

Graphics API Developer's Guide

For 5th Generation Intel® Core™ Processors

Abstract

This guide helps software developers optimize their apps for use on 5th generation Intel® Core™ processors. After an overview of the processor graphics hardware, you'll find guidance on structuring and optimizing your DirectX* and OpenGL* code. Brief tutorials on Intel® Graphics Performance Analyzers (Intel® GPA) and Intel VTune™ Amplifier XE are provided. You'll also find references to some key code samples.

Table of Contents

1	Introduction.....	1
2	Architectural Overview.....	2
2.1	Processor Graphics - An Overview of the GPU.....	2
2.2	Graphics Hardware Improvements in 5th Gen Intel® Core™ Pprocessors.....	3
2.3	Intel® Turbo Boost Technology 2.0.....	3
3	Tools.....	4
3.1	Intel® Graphics Performance Analyzers (Intel® GPA).....	4
3.2	Intel® VTune™ Amplifier XE.....	6
4	Important Optimizations	6
4.1	Test on Intel® Processor Graphics	6
4.2	Use the Newest Graphics Driver	7
4.3	Find and Fix Memory Bandwidth Problems.....	7
4.4	Choose an Appropriate Anti-Aliasing Method.....	8
4.5	Minimize State Changes.....	8
4.6	Avoid Unnecessary Clears and Copies	9
4.7	(OpenGL only) Use Vertex Buffer Objects and Frame Buffer Objects.....	9
4.8	(OpenGL only) Use Fence Objects for Synchronization	10
4.9	Avoid Interleaving Reads and Writes	10
4.10	Use Culling to Avoid Bottlenecks in the Geometry Pipeline	10

4.11	Find and Fix Shader Bottlenecks	11
4.12	Use Shader Literals Instead of Constant Buffers	11
4.13	Minimize Register Pressure	11
4.14	Use Shader Built-ins.....	12
4.15	Favor UAV Counters over Atomic Operations.....	12
4.16	(OpenGL only) Use glFinish and glFlush Correctly	12
4.17	Use Power-of-Two Mipmaps.....	12
4.18	(DirectX only) Avoid DirectX 9 Legacy Fixed-Function State.....	12
5	Detecting the Amount of Graphics Memory.....	12
6	(DirectX only) Intel® Iris™ Graphics Extensions to DirectX 11 API.....	13
6.1	Checking if Extensions are Available.....	13
6.2	Intel® Iris™ Graphics Extension for Pixel Synchronization	13
6.3	Intel® Iris™ Graphics Extension for Instant Access.....	14
7	(OpenGL only) Intel Iris Graphics Extensions to OpenGL.....	17
7.1	Fragment Shader Ordering.....	17
7.2	Map Texture.....	17
8	(DirectX only) DirectX 9 FOURCC Extensions.....	17
9	Power Efficiency	18
9.1	Power Background.....	18
9.2	Power Efficient Programming	19
9.3	Power Analysis Workflow	20
10	Sample Code.....	21
10.1	GPU Detect.....	21
11	References.....	22

1 Introduction

With the release of 5th generation Intel Core processors, developers have a new set of powerful graphics capabilities. This guide introduces the graphics hardware architecture of 5th gen Intel Core processors. It also gives best practices for writing code that uses Intel® processor graphics on those processors.

Here’s a quick comparison of the graphics capabilities of Intel Core processors, showing the evolution from the 4th gen to 5th gen processors.

Table 1-1: Comparing the best of Intel® Core™ 4th generation with Intel® Core™ 5th generation processor graphics

	4th gen Intel® Core™ processor	5th gen Intel® Core™ processor		
	Intel® Iris™ Pro Graphics 5200, Intel™ Iris™ Graphics 5100, Intel® HD Graphics 5000	Intel® HD Graphics 5300	Intel® HD Graphics 5500	Intel® HD Graphics 6000, Intel™ Iris™ Graphics 6100
APIs	DirectX* 11.1 DirectX* Shader Model 5.0 OpenGL* 4.3 OpenCL* 1.2	DirectX* 12, 11.2 DirectX* Shader Model 5.0 OpenGL* 4.3 OpenCL* 2.0		
Execution Units (EUs)	40 EUs	24 EUs	24 EUs (23 EUs in some Core i3 SKUs)	48 EUs (47 EUs in some Core i3 SKUs)
Texture sampler(s)	4	3	3	6
Package Power	15W to 47W	4.5W	15W	28W
	128 MB eDRAM (Intel Iris Pro Graphics 5200 only)			

This guide introduces the graphics hardware of these processors and walks through suggested tools. It also gives performance recommendations for DirectX and OpenGL programmers.

2 Architectural Overview

2.1 Processor Graphics – An Overview of the GPU

The graphics hardware architecture of the 5th gen Intel Core processors is similar to that of the 4th gen processors. The modular architecture allows different versions of the processors to be more easily designed. There's some fixed function graphics hardware, and the rest of the GPU is made up of "slices." More slices means more graphics processing ability. Each slice contains some shared hardware, such as a pixel front-end, pixel back-end, and level 3 cache, along with a group of execution units (EUs) and samplers. The graphics processors in 5th gen Intel Core processors have a ratio of 1 sampler for every 8 EUs, which is more favorable for high-sampling workloads like games, relative to the ratio of 1:10 in the 4th gen processors.

As with earlier generations, the graphics processor is manufactured in the same die as the CPU. For the 5th gen Intel Core processors, that means both the CPU and the graphics hardware are manufactured using the same 14nm process, giving significant power and performance advantages.

