# Tutorials Readme

Intel® Media SDK Tutorials

**Intel Corporation**

# Table of Contents

# 1 Introduction

## 1.1 Exploring Media SDK via a simplified set of samples

The Intel® Media Software Development Kit (Intel® Media SDK) gives developers access to specialized hardware acceleration capabilities for video decoding, encoding, and pre/post processing (VPP).

Intel® Media SDK helps developers rapidly write software that accesses hardware acceleration for video codecs with automatic fallback on software if hardware acceleration is not available. Intel® Media SDK is available for downloading here: http://www.intel.com/software/mediasdk/.

This quick-start tutorial illustrates how to use Intel® Media SDK by stepping from simple to complex. Concepts are added in layers. You are encouraged to use your favorite file difference tool to compare what was added in each step. While the tutorial cannot cover every possible usage, our goal is to provide starting points to shrink your TTM for a wide range of applications.

For simplicity and uniformity the tutorial focuses on the H.264 (AVC) video codec. Other codecs supported by Intel Media SDK can be utilized in a similar way, often by a single configuration parameter. This unified approach can save significant development time and is a major advantage to working with Media SDK.

Detailed comments explain the behavior of the code. You are encouraged to read and experiment. For a deeper understanding of the SDK and details on specific parameters, please refer to the Media SDK reference manual as well as the users guide. The set of samples packaged with the SDK showcase many more parameters and scenarios and are a valuable reference as well.

All tutorial samples are self-contained unless otherwise noted. For Linux* makefiles are provided. For Windows* each tutorial has a Microsoft Visual Studio* 2012 solution and project. To reduce redundancy and improve readability common code segments are located in the '`common`' folder. These are tied to functions not directly related to Intel Media SDK functionality, such as read/write of bit streams and raw frames. The '`common`' folder also contains OS-specific device handling and surface allocation implementations.

The tutorial samples were developed and tested using recent versions of Intel® Media SDK for Windows and Linux. For questions, or to report issues with the tutorial samples presented in this article, please use the Intel Media SDK Forum.

## 1.2 Windows specific notes

- Please note, some Windows tutorials also require installation of Microsoft Windows SDK*.
- Since the introduction of Microsoft Window 8, Media SDK can now also be used with DirectX 11 devices and surfaces. Media SDK relies on features in DirectX 11.1, and can therefore not be used on Microsoft Windows 7. If target application must run on Microsoft Windows 7 then Media SDK DirectX9 capabilities should be used instead. DirectX 11 example solutions/projects are created using Microsoft Visual Studio 2012 to ensure DirectX 11.1 environment support.

## 1.3 How to obtain input content

The tutorial assumes that the user has proper content to play with. Such content can be acquired from many sources on the web. One example is the Peach open movie project "Big Buck Bunny", which can be downloaded from http://www.bigbuckbunny.org. The video elementary stream must be extracted before using. This can be done using ffmpeg with:

```
ffmpeg \
  -i big_buck_bunny_1080p_h264.mov \
  -an -vcodec copy -bsf h264_mp4toannexb \
  -f h264 bbb1920x1080.264
```

You can decode this stream to raw YUV format using ffmpeg, the tutorial decode examples, or Media SDKs sample_decode.

```
ffmpeg -i bbb1920x1080.264 bbb1920x1080.yuv
sample_decode h264 -i bbb1920x1080.264 -o bbb1920x1080.yuv
```

Storing all frames in raw format can take a lot of space. However, if disk capacity is a concern, the samples and tutorials will work on a subset of frames. Number of frames processed can be controlled via -vframes for ffmpeg or changing the decode sample/tutorial code.

## 1.4 Tutorials Samples Index

Tutorials are devided into few sections:

| | |
|---|---|
| 1 | Media SDK "Hello World". Start and query a session |
| 2-4 | Decode, encode, and VPP as single component pipelines |
| 5 | Transcode: The most common compound workload |
| 6 | Advanced/compound Media SDK scenarios |

| **Name** | **Description** |
|---|---|
| simple_1_session | Sets up Intel Media SDK session and perform queries to determine selected implementation and which API version is used |
| simple_2_decode | Decodes AVC stream into YUV file using system memory surfaces, showcasing simple synchronous decode pipeline flow |
| simple_2_decode_vmem | Adds use of video memory surfaces for improved decode performance |
| simple_3_encode | Encodes YUV frames from file into AVC stream using surfaces in system memory, showcasing simple synchronous encode pipeline flow |
| simple_3_encode_vmem | Adds use of video memory surfaces for improved encode performance |
| simple_3_encode_vmem_async | Adds asynchronous operation to previous example, resulting in further improved performance |
| simple_4_vpp_resize_denoise | Showcases video frame processing (VPP) using system memory surfaces. Highlights frame resize and denoise filter processing |

