenient for user to operate with size of restored (output) than splitted (input) image.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INTERNAL_ERROR, FAST_INVALID_SIZE`

fastBayerMergerGetAllocatedGpuMemorySize

```
fastStatus_t fastBayerMergerGetAllocatedGpuMemorySize(
      fastBayerMergerHandle_t  handle,
      unsigned     *requestedGpuSizeInBytes)
```

Restores Bayer image from splitted image.

Parameters:

handle [in] – BayerMerger component handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for BayerMerger component.

Statuses:

`FAST_OK`

fastBayerMergerChangeSrcBuffer

```
fastStatus_t DLL fastBayerMergerChangeSrcBuffer(
      fastBayerMergerHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – BayerMerger component handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastBayerMergerMerge

```
fastStatus_t fastBayerMergerMerge(
      fastBayerMergerHandle_t  handle,
      unsigned     width,
      unsigned     height)
```

Returns requested GPU memory for BayerMerger component.

Parameters:

handle [in] – BayerMerger component handle

width [in] – restored (original) image width in pixels

height [in] – restored (original) image height in pixels

Notes:

If image size is greater than maximum value on creation error status `FAST_INVALID_SIZE` will be returned.

Restored Image width and height are taken from EXIF section, defined by `SplitterExif_t` structure in `ExifInfo.hpp`. Section is parsed by `ParseSplitterExif` function from `ExifInfo.hpp.`

Statuses:

`FAST_OK`

fastBayerMergerDestroy

```
fastStatus_t fastBayerMergerDestroy(fastBayerMergerHandle_t handle)
```

Destroys BayerMerger component.

Parameters:

handle [in] – BayerMerger component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

Timer functions

fastGpuTimerCreate

```
fastStatus_t fastGpuTimerCreate(fastGpuTimerHandle_t *handle)
```

Creates Timer and returns associated handle.

Parameters:

handle [out] – pointer to created Timer handle

Notes:

Allocates necessary buffers in GPU memory. In case GPU does not have enough free memory returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR`

fastGpuTimerStart

```
fastStatus_t fastGpuTimerStart(fastGpuTimerHandle_t handle)
```

Inserts start event into GPU stream.

Parameters:

handle [in] – Timer handle pointer

Notes:

Inserts start event into GPU stream.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR`

fastGpuTimerStop

```
fastStatus_t fastGpuTimerStop(fastGpuTimerHandle_t handle)
```

Inserts stop event into GPU stream.

Parameters:

handle [in] – Timer handle pointer

Notes:

Inserts stop event into GPU stream.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR`

fastGpuTimerGetTime

```
fastStatus_t fastGpuTimerGetTime(
      fastGpuTimerHandle_t          *handle,
      float                         *elapsed)
```

Synchronizes CPU thread with stop event and calculates time elapsed between start and stop events.

Parameters:

handle [in] – Timer handle pointer

elapsed [out] – time elapsed

Notes:

Synchronizes CPU thread with stop event and calculates time elapsed between start and stop events.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR`

fastGpuTimerDestroy

```
fastStatus_t fastDestroyGpuTimerHandle(fastGpuTimerHandle_t handle)
```

Destroys Timer handle.

Parameters:

handle [in] – pointer to Timer handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

Mux functions

fastMuxCreate

```
fastStatus_t fastMuxCreate(
            fastMuxHandle_t    *handle,
            fastDeviceSurfaceBufferHandle_t* srcBuffers,
            unsigned     numberOfInputs,
            fastDeviceSurfaceBufferHandle_t *dstBuffer)
```

Creates Mux and returns associated handle.

Parameters:

handle [out] – pointer to created Mux handle

srcBuffers [in] – array of linked buffer from previous component

numberOfInputs [in] – element count in *srcBuffers*

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Allocates necessary buffers in GPU memory. In case GPU does not have enough free memory returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

All input buffers have to be same size and type, else function returns `FAST_INVALID_FORMAT`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_FORMAT`

fastMuxSelect

```
fastStatus_t fastMuxSelect(
            fastMuxHandle_t    handle,
            unsigned     srcBufferIndex)
```

Selects specified input and passes it to the output.

Parameters:

handle [in] – Mux handle

srcBufferIndex [in] – index of selected input

Notes:

Index is zero-based numbering. It has to be less than *numberOfInputs* in Create function, else function returns `FAST_INVALID_SIZE`.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INVALID_SIZE`

fastMuxDestroy

```
fastStatus_t fastMuxDestroy(fastMuxHandle_t handle);
```

Destroys Mux.

Parameters:

handle [in] – Mux handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE`

SDI import and export

FastSDIImportFromHostCreate / fastSDIImportFromDeviceCreate

```
fastStatus_t fastSDIImportFromHostCreate(
            fastSDIImportFromHostHandle_t  *handle,
            fastSDIFormat_t    sdiFmt,
            unsigned    maxWidth,
            unsigned    maxHeight,
            fastDeviceSurfaceBufferHandle_t *dstBuffer)

fastStatus_t fastSDIImportFromDeviceCreate(
            fastSDIImportFromDeviceHandle_t      *handle,
            fastSDIFormat_t    sdiFmt,
            unsigned    maxWidth,
            unsigned    maxHeight,
            fastDeviceSurfaceBufferHandle_t *dstBuffer)
```

Creates SDI Import component and returns associated handle.

Parameters:

handle [out] – pointer to created SDI Import handle

sdiFmt [in] – SDI format

maxWidth [in] – maximum width of image in pixels

maxHeight [in] – maximum height of image in pixels

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Allocates necessary buffers in GPU memory. In case GPU does not have enough free memory returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If component does not support current surface format, then the function will return `FAST_UNSUPPORTED_SURFACE`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_FORMAT, FAST_UNSUPPORTED_SURFACE`

fastSDIExportToHostCreate / fastSDIExportToDeviceCreate

```
fastStatus_t fastSDIExportToHostCreate(
            fastSDIExportToHostHandle_t    *handle,
            fastSDIFormat_t    sdiFmt,
            fastSurfaceFormat_t *surfaceFmt,
            unsigned    maxWidth,
            unsigned    maxHeight,
      fastDeviceSurfaceBufferHandle_t srcBuffer)

fastStatus_t fastSDIExportToDeviceCreate(
            fastSDIExportToDeviceHandle_t  *handle,
            fastSDIFormat_t    sdiFmt,
            fastSurfaceFormat_t *surfaceFmt,
            unsigned    maxWidth,
            unsigned    maxHeight,
      fastDeviceSurfaceBufferHandle_t srcBuffer)
```

Creates SDI Export component and returns associated handle.

Parameters:

handle [out] – pointer to created SDI Import handle

sdiFmt [in] – SDI format

surfaceFmt [out] – return format of input surface

maxWidth [in] – maximum width of image in pixels

maxHeight [in] – maximum height of image in pixels

srcBuffer [in] – linked buffer from previous component

Notes:

Allocates necessary buffers in GPU memory. In case GPU does not have enough free memory returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If srcBuffer format is incompatible with selected sdiFmt, then function returns `FAST_INVALID_FORMAT`.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_FORMAT`

fastSDIImportFromHostGetAllocatedGpuMemorySize

```
fastStatus_t fastSDIImportFromHostGetAllocatedGpuMemorySize(
      fastSDIImportFromHostHandle_t  handle,
      unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory size for SDI Import component.

Parameters:

handle [in] – SDI Import component handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for SDI Import component.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE`

fastSDIImportFromDeviceGetAllocatedGpuMemorySize

```
fastStatus_t fastSDIImportFromDeviceGetAllocatedGpuMemorySize(
      fastSDIImportFromDeviceHandle_thandle,
      unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory size for SDI Import component.

Parameters:

handle [in] – SDI Import component handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for SDI Import component.

Statuses:

FAST_OK, FAST_INVALID_HANDLE

fastSDIExportToHostGetAllocatedGpuMemorySize

```
extern fastStatus_t DLL fastSDIExportToHostGetAllocatedGpuMemorySize(
      fastSDIExportToHostHandle_t    handle,
      unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory size for SDI Export component.