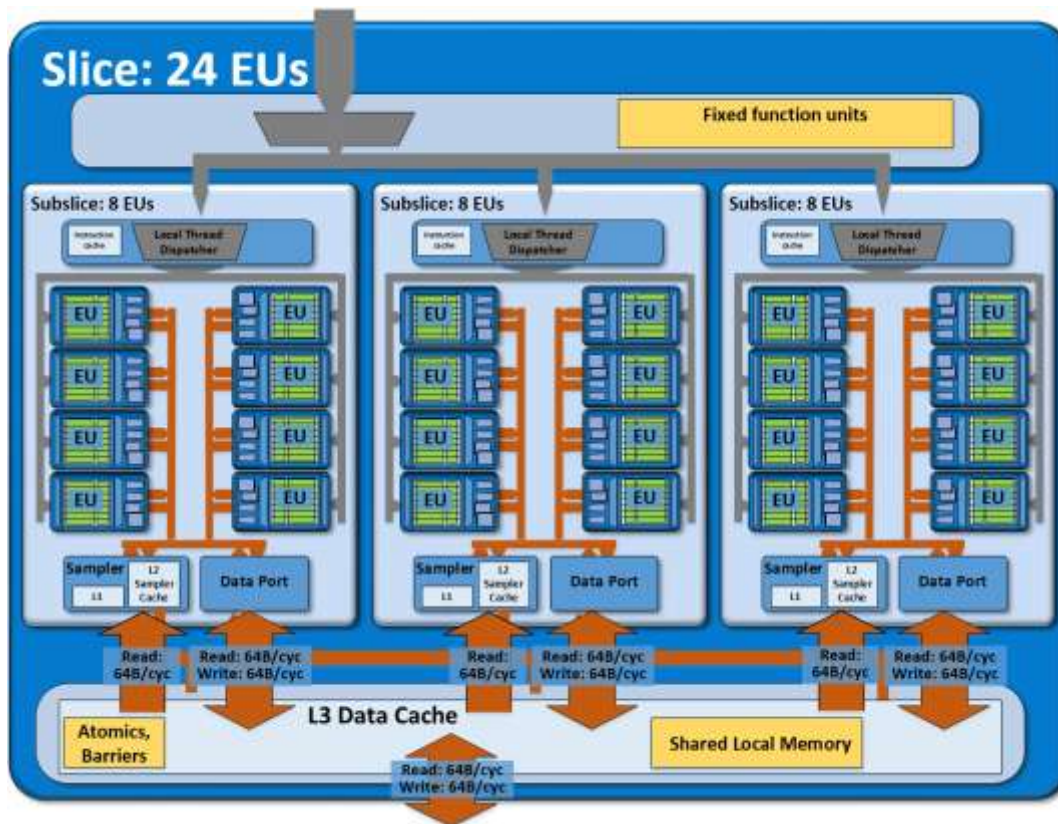


Figure 2-1: Overview of processor graphics (Intel™ HD Graphics 5500 shown)

2.2 Graphics Hardware Improvements in 5th Gen Intel® Core™ Pprocessors

The 5th gen Intel Core processor graphics provides a number of architectural improvements over the 4th gen, namely:

- **Native 2x MSAA mode** - For anti-aliasing, you should first consider a lightweight method like CMAA, but if results are not acceptable, the graphics in 5th gen Intel core processors provides a native 2x MSAA mode. Performance will be faster with 2x MSAA than 4x MSAA or higher. More detail is available in the section on Choose an Appropriate Anti-Aliasing .
- **Full speed 128bpp point sampling** - When you use a RGBA32 texture format, you'll get full throughput from each sampler (4 texels per clock) if you're not bandwidth bound.

2.3 Intel® Turbo Boost Technology 2.0

Both the CPU and processor graphics benefit from Intel® Turbo Boost 2.0 Technology! (sometimes called "Turbo mode"), which increases either the CPU or graphics frequency as needed, when the total system load allows for it. For example, if an application runs a CPU-intensive section, the processor can increase the frequency above its rated upper power level for a limited amount of time using the Intel Turbo Boost Technology. Similarly, if the CPU is not maxed out and the graphics are fully loaded, it is possible for the system to increase the graphics frequency. This technology runs automatically to give your application the best performance available within power and temperature constraints.

3 Tools

To understand how your application runs on an Intel GPU, you'll need some tools. With the correct tools, you'll be able to find and fix performance problems. Below is a list of some useful tools that you can use to identify performance issues.

3.1 Intel® Graphics Performance Analyzers (Intel® GPA)

The Intel Graphics Performance Analyzers is a suite of graphics analysis and optimization tools to help game developers make games and other graphics-intensive applications run faster. Intel GPA provides extensive functionality to allow developers to perform in-depth analysis of graphics API calls and determine where the primary performance issues arise.

Intel GPA can analyze graphics API usage of DirectX 9, 10, and 11 apps on Windows*, and OpenGL ES* 1, 2, and 3 apps on Android* devices with Intel processors. Regardless of the graphics API, you can also use Intel GPA Platform Analyzer to see the detailed CPU load of your Windows or Android app, including any OpenCL™ activity. If you want a closer look, Intel GPA has an API for adding your own instrumentation.



Figure 3-1: Intel® GPA Graphics Monitor is an overlay analysis tool for your DirectX* game.

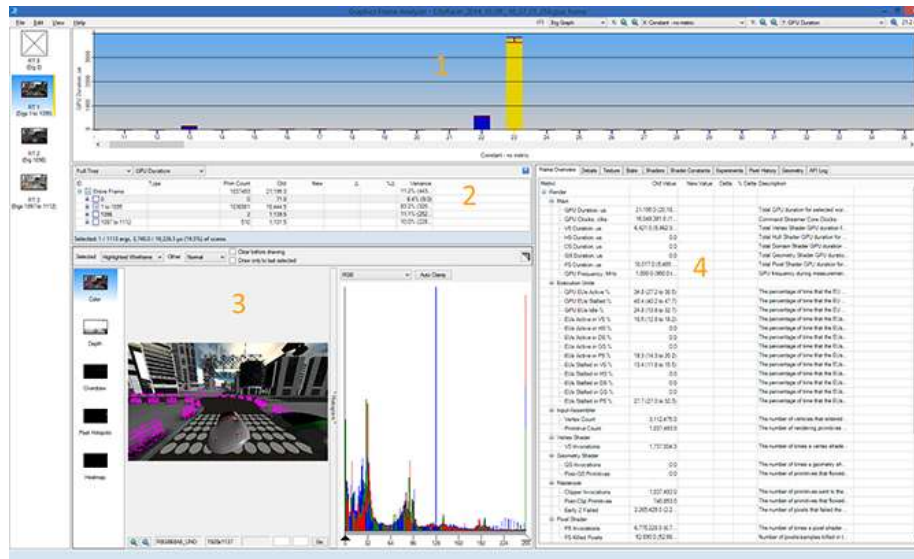


Figure 3-4: Intel® GPA Frame Analyzer shows the activity of each draw call in a frame.

Figure 3-4 shows Intel GPA Frame Analyzer, a tool for deep analysis on a captured frame. The complete frame and all its resources are contained in the frame capture. This lets you study individual draw calls and the state, geometry, textures, and shaders that make up the frame. This in-depth analysis shows a number of metrics, including data on all Intel graphics performance counters, such as the amount of time the EUs are stalled for a particular call or group of calls.