| | |
|---|---|
| simple_4_vpp_resize_denoise_vmem | Adds use of video memory surfaces for improved VPP performance |
| simple_5_transcode | Transcodes (decode+encode) AVC stream to another AVC stream using system memory surfaces |
| simple_5_transcode_opaque | Same as previous sample but uses the Intel Media SDK opaque memory feature. The opaque memory type hides surface allocation specifics and allows the SDK to select the best type for the execution in HW or SW |
| simple_5_transcode_opaque_async | Adds asynchronous operation to the transcode pipeline implementation, resulting in further improved performance |
| simple_5_transcode_vmem | Same as "simple_5_transcode" sample but uses video memory surfaces instead. While opaque surfaces use video memory internally, application-level video memory allocation is required to integrate components not in Media SDK. |
| simple_5_transcode_opaque_async_vppresize | Same as "simple_5_transcode_opaque -async" sample but pipeline includes VPP resize. |
| simple_6_decode_vpp_postproc | Similar to the simple_2_decode sample but adds VPP post-processing capabilities to showcase resize and ProcAmp |
| simple_6_encode_vmem_lowlatency | Similar to the simple_3_encode_vmem sample with additional code to illustrate how to configure an encode pipeline for low latency and how to measure latency |
| simple_6_transcode_opaque_lowlatency | Similar to the simple_5_transcode_opaque sample with additional code to illustrate how to configure a transcode pipeline for low latency and how to measure latency |
| simple_6_encode_vmem_vpp_preproc | Similar to the simple_3_encode_vmem sample but adds VPP pre-processing capabilities to show frame color conversion from RGB32(4) to NV12 |

# 2 Tutorials Section 1

## 2.1 Hello World

This tutorial sample showcases a very simple "hello world" type Intel® Media SDK use case.

The sample shows how to initialize an Intel Media SDK session (MFXVideoSession) using the target selection option "MFX_IMPL_AUTO_ANY", which is recommended as a default setting since it is appropriate for nearly all cases.

**Initialization Differences Between Windows and Linux**

There are not many differences between Media SDK for Windows and for Linux, but initialization showcases some of the main ones. Please note that the main function is the same for all operating systems but there is some OS specific code in the common directory.

**Windows**

The Windows releases contain software and hardware implementations. MFX_IMPL_AUTO_ANY implies the session will be initialized to use HW acceleration (regardless in which adapter the Intel® HD Graphics device resides) if available on the processor. If HW acceleration is not available, the Intel Media SDK defaults to SW implementation.

In the "initialize" function (called from main), associating a display handle with the session is not necessary except when using video memory surfaces.

**Linux**

The Linux server release does NOT include a software implementation. MFX_IMPL_AUTO_ANY will attempt to start the session with hardware acceleration. If the hardware implementation cannot be found initialization is not successful.

In the "initialize" function (called from main), associating a display handle with the session is ALWAYS necessary for Linux.

**Session queries (all OS)**

After initialization, the session is queried to determine the actual target (via QueryIMPL) that was selected. For Windows this could be HW or SW, though HW will be chosen if your processor and driver support accelerated media processing. For Linux the implementation can only be HW. The highest supported API version is returned via QueryVersion.

# 3 Tutorials Section 2: Decode

## 3.1 simple_2_decode

This Intel® Media SDK tutorial sample illustrates the most simplistic way of implementing HW decode using system memory surfaces.

Note: In this and many of the following tutorials Intel GPA is an ideal tool to analyze and identify potential performance bottlenecks. More details will be added on using Intel GPA soon.

The basic goal of this example is to illustrate why asynchronous operation using video memory surfaces is necessary. While it is simpler to use system memory synchronously, as in this example, this introduces unnecessary bottlenecks:

Surfaces must be copied from GPU to CPU. While this must happen in any case for decode which writes frames to disk, buffering is not as efficient in this scenario. For a single decode (or possibly even several) gaps in the processing pipeline cannot easily be filled. Since the GPU is not in constant use it may fall out of turbo mode. Based on the above analysis we should be able to improve the performance of the workload by using video memory surfaces instead of system memory surfaces. The next tutorial sample will explore such scenario.

## 3.2 simple_2_decode_vmem

This Intel® Media SDK tutorial sample operates in the same way as the previous "sample_2_decode" sample except that it uses video memory surfaces instead of system memory surfaces.

Video surfaces are required to enable allocation on the GPU. These are implemented in DirectX for Windows and VAAPI for Linux. Device creation, adapter detection and surface management can be found in the tutorial "common" folder.

Video memory surfaces allow greater efficiency by avoiding explicit copies. Further improvement may be achieved by making the decode pipeline asynchronous. Well explore this approach further when we discuss encoding workloads in the following tutorial sections. Improved GPU utilization can also be achieved by executing several decode workloads concurrently.