Parameters:

handle [in] – SDI Export component handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for SDI Export component.

Statuses:

FAST_OK, FAST_INVALID_HANDLE

fastSDIExportToDeviceGetAllocatedGpuMemorySize

```
extern fastStatus_t DLL fastSDIExportToDeviceGetAllocatedGpuMemorySize(
      fastSDIExportToDeviceHandle_t  handle,
      unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory size for SDI Export component.

Parameters:

handle [in] – SDI Export component handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for SDI Export component.

Statuses:

FAST_OK, FAST_INVALID_HANDLE

fastSDIImportFromHostCopy / fastSDIImportFromDeviceCopy

```
fastStatus_t fastSDIImportFromHostCopy(
            fastSDIImportFromHostHandle_t  handle,
            void* h_src,
            unsigned     width,
            unsigned     height)

fastStatus_t fastSDIImportFromDeviceCopy(
            fastSDIImportFromDeviceHandle_thandle,
            void* d_src,
            unsigned     width,
            unsigned     height)
```

Loads SDI formatted image to the pipeline.

Parameters:

handle [in] – SDI Import component handle

h_src [in] – SDI formatted image located on host

d_src [in] – SDI formatted image located on device

width [in] – image width in pixels

height [in] – image height in pixels

Notes:

Buffer *h_src* has to be allocated by `fastMalloc`. Buffer allocated by original `malloc` also can be used, but copy speed will degrade.

Buffer `d_src` has to be allocated in Device memory by `cudaMalloc`. If size of `d_dst` is not enough, then function will fail with segmentation fault.

If image size is greater than maximum value on creation, then error status `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastSDIExportToHostCopy / fastSDIExportToDeviceCopy

```
fastStatus_t fastSDIExportToHostCopy(
            fastSDIExportToHostHandle_t    handle,
            void* h_dst,
            unsigned     *width,
            nsigned      *height);

fastStatus_t fastSDIExportToDeviceCopy(
            fastSDIExportToDeviceHandle_t  handle,
            void* d_dst,
            unsigned     *width,
            nsigned      *height);
```

Exports SDI formatted image from pipeline to host memory.

Parameters:

handle [in] – SDI Export component handle

h_dst [out] – host buffer for exported SDI formatted image

d_dst [out] – device buffer for exported SDI formatted image

width [out] – image width in pixels

height [out] – image height in pixels

Notes:

Buffer size in Bytes can be calculated by `GetSDIBufferSize` function from `HelperSDI.hpp` (part of SDIConverterSample application).

Buffer h_dst has to be allocated by `fastMalloc`. Buffer allocated by original `malloc` also can be used, but copy speed will degrade.

Buffer `d_dst` has to be allocated in Device memory by `cudaMalloc`. If size of `d_dst` is not enough, then function will fail with segmentation fault.

User has to estimate width and height of export image and allocate buffer according to these values. Function returns real width and height.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR`

fastSDIImportToHostCopy3 / fastSDIImportToDeviceCopy3

```
extern fastStatus_t DLL fastSDIImportFromHostCopy3(
```

```
        fastSDIImportFromHostHandle_t handle,

        fastChannelDescription_t *srcY,
        fastChannelDescription_t *srcU,
        fastChannelDescription_t *srcV
);

extern fastStatus_t DLL fastSDIImportFromDeviceCopy3(
        fastSDIImportFromDeviceHandle_t handle,

        fastChannelDescription_t *srcY,
        fastChannelDescription_t *srcU,
        fastChannelDescription_t *srcV
);
```

Loads YV12 image from three separate buffers to the pipeline.

Parameters:

handle [in] – SDI Import component handle

srcY [in] – Y plane of YV12 image

srcU [in] – Cb plane of YV12 image

srcV [in] – Cr plane of YV12 image

Notes:

Srtucture *fastChannelDescription_t* defines size of plane.

```
typedef struct {
        unsigned char *data;
        unsigned width;
        unsigned pitch;
        unsigned height;
} fastChannelDescription_t;
```

Where

data – plane buffer. It has to be allocated by `fastMalloc` for host version and `cudaMalloc` for device version

width – plane width

height – plane height

pitch – size of image raw in byte. Normally aligned by 4.

If even one image plane has incorrect size , then error status `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK`, `FAST_INVALID_HANDLE`, `FAST_INTERNAL_ERROR`, `FAST_UNKNOWN_ERROR`, `FAST_INVALID_SIZE`

fastSDIExportToHostCopy3 / fastSDIExportToDeviceCopy3

```
fastStatus_t fastSDIExportToHostCopy3(
        fastSDIExportToHostHandle_t handle,

        fastChannelDescription_t *dstY,
        fastChannelDescription_t *dstU,
        fastChannelDescription_t *dstV
);
```

```
fastStatus_t fastSDIExportToDeviceCopy3(
      fastSDIExportToDeviceHandle_t handle,

      fastChannelDescription_t *dstY,
      fastChannelDescription_t *dstU,
      fastChannelDescription_t *dstV
);
```

Exports YV12 image to three separate buffers from the pipeline.

Parameters:

handle [in] – SDI Export component handle

srcY [in] – Y plane of YV12 image

srcU [in] – Cb plane of YV12 image

srcV [in] – Cr plane of YV12 image

If even one image plane has incorrect size , then error status `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, AST_INVALID_SIZE`

fastSDIImportFromHostDestroy / fastSDIImportFromDeviceDestroy

```
fastStatus_t fastSDIImportFromHostDestroy(fastSDIImportFromHostHandle_t handle)

fastStatus_t fastSDIImportFromDeviceDestroy(fastSDIImportFromDeviceHandle_t handle)
```

Destroys SDI Import component.

Parameters:

handle [in] – SDI Import component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE`

fastSDIExportToHostDestroy / fastSDIExportToDeviceDestroy

```
fastStatus_t fastSDIExportToHostDestroy(fastSDIExportToHostHandle_t handle)

fastStatus_t fastSDIExportToDeviceDestroy(fastSDIExportToDeviceHandle_t handle)
```

Destroys SDI Export component.

Parameters:

handle [in] – SDI Export component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE`

Surface converter

fastSurfaceConverterCreate

```
fastStatus_t fastSurfaceConverterCreate(
      fastSurfaceConverterHandle_t *handle,

      fastSurfaceConverter_t surfaceConverterType,
      void *staticSurfaceConverterParameters,

      unsigned maxWidth,
      unsigned maxHeight,

      fastDeviceSurfaceBufferHandle_t  srcBuffer,
      fastDeviceSurfaceBufferHandle_t *dstBuffer
);
```

Creates Surface Converter and returns associated handle.

Parameters:

handle [out] – pointer to created Surface Converter handle

surfaceConverterType[in] – surface converter type

staticFilterParameters [in] – static parameters for image filter

maxWidth [in] – maximum image width in pixels

maxHeight [in] – maximum image height in pixels

srcBuffer [in] – linked buffer from previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastSurfaceConverterCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastSurfaceConverterCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

List of supported structures for `staticFilterParameters`:

`fastBitDepthConverter_t`,

`fastSelectChannel_t`,

`fastRgbToGrayscale_t`

Structure `fastBitDepthConverter_t` is a static parameter for `FAST_BIT_DEPTH` filter.

```
typedef struct {
      unsigned bitsPerChannel;
} fastBitDepthConverter_t;
```

Where

bitsPerChannel – bit depth of destination surface.

Structure `fastSelectChannel_t` is a static parameter for `FAST_SELECT_CHANNEL` filter.

```
typedef struct {
      fastChannelType_t channel;
} fastSelectChannel_t;
```

Where

channel – selected RGB channel

Structure `fastRgbToGrayscale_t` is a static parameter for `FAST_RGB_TO_GRAYSCALE` filter.

```
typedef struct {
      float coefficientR;
      float coefficientG;
      float coefficientB;
} fastRgbToGrayscale_t;
```

Where

coefficientR – weight for R channel

coefficientG – weight for G channel

coefficientB – weight for B channel

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_VALUE, FAST_UNSUPPORTED_SURFACE`

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNSUPPORTED_SURFACE`

fastSurfaceConverterGetAllocatedGpuMemorySize

```
fastStatus_t fastSurfaceConverterGetAllocatedGpuMemorySize(
      fastSurfaceConverterHandle_t   handle,
      unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory for Surface Converter component.