See the [Intel GPA documentation](#) for more details on features and functionality. You can download Intel GPA as part of the [Intel® Integrated Native Developer Experience](#).

3.2 Intel® VTune™ Amplifier XE

If the Intel GPA Platform Analyzer doesn't go deep enough, Intel provides an even more detailed profiling tool: Intel VTune Amplifier XE. One of the Intel® Parallel Studio XE tools, Intel VTune Amplifier helps you study the CPU performance of both serial and parallel behavior, with deep, instruction-level analysis. It can be used within Microsoft Visual Studio* or on its own with the GUI Client. Intel VTune Amplifier is also useful for profiling game engine code and tuning its performance. Intel VTune Amplifier will automatically mark DirectX frames and detect slow frames for analysis, and you can add simple instrumentation to your code to mark OpenGL frames.

You can learn more about Intel VTune Amplifier and download a trial version at [the Intel VTune Amplifier site](#).

4 Important Optimizations

This section introduces some ways to optimize your apps for 5th generation Intel processor graphics. This guide is not definitive, but it outlines the issues that you are most likely to encounter, no matter which graphics API you are using.

4.1 Test on Intel® Processor Graphics

To provide an optimal user experience, you should identify the hardware platform your title is running on and set pre-defined suggested settings based on the known performance characteristics of that platform. This will

provide the best out-of-box experience for your customers. We recommend you test on the target hardware platform to verify playability, as this varies widely depending on the game genre. For some game genres, a rate of 25-30 frames per second is considered playable, but your target may be different. If it's practical, the best approach is to have a live, in-game benchmark to verify that the resolution and quality settings run well. Automatically running that benchmark during initialization will confirm that you have chosen acceptable defaults.

[GPU Detect](#) is a DirectX code sample that shows how to detect the GPU and offers some suggestions for quality pre-sets based on different device IDs. Because there are different packages for each graphics device ID, this sample also contains code for detecting the GPU's operating frequency. This, plus the device ID, will give you a better understanding of the performance of the GPU running your game. With this information, your game can set the default resolution, shader complexity, texture resolution, etc. for your game to give the best trade-off between performance and visual quality. See the GPU Detect section for more information about GPU Detect.

4.2 Use the Newest Graphics Driver

Accompanying the 5th gen processors, Intel released an updated graphics driver. The latest drivers from Intel are available at downloadcenter.intel.com, and the OEM for your device's manufacturer may offer more specialized drivers. You can find open-source drivers for Linux* at O1.org/linuxgraphics. With any of the drivers, the best rendering performance is achieved by focusing on both CPU and GPU efficiency.

The Intel driver spends less time doing complex analysis and reordering of rendering commands, as these tasks are better suited to the app, which has the context to best do these optimizations. As much as possible, the Intel driver submits rendering commands from the application to the hardware with minimal manipulation. Do not expect the driver to reorder calls.

4.3 Find and Fix Memory Bandwidth Problems

As the GPU has become more capable, bandwidth to main memory (RAM) has improved more slowly. Since the GPU now has better throughput than previous generations, memory is more likely to become a bottleneck. The memory hierarchy is shared by the CPU and GPU, so larger CPU workloads will tend to increase demand on memory.

To check if your application has a GPU memory bandwidth bottleneck:

- Using Intel GPA, check the GPU memory reads/writes counters. If the combined reads and writes are 1 GB per frame, typical RAM bandwidth of 25 GB/s would bottleneck the application at 25 FPS, even with an idle CPU.
- If the Sampler Busy % is high, it may indicate that the sampler is waiting on RAM. You can often see this by comparing the frame with the original textures, and then using small textures by selecting the "2x2 Textures" experiment.
- EU stalls can often be either directly caused by waiting on RAM, or indirectly via the sampler.

If you have memory bandwidth bottlenecks, you usually need to either manipulate less data or improve cache usage. Try one or more of the following:

- Compress textures where you can. In DirectX 11, the BC6/7 formats let you compress most static textures with minimal loss of quality.

- Reduce the bit depth of textures and other data. 32-bit floats (per component) are rarely required. Even intermediate HDR data can often be adequately represented by formats like R10G10B10A2 or R11G11B10.
- When possible, don't use lookup textures or other "gradient-like" data. Most of the time, shaders can do the math faster than a lookup. Even when artistic input is required, the result can typically still be compressed by using approximating polynomials, spherical harmonic coefficients, or similar.
- Use branching to avoid memory access. When compositing texture layers, skip the lookup if the weight of the given layer is negligible.
- Wherever possible, condense as many operations as possible into single invocations. Post-processing can be done as a single pass, instead of multiple passes through memory. If you require communication or ping-ponging, use compute shaders and split the screen into tiles (with borders if necessary). Also, prefer tiled deferred shading with a single lighting pass, instead of conventional blending-based deferred shading.
- Aggressively trim and cull particle blending passes. Discard pixels with alpha equal to zero instead of blending them to no effect. Even better, use polygons that fit the particle sprites as tightly as possible (see <http://www.humus.name/index.php?page=Cool&ID=8>, for instance).
- Render scenes at a "useful" resolution, not necessarily full-screen, and then upscale the image to the final rendered size. If you do this, render your UI separately at full resolution to minimize impact.

If your application is running on Iris Pro graphics, it will have more (cached) bandwidth available; thus, cache friendly operations like particle blending will tend to have fewer bandwidth issues. When you can, try to group rendering operations so they consume data shortly after it is generated, to increase the chances the data will still be in cache.

4.4 Choose an Appropriate Anti-Aliasing Method

Instead of starting with a traditional GPU method like multisample anti-aliasing (MSAA), consider using conservative morphological anti-aliasing (CMAA) ([article and sample](#)), which is a lightweight method developed by Intel that works well with Intel processors. If you do use MSAA, carefully consider the performance impact of MSAA and look at the tradeoff between visual improvement and cost for your game. 2x MSAA is now supported natively in the 5th gen processors.

As another alternative to MSAA, consider using post-process antialiasing solutions such as fast approximate antialiasing (FXAA). You should also consider the temporal super sampling anti-aliasing technique. This technique, shown in the [DirectX dynamic resolution rendering sample](#), uses previous frame data to increase the effective sampling rate without ghosting artifacts.

4.5 Minimize State Changes

Sort your draw calls by state to eliminate redundant state changes. One good solution is to encode the state required for each draw call into bit vectors and to radix-sort based on the resulting values. Put the most expensive state changes into highest bits (for instance, changing shaders or render targets) so that they don't change often.