Additional details for Windows developers:

Since the introduction of Microsoft Windows* 8, Intel Media SDK can be used with DirectX11 devices and surfaces. Note that Intel Media SDK relies on the features part of DirectX 11.1, and can therefore not be used on Microsoft Windows 7. If your target application must run on Microsoft Windows 7, use the DirectX 9 path via Intel Media SDK.

Tutorial samples illustrating use of D3D surfaces (such as in this sample) have two Microsoft Visual Studio* solution/project (sln/prj) files. sln/prj for DirectX9 usage created using Microsoft Visual Studio 2010 and sln/prj for DirectX11 usage created using Microsoft Visual Studio 2012. Microsoft Visual Studio* 2012 is used for DirectX11 to ensure full DirectX 11.1 environment support.

# 4 Tutorial Section 3: Encode

## 4.1 simple_3_encode

This Intel® Media SDK tutorial sample illustrates the most simplistic way of implementing HW encode using system memory surfaces.

This is intended as a simple starting point, but implicit copies and synchronous implementation limit performance.

## 4.2 simple_3_encode_vmem

This Intel® Media SDK tutorial sample operates in the same way as the "sample_3_encode" workload except that it is using video memory surfaces instead of system memory surfaces.

For more details on this topic, please refer to "simple_2_decode" sample description.

By moving from system to video memory implicit copies are eliminated, thus improving GPU load and overall performance. CPU utilization should also decrease slightly.

To improve performance and achieve greater GPU utilization we must move away from the synchronous encoding approach towards asynchronous workload behavior. The next tutorial section will explore an asynchronous encode pipeline.

## 4.3 simple_3_encode_vmem_async

This Intel® Media SDK tutorial sample keeps multiple encode tasks in flight simultaneously, and SyncOperation() is not called until absolutely necessary (when all surface input buffers have been exhausted).

For more details on implementing with video memory please refer to "simple_2_decode" sample description.

This example achieves efficient operation with video memory surfaces and asynchronous implementation, minimizing gaps in the GPU pipeline. Decode and encode operations are optimally scheduled internally by Media SDK so that fixed function hardware is fully utilized with many operations occurring simultaneously. As an added benefit of fully utilizing the GPU, this will cause it to remain in turbo mode1 providing a further boost to performance.

Marginal throughput improvements may be achieved by executing several encode workloads concurrently.

# 5 Tutorial Section 4: VPP

## 5.1 simple_4_vpp_resize_denoise

In this Intel® Media SDK tutorial sample we illustrate how to utilize Intel Media SDK to do frame processing on frame surfaces using the SDKs VPP component.

We start with the most simplistic Intel Media SDK VPP workload that uses system memory frame surfaces. This has several disadvantages, resulting in low GPU utilization and low performance (Specifically, implicit memory copies between CPU and GPU and synchronous implementation.) Note: many VPP operations are implemented on general purpose execution units (EUs) and do not directly use fixed function hardware.

To address the above performance issues lets explore a modified VPP workload that uses video memory surfaces instead.

## 5.2 simple_4_vpp_resize_denoise_vmem

This Intel® Media SDK tutorial sample operates in the exact same way as the previous tutorial sample, "simple_4_vpp_resize_denoise" except that it is using video memory surfaces.

For more details on using video memory please refer to "simple_2_decode" sample description.

As with the previous workload, the RunFrameVPPAsync() call leads to the VPP Submit operation. However, in this case the task is submitted to the GPU almost immediately since the input surface already resides on a video memory surface.

Like the encode workloads discussed earlier (such as simple_3_encode_vmem), Intel Media SDK uses a polling mechanism, via VPP Query, to determine if the GPU has fully processed the frame.The GPU is queried every 1ms until the frame is ready. Since the current workload is synchronous the SyncOperation() call will wait until the next VPP Query which can cause large gaps in GPU execution.

We already demonstrated how to achieve greater performance by making the Intel Media SDK work in an asynchronous fashion, as in the simple_3_encode_vmem_async workload. The same approach can be used for VPP processing so we will not explore this case further. It suffices to say that GPU utilization and performance can be improved significantly by applying the task concurrency approach for VPP.

This concludes the analysis of Intel Media SDK decode, encode and VPP workloads. The next tutorial sections explore the behavior of workloads combining several Intel Media SDK components.

This tutorial sample is found in the tutorial samples package under the name "simple_4_vpp_resize_denoise_vmem". The code is extensively documented with inline comments detailing each step required to setup and execute the use case.

# 6 Tutorial Section 5: Transcode

## 6.1 simple_5_transcode

In this part of the Intel® Media SDK tutorial we will explore transcode workloads, starting with the most simplistic transcode sample using system memory frame surfaces.