Parameters:

handle [in] – Surface Converter handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for Surface Converter component.

Statuses:

`FAST_OK`

fastSurfaceConverterChangeSrcBuffer

```
fastStatus_t DLL fastSurfaceConverterChangeSrcBuffer(
      fastSurfaceConverterHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – Surface Converter handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastSurfaceConverterTransform

```
fastStatus_t fastSurfaceConverterTransform(
            fastSurfaceConverterHandle_t   handle,
            void *surfaceConverterParameters,
            unsigned     width,
            unsigned     height)
```

Perform current Surface Converter transformation.

Parameters:

handle [in] – Surface Converter handle

surfaceConverterParameters [in] – parameters for current image

width [in] – image width in pixels

height [in] – image height in pixels

Notes:

If image size is greater than maximum value on creation error status `FAST_INVALID_SIZE` will be returned.

Pointer `filterParameters` can point on the following structures:

`fastBitDepthConverter_t, fastSelectChannel_t, fastRgbToGrayscale_t`

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastSurfaceConverterDestroy

```
fastStatus_t fastSurfaceConverterDestroy(
      fastSurfaceConverterHandle_t handle)
```

Destroys Surface Converter component handle.

Parameters:

handle [in] – Surface Converter component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

Histogram functions

fastHistogramCreate

```
fastStatus_t fastHistogramsCreate(
             fastHistogramsHandle_t *handle,
             fastHistogramType_t histogramType,
             void *staticParameters,

             unsigned int bins,
             unsigned int maxWidth,
             unsigned int maxHeight,
             fastDeviceSurfaceBufferHandle_t  srcBuffer)
```

Creates Histogram and returns associated handle.

Parameters:

handle [out] – pointer to created Histogram component

histogramType [in] – histogram type

staticParameters [in] – pointer to static parameter.

bins [in] – number of bins in histogram. Number is power of two.

maxWidth [in] – maximum input image width in pixels

maxHeight [in] – maximum input image height in pixels

srcBuffer [in] – linked buffer from previous component

Notes:

Function `fastHistogramCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastHistogramCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

There is no static parameter for `FAST_HISTOGRAM_COMMON` type. Static parameter has to be `null`.

Structure `fastHistogramBayer_t` is static parameter for `FAST_HISTOGRAM_BAYER` and `FAST_HISTOGRAM_BAYER_G1G2` types.

```
typedef struct {
      fastBayerPattern_t bayerPattern;
} fastHistogramBayer_t
```

where `bayerPattern` is pattern for bayer filtered image.

Structure `fastHistogramParade_t` is static parameter for `FAST_HISTOGRAM_PARADE` type.

```
typedef struct {
      unsigned int stride;
} fastHistogramParade_t
```

where `stride` is step to next image column used for parade calculation.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INTERNAL_ERROR, FAST_INVALID_SIZE, FAST_UNSUPPORTED_SURFACE`

fastHistogramGetAllocatedGpuMemorySize

```
fastStatus_t fastHistogramGetAllocatedGpuMemorySize(
      fastHistogramHandle_t    handle,
      unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory for Histogram component.

Parameters:

handle [in] – Histogram handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for Histogram component.

Statuses:

`FAST_OK`

fastHistogramChangeSrcBuffer

```
fastStatus_t fastHistogramChangeSrcBuffer(
      fastHistogramHandle_t handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – Histogram handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastHistogramCalculate

```
fastStatus_t fastHistogramCalculate(
      fastHistogramHandle_t handle,
      void *histogramParameters,
      unsigned int roiLeftTopX,
      unsigned int roiLeftTopY,
      unsigned int roiWidth,
      unsigned int roiHeight,
      unsigned int *h_histogram )
```

Calculate histogram in ROI for current image.

Parameters:

handle [in] – Histogram handle

histogramParameters [in] – pointer to dynamic parameter.

roiLeftTopX [in] – column of top left corner of ROI.
roiLeftTopY [in] – row of top left corner of ROI.
roiWidth [in] – width of ROI

roiHeight [in] – height of ROI

h_histogram [out] – cpu buffer for calculated histogram

Notes:

If *roiLeftTopX* + *roiWidth* greater than image width or *roiLeftTopY* + *roiHeight* greater than image height error status `FAST_INVALID_SIZE` will be returned.

Values *roiLeftTopX, roiLeftTopY* have to be even in case of bayer histogram.

There is no dynamic parameter for `FAST_HISTOGRAM_COMMON` type. Structure `fastHistogramBayer_t` is dynamic parameter for `FAST_HISTOGRAM_BAYER` and `FAST_HISTOGRAM_BAYER_G1G2` types. Structure `fastHistogramParade_t` is dynamic parameter for `FAST_HISTOGRAM_PARADE` type.

Buffer *h_histogram* has to be allocated with `fastMalloc`. Buffer, which is allocated by original `malloc` also can be used, but copy speed will degrade. If size of *h_histogram* is not enough, then the function will fail with segmentation fault.

Number of elements in *h_histogram* is production of bin count on number of histogram. Elements of histogram is `int` value. Number of histogram depends on histogram type, surface format and image width(for some types). Number of histogram can be calculated by `GetHistogramCount` function in `HistogramSample`.

Function `FastHistogramCalculate` copy *h_histogram* buffer asynchronously. So when function finished buffer is not ready. It is necessary to use `cudaEvent` to wait until copy has been finished. See `HistogramSample`.

Format *h_histogram* depends on histogram type. For `FAST_HISTOGRAM_COMMON`, `FAST_HISTOGRAM_BAYER`, `FAST_HISTOGRAM_BAYER_G1G2` types *h_histogram* is array of histogram where each histogram is dense array of values. Histogram order in *h_histogram* for color image is R, G, B. Histogram order in *h_histogram* for Bayer filtered image is the same. Histogram order in *h_histogram* for Bayer filtered image for `FAST_HISTOGRAM_BAYER_G1G2` is R, G1, G2, B. Where G1 is green pixel in first line of pattern, G2 is green pixel in second line of pattern. For more details see `SaveHistogramToFile` in `HistogramSample`.

For `FAST_HISTOGRAM_PARADE` *h_histogram* is array of parade. Parade order is R, G, B. Parade contains first bin for each column then second bin for each column and so on. For more details see `SaveParadeToFile` in `HistogramSample`.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastHistogramDestroy

`fastStatus_t fastHistogramDestroy(fastHistogramHandle_t handle)`

Destroys Histogram component.

Parameters:

handle [in] – Histogram component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE`

JPEG2000 Encoder functions

fastEncoderJ2kCreate

```
fastStatus_t fastEncoderJ2kCreate (
    fastEncoderJ2kHandle_t *handle,
    fastEncoderJ2kStaticParameters_t *parameters,
    fastSurfaceFormat_t surfaceFmt,
    unsigned maxWidth,
    unsigned maxHeight,
    unsigned maxBatchSize,
    fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Creates JPEG2000 Encoder and returns associated handle.

Parameters:

handle [out] – pointer to the created JPEG2000 Encoder handle

parameters [in] – structure, which specifies encoder parameters (lossy/lossless compression algorithm, number of DWT levels, codeblock size, quality coefficient etc.)

surfaceFmt [in] – surface format of input image

maxWidth [in] – maximum image width in pixels

maxHeight [in] – maximum image height in pixels

maxBatchSize [in] – maximum number of simultaneously processed images in batch

srcBuffer [in] – linked buffer from the previous component

Notes:

The same encoder component and interface functions are used for both lossy and lossless compression.

Function `fastEncoderJ2kCreate` allocates all necessary buffers in GPU memory. So, if GPU does not have enough free memory, then `fastEncoderJ2kCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Maximum dimensions of the images should be specified for Encoder creation. Thus, if size of an input image exceeds the maximum size, then error status `FAST_INVALID_SIZE` will be returned.