Using instancing and texture arrays can also significantly reduce CPU overhead by having fewer draw calls and associated state changes, so it is usually appropriate to apply this technique. As instance counts rise into the tens of thousands, performance will degrade and you'll need to switch to a lower vertex count instanced model, such as a billboard.

Inside Intel GPA Frame Analyzer, you can study API calls to check the frequency of various state changes. Ideally, only textures and constant buffers should be manipulated at similar frequencies to the frequency of draw calls, and operations such as changing shaders and render targets should be performed as rarely as possible.

4.6 Avoid Unnecessary Clears and Copies

Resource clear and copy operations have significant set-up overhead. Because of this, multiple small clears and copies are expensive to fewer large clears and copies. Limit your use of them.

While you should always clear depth and multi-sampled resources, it isn't always necessary to clear standard color render targets. For instance, when doing deferred shading, it's good enough to clear the depth buffer each frame. There is no need to clear all of the G-Buffer resources since a pixel with the depth buffer set to the depth clear value shows that no geometry is present, so the rest of the G-Buffer data can be ignored. Finally, if your rendering pass will touch all pixels in a target anyway, starting with a clear is unnecessary and may be slow.

You may see recommendations to clear a resource simply to indicate that the data in it is no longer needed, and thus to break any dependency chains. These clears can be expensive, so it's better to avoid them when possible. DirectX 11.1 introduced a "discard" operation with the proper semantics, so these extra clears are no longer required.

Full resource copies are rarely necessary. Any time resource A would be copied to B, you can usually replace references to B with references to A. If future operations would mutate A while B is still in scope, simply introduce another resource for the mutated data instead of modifying A. Since resources can usually only be bound for read *or* write in 3D APIs (the exception being via Unordered Access Views in DirectX or textures bound with `glBindImageTexture` in OpenGL), there is no need to make explicit copies of data.

This leaves sub-resource copies as the only real problem. They are often used to reorganize data, by scattering different blocks into various places in a destination buffer. If there are a large number of these operations, it may be faster to explicitly use rendering operations (e.g., draw triangles) and use the rasterizer to do the scatter operation, which will group many copies into a single draw call. Depending on the amount of data being overwritten in the destination target, it may also be faster to switch to a gather-style operation and have a pixel shader pull the relevant data for each output element.

As with other API activity, you can look for copies and clears among the API calls. Be warned, however, that the performance numbers of these ergs are misleading. For a variety of reasons, copies and clears are often executed "lazily" by the driver, and thus their performance impact may sometimes appear as part of some future operation and not where they were issued.

4.7 (OpenGL only) Use Vertex Buffer Objects and Frame Buffer Objects

When possible, use Vertex Buffer Objects (VBOs). When your code is using VBOs, use `GL_STATIC_DRAW` whenever you can. For any offscreen rendering, use Frame Buffer Objects (FBOs), rather than `EGLPbuffer` or `EGLPtexmaps`.

4.8 (OpenGL only) Use Fence Objects for Synchronization

To ensure synchronization between independent but connected data buffers (e.g., in a producer/consumer algorithm), use fence objects. For a good discussion of this subject, see <http://mobile3dgraphics.blogspot.com/2013/01/consumerproducer-approach-for.html>.

4.9 Avoid Interleaving Reads and Writes

Some resources have additional metadata that requires special handling. In particular, depth/stencil resources have associated hierarchical Z-buffer data that must be “resolved” before certain operations including:

- Binding the resource as a texture
- Copying the resource
- Mapping the resource
- Partially clearing the resource (i.e., not the full array/MIP chain) with a different value than previously

While these resolve operations are often required, try to avoid unnecessary resolves. For instance, rendering some geometry, and then sampling the depth buffer, then rendering some more geometry and sampling again should be avoided. Where it's possible, it's more efficient to group all writing to a given resource together and then do a single resolve before reading.

Depth and stencil are stored separately, even with the D24S8 format. Thus mapping the resource for read-back incurs an additional copy (to recombine depth and stencil), so it's not recommended. Sampling any stencil data (i.e., as a shader resource view) requires a resolve pass and should be avoided.

In general, unless high-frequency masking patterns are required, we recommend using scissor testing, depth-buffer based culling, or shader branching/discard instead of stencil on 5th gen Intel processors.

Similar to copies and clears, you can find these situations by looking at the API activity within a frame, but the performance overhead may not be represented where the API call is shown.

4.10 Use Culling to Avoid Bottlenecks in the Geometry Pipeline

Diagnosing geometry pipeline bottlenecks can be tricky since bottlenecks can occur in a number of different places. EUs that are idle may indicate bottlenecks in the geometry pipeline fixed-function units. Other ratios like a high percentage of culled triangles or a low ratio of pixels to triangles can also point to the need for better geometry culling or level of detail.

The fastest draw calls are the ones that don't get executed, so good culling is recommended. For those draw calls that are made, enable back-face culling. Occlusion culling can skip a large amount of work. Even though implementing occlusion culling can be a significant effort, it's often worthwhile (see the [software occlusion culling](#) sample for one technique).

It's also possible to vastly reduce geometry processing by keeping shadow maps focused on only the visible parts of the scene. See the [sample distribution shadow maps](#) article and code for one way to do this. When the CPU does not know at the time of drawing whether an object is visible, you can use predicated rendering to skip expensive draw calls on the GPU.

Finally, use an efficient view-frustum culling algorithm in your app.

Otherwise, optimizing the geometry pipeline is similar to other GPUs:

- Use indexed primitives for maximum reuse of the vertex cache.
- Optimize meshes in a way that's cache-size-independent with an index reordering algorithm like DirectX's `ID2DX10Mesh::Optimize`.
- Minimize the size and number of vertex attributes.
- When one pass needs fewer vertex attributes (e.g., only positions for depth and shadow passes), split those attributes into a separate vertex buffer and use multiple vertex streams to minimize cache thrashing.

The geometry pipeline in 5th gen Intel Core processors has mostly doubled in throughput to match the throughput increase of the rest of the graphics hardware. One exception, however, is tessellation performance, which runs at 1 domain point per clock.

On the higher-end parts, like Intel Iris graphics and Intel Iris Pro graphics, the tessellator may become a bottleneck on heavily tessellated meshes. Take care to tessellate only as much as necessary. Focus on the areas with the biggest visual impact (e.g., silhouettes). Tessellation also often increases the impact of finer grained culling. It's usually worth implementing visibility culling in the hull shader.