In this simple implementation there are barriers to full GPU utilization, as with the other examples:

System memory adds implicit copies when using hardware acceleration Synchronous implementation means less efficient internal scheduling of decode/encode stages. Not as many opportunities to keep hardware pipeline fully loaded. Since we are using system memory surfaces we must copy the decoded surface to system memory first, then before encode the surface will be copied to video memory again. Both copy operations have a large impact on CPU load and performance

As noted when exploring the encode workloads, the Encode Query polling method also introduces a slight inefficiency in the pipeline after the GPU has completed the encoding task.

The performance is also indirectly degraded by the fact that the GPU remains in lower frequency states due to the relatively low GPU activity.

In the following sections we will explore how to enhance Intel Media SDK transcode pipelines for improved GPU utilization leading to better performance.

## 6.2 simple_5_transcode_opaque

In the Intel® Media SDK tutorial sections covering encode, decode and VPP the performance issues caused by relying on system memory were removed by using video memory surfaces. "Opaque memory" surfaces offer a simplified path to video memory optimization when working with transcode pipelines implemented entirely with Media SDK. With opaque memory surfaces are managed entirely by Media SDK so it can automatically optimize for the type of session requested. Video memory will be used for hardware sessions, system memory for software sessions.

The use of opaque memory eliminates two surface copies per frame which leads to lower CPU utilization and higher GPU utilization. As with encode, synchronous implementation is insufficient to keep the hardware pipeline busy and introduces many gaps. These gaps can be filled with work from the same video sequence with asynchronous implementation. Further efficiency can be gained by working with multiple transcodes simultaneously. For best results, individual pipelines should be asynchronous with multiple sessions running simultaneously.

## 6.3 simple_5_transcode_opaque_async

In this Intel® Media SDK tutorial transcode sample we introduce asynchronous pipeline behavior using the same approach as we did in the simple_3_encode_vmem_async sample.

Overall GPU utilization can be improved significantly by implementing asynchronously. Since the GPU is highly utilized, the overall performance is also improved by the fact that the GPU is consistently residing in a high frequency state (due to Intel® Turbo Boost Technology1).

## 6.4 simple_5_transcode_opaque_async_vppresize

This Intel® Media SDK tutorial sample is essentially the same as the "simple_5_transcode_opaque_async" sample except that VPP processing is used to resize the content.

## 6.5 simple_5_transcode_vmem

This Intel® Media SDK tutorial sample is essentially the same as the "simple_5_transcode" sample except for that it uses D3D surfaces instead of system memory surfaces.

Like the "simple_2_decode" tutorial sample this sample supports both Microsoft DirectX* 9 and DirectX* 11 for Windows and VAAPI for Linux. For more details on this topic please refer to "simple_2_decode" sample description.

The use of GPU memory surfaces leads to improved performance. In essence the behavior of this workload is the same as for "simple_5_transcode_opaque" (when executed on a processor that supports HW acceleration). However, since the application manages video memory these surfaces are available to integrate with components that are not in the standard Intel Media SDK decode>process>encode pipeline.

# 7 Tutorial Section 6: Advanced Media SDK

## 7.1 How to create low latency pipelines and how to benchmark latency

Low latency video codec pipelines are important for many workloads, one example is video conferencing where minimal latency is desired. Media SDK supports configuration of encoder and decoder for low latency. Using a low latency configuration results in lower pipeline throughput, but for this specific use case its not an issue. Note that Media SDK also supports a wide range of other features useful for developing video conferencing or dynamic video streaming solutions. More information on how to use Media SDK for low latency workloads and other typical video conferencing usages can be found in this white paper: http://software.intel.com/en-us/articles/video-conferencing-features-of-intel-media-software

### 7.1.1 simple_6_encode_vmem_lowlatency

This sample illustrates how to configure an encode pipeline for low latency and how to measure latency.

### 7.1.2 simple_6_transcode_opaque_lowlatency

This sample illustrates how to configure an Intel® Media SDK transcode pipeline for low latency and how to measure latency.

The approach is very similar to the description of low latency workloads in tutorial section, simple_6_encode_d3d_lowlatency.

## 7.2 How to use VPP for pre- or post-processing purposes

Aside from using VPP in a transcode pipeline, VPP can also be used for pre or post processing of frames. Common usages are image resize, effects or enhancements before rendering to display. Such usage is illustrated in the first sample below. The second sample below showcases how to VPP for pre processing of frames, such as color space conversion (common for pipelines where frame input originates from camera).

## 7.3 simple_6_decode_vpp_postproc

This sample is similar to simple_2_decode but also adds VPP post processing capabilities, showcasing frame resize and noise reduction

## 7.4 simple_6_encode_vmem_vpp_preproc

This Intel® Media SDK tutorial sample is similar to simple_3_encode_vmem but adds VPP pre-processing capabilities, showcasing VPP color conversion from RGB32(4) to NV12.