If encoder does not support the specified surface format, then the function will return `FAST_UNSUPPORTED_SURFACE`.

Statuses:

`FAST_OK`, `FAST_INVALID_VALUE`, `FAST_UNSUPPORTED_SURFACE`, `FAST_INSUFFICIENT_DEVICE_MEMORY`, `FAST_INTERNAL_ERROR`, `FAST_UNKNOWN_ERROR`

fastEncoderJ2kGetAllocatedGpuMemorySize

```
fastStatus_t fastEncoderJ2kGetAllocatedGpuMemorySize (
      fastEncoderJ2kHandle_t handle,
      unsigned long long *allocatedGpuSizeInBytes)
```

Returns the GPU memory size allocated for JPEG2000 Encoder.

Parameters:

handle [in] – JPEG2000 Encoder handle

allocatedGpuSizeInBytes [out] – memory size in Bytes

Notes:

Statuses:

`FAST_OK`

fastEncoderJ2kTransform

```
fastStatus_t fastEncoderJ2kTransform (
      fastEncoderJ2kHandle_t handle,
      fastEncoderJ2kDynamicParameters_t *parameters,
      unsigned width,
      unsigned height,
      fastEncoderJ2kOutput_t *output,
      fastEncoderJ2kReport_t *report)
```

Encodes surface to JPEG2000 format, stores the result into host memory and returns report.

Parameters:

handle [in] – JPEG2000 Encoder handle

parameters [in] – structure, which specifies parameters for the current image: if JP2 file header is needed, quality coefficient, target stream size in bytes (optional)

width [in] – image width in pixels

height [in] – image height in pixels

output [in/out] – structure, where output JP2/J2K bytestream is copied to, taking into account the specified buffer size

report [out] – structure, where measured duration of each encoding stage and some other values (e.g., codeblock count) are written to

Notes:

The procedure takes surface from previous component of the pipeline through input linked buffer and encodes it accordingly to the parameters specified during encoder creation. JPEG2000 bytestream is placed to `output->byteStream` buffer.
Buffer for JPEG2000 bytestream must be allocated before call. Real JPEG2000 bytestream size is calculated during compression and put to `output->streamSize` field. If `bufferSize` is too small, then the stream is truncated, and the `truncated` flag is set.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INVALID_VALUE, FAST_INVALID_SIZE, FAST_INTERNAL_ERROR, FAST_EXECUTION_FAILURE, FAST_UNKNOWN_ERROR`

fastEncoderJ2kFreeSlotsInBatch

```
fastStatus_t fastEncoderJ2kFreeSlotsInBatch (
      fastEncoderJ2kHandle_t handle,
      int *value)
```

Returns the number of free slots in batch for JPEG2000 Encoder.

Parameters:

handle [in] – JPEG2000 Encoder handle

value [out] – number of free slots

Notes:

Statuses:

FAST_OK

fastEncoderJ2kUnprocessedImagesCount

```
fastStatus_t fastEncoderJ2kUnprocessedImagesCount (
      fastEncoderJ2kHandle_t handle,
      int *value)
```

Returns the number of unprocessed images in batch for JPEG2000 Encoder.

Parameters:

handle [in] – JPEG2000 Encoder handle

value [out] – number of unprocessed images

Notes:

Statuses:

FAST_OK

fastEncoderJ2kAddImageToBatch

```
fastStatus_t fastEncoderJ2kAddImageToBatch (
      fastEncoderJ2kHandle_t handle,
      fastEncoderJ2kDynamicParameters_t *parameters,
      unsigned width,
      unsigned height)
```

Adds image to queue for batch processing (if free slots are available) and returns immediately.

Parameters:

handle [in] – JPEG2000 Encoder handle

parameters [in] – structure, which specifies parameters for the current image: if JP2 file header is needed, quality coefficient, target stream size in bytes (optional)

width [in] – image width in pixels

height [in] – image height in pixels

Notes:

The procedure takes surface from the previous component of the pipeline through input linked buffer and adds it to queue for batch processing using `fastEncoderJ2kTransformBatch` procedure. When this procedure returns control, the input buffer can be used to store another image or be disposed.
The availability of free slots in the batch can be checked using `fastEncoderJ2kFreeSlotsInBatch` function.

Statuses:

FAST_OK, FAST_INVALID_HANDLE, FAST_INVALID_VALUE, FAST_INVALID_SIZE, FAST_INTERNAL_ERROR, FAST_EXECUTION_FAILURE, FAST_UNKNOWN_ERROR

fastEncoderJ2kTransformBatch

```
fastStatus_t fastEncoderJ2kTransformBatch (
     fastEncoderJ2kHandle_t handle,
     fastEncoderJ2kOutput_t *output,
     fastEncoderJ2kReport_t *report)
```

Encodes multiple surfaces to JPEG2000 format using batch processing.

Parameters:

handle [in] – JPEG2000 Encoder handle

output [in/out] – structure, where the first output JP2/J2K bytestream is copied to, taking into account the specified buffer size

report [out] – structure, where elapsed time is written to

Notes:

The procedure takes surfaces, which have been added using `fastEncoderJ2kAddImageToBatch` procedure, encodes them and stores the first of the resulted JPEG2000 bytestreams into host memory. The rest bytestreams should be obtained via `fastEncoderJ2kGetNextEncodedImage` function.
Buffer for the returned JPEG2000 bytestream must be allocated before call. Real JPEG2000 bytestream size is calculated during compression and put to `output->streamSize` field. If `bufferSize` is too small, then the stream is truncated, and the `truncated` flag is set.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INVALID_VALUE, FAST_INVALID_SIZE, FAST_INTERNAL_ERROR, FAST_EXECUTION_FAILURE, FAST_UNKNOWN_ERROR`

fastEncoderJ2kGetNextEncodedImage

```
fastStatus_t fastEncoderJ2kGetNextEncodedImage (
      fastEncoderJ2kHandle_t handle,
      fastEncoderJ2kOutput_t *output,
      fastEncoderJ2kReport_t *report
      int *imagesLeft
);
```

Returns the next compressed image during batch processing.

Parameters:

handle [in] – JPEG2000 Encoder handle

output [in/out] – structure, where the successive output JP2/J2K bytestream is copied to, taking into account the specified buffer size

imagesLeft [out] – number of compressed images, which can be returned by successive calls

report [out] – structure, where elapsed time is written to

Notes:

The procedure takes surfaces, which have been added using `fastEncoderJ2kAddImageToBatch` procedure, encodes them and stores the first of the resulted JPEG2000 bytestreams into host memory. The rest bytestreams should be obtained via `fastEncoderJ2kGetNextEncodedImage` function.
Buffer for the returned JPEG2000 bytestream must be allocated before call. Real JPEG2000 bytestream size is calculated during compression and put to `output->streamSize` field. If `bufferSize` is too small, then the stream is truncated, and the `truncated` flag is set.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_EXECUTION_FAILURE, FAST_UNKNOWN_ERROR`

fastEncoderJ2kDestroy

fastStatus_t fastEncoderJ2kDestroy (fastEncoderJ2kHandle_t handle)

Destroys JPEG2000 Encoder.

Parameters:

handle [in] – JPEG2000 Encoder handle

Notes:

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

JPEG2000 Decoder functions

fastDecoderJ2kCreate

```
fastStatus_t fastDecoderJ2kCreate (
      fastDecoderJ2kHandle_t *handle,
      fastDecoderJ2kStaticParameters_t *staticParameters,
      fastSurfaceFormat_t surfaceFmt,
      unsigned maxWidth,
      unsigned maxHeight,
      unsigned maxBatchSize,
      fastDeviceSurfaceBufferHandle_t *dstBuffer)
```

Creates JPEG Decoder and returns associated handle.