4.11 Find and Fix Shader Bottlenecks

Although increasing EU counts mean many more shader operations can be calculated per frame, complex scenes will still hit bottlenecks. You should optimize your shaders (especially pixel shaders) to save time and power over their many invocations. In particular, you will need to find the best balance between large, branching shaders (which reduce shader state changes) and more specialized shaders (which minimize shader computation). To spot the shaders that are ripe for optimization, use Intel GPA to look for EUs that are active most of the time or a combination of high EU stalls and a mostly idle texture unit.

4.12 Use Shader Literals Instead of Constant Buffers

The hardware can pre-load constant buffer values (also known as uniform buffer objects) into registers if they're indexed by immediate values. Pre-loading can occur when the constants are referenced statically in the shader, instead of with a dynamic array offset. This can be much faster at runtime, but it does increase the amount of registers used, or "register pressure."

When you can, use literal values in the shaders (compile-time constants) instead of reading from constant buffers, especially for values used in loop iteration conditions or for indexing buffers, inputs, or outputs. This is especially useful in compute shaders, where you cannot use the constant pre-loading optimization.

4.13 Minimize Register Pressure

As with any GPU, if you minimize register pressure, you can improve performance since you'll avoid spills and fills. You can minimize register pressure in a variety of ways and eliminate register spill/fills:

- Avoid creating large indexed lookup tables in shaders.
- Use a maximum group size of 512 for compute shaders. Larger group sizes require wide SIMD execution modes that interfere with the compiler's ability to choose the best SIMD size for the given register requirements.

- If you have branches that are complex (i.e., have heavy register use) but those branches are rarely taken, you should split them into separate shaders that run over only the relevant portions of the screen. Expensive blocks may reduce the performance of the entire shader, even if the branches are never taken.

4.14 Use Shader Built-ins

When possible, use shader built-in functions instead of writing your own.

4.15 Favor UAV Counters over Atomic Operations

Minimize the use of global atomic operations (i.e., `InterlockedAdd`) on the same address. Instead, use UAV counters (i.e., `IncrementCounter`) that are optimized for high contention access. To find this issue, look for excessive EU stalls and atomic usage in Intel GPA or use the VTune analyzer to study your atomic operations.

4.16 (OpenGL only) Use glFinish and glFlush Correctly

Many developers confuse when to use `glFinish` and `glFlush`.

- `glFinish` will block until the command stream is fully processed.
- `glFlush` submits the command stream to the hardware and then flushes the memory cache, but does not wait for all commands to be processed.

Check your code to be sure you use the correct one.

4.17 Use Power-of-Two Mipmaps

When mipmapping, use textures that are a power of 2 in both width and height. Textures that aren't a power of 2 can be inefficient.

4.18 (DirectX only) Avoid DirectX 9 Legacy Fixed-Function State

The graphics architecture in 5th gen Intel Core processors does not include special hardware to implement a number of the legacy fixed-function states used by DirectX 9. These must be implemented by the shader compiler instead. Changes to these states require recompiling shaders, which can result in visible hitching and jitter.

Intel recommends avoiding the following states: fog, user clip planes, and alpha test. Instead of using these states, implement equivalent functionality in shaders that need it.

5 Detecting the Amount of Graphics Memory

On first run, most games will attempt to auto-detect various features of the graphics processor for purposes of choosing initial settings. One of the common checks is for graphics memory, which can be misreported on Intel processor graphics when done incorrectly. Since the graphics share the memory controller and last-level cache with the CPU, the GPU has full access to system memory. Because of this, simple queries for "dedicated" video memory give an inaccurate picture of how much memory is actually available for the GPU.

The [GPU Detect](#) sample also shows how to correctly check available video memory for Intel processor graphics using DirectX. Use the methods shown in GPU Detect to detect the total amount of video memory. Other

methods may return only the dedicated graphics memory, or return the sum of the dedicated memory and the shared memory.

Unfortunately, there is no reliable programmatic method for OpenGL or OpenGL ES apps to accurately detect the amount of graphics memory.

6 (DirectX only) Intel® Iris™ Graphics Extensions to DirectX 11 API

The D3D11 driver now supports the Intel Iris graphics extensions to DirectX 11 API, extended features that are not available in the standard API. Since the API does not have a mechanism to expose vendor-specific extensions, Intel provides ways to test if the extensions are available and access the features.

To easily access the extensions, load one of the extension samples (listed in the References) and look at the files `IGFXExtensionHelper.h` and `IGFXExtensionHelper.cpp`.

6.1 Checking if Extensions are Available

Once the helper header and source files have been added into your app, it's a simple matter to check if the extensions are available.

First, call `Init`:

```
HRESULT IGFX::Init( ID3D11Device *pDevice );
```

If that succeeds, call `getAvailableExtensions`:

```
IGFX::Extensions IGFX::getAvailableExtensions( ID3D11Device  
                                                *pd3dDevice );
```

The `IGFX::Extensions` result struct will contain a boolean for each feature if it's available on your current platform. Be sure to check that an extension is available before you use the extension interface.

6.2 Intel® Iris™ Graphics Extension for Pixel Synchronization

With Direct3D 11, the programmable pipeline gained access to random access buffers in the form of unordered access views (UAVs). These UAVs allowed a new set of algorithms, typically relying on atomics or UAV counters to allocate or consume memory from their underlying buffers. However, the API does not guarantee any ordering of shader execution, nor exclusive access to the underlying memory.

Pixel synchronization provides these two guarantees for invocations of pixel shaders that happen on the same pixel location.

To use pixel synchronization, you typically index UAVs based on the current pixel location. This allows for a fixed-size per-pixel memory, with exclusive and primitive-ordered read-modify-write, allowing a host of new algorithms. Intel has created DirectX sample code for this feature at <https://software.intel.com/en-us/articles/oit-approximation-with-pixel-synchronization-update-2014>.

To use pixel synchronization, you must make some small additions to your app's HLSL. To start, include `IntelExtensions.hls1`. Then, initialize the extension by calling:

```
IntelExt_Init();
```

The init step in the HLSL initializes the Intel extension framework, so that the shader compiler will look for extension uses.

```
IntelExt_BeginPixelShaderOrdering();
```

Code following this call, until the end of the shader execution, benefits from the pixel synchronization guarantees. Any UAV access based on the current pixel location will be exclusive and ordered.

Since the region of code after the `IntelExt_BeginPixelShaderOrdering` is executed exclusively for that one pixel location, minimize the amount of code in that region to maximize the EU usage. You can check with Intel GPA to be sure the EUs aren't under-utilized while using the extension.

With arbitrary read-modify-write per-pixel data structures, you can do a variety of things. Some example use are:

1. A limited form of "Render Target Read" (where the render target is accessed through a UAV), as seen in the [programmable blend sample](#) that shows RGBE blending
2. A memory-bounded implementation of [adaptive volumetric shadow maps](#) (AVSM)

You can use this extension for many things, including:

1. A memory-bounded implementation of adaptive order independent transparency (AOIT, as described in the [adaptive transparency paper](#))
2. Normal map decal blending for G-buffers

6.3 Intel® Iris™ Graphics Extension for Instant Access

The 5th gen Intel Core processors use a Unified Memory Architecture (UMA), so the GPU and the CPU share the same physical memory. Therefore, it is a waste of resources (bandwidth and power) to force memory copies when the CPU writes to or reads from GPU resources.

The D3D runtime eliminates some of those resource copies when it can, but there are specific cases that are not optimized. The main cases are Texture and Render-Target resources. Instant Access provides a way to map those resources directly, avoiding any additional memory copy.

Instant Access resources fit within the usual D3D resource model. You create both CPU (staging) and GPU resources, to allow CPU manipulation of resources. However, Instant Access resources are created with special initial parameters.

The easiest way to create the mapping between resources is with the helper function. As before, this is found in the files `IGFXExtensionHelper.h` and `IGFXExtensionHelper.cpp`. Create two textures to share:

```
CreateSharedTexture2D(device, const D3D11_TEXTURE2D_DESC *tex2d,  
                      ID3D11Texture2D **pCPUSharedTexture2D,
```

```
ID3D11Texture2D **pGPUSharedTexture2D, ...);
```

They are then linked together through a call to `CopyResource`.

```
CopyResource(pCPUSharedTexture2D, pGPUSharedTexture2D);
```

Then, you can use the GPU resource as usual for rendering and the CPU resource for CPU access, with both pointing to the same underlying memory.

The data returned in the `pData` field of the `D3D11_MAPPED_SUBRESOURCE` structure is actually a `RESOURCE_EXTENSION_DIRECT_ACCESS::MAP_DATA` structure. This new structure contains information about the GPU resource including the CPU-side pointer to the memory.

```
struct RESOURCE_EXTENSION_DIRECT_ACCESS::MAP_DATA
{
    void*    pBaseAddress;
    UINT     XOffset;
    UINT     YOffset;

    UINT     TileFormat;
    UINT     Pitch;
    UINT     Size;
};
```

`XOffset` and `YOffset` comprise a 2D offset to the start of the sub-resource within the full surface (which can hold multiple sub-resources for a single resource).

The memory referenced by `pBaseAddress` is pointing directly to the underlying memory for the GPU resource.

Instant Access 2D resources like textures and render-targets, unless specifically requested as linearly allocated, are *not* organized in a linear pattern, but instead in an intertwined pattern called tiling. Additionally, the memory addressing used can go through an extra operation called CSX swizzling. The tiling and swizzling operations required for the mapped surface are described through the `MAP_DATA::TileFormat` field.

Various tiling patterns are supported by the hardware, depending on the usage pattern, as detailed in the next sections.

`MAP_TILE_TYPE_LINEAR` is typically used for buffers, or other resources that have been allocated with the flag `RESOURCE_EXTENSION_DIRECT_ACCESS::CREATION_FLAGS_LINEAR_ALLOCATION`. They get addressed as any linear resource, where the element location (x, y) is at memory location

```
pBaseAddress + (YOffset + y) * Pitch + (Xoffset + x) * pixel_size
```

`MAP_TILE_TYPE_TILE_Y_NO_CSX_SWIZZLE` organizes the full surface in 4 KB tiles (a footprint of 128Bx32 rows) that are organized in a row-major pattern, and `MAP_DATA::Pitch` corresponds to the byte count corresponding to a full row of tiles.

The following explains the organization of the tiled data; however, it is **not the recommended way of writing performance code**. See the next section for how to access memory in a high performance way.

The memory address for element (x, y) can be constructed in three steps:

1. Finding the tile containing the element.
2. Finding the offset within the tile of that element.
3. Recombining all the parts together.

Step 1 can conceptually be achieved like this:

```
tileOfInterestX = ((Xoffset + x) * pixel_size) / 128
tileOfInterestY = (Yoffset + y) / 32
```

The 2D address within the tile is:

```
XwithinTile = ((Xoffset + x) * pixel_size) % 128
YwithinTile = (Yoffset + y) % 32
```

To perform Step 2, you need to know the addressing pattern within the tile. It intertwines those 2 offsets in the following bit pattern: $X_6X_5X_4Y_4Y_3Y_2Y_1Y_0X_3X_2X_1X_0$, where $X_6X_5X_4X_3X_2X_1X_0$ is the binary representation of $X_{withinTile}$, and $Y_4Y_3Y_2Y_1Y_0$ is the binary representation of $Y_{withinTile}$.

Thus the offset within the tile can be computed as follows:

```
Offset_x = (XwithinTile & 0xF) | ((XwithinTile & 0x70) << 5)
Offset_y = (YwithinTile & 0x1F) << 4
Offset = Offset_x | Offset_y
```

The full address of the pixel can then be reconstructed:

```
Address = (tileOfInterestY * Pitch) + (tileOfInterestX * 4096) + Offset
```

MAP_TILE_TYPE_TILE_Y adds a simple operation to the way that MAP_TILE_TYPE_TILE_Y_NO_CSX_SWIZZLE generates addresses. It adds a bit operation on the final address, with bits 6 and 9 of the final address XORed together. This is the CSX swizzle operation.

```
FinalAddress = Address ^ ((Address & (1<<9)) >> 3)
```

The sample code for [CPU texture compositing](#) shows how to load data directly to a TileY resource in a performant manner. It builds upon a [method documented](#) by Fabian Giesen. The one major change to the method is making the code tiling-pattern-aware by changing the amount of data processed in the inner loop to process a cacheline-worth of data.

As of the first instantiation of the Instant Access extension, the memory mapping that is returned is Uncached Speculative Write-combine (USWC). As a result, contiguous writes (i.e., consecutive writes to incrementing addresses with no gaps) are required to see the performance improvements from the extension. Also, reads from the resource need to take special care to read the data in cacheable memory for increased throughput.