Parameters:

handle [out] – pointer to the created JPEG2000 Decoder handle

staticParameters [in] – structure, which specifies decoder parameters (number of DWT levels, maximum tile size, truncation parameters, Tier-1 errors detection, verbose levels etc.)

surfaceFmt [in] – type of surface for decoded image

maxWidth [in] – maximum image width in pixels

maxHeight [in] – maximum image height in pixels

maxBatchSize [in] – maximum number of simultaneously processed images in batch

dstBuffer [out] – pointer for output buffer of the current component (which is also linked buffer for the next component)

Notes:

Function `fastDecoderJ2kCreate` allocates all necessary buffers in GPU memory. Thus in case GPU does not have enough free memory, then `fastDecoderJ2kCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

Maximum dimensions of the image are set to Decoder during creation. Thus, if size of the decoded image exceeds the maximum size, then error status `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_UNSUPPORTED_SURFACE, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR`

fastDecoderJ2kGetAllocatedGpuMemorySize

```
fastStatus_t fastDecoderJ2kGetAllocatedGpuMemorySize (
      fastDecoderJ2kHandle_t handle,
      unsigned long long *allocatedGpuSizeInBytes)
```

Returns the GPU memory size allocated for JPEG2000 Decoder.

Parameters:

handle [in] – JPEG2000 Decoder handle

allocatedGpuSizeInBytes [out] – memory size in Bytes

Statuses:

FAST_OK

fastDecoderJ2kTransform

```
fastStatus_t fastDecoderJ2kTransform (
      fastDecoderJ2kHandle_t handle,
      unsigned char *byteStream,
      long streamSize,
      fastDecoderJ2kReport_t *report)
```

Decodes JPEG2000 image to surface and returns report.

Parameters:

handle [in] – pointer to JPEG2000 Decoder

byteStream – input bytestream with compressed image

streamSize – size of input bytestream

report [out] – structure, where measured duration of each decoding stage and some other values (e.g., codeblock count, output image parameters) are written to

Notes:

The procedure takes JPEG2000 bytestream and decodes it. Decoded surface is placed to the output linked buffer and the following component of the pipeline consumes it.

Statuses:

FAST_OK, FAST_INVALID_HANDLE, FAST_INVALID_VALUE, FAST_INVALID_SIZE, FAST_BITSTREAM_CORRUPT, FAST_INTERNAL_ERROR, FAST_EXECUTION_FAILURE, FAST_UNKNOWN_ERROR

fastDecoderJ2kFreeSlotsInBatch

```
fastStatus_t fastDecoderJ2kFreeSlotsInBatch (
      fastDecoderJ2kHandle_t handle,
      int *value)
```

Returns the number of free slots in batch for JPEG2000 Decoder.

Parameters:

handle [in] – JPEG2000 Decoder handle

value [out] – number of free slots

Notes:

Statuses:

FAST_OK

fastDecoderJ2kUnprocessedImagesCount

```
fastStatus_t fastDecoderJ2kUnprocessedImagesCount (
      fastDecoderJ2kHandle_t handle,
      int *value)
```

Returns the number of unprocessed images in batch for JPEG2000 Decoder.

Parameters:

handle [in] – JPEG2000 Decoder handle

value [out] – number of unprocessed images

Notes:

Statuses:

`FAST_OK`

fastDecoderJ2kAddImageToBatch

```
fastStatus_t fastDecoderJ2kAddImageToBatch (
      fastDecoderJ2kHandle_t handle,
      unsigned char *byteStream,
      long streamSize)
```

Adds image to queue for batch processing (if free slots are available) and returns immediately.

Parameters:

handle [in] – JPEG2000 Decoder handle

byteStream – input bytestream with compressed image

streamSize – size of input bytestream

Notes:

The procedure takes JPEG2000 bytestream and adds it to queue for batch processing using `fastDecoderJ2kTransformBatch` procedure. When this procedure returns control, the input buffer can be used to store another image or be disposed.
The availability of free slots in the batch can be checked using `fastDecoderJ2kFreeSlotsInBatch` function.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INVALID_VALUE, FAST_INVALID_SIZE, FAST_BITSTREAM_CORRUPT, FAST_INTERNAL_ERROR, FAST_EXECUTION_FAILURE, FAST_UNKNOWN_ERROR`

fastDecoderJ2kTransformBatch

```
fastStatus_t fastDecoderJ2kTransformBatch (
      fastDecoderJ2kHandle_t handle,
      fastDecoderJ2kReport_t *report)
```

Decodes multiple JPEG2000 images to surfaces using batch processing.

Parameters:

handle [in] – JPEG2000 Decoder handle

report [out] – structure, where measured duration of each decoding stage and some other values (e.g., codeblock count, output image parameters) are written to

Notes:

The procedure takes images, which have been added using `fastDecoderJ2kAddImageToBatch` procedure, decodes them and stores the first of the resulted surfaces. Decoded surface is placed to the output linked buffer and the following component of the pipeline consumes it. The rest surfaces should be obtained via `fastDecoderJ2kGetNextDecodedImage` function.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INVALID_VALUE, FAST_INVALID_SIZE, FAST_BITSTREAM_CORRUPT, FAST_INTERNAL_ERROR, FAST_EXECUTION_FAILURE, FAST_UNKNOWN_ERROR`

fastDecoderJ2kGetNextDecodedImage

```
fastStatus_t fastDecoderJ2kGetNextDecodedImage (
      fastDecoderJ2kHandle_t handle,
      fastDecoderJ2kReport_t *report,
      int *imagesLeft
);
```

Returns the next decoded surface during batch processing.

Parameters:

handle [in] – JPEG2000 Decoder handle

report [out] – structure, where measured duration of each decoding stage and some other values (e.g., codeblock count, output image parameters) are written to

imagesLeft [out] – number of compressed images, which can be returned by successive calls

Notes:

The procedure takes images, which have been added using `fastDecoderJ2kAddImageToBatch` procedure, encodes them and stores the first of the resulted surfaces. Decoded surface is placed to the output linked buffer and the following component of the pipeline consumes it.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_EXECUTION_FAILURE, FAST_UNKNOWN_ERROR`

fastDecoderJ2kDestroy

```
fastStatus_t fastDecoderJ2kDestroy (fastDecoderJ2kHandle_t handle)
```

Destroys JPEG2000 Decoder.

Parameters:

handle [in] – pointer to JPEG2000 Decoder

Notes:

Statuses:

`FAST_OK, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR`

NppFilter functions

fastNppFilterCreate

```
fastStatus_t fastNppFilterCreate(
	fastNppFilterHandle_t *handle,
	fastNPPImageFilterType_t filterType,
	void *staticFilterParameters,
	unsigned maxWidth,
	unsigned maxHeight,
	fastDeviceSurfaceBufferHandle_t  srcBuffer,
	fastDeviceSurfaceBufferHandle_t *dstBuffer
);
```

Creates NppFilter and returns associated handle.

Parameters:

handle [out] – pointer to created NppFilter component

filterType [in] – filter type

staticFilterParameters [in] – pointer to static parameter.

maxWidth [in] – maximum input image width in pixels

maxHeight [in] – maximum input image height in pixels

srcBuffer [in] – linked buffer from previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastNppFilterCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastNppFilterCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

There are three filters: `NPP_GAUSSIAN_SHARPEN, NPP_UNSHARP_MASK_SOFT, NPP_UNSHARP_MASK_HARD`.

Filter `NPP_GAUSSIAN_SHARPEN` supports next surfaces: `I8, RGB8, I16, RGB16.`

Filters `NPP_UNSHARP_MASK_*` support next surfaces: `RGB8, RGB12, RGB16.`

Structure `fastNPPGaussianFilter_t` is static parameter for `NPP_GAUSSIAN_SHARPEN` filter type.

```
typedef struct{
	double radius;
	double sigma;
} fastNPPGaussianFilter_t;
```

where `radius` is a radius of gaussian kernel, `sigma` is parameter for kernel value.

In general `radius` of kernel should be `3 * sigma` but for performance optimization `radius` can be less than `3 * sigma.` Liner size of kernel can be estimated like `2 * radius + 1`.

Filters `NPP_UNSHARP_MASK_SOFT` and `NPP_UNSHARP_MASK_HARD` have the same function

$$out=val+amount*(val-blur)*envelop(val)$$

where *val* is original image value, *blur* is a pixel of blurred image, *amount* controls how much contrast is added at the edges. *Envelop* function allow to make amount parameter depended on pixel brightness. Also there is additional parameter threshold. The threshold controls the minimum brightness change that will be sharpened. If *(val-blur)* less threshold then amount will be set to zero.

Filter `NPP_UNSHARP_MASK_SOFT` ignores threshold parameter.

Structure `fastNPPUnsharpMaskFilter_t` is static parameter for `NPP_UNSHARP_MASK_SOFT` and `NPP_UNSHARP_MASK_HARD` filter types.

```
typedef struct {
	float amount;
	float sigma;
	float envelopMedian; /*(0;1)*/
	float envelopSigma; /*(0;), 0.5 - mean median of value interval*/
	int envelopRank; /*2,4,6,8,12*/
	float envelopCoef; /*(;0)*/
	float threshold; /*(0;1)*/
} fastNPPUnsharpMaskFilter_t;
```

where *Amount* is filter function parameter. *Sigma* defines blur kernel size and value. There is no radius as a parameter for the filter. Radius is always equal to *3*sigma*. Threshold is relative to pixel range. Description envelop function and its parameters you can find in the chapter about NPP component

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNSUPPORTED_SURFACE`

fastNppFilterGetAllocatedGpuMemorySize

```
fastStatus_t fastNppFilterGetAllocatedGpuMemorySize(
	fastNppFilterHandle_t *handle,
	unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory for NppFilter component.

Parameters:

handle [in] – NppFilter handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for NppFilter component.

Statuses:

`FAST_OK`

fastNppFilterChangeSrcBuffer

```
fastStatus_t DLL fastNppFilterChangeSrcBuffer(
	fastNppFilterHandle_t *handle,
	fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – NppFilter handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastNppFilterFiltersTransform

```
fastStatus_t fastNppFilterTransform(
            fastNppFilterHandle_t handle
            unsigned     width,
            unsigned     height,
            void   *filterParameters)
```

Perform current NppFilter transformation.

Parameters:

handle [in] – ImageFilter component handle

width [in] – image width in pixels

height [in] – image height in pixels

filterParameters [in] – filter parameters for current image

Notes:

If image size is greater than maximum value on creation error status `FAST_INVALID_SIZE` will be returned.

Pointer `filterParameters` can point on the following structures:

`fastNPPGaussianFilter_t, fastNPPUnsharpMaskFilter_t.`

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastNppFilterDestroy

```
fastStatus_t fastNppFilterDestroy(fastNppFilterHandle_t handle)
```

Destroys NppFilter component.

Parameters:

handle [in] – NppFilter component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE`

NppGeometry functions

fastNppGeometryCreate

```
fastStatus_t fastNppGeometryCreate(
	fastNppGeometryHandle_t *handle,
	fastNppGeometryTransformationType_t transformationType,
	fastNPPImageInterpolation_t interpolationMode,
	void *filterParameters,
	unsigned maxDstWidth,
	unsigned maxDstHeight,
	fastDeviceSurfaceBufferHandle_t  srcBuffer,
	fastDeviceSurfaceBufferHandle_t *dstBuffer
);
```

Creates NppGeometry and returns associated handle.

Parameters:

handle [out] – pointer to created NppGeometry component

transformationType [in] – transformation type

interpolationMode [in] – interpolation mode

filterParameters [in] – pointer to static parameter.

maxWidth [in] – maximum input image width in pixels

maxHeight [in] – maximum input image height in pixels

srcBuffer [in] – linked buffer from previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastNppGeometryCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastNppGeometryCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

There are two transformations: `FAST_NPP_GEOMETRY_REMAP`, `FAST_NPP_GEOMETRY_REMAP3`.

Transformation `FAST_NPP_GEOMETRY_REMAP3` allows individual transformation for each RGB channels.

Filter `FAST_NPP_GEOMETRY_REMAP` supports next surfaces: `I12, RGB12, I16, RGB16`.

Filter `FAST_NPP_GEOMETRY_REMAP3` supports next surfaces: `RGB12, RGB16`.

Structure `fastNPPRemap_t` is static parameter for `FAST_NPP_GEOMETRY_REMAP` filter type.

```
typedef struct {
	fastNPPRemapMap_t *map;
	fastNPPRemapBackground_t *background;
} fastNPPRemap_t;
```

Structure `fastNPPRemap3_t` is static parameter for `FAST_NPP_GEOMETRY_REMAP3` filter type.

```
typedef struct {
      fastNPPRemapMap_t *map[3];
      fastNPPRemapBackground_t *background;
} fastNPPRemap3_t;
```

where *map* field stores information about transformation, *background* stores RGB value for background.

Structure `fastNPPRemap3_t` has individual transformation for each RGB channels, respectively.

Remap transformation is the process of taking pixels from one place in the image and locating them in another position in a new image. To accomplish the mapping process, it might be necessary to do some interpolation for non-integer pixel locations, since there will not always be a one-to-one-pixel correspondence between source and destination images.

Structure `fastNPPRemapMap_t` stores information about transformation.

```
typedef struct {
      float *mapX;
      float *mapY;

      unsigned dstWidth;
      unsigned dstHeight;
} fastNPPRemapMap_t;
```

where *dstWidth, dstHeight* are new image width and height respectively. Arrays *mapX, mapY* have size as a source image. Value in the array is position of pixel in new image. Array *mapX* stores new X position (column) , Array *mapY* stores new Y position (row).

During transformation some new pixel will be defined by old pixel. This pixel will be filled by background color.

Structure `fastNPPRemapBackground_t` defines color for background.

```
typedef struct {
      unsigned R;
      unsigned G;
      unsigned B;
      bool isEnabled;
} fastNPPRemapBackground_t;
```

If background is enabled then destination image before transformation will be filled by background. If background is disabled then non initialized pixels can contain noise or pixel of previous images.

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INVALID_VALUE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_UNSUPPORTED_SURFACE`

fastNppGeometryGetAllocatedGpuMemorySize

```
fastStatus_t fastNppGeometryGetAllocatedGpuMemorySize(
      fastNppGeometryHandle_t *handle,
      unsigned    *requestedGpuSizeInBytes)
```

Returns requested GPU memory for NppGeometry component.

Parameters:

handle [in] – NppGeometry handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:
Function returns requested memory size in Bytes for NppGeometry component.

Statuses:
`FAST_OK`

fastNppGeometryChangeSrcBuffer

```
fastStatus_t fastNppGeometryChangeSrcBuffer(
      fastNppGeometryHandle_t *handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – NppGeometry handle

srcBuffer [in] – new source buffer

Notes:
MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:
`FAST_OK, FAST_INVALID_SIZE`

fastNppGeometryTransform

```
fastStatus_t fastNppGeometryTransform(
      fastNppGeometryHandle_t handle,
      void *filterParameters,
      unsigned width,
      unsigned height    )
```

Perform current NppGeometry transformation.

Parameters:

handle [in] – ImageFilter component handle

filterParameters [in] – filter parameters for current image

width [in] – NppGeometry width in pixels

height [in] – image height in pixels

Notes:
If image size is greater than maximum value on creation error status `FAST_INVALID_SIZE` will be returned.
Pointer `filterParameters` can point on the following structures:
`fastNPPRemap_t, fastNPPRemap3_t.`

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastNppGeometryDestroy

`fastStatus_t fastNppGeometryDestroy(fastNppGeometryHandle_t handle)`

Destroys NppGeometry component.

Parameters:

handle [in] – NppGeometry component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE`

NppResize functions

fastNppResizeCreate

```
fastStatus_t fastNppResizeCreate(
      fastNppResizeHandle_t *handle,

      unsigned resizedWidth,
      unsigned resizedHeight,

      fastDeviceSurfaceBufferHandle_t  srcBuffer,
      fastDeviceSurfaceBufferHandle_t *dstBuffer
);
```

Creates NppResize and returns associated handle.

Parameters:

handle [out] – pointer to created Resizer component

resizedWidth [in] – maximum destination (cropped) image width in pixels

resizedHeight [in] – maximum destination (cropped) image height in pixels

srcBuffer [in] – linked buffer from previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastNppResizeCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastNppResizeCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

Component supports next surfaces: `RGB8, RGB12, RGB16.`

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INTERNAL_ERROR, FAST_INVALID_SIZE, FAST_UNSUPPORTED_SURFACE`

fastNppResizeGetAllocatedGpuMemorySize