While the extension allows sharing memory, it still requires that you Map and Unmap your resource to get the pointer to the underlying memory. This acts as both a synchronization point for resources written to (so that the GPU can finish writing to the resource before the CPU can see the new data), as well as a cache coherency notification to the driver.

The Map call follows the same synchronization rules as the regular Map call, with the D3D types and flags providing the typical synchronization features (`D3D11_MAP_WRITE_DISCARD`, `D3D11_MAP_WRITE_NO_OVERWRITE`, `D3D11_MAP_FLAG_DO_NOT_WAIT`, etc.).

7 (OpenGL only) Intel® Iris™ Graphics Extensions to OpenGL

Two OpenGL extensions are similar to those discussed above for DirectX. One extension lets you efficiently enforce the ordering of pixel accesses within fragment shaders. The other lets CPU code access the GPU's memory for faster texture uploads and accesses.

7.1 Fragment Shader Ordering

Fragment shaders may run in parallel but access the same coordinates. This can cause non-deterministic behavior. To synchronize access to shared coordinates, use this extension. When you enable the extension in your fragment shader, it blocks fragment shader invocations until invocations from previous primitives (that map to the same coordinates) are done executing. This also blocks the same sample, when per-sample shading is active.

For more details on how to enable and control this extension, see the description in the OpenGL registry at http://www.opengl.org/registry/specs/INTEL/fragment_shader_ordering.txt.

7.2 Map Texture

Since the CPU and GPU share the same physical memory, it is possible to speed up texture upload and access. With the map texture extension, the CPU has direct access to the GPU texture memory.

One challenge with this extension is that textures are often tiled. Texels are kept in a specific layout to improve locality of reference for fast texturing. When using this extension, textures are linear. Although this may slow down texture sampling, it can be more than offset by the performance gained when uploading the textures in the first place.

To learn more about this extension, see the OpenGL registry at http://www.opengl.org/registry/specs/INTEL/map_texture.txt.

8 (DirectX only) DirectX 9 FOURCC Extensions

For DirectX 9, Table 8-1 shows the supported format extension. Refer to the code in GPU Detect for run-time detection on Intel hardware, as support may also depend on the driver version.

Table 8-1: Supported DirectX* 9 Format Extensions.

DirectX* 9 Format	Description
NULLRT	Null render target for use when rendering to depth or stencil buffer with no color writes. Consumes no memory.
ATI1N	One component-compressed texture format similar to DXGI_FORMAT_BC4.
ATI2N	Two component-compressed texture formats similar to DXGI_FORMAT_BC5.
INTZ	Allows native depth buffer to be used as a texture.
RESZ	Allows a multisampled depth buffer to be resolved into a depth stencil texture.
DF24	Allows native depth buffer to be used as a texture.
DF16	Allows native depth buffer to be used as a texture.
ATOC	Alpha to coverage for transparency multisampling.
ATIATOC	Alpha to coverage for transparency multisampling (alternate implementation).

9 Power Efficiency

Mobile and ultra-mobile computing are becoming ubiquitous. As a result, battery life has become a significant issue. As manufacturing processes continue to shrink and improve, we see improved performance-per-watt characteristics of CPUs and processor graphics. However, there are many ways that software can reduce power use on mobile devices.

9.1 Power Background

The Advanced Configuration and Power Interface (ACPI) is the modern standard used by operating systems to optimize power use. The processor implements power and thermal management features to correspond to the ACPI features. Processors execute in different power states, known as "P-States" and "C-States." These power states define how much the processor is sleeping.

How do you determine power state behavior? You measure how much time your app is spending in each state. Since each state uses a different amount of power, you'll get a picture over time of your app's power use.

To start, measure your app's power baseline usage in three cases:

1. At idle, for example in the UI
2. Under average load, for example during an average scene with average effects
3. Under worst-case load, for example in a maximum scene with maximum effects

These are your app's Idle, Average, and Max Power.

Your worst-case load may not be where you think it is. We have seen very high frame rates (1000 FPS) for cut-scene playback in shipping apps. This causes the GPU and CPU to use unnecessary power.

You should also measure how long (on average) your app can run on battery, and you should compare your app with other similar apps.

Measuring power consumption regularly will inform you if any recent changes caused your app to use more power.

Battery Life Analyzer (BLA) is a good (Windows-only) tool for this pre-work. See this [article showcasing BLA](#) to collect high-level data and analyze the app's power use. If this data shows you have issues residing in the wrong C-States for too long, then it's time to look deeper.

9.2 Power Efficient Programming

You write more power-efficient code by taking some simple steps.

Be aware of system power settings and power profile, and scale your app's settings

Although it was once necessary to poll for this data (e.g., using `GetSystemPowerStatus()`), since Windows Vista*, Windows supports asynchronous power notification APIs. Use `RegisterPowerSettingNotification()` with the appropriate GUID to track changes.

Scale your app's settings and behavior based on the power profile and whether your device is plugged in to power. Consider scaling resolution, reducing the max frame rate to a cap and reducing quality settings. Review this article, which [shows possible power efficiency gains](#) by watching for changed power states.

If your app has high frame rates during cut-scenes, menus, or other low-GPU-intensive parts, it would still look fine if you locked the Present interval to a 60 Hz display refresh rate. Watch for this behavior in menus, loading screens, and other low-GPU-intensive parts of games.

You could use the V-Sync mechanism in DirectX or OpenGL, but you can also manage frame rate and resolution yourself. The [dynamic resolution rendering](#) sample shows how to adjust frame resolution to maintain a frame rate.

Run as slow as you can, while remaining responsive

Detect when you are in a power-managed mode and limit frame rate. This will prolong battery life and also allow your system to run cooler. For benchmarking, it is sensible to be able to disable the frame rate limit, but warn your player that they will discharge the battery quickly. You may also want to let the player control the frame rate cap.

The UI usually changes much more slowly than the game scene. Often the UI is limited to small panels for displays like health, powerups, and other status. Use off-screen buffers and do smart compositing. Here again, Dynamic Resolution Rendering may be useful, to decouple UI rendering from main scene rendering.

Running at 30 Hz instead of 60 Hz can save significant power. Consider reducing the scene to 30 Hz when on battery.