```
fastStatus_t fastNppResizeGetAllocatedGpuMemorySize(
      fastNppResizeHandle_t *handle,
      unsigned    *requestedGpuSizeInBytes)
```

Returns requested GPU memory for NppResize component.

Parameters:

handle [in] – NppResize handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for NppResize component.

Statuses:

FAST_OK

fastNppResizeChangeSrcBuffer

```
fastStatus_t fastNppResizeChangeSrcBuffer(
      fastNppResizeHandle_t *handle,
      fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – NppResize handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

FAST_OK, FAST_INVALID_SIZE

fastNppResizeTransform

```
fastStatus_t fastNppResizeTransform(
      fastNppResizeHandle_t handle,

      fastNPPImageInterpolation_t resizeType,

      unsigned width,
      unsigned height,

      unsigned resizedWidth,
      unsigned *resizedHeight,

      double shiftX,
      double shiftY
);
```

Perform current NppResize transformation.

Parameters:

handle [in] – NppResizer handle

resizeType [in] – type of resize interpolation.

width [in] – input image width in pixels

height [in] – input image height in pixels

resizedWidth [in] – width of resized image in pixels

resizedHeight [out] – height of resized image in pixels

shiftX [in] – shift between source and destination grids by x coordinate. Currently ignored, should be 0,0.

shiftY [in] – shift between source and destination grids by y coordinate. Currently ignored, should be 0,0.

Notes:

If size of input image or size of resized image are greater than maximum value on creation error status `FAST_INVALID_SIZE` will be returned.

Height of resized image is calculated by the function and then customer application gets it in resizedHeight.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastNppResizeTransformStretch

```
fastStatus_t fastNppResizeTransformStretch(
      fastNppResizeHandle_t handle,

      fastNPPImageInterpolation_t resizeType,

      unsigned width,
      unsigned height,

      unsigned resizedWidth,
      unsigned resizedHeight,

      double shiftX,
      double shiftY
);
```

Perform current NppGeometry transformation.

Parameters:

handle [in] – NppResizer handle

resizeType [in] – type of resize interpolation.

width [in] – input image width in pixels

height [in] – input image height in pixels

resizedWidth [in] – width of resized image in pixels

resizedHeight [in] – height of resized image in pixels

shiftX [in] – shift between source and destination grids by x coordinate. Currently ignored, should be 0,0.

shiftY [in] – shift between source and destination grids by y coordinate. Currently ignored, should be 0,0.

Notes:

If size of input image or size of resized image are greater than maximum value on creation error status `FAST_INVALID_SIZE` will be returned.

Function allows upscale one dimension and downscale other dimension.

Statuses:

`FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE`

fastNppResizeDestroy

```
fastStatus_t fastNppResizeDestroy(fastNppResizeHandle_t handle)
```

Destroys NppResize component.

Parameters:

handle [in] – NppResize component handle

Notes:

Procedure frees all device memory.

Statuses:

`FAST_OK, FAST_INVALID_HANDLE`

NppRotate functions

fastNppRotateCreate

```
fastStatus_t fastNppRotateCreate(
      fastNppRotateHandle_t *handle,
      fastNPPImageInterpolation_t interpolationMode,

      fastDeviceSurfaceBufferHandle_t  srcBuffer,
      fastDeviceSurfaceBufferHandle_t *dstBuffer
);
```

Creates NppRotate and returns associated handle.

Parameters:

handle [out] – pointer to created NppRotate component

interpolationMode [in] – type of rotate interpolation.

srcBuffer [in] – linked buffer from previous component

dstBuffer [out] – pointer for linked buffer for the next component (output buffer of current component)

Notes:

Function `fastNppRotateCreate` allocates all necessary buffers in GPU memory. So in case GPU does not have enough free memory, then `fastNppRotateCreate` returns `FAST_INSUFFICIENT_DEVICE_MEMORY`.

If component does not support current surface format then the function will return `FAST_UNSUPPORTED_SURFACE`.

Component supports next surfaces: `I8,RGB8, I12, RGB12, I16, RGB16.`

Statuses:

`FAST_OK, FAST_INSUFFICIENT_DEVICE_MEMORY, FAST_INTERNAL_ERROR, FAST_INVALID_SIZE, FAST_UNSUPPORTED_SURFACE`

fastNppRotateGetAllocatedGpuMemorySize

```
fastStatus_t fastNppRotateGetAllocatedGpuMemorySize(
      fastNppRotateHandle_t *handle,
      unsigned     *requestedGpuSizeInBytes)
```

Returns requested GPU memory for NppRotate component.

Parameters:

handle [in] – NppRotate handle

requestedGpuSizeInBytes *[out]* – memory size in Bytes

Notes:

Function returns requested memory size in Bytes for NppRotate component.

Statuses:

`FAST_OK`

fastNppRotateChangeSrcBuffer

```
fastStatus_t DLL fastNppRotateChangeSrcBuffer(
```

```
    fastNppRotateHandle_t *handle,
    fastDeviceSurfaceBufferHandle_t srcBuffer
);
```

Set new source buffer

Parameters:

handle [in] – NppRotate handle

srcBuffer [in] – new source buffer

Notes:

MaxWidth and MaxHeight of new buffer should be equal to appropriate values of current buffer otherwise `FAST_INVALID_SIZE` will be returned.

Statuses:

`FAST_OK, FAST_INVALID_SIZE`

fastNppRotateGetRotateQuad

```
fastStatus_t fastNppRotateGetRotateQuad(
    fastNppRotateHandle_t handle,

    unsigned width,
    unsigned height,

    double rotateAngle,

    double shiftX,
    double shiftY,

    NppQuadCorners_t *quadCorners
);
```

Returns coordinates of rectangle that envelop rotated image.

Parameters:

handle [in] – NppRotate handle

width [in] – input image width in pixels

height [in] – input image height in pixels

rotateAngle[in] – rotation angle in degrees

shiftX [in] – shift by X for upper left corner of image.

shiftY [in] – shift by Y for upper left corner of image.

quadCorners[out] – pointer to result coordinates. Coordinate can be negative.

Notes:

The function is wrapper for `nppiGetRotateQuad` function. It allows to calculate real size of rotated image. It is not so obvious because image size depend on rotation angle.

Statuses:

`FAST_OK, FAST_INVALID_SIZE, FAST_INTERNAL_ERROR`

fastNppRotateTransform

```
fastStatus_t fastNppRotateTransform(
```

```
        fastNppRotateHandle_t handle,

        unsigned width,
        unsigned height,

        unsigned dstRoiX,
        unsigned dstRoiY,

        double rotateAngle,

        double shiftX,
        double shiftY
);
```

Perform current NppRotate transformation.

Parameters:

handle [in] – NppRotate handle

width [in] – input image width in pixels

height [in] – input image height in pixels

dstRoiX [in] – width of rotated image in pixels

dstRoiX [out] – height of rotated image in pixels

rotateAngle[in] – rotation angle in degrees

shiftX [in] – shift by X for upper left corner of image.

shiftY [in] – shift by Y for upper left corner of image.

Notes:

The function is wrapper for `nppiRotate*` function. Parameters *dstRoiX, dstRoiY, shiftX, shiftY* calculated on result of `fastNppRotateGetRotateQuad` function. See sample of rotation in NppSample.

Statuses:

FAST_OK, FAST_INVALID_VALUE, FAST_INVALID_HANDLE, FAST_INTERNAL_ERROR, FAST_UNKNOWN_ERROR, FAST_EXECUTION_FAILURE, FAST_INVALID_SIZE

fastNppRotateDestroy

```
fastStatus_t fastNppRotateDestroy(fastNppRotateHandle_t handle)
```

Destroys NppRotate component.

Parameters:

handle [in] – NppRotate component handle

Notes:

Procedure frees all device memory.