Manage timers and respect system idle

Reduce your app's reliance on high-resolution periodic timers. Avoid use of `Sleep()` calls in tight loops and instead use `Wait*()` APIs since using `Sleep()` or any other "busy-wait" API can cause the OS to keep the machine from being in the Idle state. The [mobile platform idle optimization](#) article has a good section on which APIs to use and not use.

```

HRESULT res;

IDirect3DQuery9 *pQuery;

// create a query
res = pDevice->CreateQuery(..., &pQuery);

...

// busy-wait for query data
while ( (res = pQuery->GetData(..., 0)) == S_FALSE);

```

Figure 9-1: Power Inefficient Graphics Busy Wait Loop, using DirectX*.

In addition to `Sleep()` calls, avoid any “busy-wait” calls. In Figure 9-1, the D3D query will be called repeatedly, causing unnecessary power use. There’s no way for the OS or the power management hardware to tell that the code does nothing useful.

Multithread sensibly

Balanced threading has performance benefits, but you need to consider this along with the GPU. Imbalanced threading can result in lower performance and reduced power efficiency. Try to avoid affinitizing threads, so the OS can schedule threads directly. If you must, provide hints using `SetIdealProcessor()`.

Make sensible use of hardware blocks and devices

If your data I/O and network usage is poor, the optimizations you performed on the CPU and GPU will be diminished. This [article](#) does a great power analysis of various I/O strategies, and this [article](#) adds a bit more data in the context of SATA hard disk reads. The [latter article](#) also has a section on data transfer across a network and the costs/benefits of compression strategies.

Use the right loops and algorithms

[Avoid use of tight loops.](#) If you have a polling architecture that uses a tight loop, convert it to an event-driven architecture. If you must poll, use the largest polling interval possible.

Study your algorithms; favor ones that finish sooner, allowing the processor to reach idle. Be smart about use of memory, and take advantage of your caches through choice of algorithm and data decomposition.

9.3 Power Analysis Workflow

How can you tell if it’s working?

Baseline

Start with [BLA](#). Measure baselines under idle, average, and worst-case loads. Measure similar apps for comparison.

If your app is reported as “deficient” or there are unexpected wakeups, then you should start optimizing for power. Look at [Windows Performance Analyzer](#). This article [showcases workflow using WPA](#) for CPU analysis.

Measure during development, to avoid surprises

Measure regularly and compare against the baselines.

Use platform power tools to optimize usage

If code modifications introduce a new power deficiency, restart your optimization process. VTune Amplifier XE is also useful to get power call stacks since it can identify the cause of the wake-up. Use this data to reduce or consolidate wake-ups, thus remaining in a lower power state longer.

“Rinse and repeat”

As with other kinds of optimization, power optimization needs to be done frequently. Repeat when your measurements show the need.

10 Sample Code

10.1 GPU Detect

This DirectX sample demonstrates how to get the vendor and ID from the GPU. For Intel processor graphics, the sample also demonstrates a default graphics quality preset (low, medium, or high), support of DX9 and DX11 extensions, the recommended method for querying the amount of video memory, and if supported by the hardware and driver, the recommended method for querying the minimum and maximum frequencies.

The sample uses a config file that lists many Intel GPUs, by vendor ID and device ID, along with a suggested graphics quality level for that device. You should test some representative devices with your app and decide what quality level is right for each.

If you need to know how much video memory is available, call `getVideoMemory()`.

For the device ID, call `getGraphicsDeviceInfo()` and then use `getDefaultFidelityPresets()` to get the suggested configuration from the config file. To see the supported extensions with your current device and driver, look at `checkDX9Extensions()` and `CheckDxExtensionVersion()`. To measure the GPU frequency, use `getGPUFrequency()`. The combination of these results for your device and driver will help you understand how you should set defaults for your app.

Intel's newest processors may have large variations in GPU frequency, even between devices with the same device ID. This is because there are many different packages, with different power and frequency options to support a wide variety of devices in the market. While this was true in earlier generations, this generation has more variations than ever.

So, don't forget to check the GPU frequency in addition to the device ID when you choose resolution and quality settings.

Visit the [GPU Detect page](#) for details.

11 References

To find the most recent graphics developer guide, as well as earlier versions, see the [developer guide page](#).

To see the latest code samples for Intel processor graphics, see the [Intel® Game Developer Community](#).

Other samples you may find useful:

- [Dynamic resolution rendering sample](#)
- [Software occlusion culling sample](#)
- [Sample distribution shadow maps](#)
- [Programmable blend sample](#)
- [Adaptive volumetric shadow maps](#)
- [Adaptive transparency paper](#)
- [CPU texture compositing](#)
- [Dynamic resolution rendering](#)
- [Conservative morphological anti-aliasing article and sample](#)
- [OpenGL fragment shader ordering extension](#)
- [OpenGL map texture extension](#)

Power references:

- [ACPI Basics](#)
- [C-States](#)
- [Power Community](#)
- [Energy Efficient Guidelines](#)
- [Creating Energy Efficient Software Part 1](#)
- [Creating Energy Efficient Software Part 2](#)
- [Creating Energy Efficient Software Part 3](#)
- [Creating Energy Efficient Software Part 4](#)
- [Energy Efficient Platforms - Considerations for Applications](#)
- [Using BLA for Studying Power](#)
- [Maximizing Power on Mobile Platforms](#)
- [Mobile Platform Idle Optimization](#)
- [Idle Software Battery Life](#)
- [Enabling Games for Power](#)
- [Power Analysis of Disk IO Methodologies](#)
- [Power Enabling on Windows Vista](#)
- [Windows 8 Power Optimization](#)
- [Optimizing Windows 8 for Connected Standby](#), good discussion of tools and flow especially WPA
- [Power Analysis Guide for Windows](#), has a good set of steps
- [Fine-grained Analysis](#), follows similar steps to those recommended here and elsewhere

Other tools you may find useful:

- [Request access](#) to the Intel Battery Life Analyzer
- [Intel® PowerGadget](#)

Developer's Guide for Intel® Processor Graphics

- [Intel® Energy Checker SDK](#)
- Microsoft [Windows Performance Analyzer](#)
- [Intel® PowerInformer](#)

Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: <http://www.intel.com/design/literature.htm>

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel, the Intel logo, Intel Core, Iris, and VTune are trademarks of Intel Corporation in the U.S. and/or other countries.

Copyright © 2015 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

OpenCL and the OpenCL logo are trademarks of Apple Inc and are used by permission by Khronos.

ⁱ Requires a system with Intel® Turbo Boost Technology. Intel Turbo Boost Technology and Intel Turbo Boost Technology 2.0 are only available on select Intel® processors. Consult your PC manufacturer. Performance varies depending on hardware, software, and system configuration. For more information, visit <http://www.intel.com/go/turbo>.