Statuses:

FAST_OK, FAST_INVALID_HANDLE

Source Code for Sample Applications

The source code for sample applications and all other sample programs are located in the samples directory of SDK. Microsoft Visual Studio (2015) solution and Linux makefiles are included with the source code. Qt solution is coming soon.

Other Sample Applications

Command-line sample applications are provided with the current SDK. The executables for all sample applications are in the SDK /bin directory. Each sample can be executed without any parameters to display a message which briefly describes all available parameters. To build the above applications, one have to utilize the following files:

- all h-files are in the SDK inc directory
- all lib-files are in the SDK lib directory

Examples of command line for DebayerSample application

Debayering with DebayerSample application:

DebayerSample.exe -i kodim19.pgm -o kodim19.ppm -type DFPD -pattern RGGB -info

The application takes kodim19.pgm image from current directory and runs debayer with DFPD algorithm and pattern RGGB to create an image kodim19.ppm. Parameter -info means that we will get detailed info about timing and performance.

.\x64\Release\DebayerSample.exe -if .\..\Images*.pgm -o .\x64\Release*.ppm -type DFPD -pattern RGGB -maxWidth 4096 -maxHeight 4096 -info

The application takes all PGM images from folder .\..\Images\ and does debayering with DFPD algorithm and pattern RGGB to create corresponding images with PPM extension. Maximum width and height of all images can't exceed 4096 in that particular case. Real maximum height and width depend on GPU memory size.

Examples of command line for JpegSample application

Decoding with JpegSample application:

JpegSample.exe -i input.jpg -o output.ppm -info

The application takes input.jpg image from current directory and decodes it to file output.ppm. Parameter -info means that application will output detailed info about timing and performance.

.\x64\Release\JpegSample.exe -if .\..\Images*.jpg -o .\x64\Release*.ppm -maxWidth 4096 -maxHeight 4096 -info

The application takes all JPEG images from folder .\..\Images\ and decodes them into corresponding images with PPM extension. Maximum width and height of all images in that particular case can't exceed 4096. Real maximum height and width depend on GPU memory size.

Encoding with JpegSample application:

JpegSample.exe -i input.ppm -o output.jpg -q 90 -s 444 -info

The application takes input.ppm image from current directory and encodes it to file output.jpg with JPEG quality 90%, subsampling 4:4:4 and optimum restart interval is computed automatically. Parameter -info means that application will output detailed info about timing and performance.

Example of command line for DebayerJpegSample application

That application makes Demosaicing and JPEG compression in one pipeline:

DebayerJpegSample.exe -i input.pgm -o output.jpg -type DFPD -pattern RGGB -q 90 -s 420 -info

The application takes input.pgm image from current directory and converts it to output.jpg. Parameter -info means that application will output detailed info about timing and performance.

Examples of command line for SDIConverterSample application

That application takes an input image from current directory and runs SDI transforms RGB to YCbCr or YCbCr to RGB. Parameter -info means that we will get detailed info about timing and performance.

Command line for SDIConverterSample:

SDIConverterSample.exe -i <input image> -o <output image> -format <format> -width <width> -height <height> -d <device ID> -info

Command line for import:

SDIConverterSample.exe -i <input image> -o <output image> -format CbYCr422_709 -width 1920 -height 1080 -d <device ID> -info

Command line for export:

SDIConverterSample.exe -export -i <input image> -o <output image> -format CbYCr422_709 -width 1920 -height 1080 -d <device ID> -info

Example of command line for PhotoHostingSample application

That application makes the following pipeline: JPEG decoding, cropping, resizing, sharpening, JPEG encoding

PhotoHostingSample.exe -i input.jpg -o output.crop.1023.jpg -outputWidth 1023 -crop 1900x1000+12+10 -q 90 -s 444 -info

The application takes input.jpg image from current directory, crops it with offsets 12 and 10 with final resolution 1900x1000, then does resize to final width 1023 and converts it to output.crop.1023.jpg. Parameter -info means that application will output detailed info about timing and performance.

This is an example for batch image processing:

.\x64\Release\PhotoHostingSample.exe -if .\..\Images*.jpg -o .\x64\Release*.512.jpg -outputWidth 511 -maxWidth 4096 -maxHeight 4096 -q 90 -s 444 -info

Troubleshooting

Debug version of SDK which allows generation of a trace/debug log is also available. It can be helpful for problem detection and identification. The debug DLL file name is the same as the non-debug version, but with the addition of the letter ‘d’ at the end of filename prior to the file extension.

We welcome and encourage your questions and comments concerning our products use from evaluation through deployment of your application. Contact us at info@fastcompression.com

Disclaimer of Warranty

In addition to the provisions of the Standard License Agreement (SLA) of FASTVIDEO the following apply:

Although FASTVIDEO has taken care to ensure the accuracy of the information contained herein it accepts no responsibility for the consequences of any use thereof and also reserves the right to change the specification without prior notice.

FASTVIDEO does not assume any liability for damage that is the result of improper use of its products or failure to comply with the operating manuals or the applicable rules and regulations.

List of Trademarks

FASTVIDEO is a registered trademark of FASTVIDEO LLC in Russia.

Microsoft, Windows, Windows 10, Windows 8, Windows 7, Windows Vista, and Microsoft Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

NVIDIA, CUDA, NPP, Tegra, Linux4Tegra, GeForce, Quadro, Tesla are trademarks of NVIDIA Corporation.

OpenGL is a registered trademark of Silicon Graphics, Inc.

Linux is a trademark of Linus Torvalds.

Intel, Core i3 and i7 are trademarks of Intel Corporation.

FFmpeg is a trademark of Fabrice Bellard, originator of the FFmpeg project.

Qt is a registered trademark of Qt Company Ltd.

All other brands, service provision brands and logos referred to are brands, service provision brands and logos belonging to their respective owners.

Application Notes

All functions of the software can work only with NVIDIA GPUs with Kepler, Maxwell and Pascal architectures with the latest drivers installed. We don't support GPUs with cc < 3.0 from NVIDIA. The software can't work neither with Intel nor AMD/ATI GPUs.

The software can't work with Windows 2000 and XP. Command line applications are compiled for Windows-7/8/10, 64-bit.

Before testing SDK components or sample applications, please check that CUDA runtime libraries and GPU drivers are installed.

CUDA JPEG codec can work with JPEG images only according to specification of the Baseline part of JPEG Standard. CUDA JPEG encoder can work with 12-bit image format as well. Arithmetic coding, lossless JPEG (JPEG_LS), progressive JPEG and other extended features are beyond codec's capabilities.

CUDA JPEG codec could be exceptionally fast at JPEG decoding only in the case when input images have built-in Restart Markers. These markers ensure fully parallel implementation of JPEG decoding on GPU. If compressed input image doesn't have Restart Markers, the codec can perform Huffman decoding stage of JPEG Baseline algorithm on CPU and then carry on with GPU. This solution is still much faster than full JPEG decompression on CPU. If you do image compression with CUDA JPEG codec, the software automatically adds Restart Markers into compressed bitstream and you will not have any problems with performance at the decompression stage.

To get maximum performance, GPU needs as much data as possible. For small images (resolution less than 300x300) one can get moderate performance results even with very powerful GPU.

There is an upper limit for resolution for big images due to size of available GPU RAM. Please test your images on demo sample application to be sure that your GPU can work with images with desired resolution. There is also a lower limit for image resolution which is quite small.

Please note that in the case when you are utilizing discrete GPU at laptop, be sure that your laptop is directly connected to external power supply. Without external power, GPU performance will degrade.

If you implement your own software with FASTVIDEO SDK, please check that your application is running just one instance of fastvideo_sdk.dll per GPU, though it's possible (with less performance) to run several instances as well. The most efficient way is to run just one instance per GPU, because the software is highly optimized and GPU occupancy is very high, so at each moment just one image could be processed on particular GPU. It could be a good idea to organize input and output queues which are working with the same instance of fastvideo_sdk.dll to ensure maximum performance.

To get better performance we also recommend to utilize big images and multiple CUDA Streams to ensure overlapping between data copy and computations.